INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal , Hyderabad -500 043

DATABASE MANAGEMENT SYSTEMS

2018 -2019

Subject Code: ACS005

Regulations : R16

Class: II B.Tech

Branch: IT

Prepared By

Ms. K Laxmi Narayanamma
Mr.N Bhaswanth

DATABASE MANAGEMENT SYSTEMS

UNIT-I

Introduction-Database System Applications

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

Purpose of Database Systems

- In the early days, database applications were built directly on top of file systems
- Drawbacks of using file systems to store data:
  - Data redundancy and inconsistency
  - Difficulty in accessing data
  - Data isolation — multiple files and formats
  - Integrity problems
  - Atomicity of updates
    - Example: Transfer of funds from one account to another should either complete or not happen at all
  - Concurrent access by multiple users
    - Example: Two people reading a balance and updating it at the same time
  - Security problems

Files vs. DBMS

	DISADVANTAGES OF FILE SYSTEMS	ADVANTAGES OF DBMS
1	Data v/s program problem: Different programs access different files	One set of programs access all data
2	Data inconsistency problem As same data resides in many different files across the programs data inconsistency increases	Related data resides in same storage location minimizing data inconsistency
3	Data isolation problem As data is scattered in various files and in different formats it is difficult to write new programs to retrieve appropriate data	As data resides in same storage location it is easy to write new programs to retrieve appropriate data
4	Security problem:Every user can acces all data	Every user can access only needed data

5	Integrity problem:Develop new consistent range in exixting systems appropriate code must be added in various application program	Integrity Solution:appropriate code must be added in one application program that access all data at one time
6	Problem in accessing data:new appropriate program has to be written each time	DBMS consists of one or more programs to extract needed information
7	Atomicity problem:If system fails it must ensure data are restored to consistent state	It ensures atomicity
8	Data Redundancy:same information is dupliacated in several files ,so higher storage and access cost	One copy of data resides so minimium storage and access cost
9	Concurrency problem:Due to redundant data if many users access same copy leads to concurrency problem	Avoids concurrency problem since data last changed remains permanent

Levels of Abstraction

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

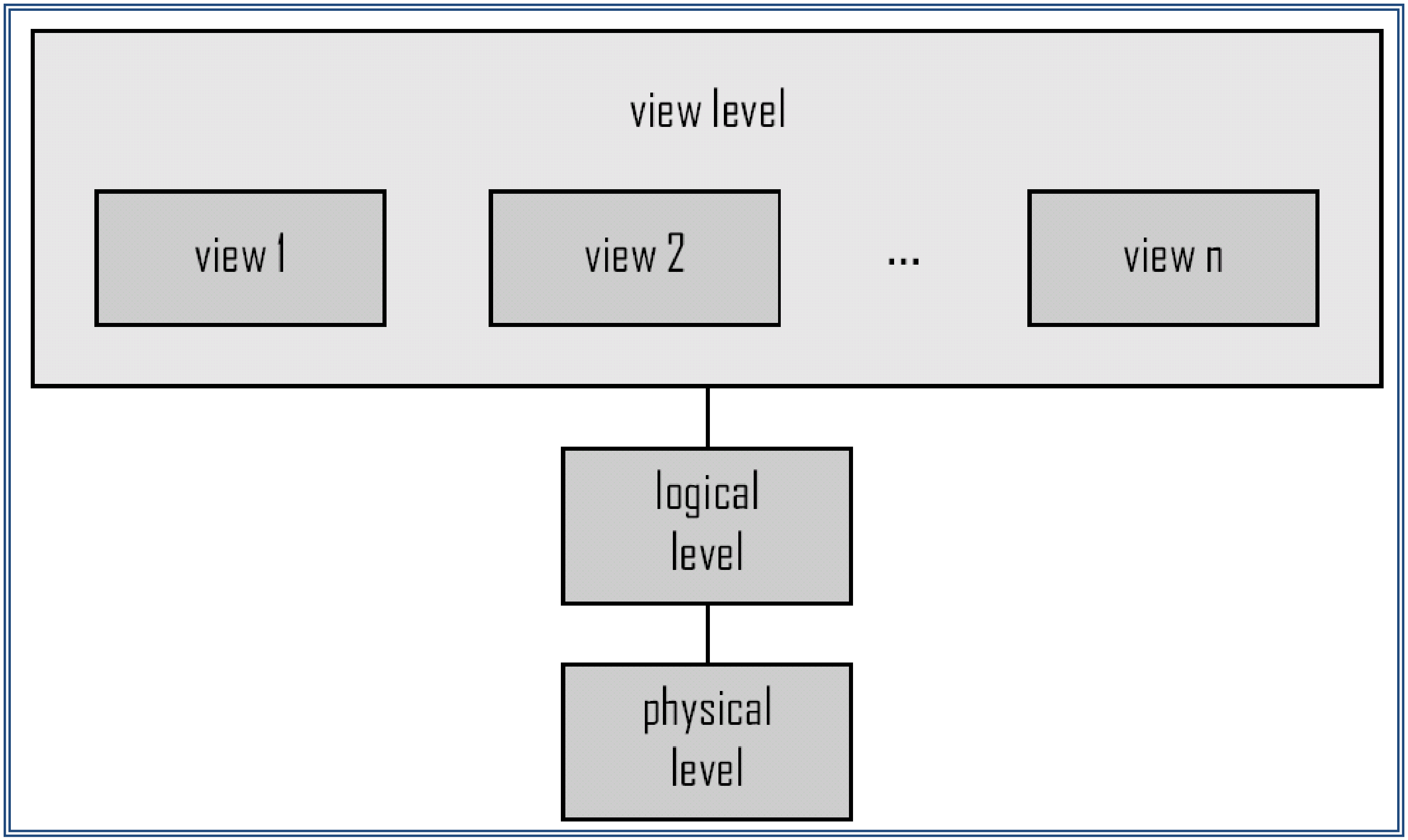
  type *customer* = record

  customer_id : string;
  customer_name : string;
  customer_street : string;
  customer_city : string;

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

View of Data

An architecture for a database system



Instances and Schemas

- Instance – the actual content of the database at a particular point in time
  - Analogous to the value of a variable

- Similar to types and variables in programming languages
- Schema – the logical structure of the database

  - Example: The database consists of information about a set of customers and accounts and the relationship between them)

  - Physical schema: database design at the physical level

  - Logical schema: database design at the logical level

Example: University Database

- Conceptual schema:
  - *Students(sid: string, name: string, login: string,age: integer, gpa:real)*
  - *Courses(cid: string, cname:string, credits:integer)*
  - *Enrolled(sid:string, cid:string, grade:string)*
- Physical schema:
  - Relations stored as unordered files.
  - Index on first column of Students.
- External Schema (View):
  - *Course_info(cid:string,enrollment:integer)*

Data Independence

- The ability to modify the schema in one level without affecting the schema in next higher level is called data independence.

- **Logical data independence**: The ability to modify the logical schema without affecting the schema in next higher level (external schema.)

- **Physical Data Independence** – the ability to modify the physical schema without changing the logical schema

Data Models

- Underlying the structure of database is data model.
- It is a collection of tools for describing
  - Data ,Data relationships,Data semantics & consistency constraints

  Data model types
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semi structured data model (XML)
- Other older models:
  - Network model
  - Hierarchical model

Relational Model

- Example of tabular data in the relational model

Attributes

customer_id	customer_name	customer_street	customer_city	account_number
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-101
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-201
677-89-9011	Hayes	3 Main St.	Harrison	A-102
182-73-6091	Turner	123 Putnam St.	Stamford	A-305
321-12-3123	Jones	100 Main St.	Harrison	A-217
336-66-9999	Lindsay	175 Park Ave.	Pittsfield	A-222
019-28-3746	Smith	72 North St.	Rye	A-201

A Sample Relational Database

customer_id	*customer_name*	*customer_street*	*customer_city*
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

account_number	*balance*
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

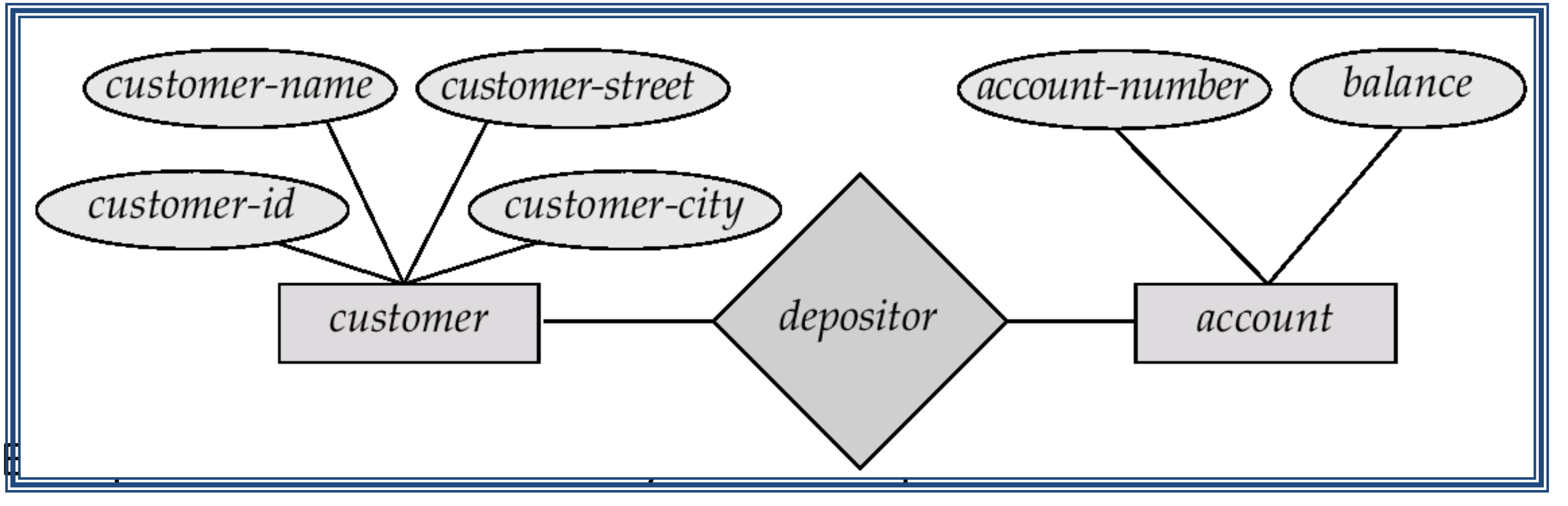
customer_id	*account_number*
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Entity-Relationship Model

An entity is a thing or object in the real world that is distinguishable from other objects.

- Rectangles represent entities
- Diamonds represent relationship among entities.
- Ellipse represent attributes
- Lines represent link of attributes to entities to relationships.



Object based data models

- It is based on object oriented programming language paradigm.
- Inheritance,object identity and encapsulations
- It can be seen as extending the E-R model with opps concepts.
- Semi structured data models
- Semi structured data models permit the specification of data where individual data items of same type may have different set of attributes.
- XML language is widely used to represent semi structured data

Database languages

2 types:

- Data definition language- to define the data in the database
- Data Manipulation language- to manipulate the data in the database

Data Definition Language (DDL)

- Specification notation for defining the database schema
- DDL is used to create the database, alter database and delete database.

  Example: create table *account* (
  account_number char(10),
  branch_name char(10),
  balance integer)
- DDL compiler generates a set of tables stored in a *data dictionary*
- Data dictionary contains metadata (i.e., data about data)
  - DDL is used by conceptual schema
  - The internal DDL or also known as Data *storage and definition* language specifies the storage structure and access methods used
    - DDl commands are Create, Alter and Drop only.

- Data values that are stored in database must satisfy certain consistency constraints

- Domain constraints(DC):A domain of possible values must be associated with every attribute

- Referential Integrity

- Assertions:conditions that database must always satisfy

- Authorization

Data Manipulation Language (DML)

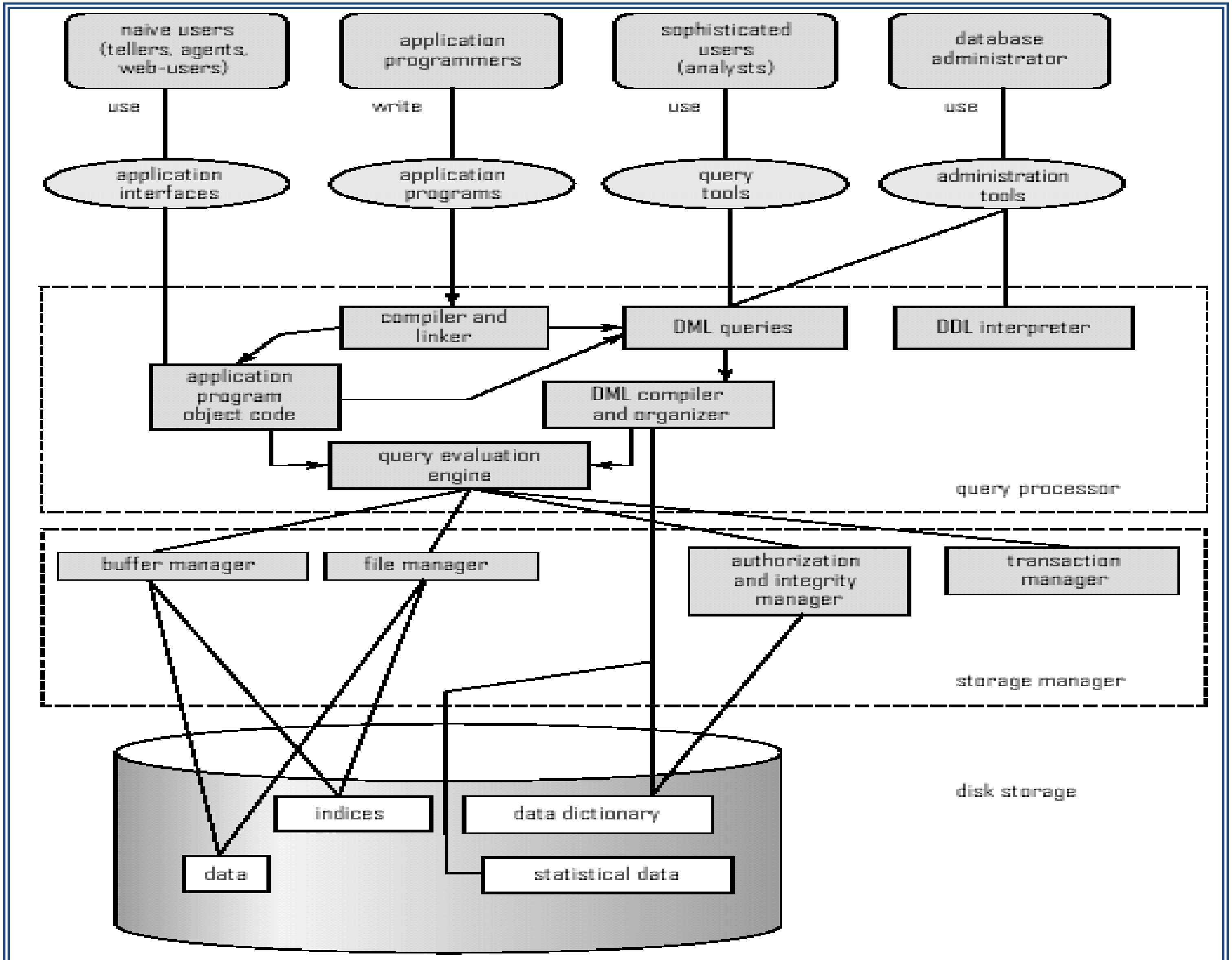
- Language for accessing and manipulating the data organized by the appropriate data model
  - DML also known as query language
  - DML is used to retrieve data from database, insertions of new data into database ,deletion or modification of existing data.
- Two classes of languages
  - Procedural – user specifies what data is required and how to get those data
  - Declarative (nonprocedural) – user specifies what data is required without specifying how to get those data
- SQL is the most widely used query language

Database access from application programs

- To access db, DML stmts need to be executed from host lang.
- 2 ways- a)by providing appn prgm interface that can b used to send DML and DDL stmts to database and retrieve results. Ex:ODBC & JDBC

- B)By extending host language syntax to embed DML calls within the host lang prgm.

Overall System Structure



Data storage and Querying

- Storage management
- Query processing
- Transaction processing

Storage Management

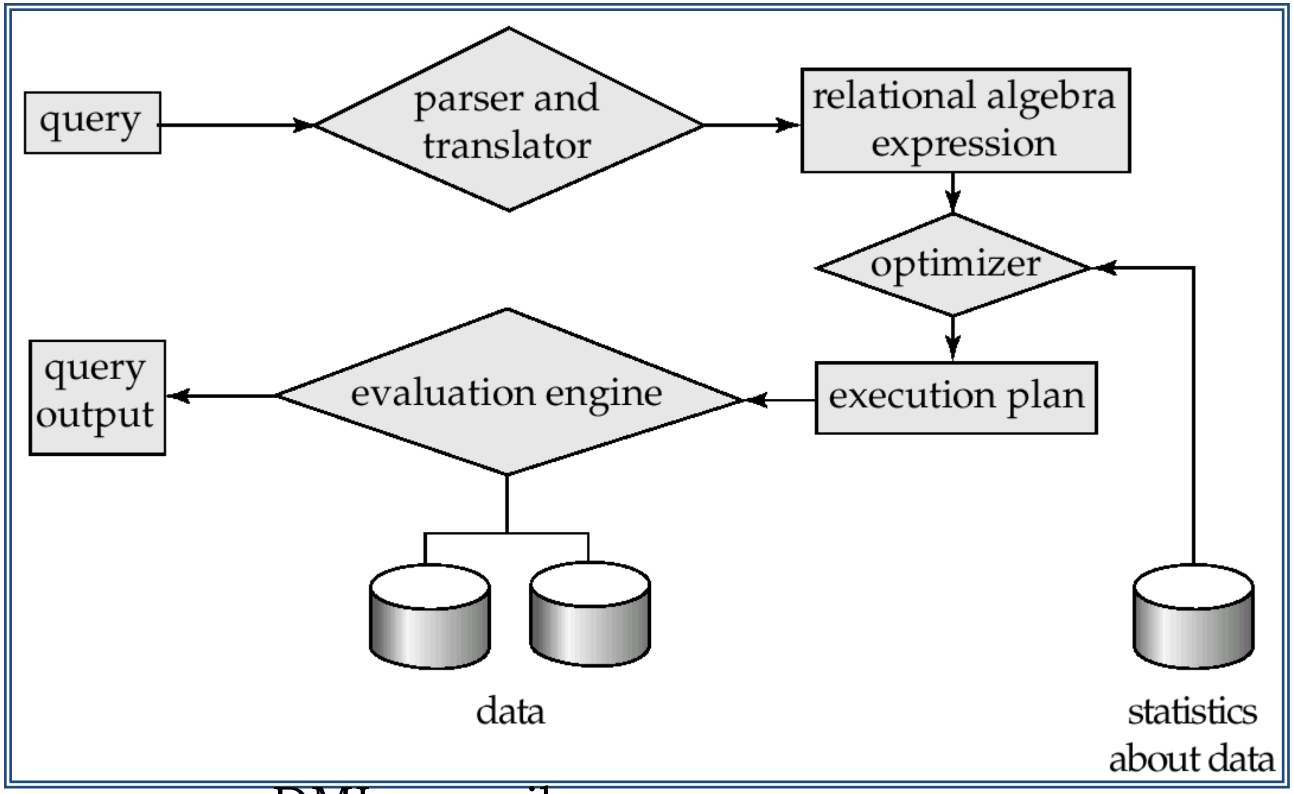
- Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
  - Interaction with the file manager
  - Efficient storing, retrieving and updating of data
- Storage mngr implements several data structures
  - Data files
  - Data dictionary
  - Indices

- Authorization and integrity mngr tests for satisfaction of integrity constraints and checks the authority of users to access the data
- Transaction mngr ensures databse remains in consistent state despite system failures and concurrent transaction executions proceed without conflicting
- File mngr manages allocation of space on disk storage and the data structures used to represent data on disk
- Buffer mngr which is responsible for fetching data from disk storage into main memory and deciding what data to cache in main memory

Query Processing

DDL interpreter interprets DDL stmts and records the definitions in data dictionary

1. Parsing and translation
2. Optimization
3. Evaluation



DML compiler

Transaction Management

- A transaction is a collection of operations that performs a single logical function in a database application
- Transaction-management component ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- Concurrency-control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Database Users

Users are differentiated by the way they expect to interact with the system

- Application programmers –are computer professionals who write appn prgms. They use RAD tools to construct forms and reports with minimum programming effect.
- Sophisticated users – interact with the system without writing programs, instead they form their requests in a database query language
- Specialized users – write specialized database applications that do not fit into the traditional data processing framework
- Ex:Computer aided design systems, knowledgebase expert systems.
- Naïve users – invoke one of the permanent application programs that have been written previously
  - Examples, people accessing database over the web, bank tellers, clerical staff

Database Administrator

- Has central control of both data and programs to access that data.
- Coordinates all the activities of the database system
  - has a good understanding of the enterprise’s information resources and needs.
- Database administrator's duties include:
  - Storage structure and access method definition
  - Schema and physical organization modification
  - Granting users authority to access the database
  - Backing up data
  - Monitoring performance and responding to changes
  - Periodically backing up the database,either on tapes or onto remote servers.

History of Database Systems

- **1950s and early 1960s:**
- First general purpose DBMS was designed by charles bachman at general electric was called Integrated data store. He is first to receive ACM'S turing award(1973).
  - Data processing using magnetic tapes for storage
    - Tapes provide only sequential access
  - Punched cards for input
- **Late 1960s and 1970s:**
- In late 1960's IBM developed information mangmt system(IMS) DBMS used even today in major installations.
  - Hard disks allow direct access to data
  - Network and hierarchical data models in widespread use
  - In 1970 Edgar Codd defined new data representation framework - relational data model.
  - ACM'S turing award(1981).

- **1980s:**
  - Research relational prototypes evolve into commercial systems
    - SQL becomes industry standard
  - Parallel and distributed database systems
  - Object-oriented database systems
- **1990s:**
  - Large decision support and data-mining applications
  - Large multi-terabyte data warehouses
  - Emergence of Web commerce
- **2000s:**
  - XML and XQuery standards
  - Automated database administration
  - Increasing use of highly parallel database systems
  - Web-scale distributed data storage systems

Introduction to Database design and ER diagrams

- The database design can be divided into 6 steps.ER model is relevent to first 3 steps

1.Requirement analysis

2.Conceptual database design

3.Logical database design

4.Schema refinement

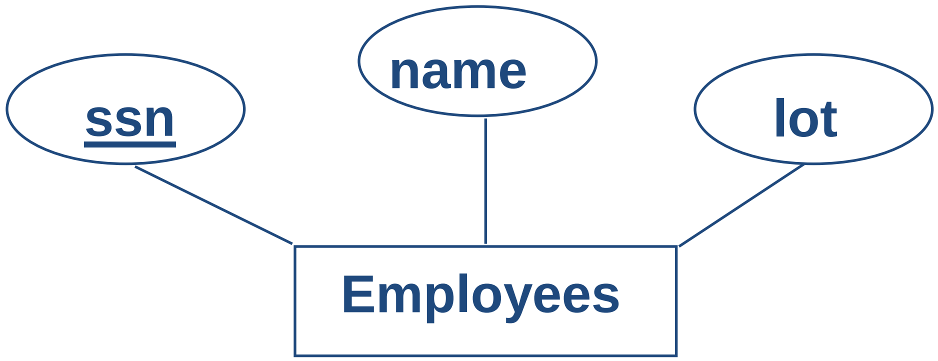
5.Physical database design:.Ex:Indexes

6.Application and security design

Database Design

- <u>*Conceptual design*</u>: *(ER Model is used at this stage.)*
  - What are the *entities* and *relationships* in the enterprise?
  - What information about these entities and relationships should we store in the database?
  - What are the *integrity constraints* or *business rules* that hold?
  - A database `schema’ in the ER Model can be represented pictorially (*ER diagrams*).
  - Can map an ER diagram into a relational schema.

ER Model Basics



- *Entity:* Real-world object distinguishable from other objects. An entity is described (in DB) using a set of *attributes*.

- *Entity Set*: A collection of similar entities. E.g., all employees.
  - All entities in an entity set have the same set of attributes.
  - Each entity set has a *key*.(minimal set of attributes whose values uniquely identify entity in set)
  - Each attribute has a *domain*.

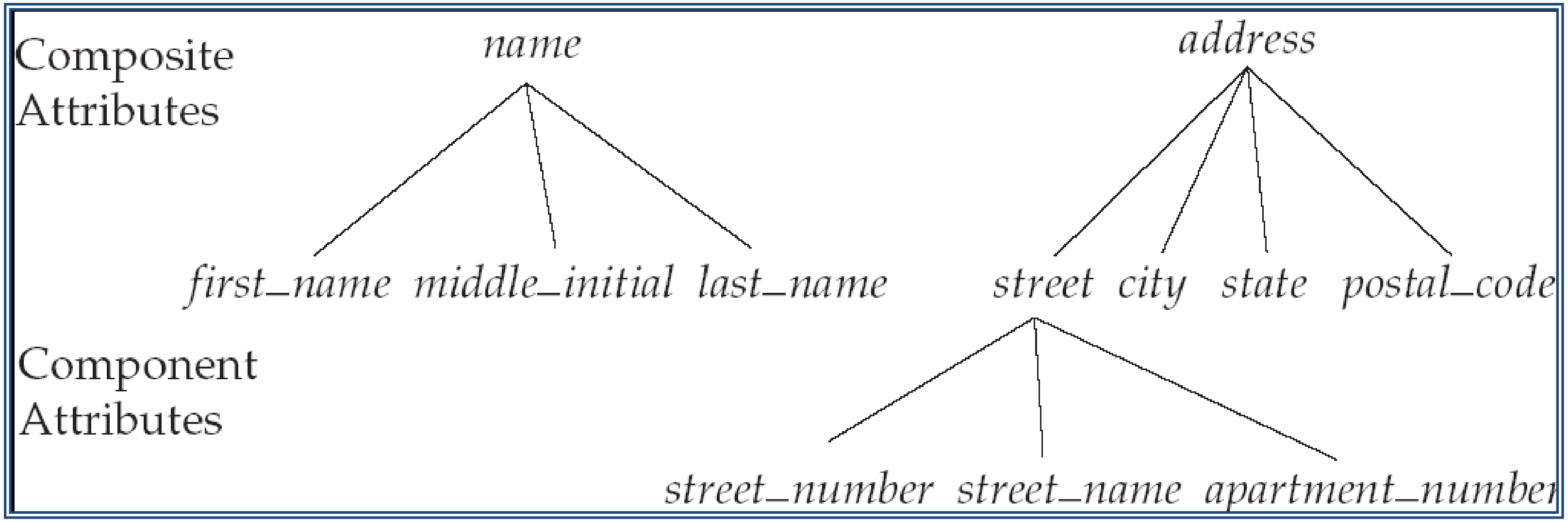
Attributes

- An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.

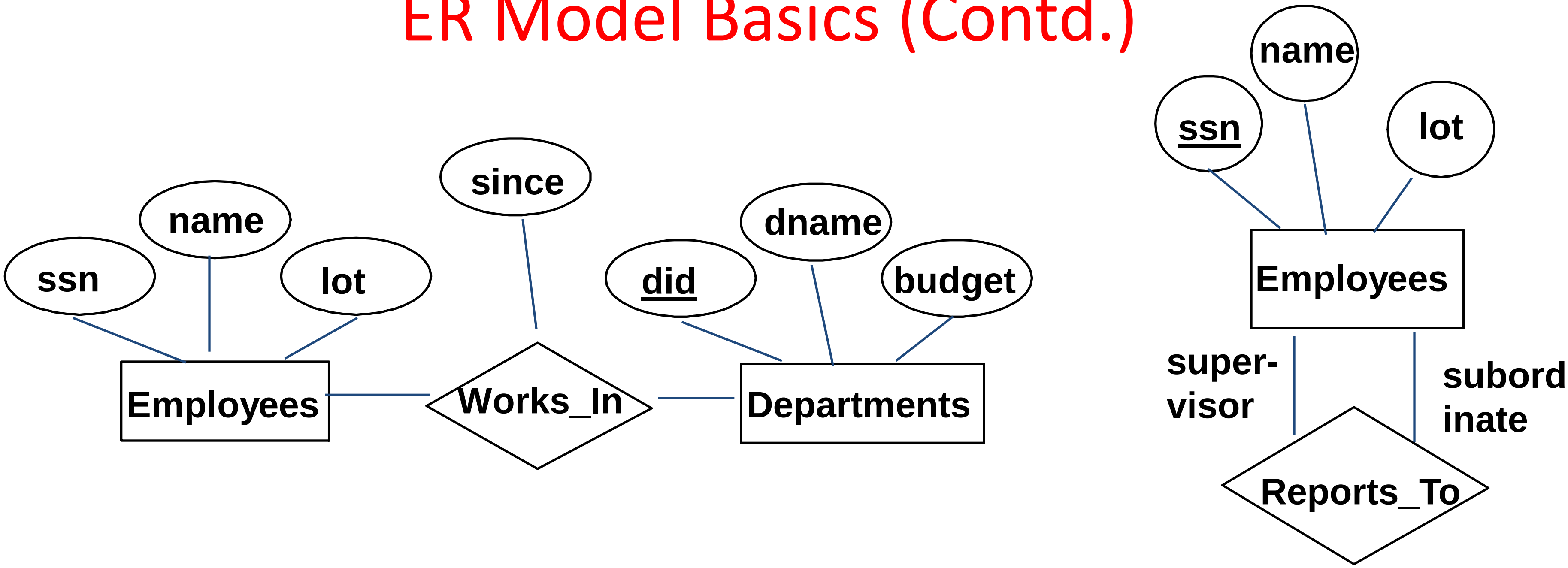
  Example:

  customer = (*customer_id, customer_name,*
  customer_street, customer_city)
  loan = (*loan_number, amount*)
- **Domain** – the set of permitted values for each attribute
- Attribute types:
  - *Simple* and *composite* attributes.
  - *Single-valued* and *multi-valued* attributes
    - Example: multivalued attribute: *phone_numbers*
  - *Derived* attributes
    - Example: age, given date_of_birth

Composite Attributes



ER Model Basics (Contd.)



- *Relationship*: Association among two or more entities. E.g., Attishoo works in Pharmacy department.
- *Relationship Set*: Collection of similar relationships.
- {(e1,…e2)|e1ЄE1, e2ЄE2….. enЄEn}

Relationship Sets

- A relationship is an association among several entities

  Example:

Hayes	*depositor*	A-102
customer entity	relationshipset	*account* entity

- A relationship set is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

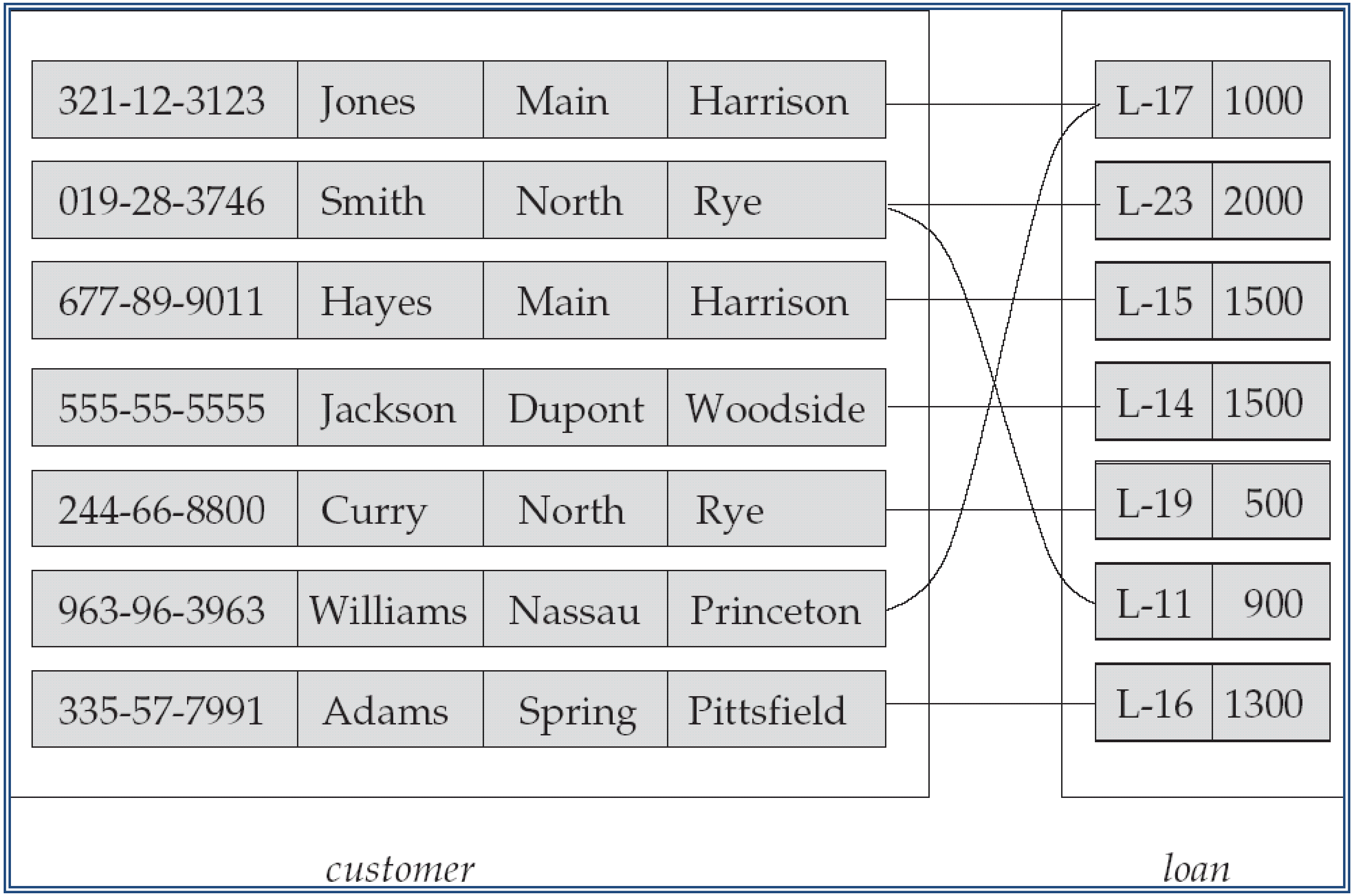
$$\{(e_1, e_2, \ldots e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

  where $(e_1, e_2, \ldots, e_n)$ is a relationship

  – Example:

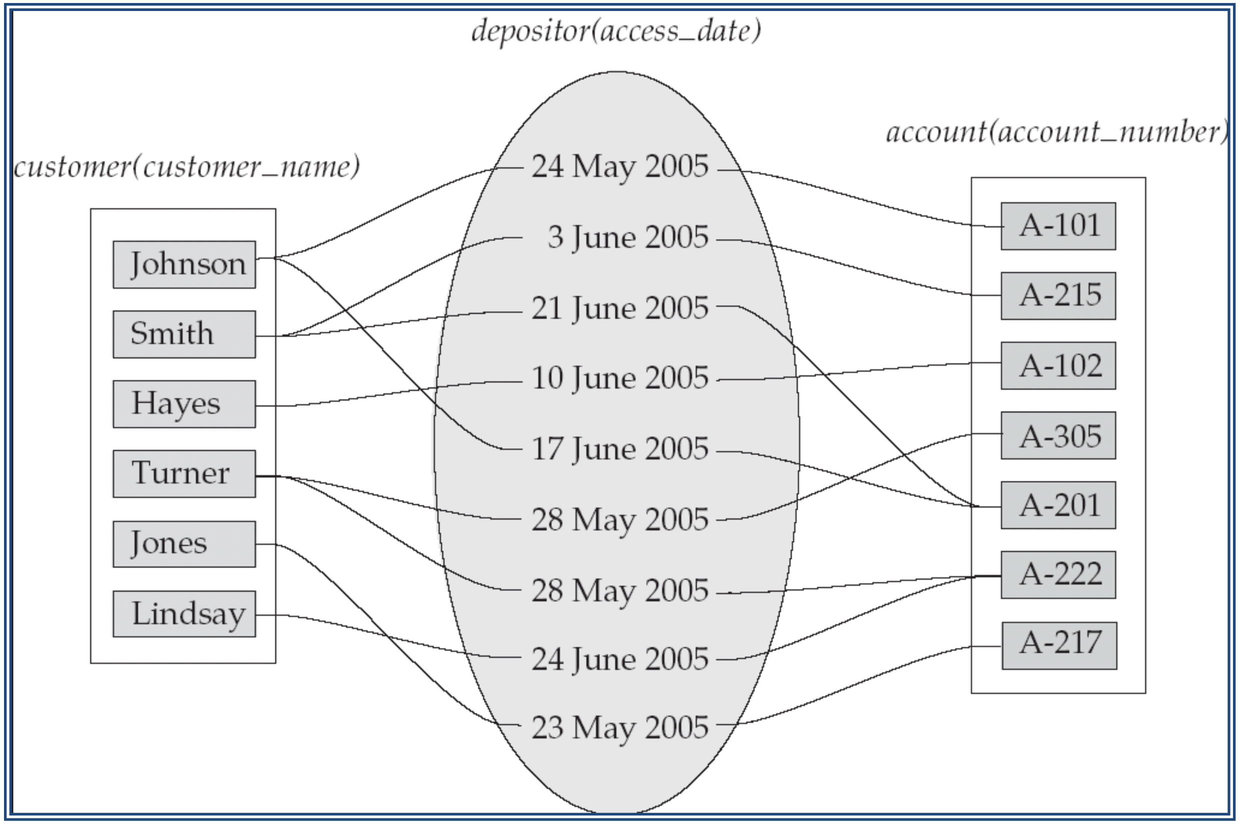
  (Hayes, A-102) $\in$ *depositor*

Relationship Set *borrower*



Relationship Sets (Cont.)

- An attribute can also be property of a relationship set.
- For instance, the *depositor* relationship set between entity sets *customer* and *account* may have the attribute *access-date*



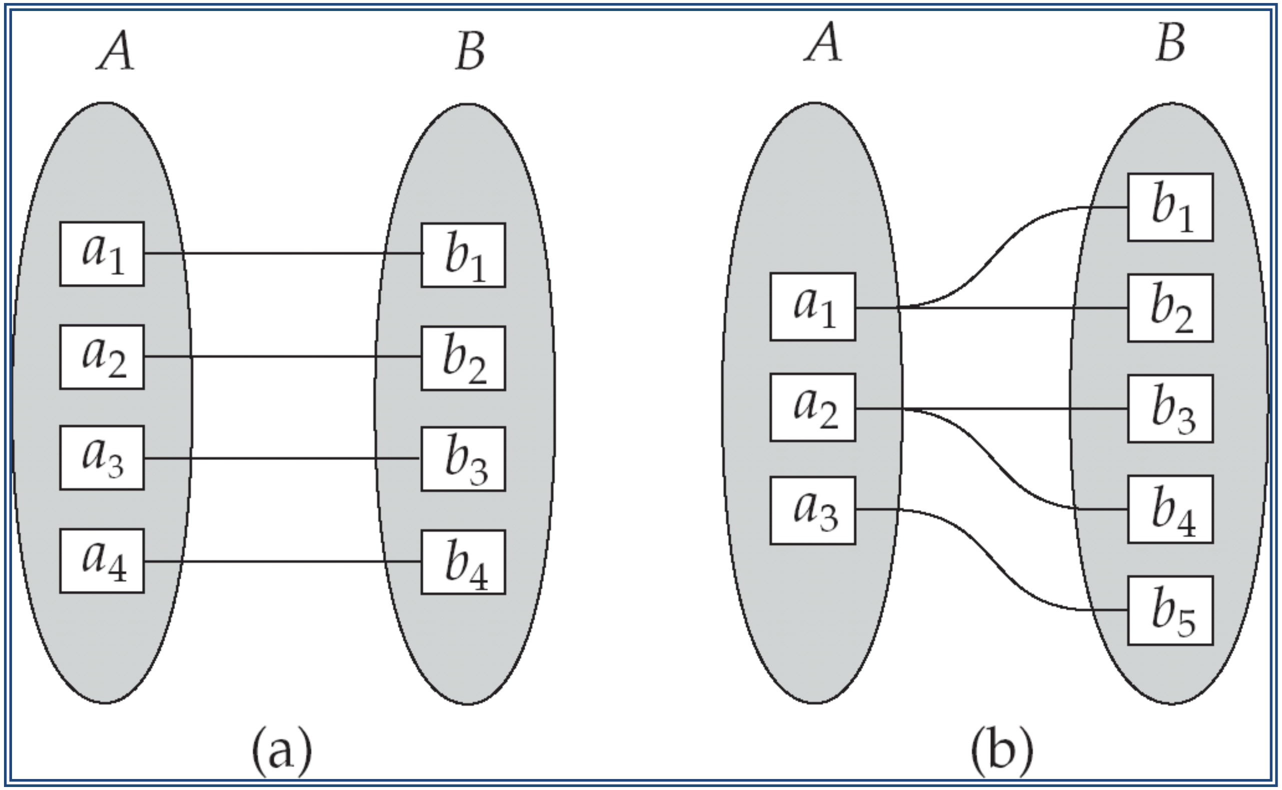
Degree of a Relationship Set

- Refers to number of entity sets that participate in a relationship set.

- Relationship sets that involve two entity sets are binary (or degree two).

- Relationship sets may involve more than two entity sets.

Mapping Cardinalities

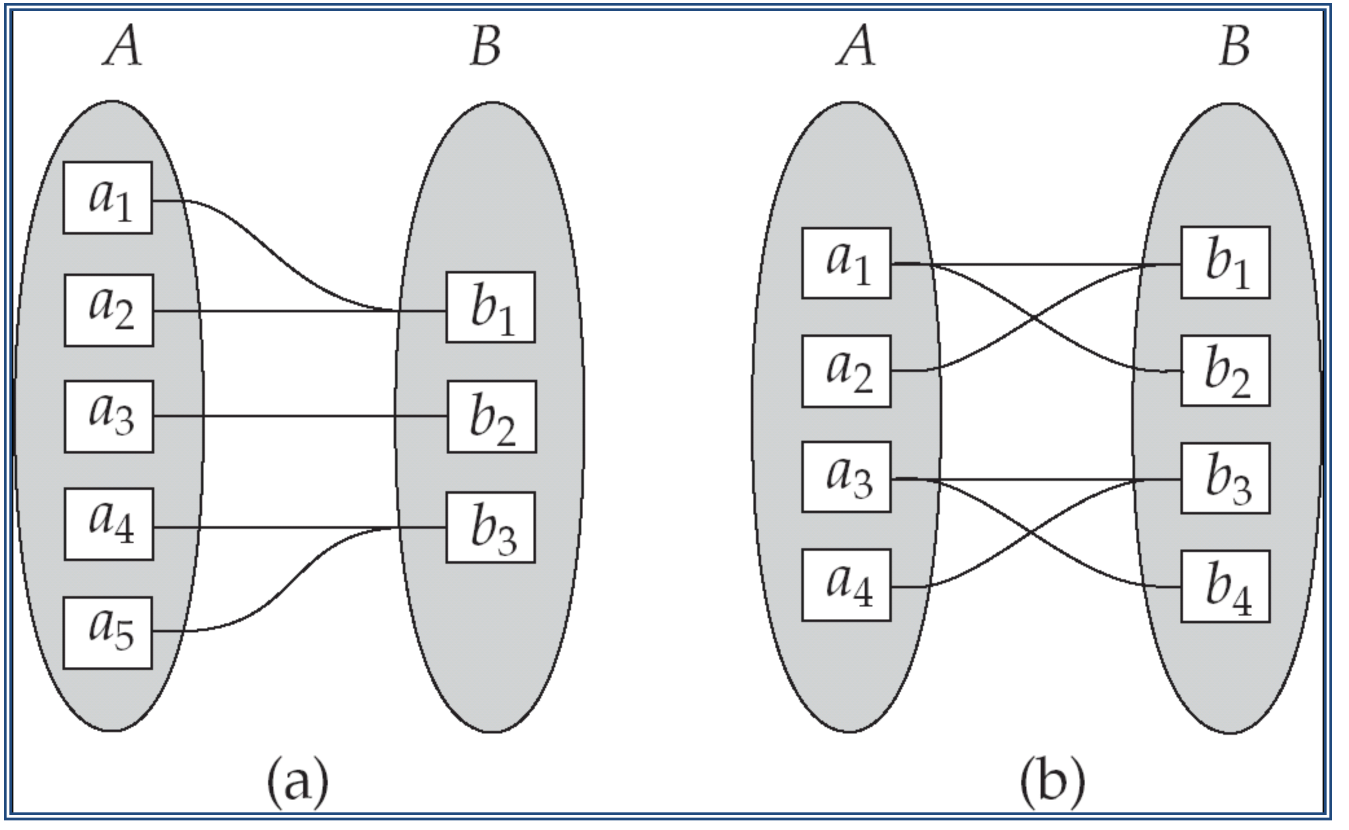


One to one

One to many

Note: Some elements in *A* and *B* may not be mapped to any elements in the other set

Mapping Cardinalities



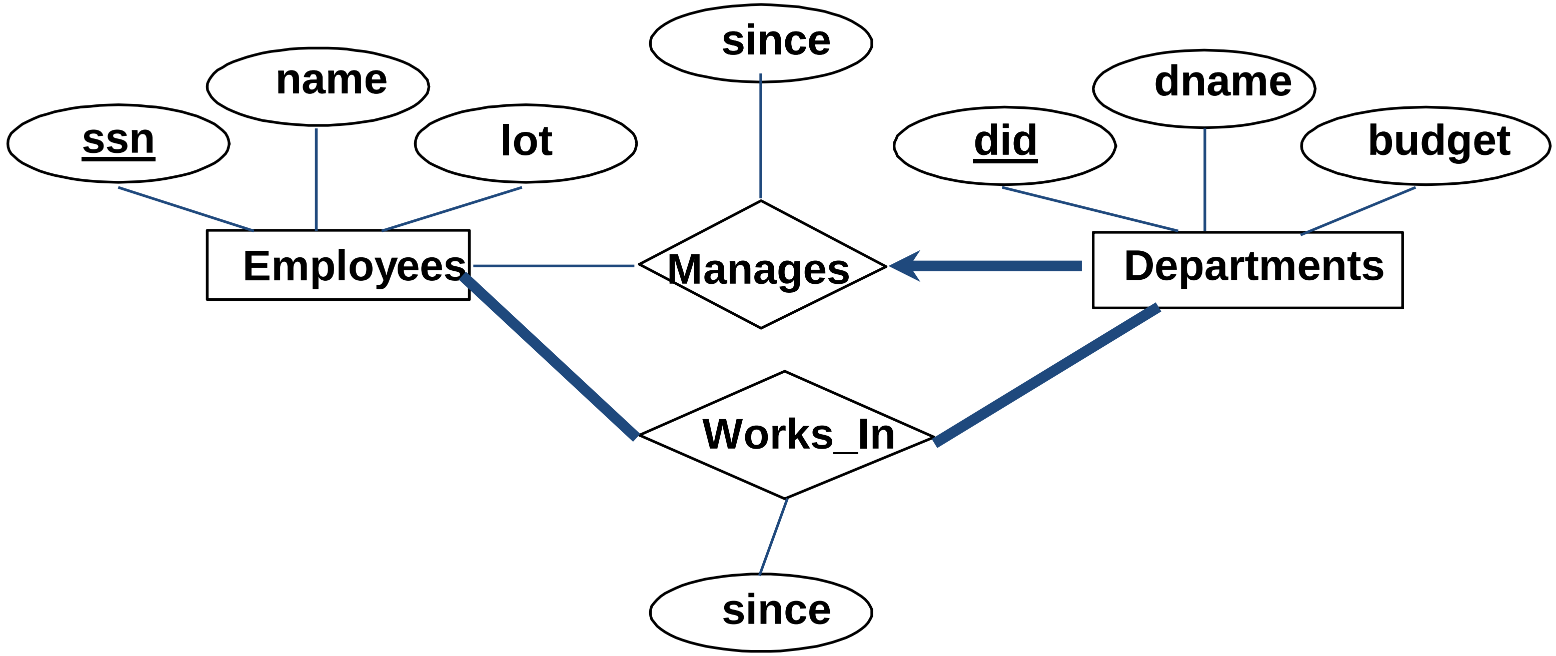
Many to one

Many to many

Note: Some elements in A and B may not be mapped to any elements in the other set

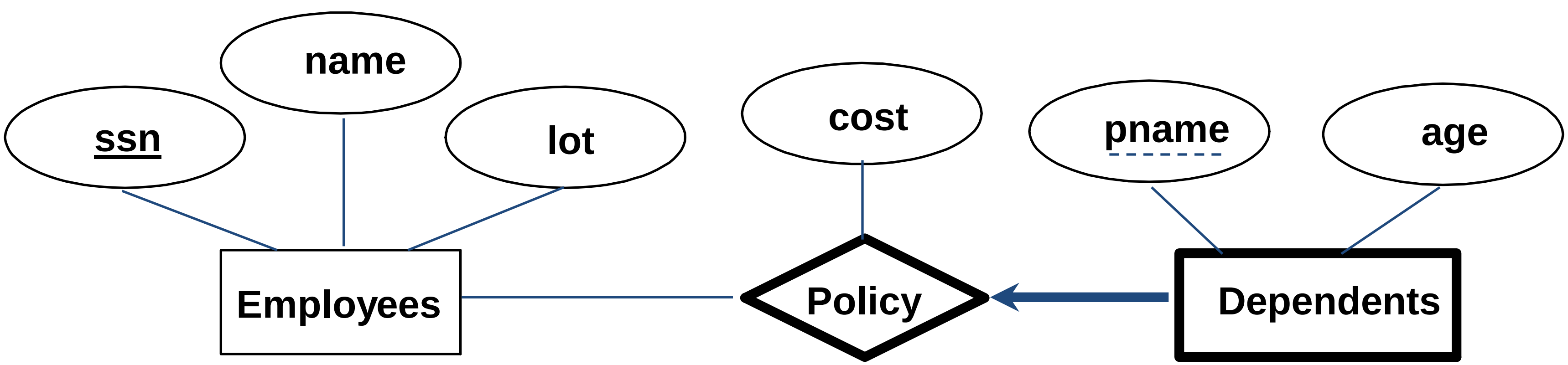
Participation Constraints

- Does every department have a manager?
  - If so, this is a *participation constraint*: the participation of Departments in Manages is said to be *total* (vs. *partial*).
    - Every Departments entity must appear in an instance of the Manages relationship.



Weak Entities

- A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.
- Restrictions
  - Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).
  - Weak entity set must have total participation in this *identifying* relationship set.



Weak Entity Sets

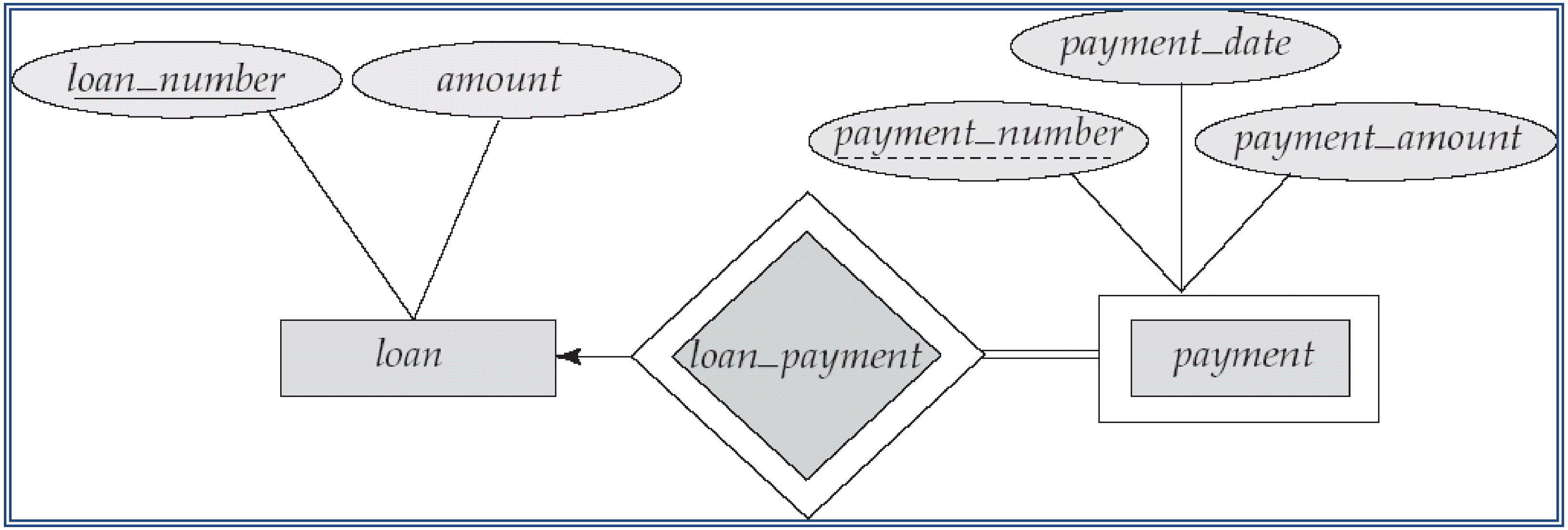
- An entity set that does not have a primary key is referred to as a weak entity set.

- The existence of a weak entity set depends on the existence of a identifying entity set

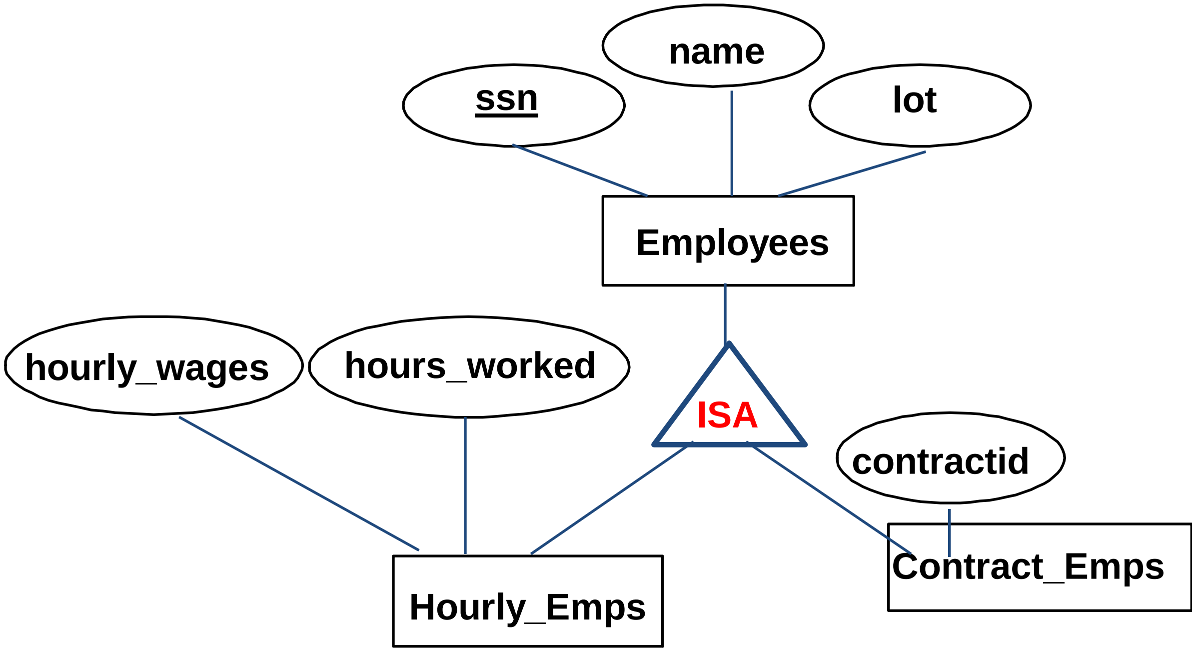
  - it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set
  - Identifying relationship depicted using a double diamond
- The discriminator (*or partial key)* of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.

Weak Entity Sets (Cont.)

- We depict a weak entity set by double rectangles.
- We underline the discriminator of a weak entity set with a dashed line.
- payment_number – discriminator of the *payment* entity set
- Primary key for *payment* – (*loan_number, payment_number*)



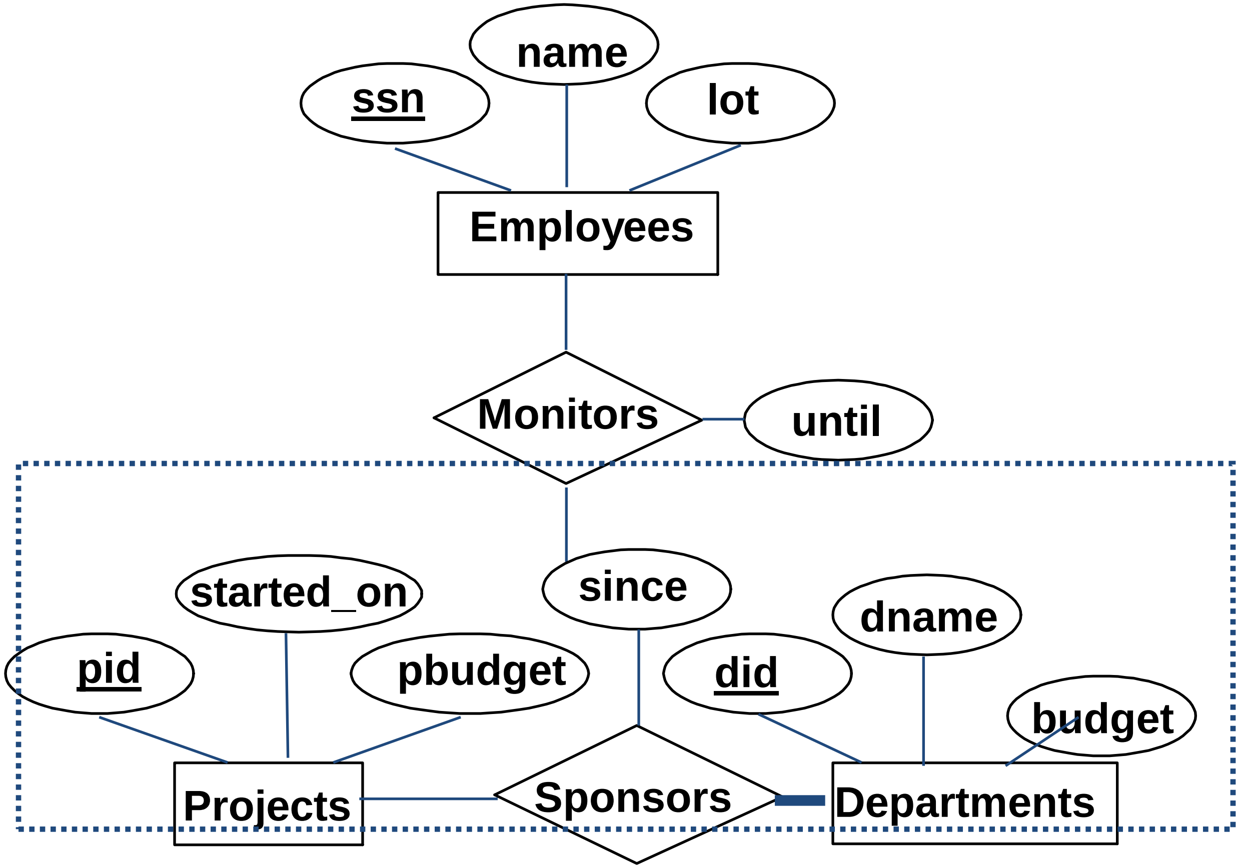
ISA (`is a') Hierarchies



- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? (*Allowed/disallowed*)
- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? *(Yes/no)*
- Reasons for using ISA:
  - To add descriptive attributes specific to a subclass.
  - To identify entities that participate in a relationship.

Aggregation

- Used when we have to model a relationship involving (entity sets and) a *relationship set*.
  - *Aggregation* allows us to treat a relationship set as an entity set for purposes of participation in (other) relationships.

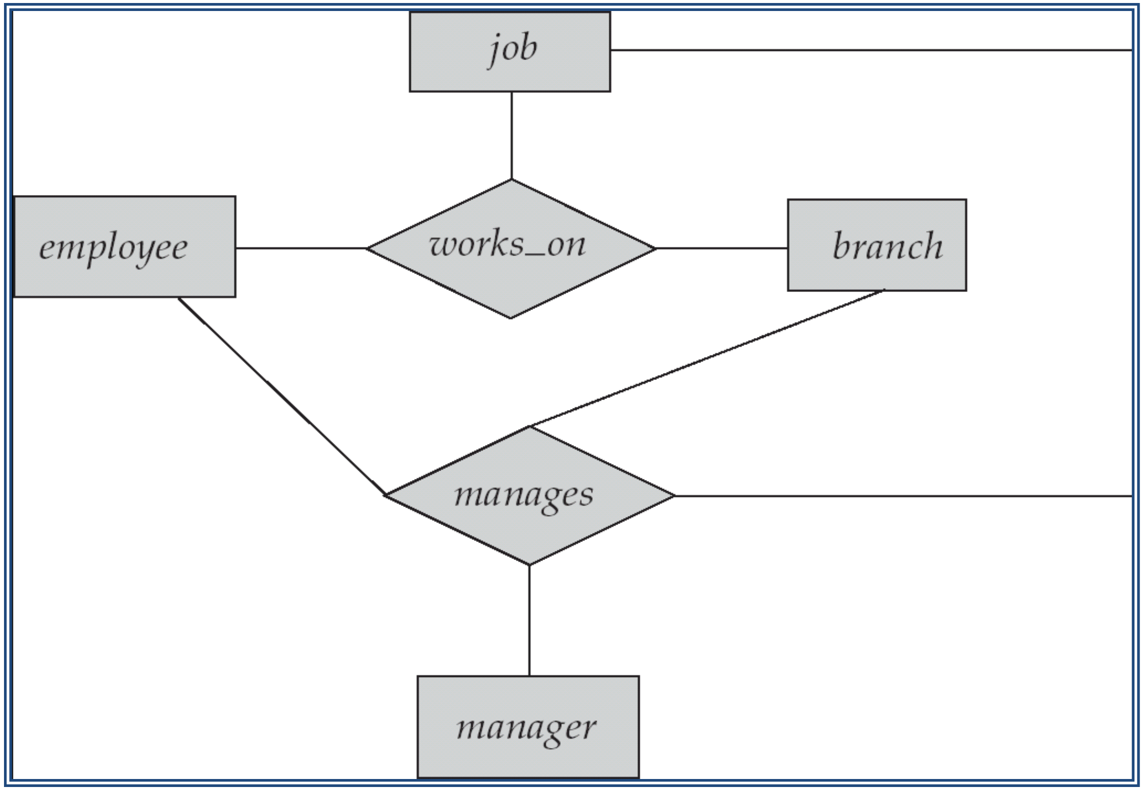


Monitors is a distinct relationship, with a descriptive attribute.

- Also, can say that each sponsorship is monitored by at most one employee.

Aggregation

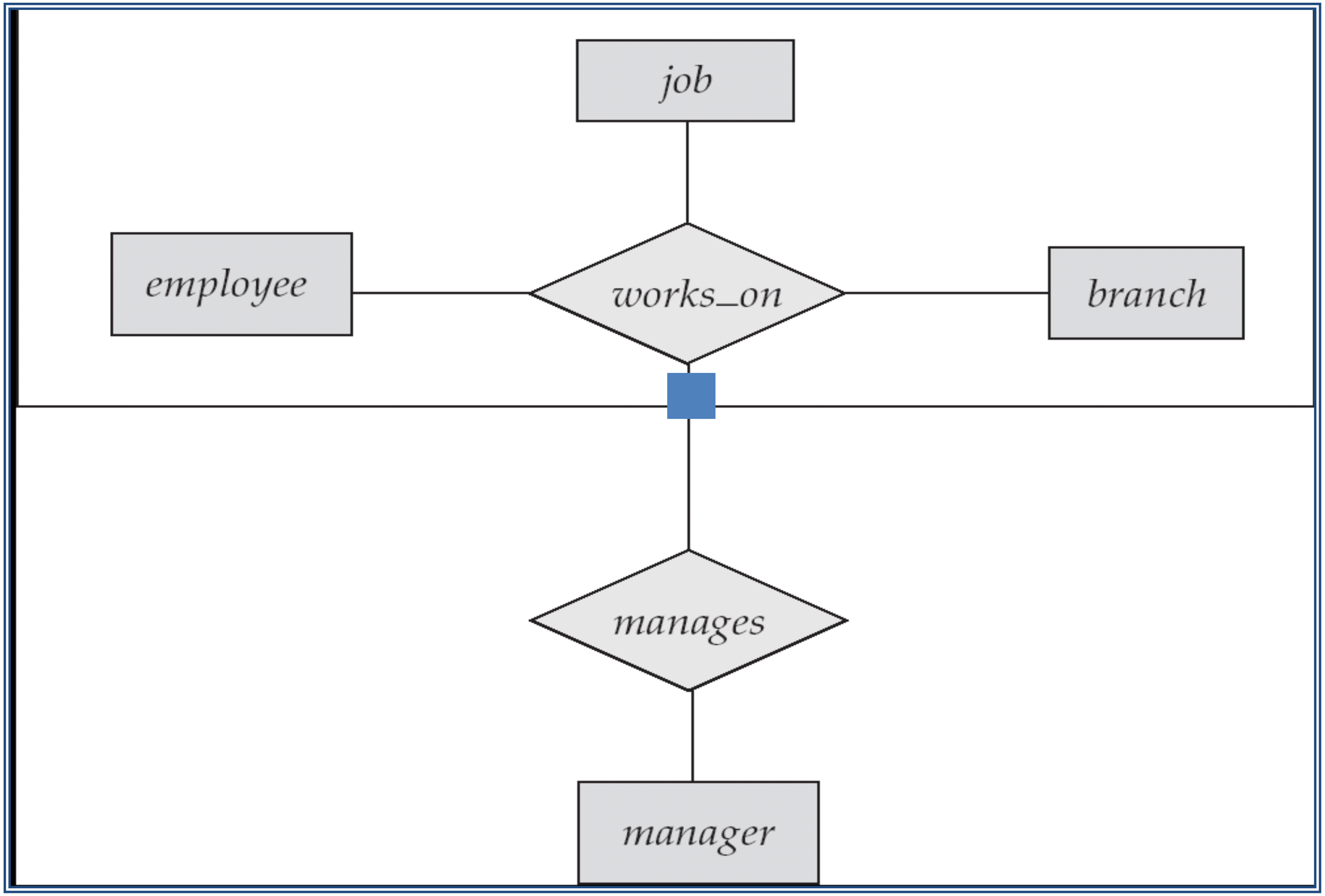
- Consider the ternary relationship *works_on*, which we saw earlier
- Suppose we want to record managers for tasks performed by an employee at a branch



Aggregation (Cont.)

- Relationship sets *works_on* and *manages* represent overlapping information
  - Every *manages* relationship corresponds to a *works_on* relationship
  - However, some *works_on* relationships may not correspond to any *manages* relationships
    - So we can't discard the *works_on* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity

E-R Diagram With Aggregation



Conceptual Design Using the ER Model

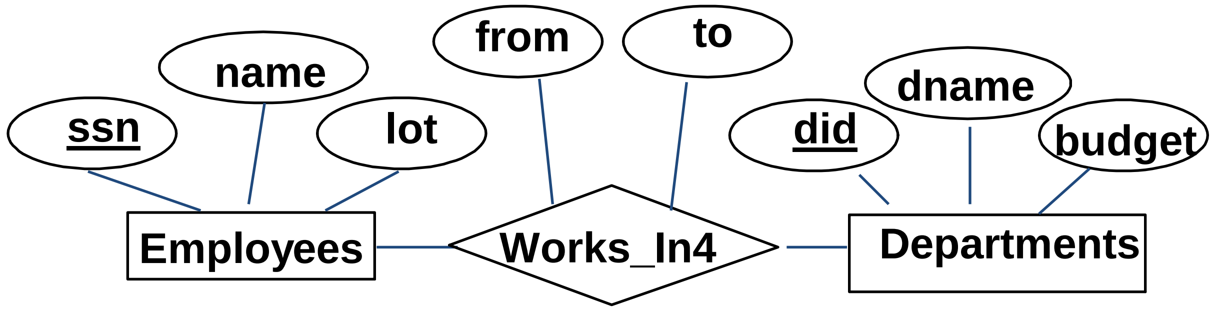
- <u>Design choices:</u>
  - Should a concept be modeled as an entity or an attribute?
  - Should a concept be modeled as an entity or a relationship?
  - Identifying relationships: Binary or ternary? Aggregation?
- Constraints in the ER Model:
  - A lot of data semantics can (and should) be captured.
  - But some constraints cannot be captured in ER diagrams.

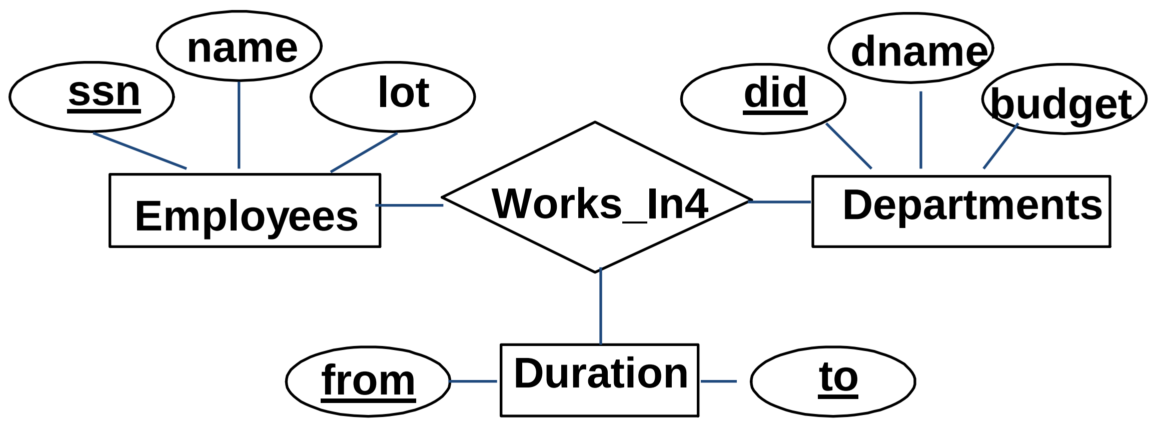
Entity vs. Attribute

- Should *address* be an attribute of Employees or an entity (connected to Employees by a relationship)?
- Depends upon the use we want to make of address information, and the semantics of the data:
    - If we have several addresses per employee, *address* must be an entity (since attributes cannot be set-valued).
    - If the structure (city, street, etc.) is important, e.g., we want to retrieve employees in a given city, *address* must be modeled as an entity (since attribute values are atomic).

Entity vs. Attribute (Contd.)

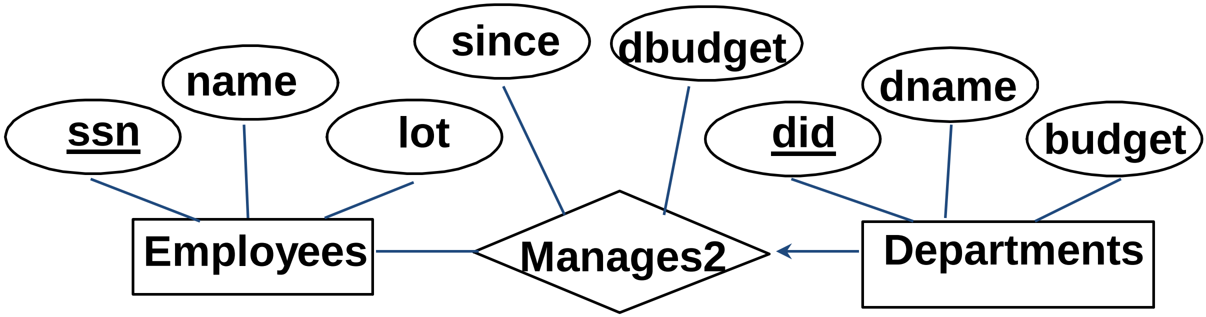
- Works_In4 does not allow an employee to work in a department for two or more periods.
- Similar to the problem of wanting to record several addresses for an employee: We want to record *several values of the descriptive attributes for each instance of this relationship.* Accomplished by introducing new entity set, Duration.

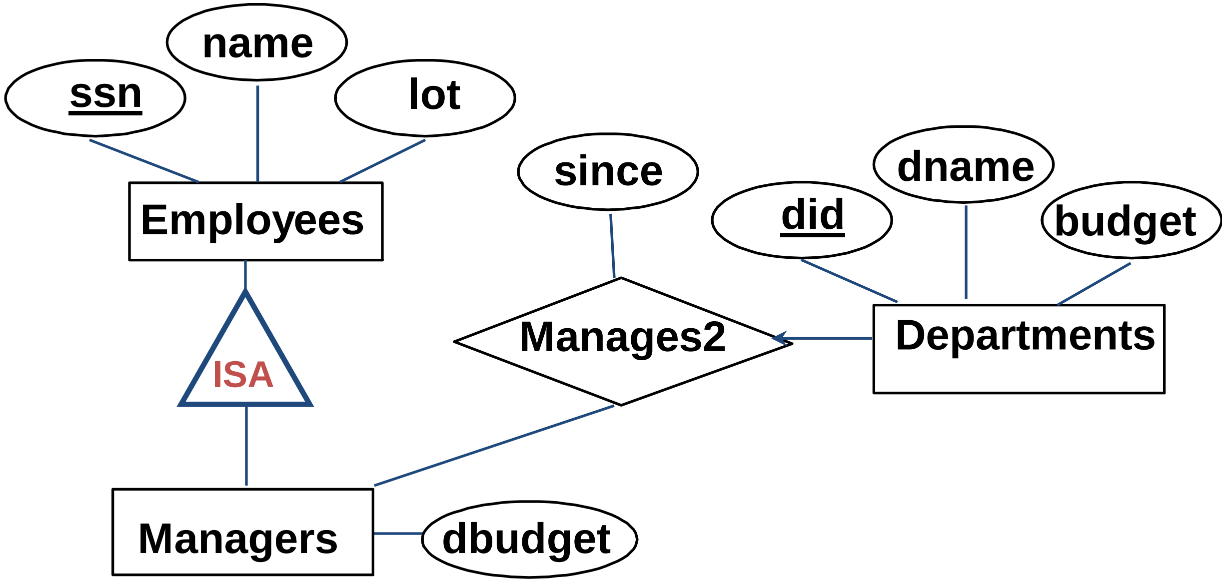




Entity vs. Relationship

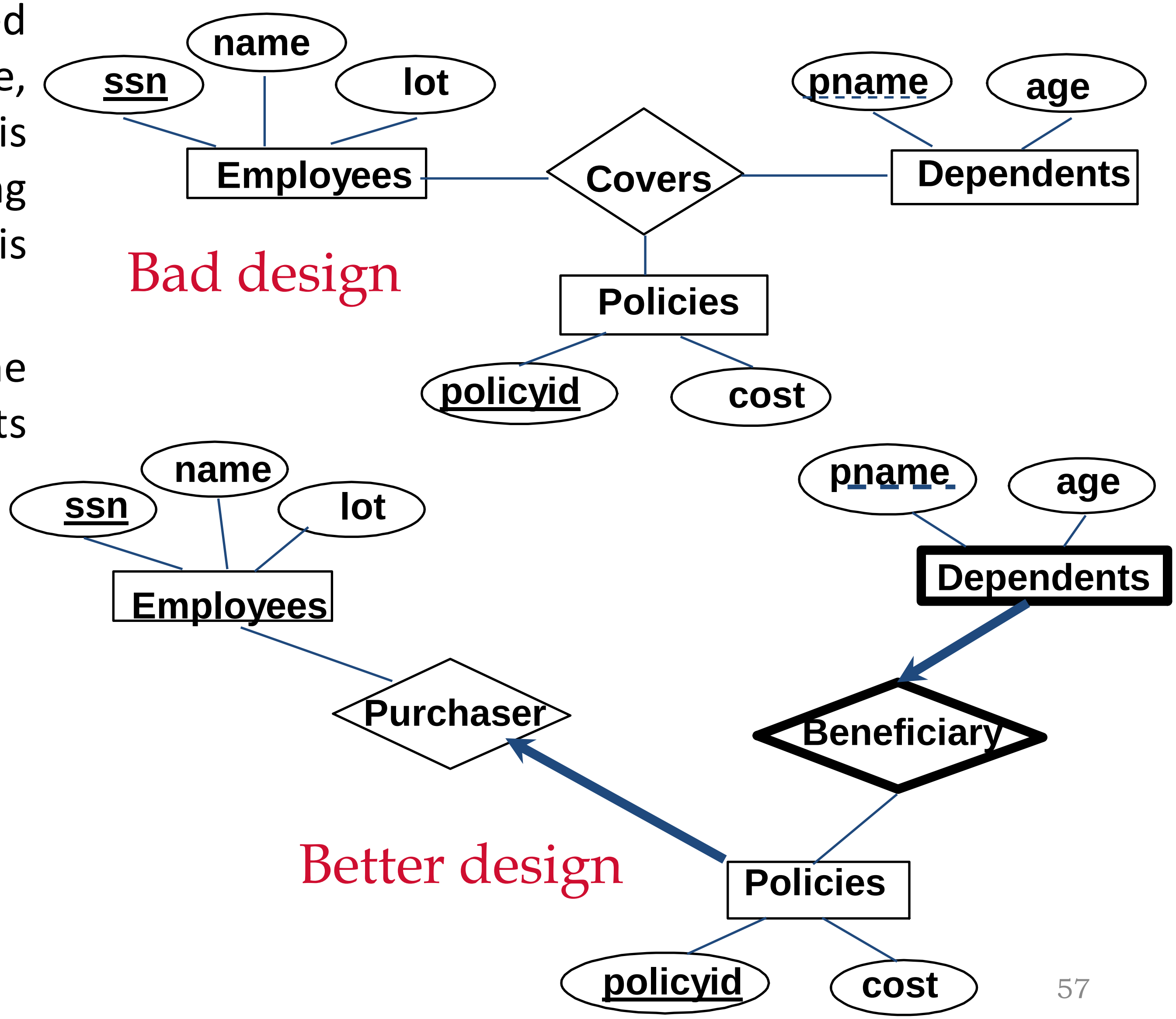
- First ER diagram OK if a manager gets a separate discretionary budget for each dept.
- What if a manager gets a discretionary budget that covers *all* managed depts?
  - Redundancy: *dbudget* stored for each dept managed by manager.
  - Misleading: Suggests *dbudget* associated with department-mgr combination.





Binary vs. Ternary Relationships

- If each policy is owned by just 1 employee, and each dependent is tied to the covering policy, first diagram is inaccurate.
- What are the additional constraints in the 2nd diagram?



Binary vs. Ternary Relationships (Contd.)

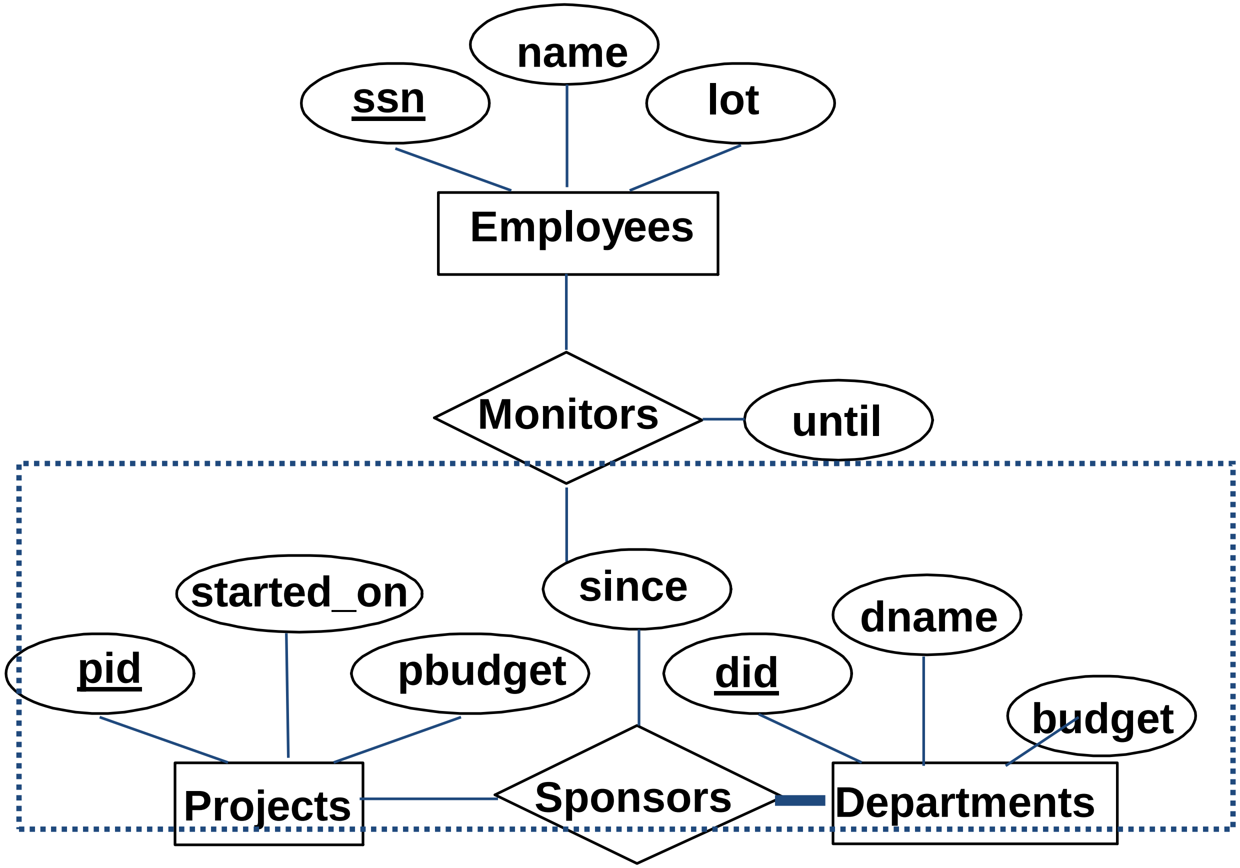
- Previous example illustrated a case when two binary relationships were better than one ternary relationship.

- An example in the other direction: a ternary relation Contracts relates entity sets Parts, Departments and Suppliers, and has descriptive attribute *qty*.

  – S “can-supply” P, D “needs” P, and D “deals-with” S does not imply that D has agreed to buy P from S.
  – How do we record *qty*?

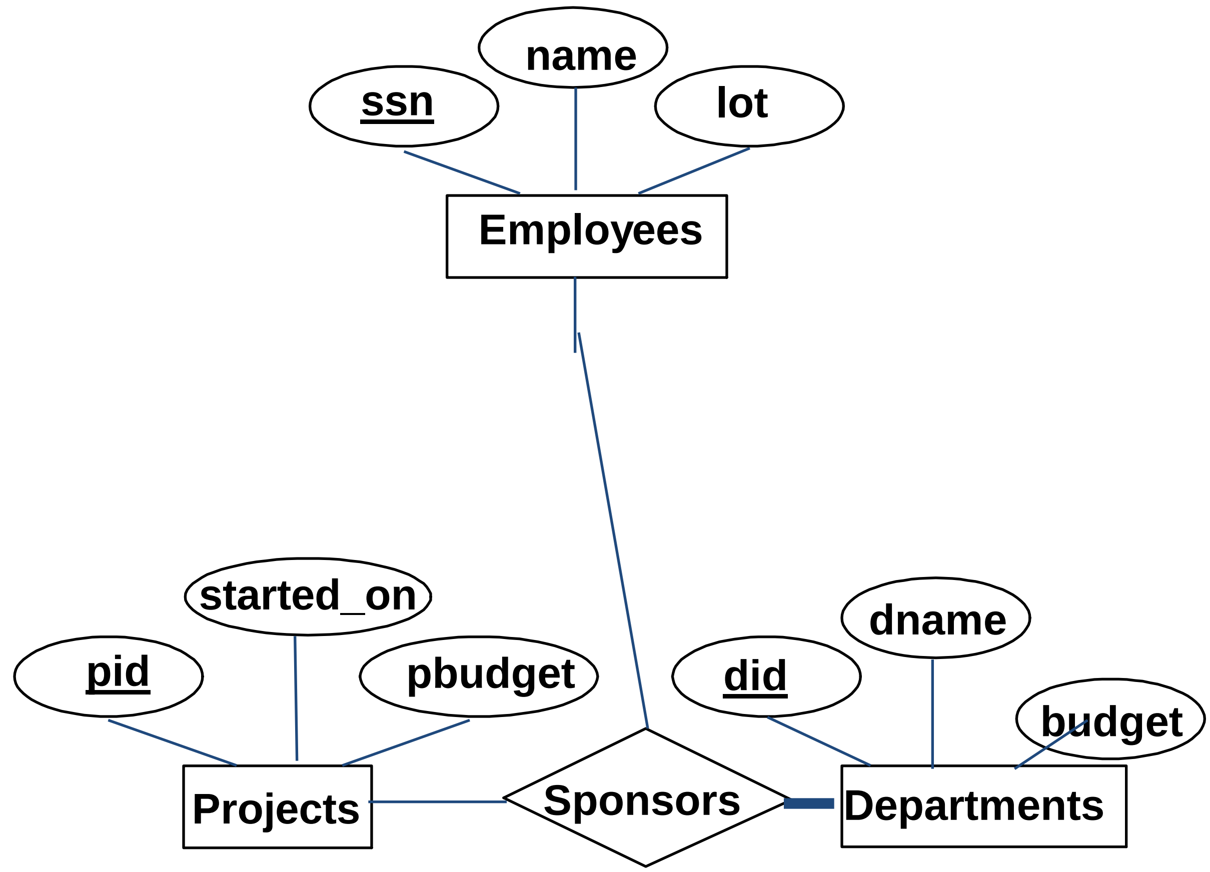
Aggregation v/s ternary relationship

- The choice between using aggregation or ternary relationship is mainly determined by existence of a relationship that relates relationship set to entity set.



The choice may also be guided by certain integrity constraints that we want to express.

Aggregation v/s ternary relationship



- Using ternary relationship instead of aggregation

Conceptual design for large enterprises

- For large enterprise the design may require efforts of more than one designer and span data and application code used by number of user groups.
- ER diagrams for Conceptual design offers additional advantage that high level design can be diagramatically represented and easily understood by many people.

2 approaches:

- Usual approach: requirements of various user groups are considered,any conflicting requirements are somehow resolved and single set of global requirements is generated at the end of requirements phase
- Alternative approach: is to develop separate conceptual schemas for different user groups and then integrate these conceptual schemas

Relational Database: Definitions

- *Relational database:* a set of *relations*
- *Relation:* made up of 2 parts:
- *Relation schema and relational instance.*
  - *Instance* : a *table*, with rows and columns.
  - Set of tuples also called as records
  - #Rows = *cardinality*, #fields = *degree / arity.*
  - A domain is referred by domain name consisting of set of associated values.
  - *Schema* : specifies name of relation, plus name and type of each column.
    - E.G. Students (*sid*: string, *name*: string, *login*: string, *age*: integer, *gpa*: real).
- Can think of a relation as a *set* of rows or *tuples* (i.e., all rows are distinct).

Example Instance of Students Relation

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

- Cardinality = 3, degree = 5, all rows distinct

- Do all columns in a relation instance have to be distinct?

Creating Relations in SQL

- Creates the Students relation. Observe that the type of each field is specified, and enforced by the DBMS whenever tuples are added or modified.
- As another example, the Enrolled table holds information about courses that students take.

```
CREATE TABLE Students
        (sid: CHAR(20),
         name: CHAR(20),
         login: CHAR(10),
         age: INTEGER,
         gpa: REAL)
```

```
CREATE TABLE Enrolled
        (sid: CHAR(20),
         cid: CHAR(20),
         grade: CHAR(2))
```

Destroying and Altering Relations

DROP TABLE Students

- Destroys the relation Students. The schema information *and* the tuples are deleted.

ALTER TABLE Students
ADD COLUMN firstYear: integer

- The schema of Students is altered by adding a new field; every tuple in the current instance is extended with a *null* value in the new field.

Adding and Deleting Tuples

- **Can insert a single tuple using:**

```
INSERT INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
```

- Can delete all tuples satisfying some condition (e.g., name = Smith):

```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```

Integrity Constraints (ICs)

- IC: condition that must be true for *any* instance of the database; e.g., *domain constraints.*
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A *legal* instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!

Primary Key Constraints

- A set of fields is a *key* for a relation if :
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key.
  - Part 2 false? A *superkey*.
  - If there's >1 key for a relation, one of the keys is chosen (by DBA) to be the *primary key*.
- E.g., *sid* is a key for Students. (What about *name*?) The set {*sid, gpa*} is a superkey.

Primary and Candidate Keys in SQL

- Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.

- "For a given student and course, there is a single grade." vs. "Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade."
- Used carelessly, an IC can prevent the storage of database instances that arise in practice!

```
CREATE TABLE Enrolled
  (studid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
 PRIMARY KEY  (sid,cid) )
```

```
CREATE TABLE Enrolled
  (studidid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid),
   UNIQUE (cid, grade) )
```

Foreign Keys, Referential Integrity

- *<u>Foreign key</u>* : Set of fields in one relation that is used to `refer' to a tuple in another relation. (Must correspond to primary key of the second relation.) Like a `logical pointer'.
- CREATE TABLE Students(sid: CHAR(20), name: CHAR(20),login:CHAR(10), age: INTEGER, gpa: REAL)

- E.g. *studid* is a foreign key referring to Students:
  - Enrolled(*studid*: string, *cid*: string, *grade*: string)

  - If all foreign key constraints are enforced, *<u>referential integrity</u>* is achieved, i.e., no dangling references.

Foreign Keys in SQL

- Only students listed in the Students relation should be allowed to enroll for courses.

```
CREATE TABLE Enrolled
(sid CHAR(20),  cid CHAR(20),  grade CHAR(2),
  PRIMARY KEY  (sid,cid),
  FOREIGN KEY (stuid) REFERENCES Students(sid )
```

Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B

Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

General constraints

- Current relational database systems support such general constraints in 2 forms
- Constraint table: It is associated with single table and checked whenever that single table is modified
- Assertions: include several tables and are checked whenever any of these tables is modified.
- Domain constraints: domains can have some constraints called Domain constraints
- Column constraints: the value in any column of any table should be controlled by column constraints
- User defined IC: it allows business rules to be specified centrally to database, so that when certain action is performed on a set of data, other actions are automatically performed

Enforcing Referential Integrity

- Consider Students and Enrolled; *sid* in Enrolled is a foreign key that references Students.
- What should be done if an Enrolled tuple with a non-existent student id is inserted? (*Reject it!*)
- What should be done if a Students tuple is deleted?
  - Also delete all Enrolled tuples that refer to it.
  - Disallow deletion of a Students tuple that is referred to.
  - Set sid in Enrolled tuples that refer to it to a *default sid*.
  - (In SQL, also: Set sid in Enrolled tuples that refer to it to a special value *null,* denoting *`unknown'* or *`inapplicable'*.)
- Similar if primary key of Students tuple is updated.

Referential Integrity in SQL

- SQL/92 and SQL:1999 support all 4 options on deletes and updates.
  - Default is NO ACTION (*delete/update is rejected*)
  - CASCADE (also delete all tuples that refer to deleted tuple)
  - SET NULL / SET DEFAULT (sets foreign key value of referencing tuple)

```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid),
   FOREIGN KEY (sid)
     REFERENCES Students
        ON DELETE CASCADE
        ON UPDATE SET
       DEFAULT )
```

Transactions and constraints

- In SQL a constraint is checked at the end of every SQL statement that could lead to viloation and if there is a violation,the statement is rejected,this approach is inflexible
- SQL allows a constraint to be in deferred or immediate mode
- Syntax: set constraint constraintname Immediate/Deffered

The SQL Query Language

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

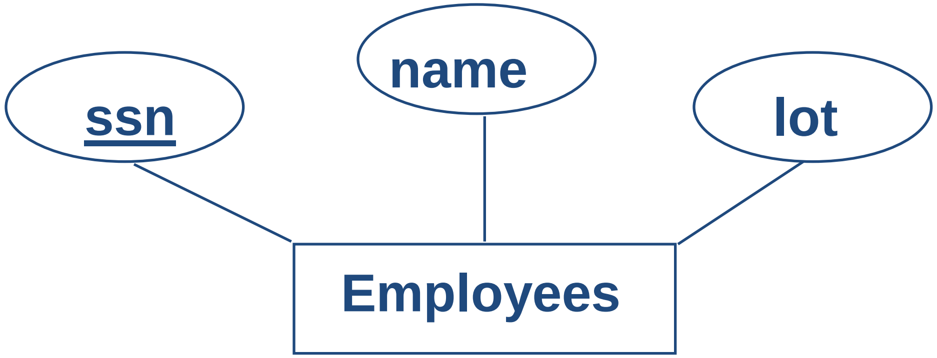
```
SELECT  *
FROM  Students S
WHERE  S.age=18
```

- To find just names and logins, replace the first line:

```
SELECT  S.name, S.login
```

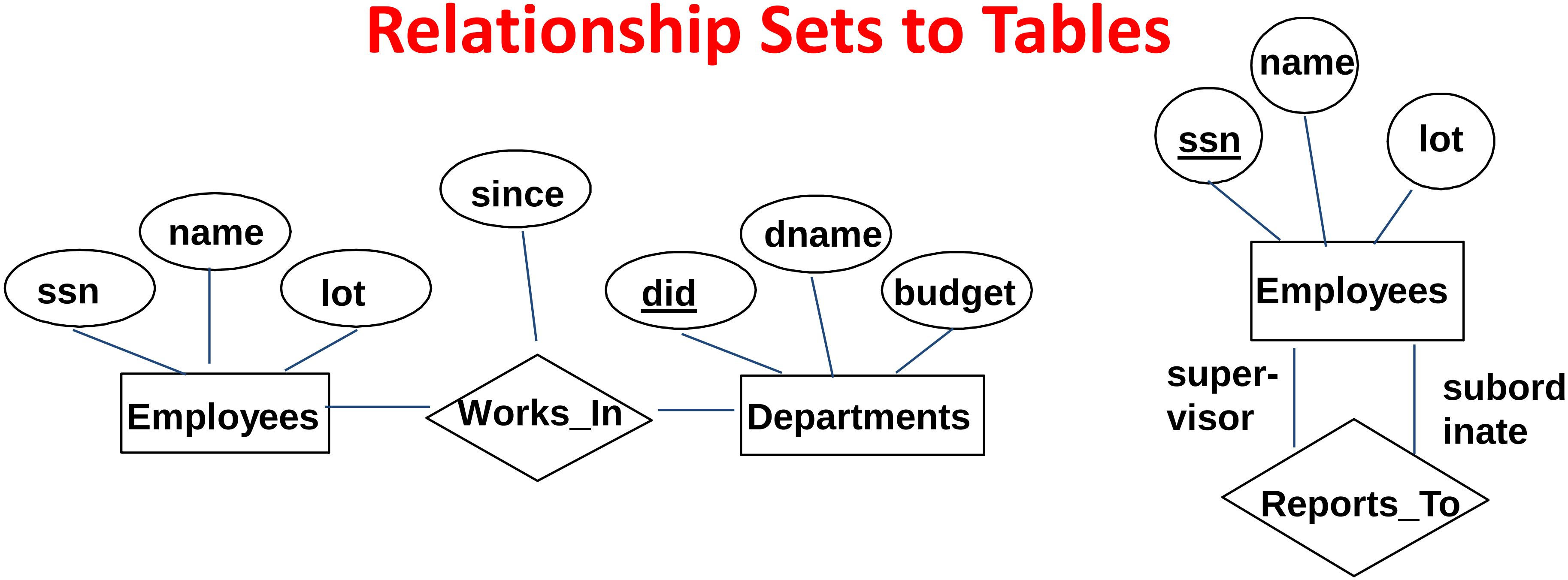
Logical DB Design: ER to Relational

- Entity sets to tables:



CREATE TABLE Employees
(ssn CHAR(11),
name CHAR(20),
lot INTEGER,
PRIMARY KEY (ssn))

Relationship Sets to Tables



Create table reportsTo(supervisor_ssn char(10), subordinate_ssn char(10),
primary key(supervisor_ssn, subordinate_ssn),
foreign key(supervisor_ssn) references employees(ssn)
foreign key(subordinate_ssn) references employees(ssn))

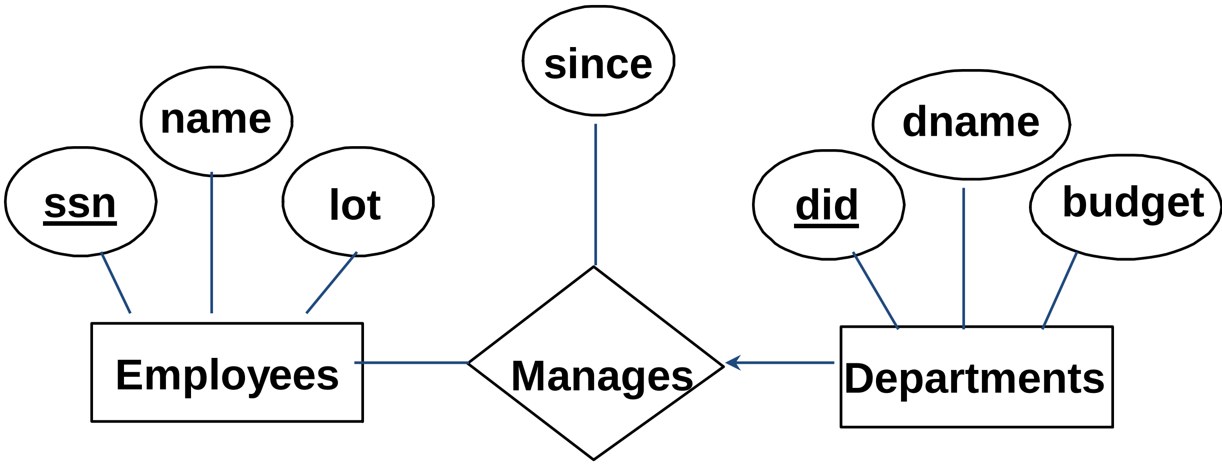
Relationship Sets to Tables

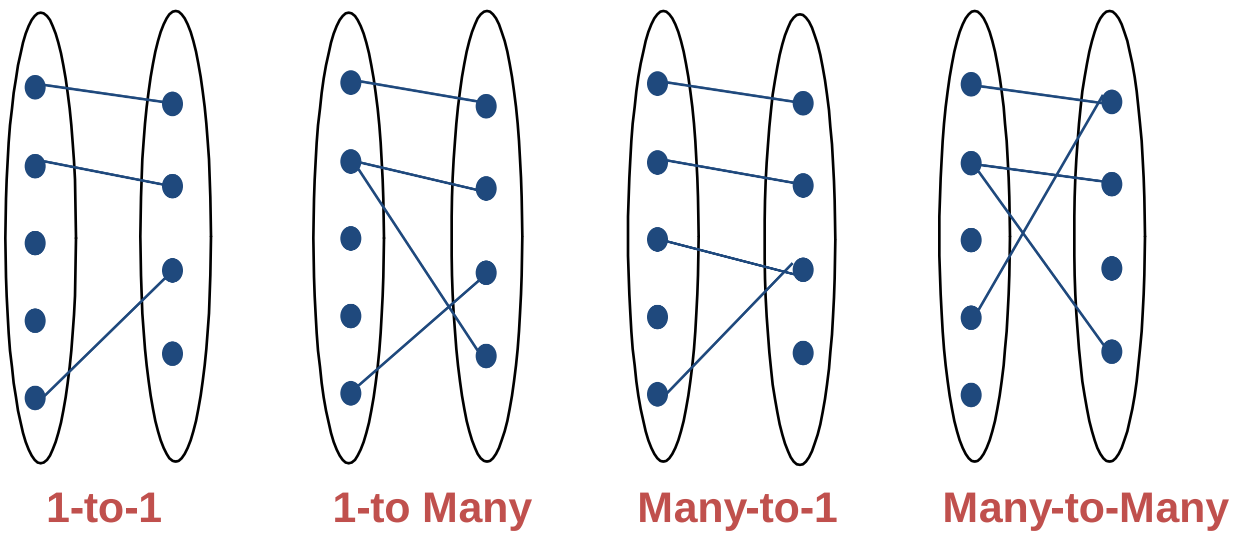
- In translating a relationship set to a relation, attributes of the relation must include:
  - Keys for each participating entity set (as foreign keys).
    - This set of attributes forms a *superkey* for the relation.
  - All descriptive attributes.

```
CREATE TABLE Works_In(
  ssn  CHAR(11),
  did  INTEGER,
  since  DATE,
  PRIMARY KEY (ssn, did),
  FOREIGN KEY (ssn)
      REFERENCES Employees,
  FOREIGN KEY (did)
      REFERENCES Departments)
```

Review: Key Constraints

- Each dept has at most one manager, according to the *key constraint* on Manages.





Translation to relational model?

Translating ER Diagrams with Key Constraints

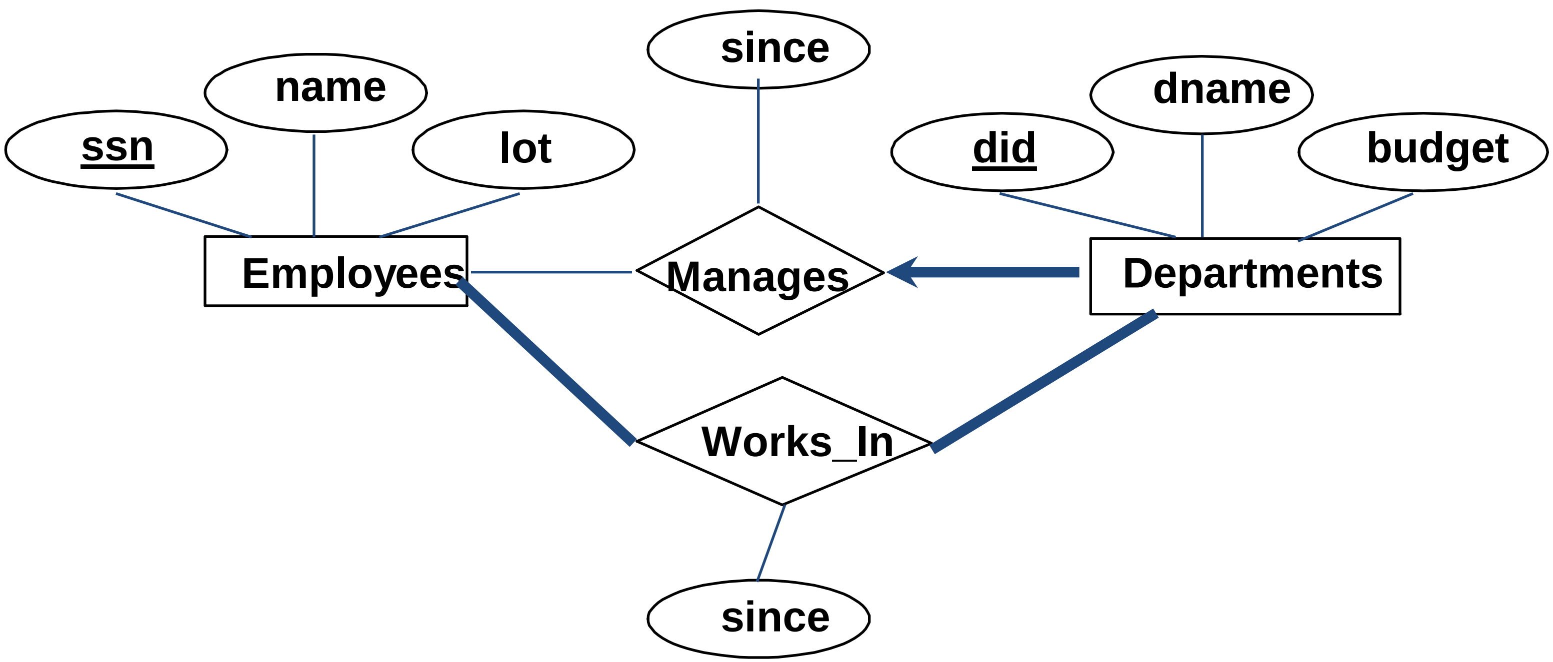
- Map relationship to a table:
  - Note that did is the key now!
  - Separate tables for Employees and Departments.
- Since each department has a unique manager, we could instead combine Manages and Departments.

```
CREATE TABLE  Manages(
  ssn  CHAR(11),
  did  INTEGER,
  since  DATE,
  PRIMARY KEY  (did),
  FOREIGN KEY (ssn) REFERENCES Employees,
  FOREIGN KEY (did) REFERENCES Departments)
```

```
CREATE TABLE  Dept_Mgr(
  did  INTEGER,
  dname  CHAR(20),
  budget  REAL,
  ssn  CHAR(11),
  since  DATE,
  PRIMARY KEY  (did),
  FOREIGN KEY (ssn) REFERENCES Employees)
```

Review: Participation Constraints

- Does every department have a manager?
  - If so, this is a ***participation constraint***: the participation of Departments in Manages is said to be *total* (vs. *partial*).
    - Every *did* value in Departments table must appear in a row of the Manages table (with a non-null *ssn* value!)



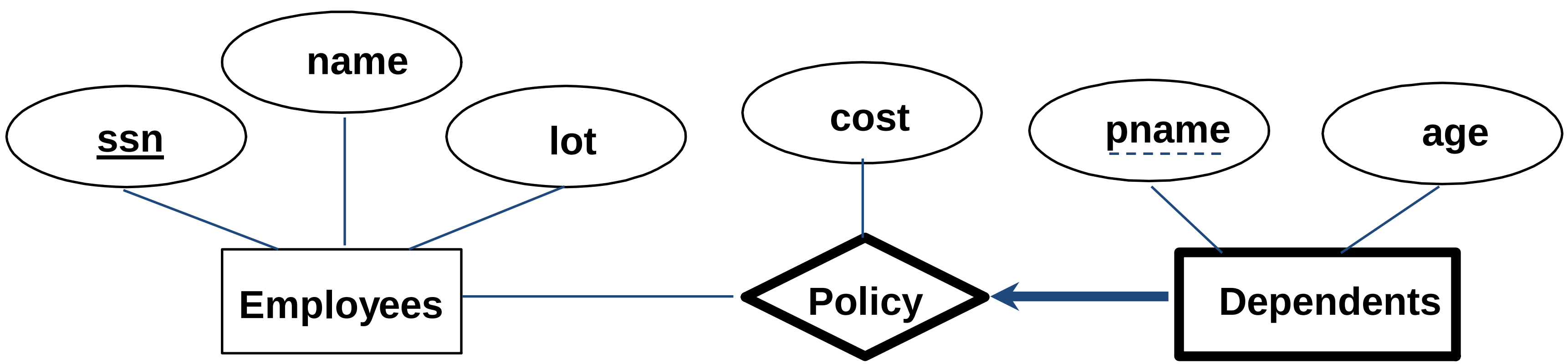
Participation Constraints in SQL

- We can capture participation constraints involving one entity set in a binary relationship, but little else (without resorting to CHECK constraints).

```
CREATE TABLE Dept_Mgr(
  did INTEGER,
  dname CHAR(20),
  budget REAL,
  ssn CHAR(11) NOT NULL,
  since DATE,
  PRIMARY KEY (did),
  FOREIGN KEY (ssn) REFERENCES Employees,
    ON DELETE NO ACTION)
```

Review: Weak Entities

- A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.
  - Owner entity set and weak entity set must participate in a one-to-many relationship set (1 owner, many weak entities).
  - Weak entity set must have total participation in this *identifying* relationship set.



Translating Weak Entity Sets

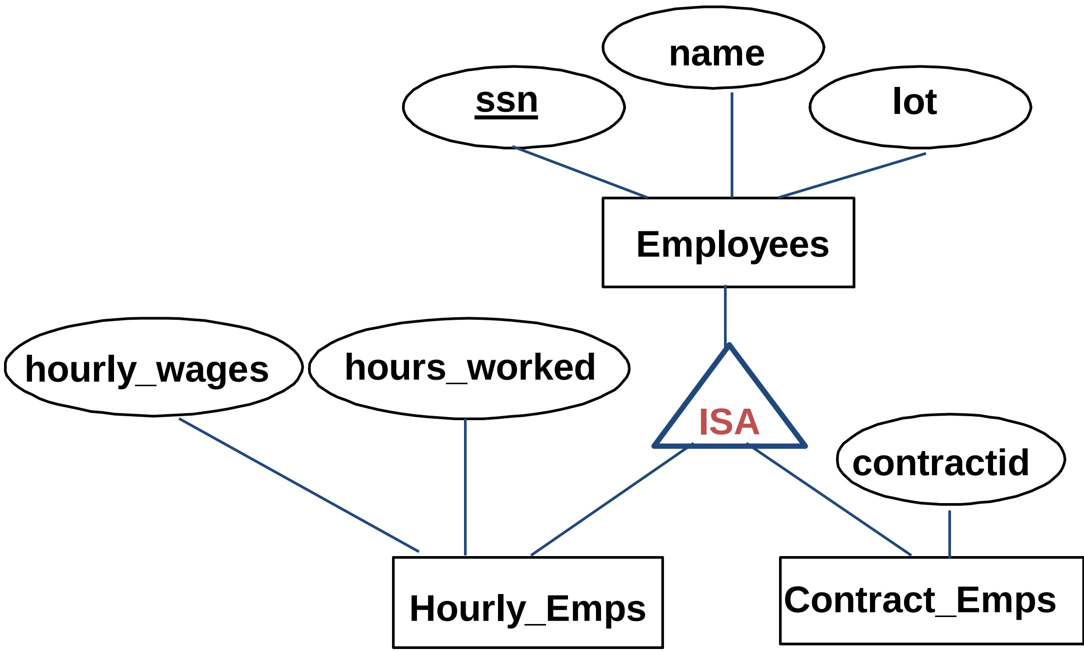
- Weak entity set and identifying relationship set are translated into a single table.
  - When the owner entity is deleted, all owned weak entities must also be deleted.

```
CREATE TABLE  Dep_Policy (
  pname  CHAR(20),
  age  INTEGER,
  cost  REAL,
  ssn  CHAR(11) NOT NULL,
  PRIMARY KEY  (pname, ssn),
  FOREIGN KEY  (ssn) REFERENCES Employees,
    ON DELETE CASCADE)
```

Review: ISA Hierarchies

❖ As in C++, or other PLs, attributes are inherited.

❖ If we declare A ISA B, every A entity is also considered to be a B entity.



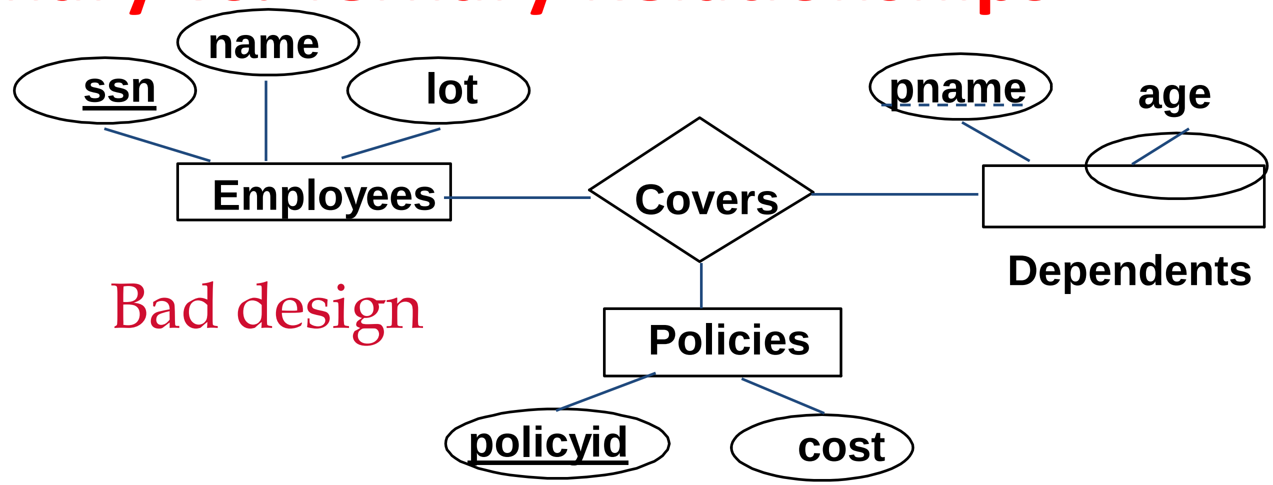
- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? (*Allowed/disallowed*)
- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? *(Yes/no)*

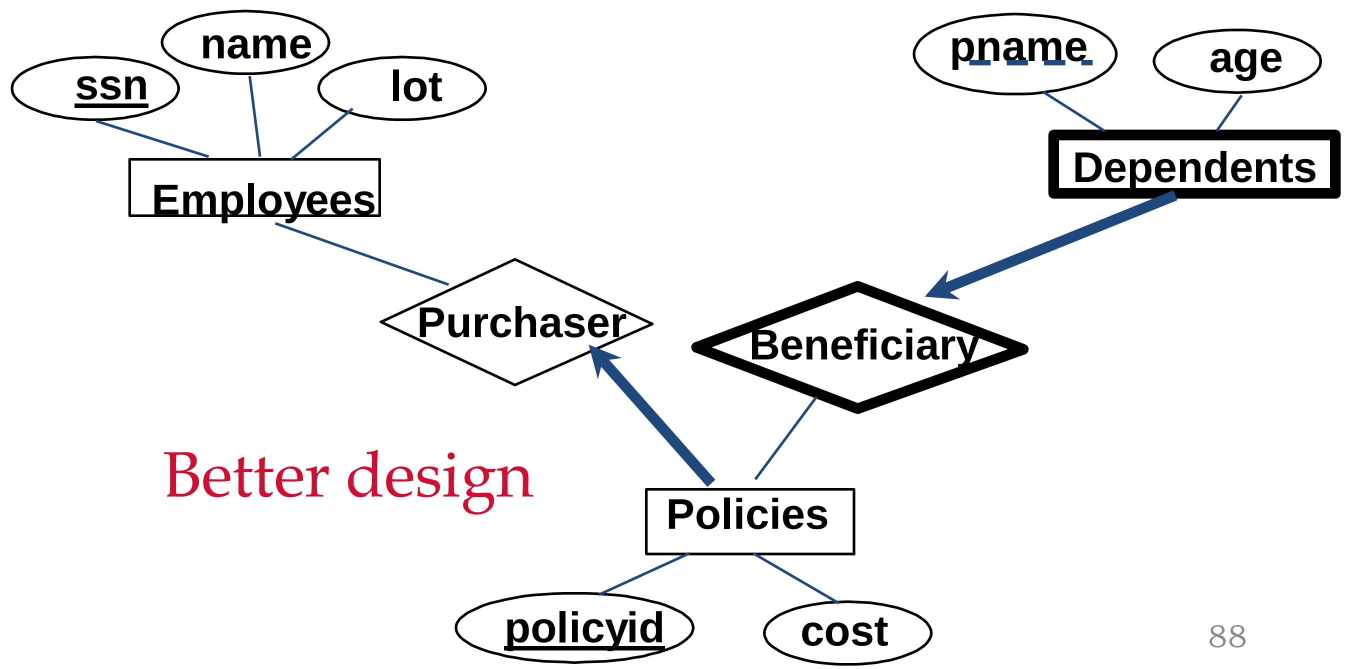
Translating ISA Hierarchies to Relations

- *General approach:*
  - 3 relations: Employees, Hourly_Emps and Contract_Emps.
    - *Hourly_Emps*: Every employee is recorded in Employees. For hourly emps, extra info recorded in Hourly_Emps (*hourly_wages*, *hours_worked*, *ssn)*; must delete Hourly_Emps tuple if referenced Employees tuple is deleted).
    - Queries involving all employees easy, those involving just Hourly_Emps require a join to get some attributes.
- Alternative: Just Hourly_Emps and Contract_Emps.
  - *Hourly_Emps*: *ssn, name, lot, hourly_wages, hours_worked.*
  - Each employee must be in one of these two subclasses.

Review: Binary vs. Ternary Relationships

- What are the additional constraints in the 2nd diagram?





Binary vs. Ternary Relationships (Contd.)

- The key constraints allow us to combine Purchaser with Policies and Beneficiary with Dependents.
- Participation constraints lead to NOT NULL constraints.
- What if Policies is a weak entity set?

```
CREATE TABLE  Policies (
  policyid  INTEGER,
  cost  REAL,
  ssn  CHAR(11)  NOT NULL,
  PRIMARY KEY (policyid).
  FOREIGN KEY (ssn) REFERENCES Employees,
    ON DELETE CASCADE)
```

```
 CREATE TABLE Dependents (
pname  CHAR(20),
age  INTEGER,
policyid  INTEGER,
PRIMARY KEY (pname, policyid).
FOREIGN KEY (policyid) REFERENCES Policies,
  ON DELETE CASCADE)
```

View Definition

- A relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a view.
- A view is defined using the create view statement which has the form

  create view *v* as < query expression >

  View is stored only as definition .When a reference is made to a view its definition is scanned, base table is opened and view is created on top of table.
- If a view is used to only look at table data and nothing else and view is called Read only view
- If a view is used to only look at table data as well as insert,update and delete table data is called Updatable view

Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
- When data redundancy is to be kept minimum while maintaining security.
  - Given YoungStudents, but not Students or Enrolled, we can find students s who have are enrolled, but not the *cid's* of the courses they are enrolled in.

```
CREATE  VIEW  YoungActiveStudents (name, grade)
        AS  SELECT   S.name, E.grade
        FROM  Students S, Enrolled E
        WHERE  S.sid = E.sid and S.age<21
```

Example Queries

- **A view consisting of branches and their customers**

```
create view all_customer as
        (select branch_name, customer_name
         from depositor, account
        where depositor.account_number =
                account.account_number )
        union
        (select branch_name, customer_name
        from borrower, loan
```

- **Find all customers of the Perryridge branch**

```
select customer_name
        from all_customer
        where branch_name = 'Perryridge'
```

Processing of Views

- When a view is created
  - the query expression is stored in the database along with the view name
  - the expression is substituted into any query using the view
- Views definitions containing views
  - One view may be used in the expression defining another view
  - A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
  - A view relation v is said to be *recursive* if it depends on itself.

Updatable views

A view is updatable if the following conditions are satisfied:

- From clause has only one database relation
- Select clause contains only attribute name of relation and does not have any expressions, aggregates or distinct specification
- Any attribute not listed in select clause can be set to null
- Query does not have a groupby or having clause.
- If user wants to insert records with help of a view then primary key column and all the not null columns must be included in view
- User can update, delete records with help of view even if primary key column and not null columns are excluded from view definition.

Update of a View

- Create a view of all loan data in the *loan* relation, hiding the *amount* attribute

  create view *loan_branch* as
  select *loan_number, branch_name*
  from *loan*

- Add a new tuple to *loan_branch*

  insert into *loan_branch*
  values ('L-37‘, 'Perryridge‘)

  This insertion must be represented by the insertion of the tuple

  ('L-37', 'Perryridge', *null*)

  into the *loan* relation

Views defined from multiple tables

- If a view is created from multiple tables which were not created using referencing clause
- Insert,update or delete operation is not allowed
- If a view is created from multiple tables which were created using referencing clause
- Insert operation is not allowed
- Delete or modify operations do not affect master table
- View can be used to modify columns of detail table included in view

- <u>Destroying a view</u>

- Syntax: Drop view view_name
- Ex:drop view v1;

DATABASE MANAGEMENT SYSTEMS

UNIT-II

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)

Example Instances

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation, assume that names of fields in query results are `inherited' from names of fields in query input relations.

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Relational Algebra

- **Basic operations:**
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Cross-product* ($\times$) Allows us to combine two relations.
  - *Set-difference* (-) Tuples in reln. 1, but not in reln. 2.
  - *Union* (U) Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, *join*, division, renaming

Projection

- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it.

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$\pi_{sname,rating}(S2)$

age
35.0
55.5

$\pi_{age}(S2)$

Selection

- Selects rows that satisfy *selection condition*.
- No duplicates in result! (Why?)
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$\sigma_{rating > 8}(S2)$$

sname	rating
yuppy	9
rusty	10

$$\pi_{sname,rating}(\sigma_{rating > 8}(S2))$$

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

Cross-Product

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.

Conflict: Both S1 and R1 have a field called *sid*.

<u>S1 X R1</u>

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

- ***<u>Renaming operator(</u>*ρ**):

_ρ (old name -> new name) or
_ρ (position -> new name)

$$\rho\ (C(1 \to sid1, 5 \to sid2),\ S1 \times R1)$$

Joins

- *Condition Join*: $R \bowtie_c S = \sigma_c (R \times S)$

$$S1 \bowtie_{S1.sid < R1.sid} R1$$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join*.

Joins

- *Equi-Join*: A special case of condition join where the condition *c* contains only *equalities*.

$$S1 \bowtie_{sid} R1$$

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

- *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.
- *Natural Join*: Equijoin on *all* common fields.
- If two relations have no attributes in common,natural join is simply cross product.

Division

- Not supported as a primitive operator, but useful for expressing queries like:

  Find sailors who have reserved all boats.

- Let *A* have 2 fields, *x* and *y*; *B* have only field *y*:
  - *A/B* = $\left\{ \langle x \rangle \mid \exists \langle x, y \rangle \in A \ \ \forall \langle y \rangle \in B \right\}$
  - i.e., *A/B* contains all *x* tuples (sailors) such that for <u>*every*</u> *y* tuple (boat) in *B*, there is an *xy* tuple in *A*.
  - *Or*: If the set of *y* values (boats) associated with an *x* value (sailor) in *A* contains all *y* values in *B*, the *x* value is in *A/B*.
- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and *x* *y* is the list of fields of *A*.

Examples of Division A/B

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

A

pno
p2

B1

pno
p2
p4

B2

pno
p1
p2
p4

B3

sno
s1
s2
s3
s4

A/B1

sno
s1
s4

A/B2

sno
s1

A/B3

Relational Calculus

- Comes in two flavors: *Tuple relational calculus* (TRC) and *Domain relational calculus* (DRC).
- Calculus has *variables, constants, comparison ops*, *logical connectives* and *quantifiers*.
  - *TRC*: Variables range over (i.e., get bound to) *tuples*.
  - *DRC*: Variables range over *domain elements* (= field values).
  - Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called *formulas*. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to *true*.

Tuple relational calculus

- A tuple rc query has the form {T|P(T)} where T is a tuple variable and P(T) denotes a formula that describes T.

- Find all sailors with rating above 7
- {S|S € Sailors Л s.rating>7}

- Let Rel be a relation name, R & S be tuple variables,'a' be an attribute of R and 'b' be attribute of S. Let op denote operator.
- An atomic formula is one of the following
- R € Rel, R.a € S.b, R.a op constant or constant op R.a

Tuple relational calculus

- A formula is recursively defined to be one of the following
  - -- any atomic formula
  - -- ┐P,PЛQ,P V Q or P=>Q
  - -- эR(P(R)) where R is tuple variable
  - -- forall R(P(R)) where R is tuple variable
- A variable is said to be free in formula if it does not contain an occurence of quantifiers that bind it.

- Find the names and ages of sailors with rating above 7
- {P| эS є Sailors(S.Rating >7 Л P.name=S.Sname Л P.age=S.age)

Queries

- Find the sailor name,boat id and reservation date for each reservation
- {P|эR є Reserves эS є Sailors (R.Sid=S.sid Л P.bid=R.bid Л P.day=R.day Л P.sname=S.sname)

- Find the names of sailors who have reserved boat 103
- {P|эR є Reserves эS є Sailors (R.Sid=S.sid Л R.bid=103 Л P.sname=S.sname)
- Find the names of sailors who have reserved boat 103
- {P|эR є Reserves эS є Sailors (R.Sid=S.sid Л P.sname=S.sname Л эB є Boats(B.bid=R.bid Л B.color='red'))}

Free and Bound Variables

- The use of quantifiers $\exists X$ and $\forall x$ in a formula is said to ***bind*** X.
  - A variable that is not bound is free.
- Let us revisit the definition of a query:

$$\left\{ \left\langle x1, x2, \ldots, xn \right\rangle \mid p\left(\left\langle x1, x2, \ldots, xn \right\rangle \right) \right\}$$

❖ There is an important restriction: the variables x1, ..., xn that appear to the left of `|' must be the *only* free variables in the formula p(...).

Find sailors rated > 7 who've reserved a red boat

$$\left\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge T > 7 \wedge \right.$$

$$\exists\ Ir,Br,D \left(\langle Ir,Br,D\rangle \in \mathrm{Reserves} \wedge Ir = I \wedge \right.$$

$$\left.\left.\exists\ B,BN,C \left(\langle B,BN,C\rangle \in Boats \wedge B = Br \wedge C = 'red'\right)\right)\right\}$$

- Observe how the parentheses control the scope of each quantifier's binding.
- Find names of sailors who've reserved a red boat

$$\{\rangle < N > | I, T, A \langle\rangle I, N, T, A \langle \in Sailors$$

$$\{\wedge$$

$$\exists < (\rangle I, Br, D \langle \in \mathrm{Re}\ serves\ \wedge < Br, BN, 'red\epsilon' > Boats$$

Find sailors who’ve reserved all boats

$$\left\{ \langle I, N, T, A \rangle \mid \langle I, N, T, A \rangle \in Sailors \wedge \right.$$

$$\forall \; B, BN, C \left(\neg \left(\langle B, BN, C \rangle \in Boats \right) \vee \right.$$

$$\left. \left. \left(\exists \; Ir, Br, D \left(\langle Ir, Br, D \rangle \in \text{Reserves} \wedge I = Ir \wedge Br = B \right) \right) \right) \right\}$$

- **Find sailors who've reserved all boats (again!)**

$$\left\{ \langle I, N, T, A \rangle \mid \langle I, N, T, A \rangle \in Sailors \wedge \right.$$

$$\forall \ \langle B, BN, C \rangle \in Boats$$

$$\left. \left(\exists \ \langle Ir, Br, D \rangle \in \mathrm{Re}serves \left(I = Ir \wedge \ Br = B \right) \right) \right\}$$

- **To find sailors who've reserved all red boats:**

$$\ldots. \quad \left. \left(C \neq 'red' \vee \exists \ \langle Ir, Br, D \rangle \in \mathrm{Re}serves \left(I = Ir \wedge \ Br = B \right) \right) \right\}$$

Unsafe Queries, Expressive Power

- It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called ***unsafe***.
  - e.g., $\left\{ S \mid \neg \left(S \in Sailors \right) \right\}$
- It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC; the converse is also true.
- ***Relational Completeness***: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.

Data Definition Language

Allows the specification of:

- The schema for each relation, including attribute types.
- Integrity constraints
- Authorization information for each relation.
- Non-standard SQL extensions also allow specification of
  - The set of indices to be maintained for each relations.
  - The physical storage structure of each relation on disk.

Create Table Construct

- An SQL relation is defined using the create table command:

  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of attribute A_i

  Example:

```
create table branch
    (branch_name    char(15),
     branch_city    char(30),
     assets         integer)
```

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.

Integrity Constraints on Tables

- not null
- primary key (A_1, ..., A_n)

Example: Declare *branch_name* as the primary key for *branch*

.

```
create table branch
        (branch_name  char(15),
         branch_citychar(30) not null,
         assets            integer,
         primary key (branch_name))
```

primary key declaration on an attribute automatically ensures not null in SQL-92 onwards, needs to be explicitly stated in SQL-89

Basic Insertion and Deletion of Tuples

- Newly created table is empty
- Add a new tuple to *account*

  insert into *account*
  values ('A-9732', 'Perryridge', 1200)

  - Insertion fails if any integrity constraint is violated
- Delete *all* tuples from *account*

  delete from *account*

Drop and Alter Table Constructs

- The drop table command deletes all information about the dropped relation from the database.
- The alter table command is used to add attributes to an existing relation:

  alter table *r* add *A D*

  where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
  - All tuples in the relation are assigned *null* as the value for the new attribute.
- The alter table command can also be used to drop attributes of a relation:

  alter table *r* drop *A*

  where *A* is the name of an attribute of relation *r*
  - Dropping of attributes not supported by many databases

Basic Query Structure

- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.

- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

- The result of an SQL query is a relation.

The select Clause

- The select clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all branches in the *loan* relation:

  select *branch_name*
  from *loan*

- In the relational algebra, the query would be:

$$\Pi_{branch\_name}(loan)$$

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name*
  - Some people use upper case wherever we use bold font.

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword distinct after select.
- Find the names of all branches in the *loan* relations, and remove duplicates

select distinct *branch_name*
from *loan*

- The keyword all specifies that duplicates not be removed.

select all *branch_name*
from *loan*

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- **To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.**

  select *loan_number*
  from *loan*
  where *branch_name = '*Perryridge' **and** *amount* > 1200
- Comparison results can be combined using the logical connectives **and, or,** and **not.**

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower X loan*

select *
from *borrower, loan*

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

select *customer_name, borrower.loan_number, amount*
from *borrower, loan*
where *borrower.loan_number = loan.loan_number* **and**
branch_name = 'Perryridge'

The Rename Operation

- SQL allows renaming relations and attributes using the as clause:

  old-name as *new-name*

- E.g. Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id.*

select *customer_name, borrower.loan_number* **as** *loan_id, amount*
from *borrower, loan*
where *borrower.loan_number = loan.loan_number*

Tuple Variables

- Tuple variables are defined in the from clause via the use of the as clause.
- Find the customer names and their loan numbers and amount for all customers having a loan at some branch.

select *customer_name, T.loan_number, S.amount*
from *borrower* **as** *T, loan* **as** *S*
where *T.loan_number = S.loan_number*

- Find the names of all branches that have greater assets than some branch located in Brooklyn.

select distinct *T.branch_name*
from *branch* as *T, branch* as *S*
where *T.assets > S.assets* and *S.branch_city* = 'Brooklyn'

- Keyword as is optional and may be omitted

borrower as *T* ≡ *borrower T*

  - Some database such as Oracle *require* as to be omitted

Example Instances

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

- We will use these instances of the Sailors and Reserves relations in our examples.
- If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Find sailors who've reserved at least one boat

```
SELECT  S.sid
FROM  Sailors S, Reserves R
WHERE  S.sid=R.sid
```

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

- Illustrates use of arithmetic expressions and string pattern matching: ***Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.***
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters.

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator "like" uses patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Find the names of all customers whose street includes the substring "Main".

  select *customer_name*
  from *customer*
  where *customer_street* like '% Main%'

- Match the name "Main%"

  like 'Main\%' escape '\'

- SQL supports a variety of string operations such as
  - concatenation (using "||")
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- List in alphabetic order the names of all customers having a loan in Perryridge branch

  select distinct *customer_name*
  from *borrower, loan*
  where *borrower loan_number* = *loan.loan_number* and
  branch_name = 'Perryridge'
  order by *customer_name*

- We may specify desc for descending order or asc for ascending order, for each attribute; ascending order is the default.
  - Example: order by *customer_name* desc

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- Multiset versions of some of the relational algebra operators – given multiset relations r_1 and r_2:
  1. $\sigma_\theta(r_1)$*:* If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections $\sigma_{\theta,}$, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
  2. $\Pi_A(r)$: For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.
  3. $r_1 \times r_2$: If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \times c_2$ copies of the tuple $t_1.\ t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

- Example: **Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:**

  $$r_1 = \{(1, a)\ (2,a)\} \qquad r_2 = \{(2), (3), (3)\}$$

- **Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1)$ x r_2 would be**

  $$\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$$

- SQL duplicate semantics:

  select $A_{1,}, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  is equivalent to the *multiset* version of the expression:

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

Set Operations

- The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

  Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:
  - $m + n$ times in *r* **union all** *s*
  - min(*m,n)* times in *r* **intersect all** *s*
  - max(0, $m - n$) times in *r* **except all** *s*

Nested Queries

Find names of sailors who've reserved boat #103:

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.sid IN  (SELECT  R.sid
                  FROM  Reserves R
                  WHERE  R.bid=103)
```

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- To find sailors who've *not* reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a <u>*nested loops*</u> evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

Nested Queries with Correlation

Find names of sailors who've reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
              FROM Reserves R
              WHERE R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN.
- If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by *R.bid*?)
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

Aggregate Functions (Cont.)

- **Find the average account balance at the Perryridge branch.**

```
select avg (balance)
    from account
    where branch_name = 'Perryridge'
```

- Find the number of tuples in the *customer* relation.

```
select count (*)
    from customer
```

- Find the number of depositors in the bank.

```
select count (distinct customer_name)
    from depositor
```

Aggregate Functions – Group By

- **Find the number of depositors for each branch.**

select *branch_name,* **count (distinct** *customer_name)*
from *depositor, account*
where *depositor.account_number = account.account_number*
group by *branch_name*

Note: Attributes in select clause outside of aggregate functions must appear in group by list

Aggregate Functions – Having Clause

- **Find the names of all branches where the average account balance is more than $1,200.**

select *branch_name,* avg (*balance*)
from *account*
group by *branch_name*
having avg (*balance*) > 1200

Note: predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A subquery is a select-from-where expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

"In" Construct

- **Find all customers who have both an account and a loan at the bank.**

```
select distinct customer_name
        from borrower
        where customer_name in (select customer_
                                        from depositor )
```

- **Find all customers who have a loan at the bank but do not have an account at the bank**

```
select distinct customer_name
        from borrower
        where customer_name not in (select customer
                                        from depositor
```

Example Query

- **Find all customers who have both an account and a loan at the Perryridge branch**

```
select distinct customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number and
          branch_name = 'Perryridge'  and
          (branch_name, customer_name ) in
                (select branch_name, customer_name
                 from depositor, account
                 where depositor.account_number =
                        account.account_number )
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

“Some” Construct

- **Find all branches that have greater assets than some branch located in Brooklyn.**

```
select distinct T.branch_name
        from branch as T, branch as S
        where T.assets > S.assets and
              S.branch_city = 'Brooklyn'
```

- Same query using > **some** clause

```
select branch_name
        from branch
        where assets > some
              (select assets
               from branch
               where branch_city = 'Brooklyn')
```

“All” Construct

- **Find the names of all branches that have greater assets than all branches located in Brooklyn.**

```
select branch_name
        from branch
        where assets > all
                (select assets
                from branch
                where branch_city = 'Brooklyn')
```

"Exists" Construct

- **Find all customers who have an account at all branches located in Brooklyn.**

```
select distinct S.customer_name
        from depositor as S
        where not exists (
                (select branch_name
                from branch
                where branch_city = 'Brooklyn')
                except
                (select R.branch_name
                from depositor as T, account as R
                where T.account_number = R.account_number and
                        S.customer_name = T.customer_name ))
```

- Note that $X - Y = \varnothing \iff X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- **Find all customers who have at most one account at the Perryridge branch**.

```
select T.customer_name

 from depositor as T
 where unique (
     select R.customer_name
     from account, depositor as R
     where T.customer_name = R.customer_name and
             R.account_number = account.account_number
and
             account.branch_name = 'Perryridge')
```

Example Query

- **Find all customers who have at least two accounts at the Perryridge branch.**

select distinct *T.customer_name*
from *depositor* **as** *T*
where not unique (
  select *R.customer_name*
  from *account, depositor* **as** *R*
  where *T.customer_name* = *R.customer_name* **and**
    R.account_number = *account.account_number* **and**
    account.branch_name = 'Perryridge')

- Variable from outer level is known as a **correlation variable**

Modification of the Database – Deletion

- **Delete all account tuples at the Perryridge branch**

  delete from *account*
  where *branch_name =* 'Perryridge'

- **Delete all accounts at every branch located in the city 'Needham'.**

  delete from *account*
  where *branch_name* **in (select** *branch_name*
  from *branch*
  where *branch_city =*
  'Needham')

Example Query

- **Delete the record of all accounts with balances below the average at the bank.**

```
delete from account
        where balance < (select avg (balance )
                             from account )
```

- **Problem: as we delete tuples from deposit, the average balance changes**
- **Solution used in SQL:**

  1. First, compute avg balance and find all tuples to delete

  2. Next, delete all tuples found above (without recomputing avg or retesting the tuples)

Modification of the Database – Insertion

- Add a new tuple to *account*

  insert into *account*
  values ('A-9732', 'Perryridge', 1200)

  or equivalently

  insert into *account* (*branch_name, balance, account_number*)
  values ('Perryridge', 1200, 'A-9732')

- Add a new tuple to *account* with *balance* set to null

  insert into *account*
  values ('A-777','Perryridge', *null*)

Modification of the Database – Insertion

- **Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account**

  insert into *account*
      select *loan_number, branch_name,* 200
      from *loan*
      where *branch_name* = 'Perryridge'
  insert into *depositor*
      select *customer_name, loan_number*
      from *loan, borrower*
      where branch_name = 'Perryridge'
          and *loan.account_number* = *borrower.account_number*

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation
  - Motivation: **insert into** *table*1 **select** * **from** *table*1

Modification of the Database – Updates

- **Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.**
  - Write two **update** statements:

    update *account*
    set *balance* = *balance* ∗ 1.06
    where *balance* > 10000

    update *account*
    set *balance* = *balance* ∗ 1.05
    where *balance* ≤ 10000
  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- **Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.**

```
update account
set balance = case
                  when balance <= 10000 then balance *1.05
                  else   balance * 1.06
              end
```

More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op* ANY, *op* ALL, *op* IN $>,<,=,\geq,\leq,\neq$
- **Find sailors whose rating is greater than that of some sailor called Horatio:**

```
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
                      FROM Sailors S2
                      WHERE S2.sname='Horatio')
```

Rewriting INTERSECT Queries Using IN

Find sid's of sailors who've reserved both a red and a green boat:

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
          AND S.sid IN  (SELECT  S2.sid
                         FROM  Sailors S2, Boats B2, Reserves R2
                         WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
                                AND  B2.color='green')
```

- Similarly, EXCEPT queries re-written using NOT IN.
- To find *names* (not *sid*'s) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in SELECT clause. (What about INTERSECT query?)

Division in SQL

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS
          ((SELECT  B.bid
            FROM  Boats B)
           EXCEPT
            (SELECT  R.bid
             FROM  Reserves R
             WHERE  R.sid=S.sid))
```

(1)
Find sailors who've reserved all boats.

- Let's do it the hard way, without EXCEPT:

(2)
```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS  (SELECT  B.bid
                    FROM  Boats B
                    WHERE  NOT EXISTS  (SELECT  R.bid
                                        FROM  Reserves R
                                        WHERE  R.bid=B.bid
                                          AND R.sid=S.sid))
```

Sailors S such that ...

there is no boat B without ...

a Reserves tuple showing S reserved B

Aggregate Operators

- Significant extension of relational algebra.

```
COUNT (*)
COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)
```

single column

```
SELECT  COUNT (*)
FROM  Sailors S
```

```
SELECT  AVG (S.age)
FROM  Sailors S
WHERE  S.rating=10
```

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.rating= (SELECT  MAX(S2.rating)
                  FROM  Sailors S2)
```

```
SELECT  COUNT (DISTINCT S.rating)
FROM  Sailors S
WHERE S.sname='Bob'
```

```
SELECT  AVG ( DISTINCT S.age)
FROM  Sailors S
WHERE  S.rating=10
```

Find name and age of the oldest sailor(s)

- The first query is illegal! (We'll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
SELECT S.sname, MAX (S.age)
FROM Sailors S
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age =
        (SELECT MAX (S2.age)
         FROM Sailors S2)
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
        FROM Sailors S2)
        = S.age
```

Motivation for Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: ***Find the age of the youngest sailor for each rating level.***
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

For $i = 1, 2, \ldots, 10$:

```
SELECT MIN (S.age)
FROM Sailors S
WHERE S.rating = i
```

Queries With GROUP BY and HAVING

```
SELECT      [DISTINCT]  target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

- The *target-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
  - The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group,* and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors

```
SELECT S.rating, MIN (S.age)
                     AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors.

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

rating	minage
3	25.5
7	35.0
8	25.5

Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 <u>such</u> sailors and with every sailor under 60.

HAVING COUNT (*) > 1 AND EVERY (S.age <=60)

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

rating	minage
7	35.0
8	25.5

What is the result of changing EVERY to ANY?

Find age of the youngest sailor with age $\geq$ 18, for each rating with at least 2 sailors between 18 and 60.

```
SELECT  S.rating,  MIN (S.age)
                        AS minage
FROM  Sailors S
WHERE  S.age >= 18 AND S.age <= 60
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

For each red boat, find the number of reservations for this boat

```
SELECT B.bid, COUNT (*) AS scount
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```

- Grouping over a join of three relations.
- What do we get if we remove *B.color='red'* from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?

Find age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

```
SELECT  S.rating,  MIN (S.age)
FROM  Sailors S
WHERE  S.age > 18
GROUP BY  S.rating
HAVING  1  <  (SELECT  COUNT (*)
               FROM  Sailors S2
               WHERE  S.rating=S2.rating)
```

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18!
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) >1

Find those ratings for which the average age is the minimum over all ratings

- **Aggregate operations cannot be nested! WRONG:**

```
SELECT S.rating
FROM Sailors S
WHERE S.age = (SELECT MIN (AVG (S2.age)) FROM Sailors S2)
```

- Correct solution (in SQL/92):

```
SELECT Temp.rating, Temp.avgage
FROM (SELECT S.rating, AVG (S.age) AS avgage
      FROM Sailors S
      GROUP BY S.rating) AS Temp
WHERE Temp.avgage = (SELECT MIN (Temp.avgage)
                     FROM Temp)
```

Null Values

- **Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).**
  - **SQL provides a special value *null* for such situations.**
- **The presence of *null* complicates many issues. E.g.:**
  - **Special operators needed to check if value is/is not *null*.**
  - **Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?**
  - **We need a 3-valued logic (true, false and *unknown*).**
  - **Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)**
  - **New operators (in particular, *outer joins*) possible/needed.**

Null Values and Three Valued Logic

- **Any comparison with *null* returns *unknown***
  - **Example*: 5 < null   or   null <> null   or   null = null***
- **Three-valued logic using the truth value *unknown*:**
  - **OR: (*unknown* or *true*)   = *true*,**
    (*unknown* or *false*) = *unknown*
    (*unknown* or *unknown*) = *unknown*
  - **AND: *(true* and *unknown) = unknown,***
    (false* and *unknown) = false,
    (unknown* and *unknown) = unknown
  - **NOT*: (*not *unknown) = unknown***
  - **"*P* is unknown" evaluates to true if predicate *P* evaluates to *unknown***
- **Result of where clause predicate is treated as *false* if it evaluates to *unknown***

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate is null can be used to check for null values.
  - Example: Find all loan number which appear in the *loan* relation with null values for *amount.*

    select *loan_number*
    from *loan*
    where *amount* is null
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- However, aggregate functions simply ignore nulls

Null Values and Aggregates

- Total all loan amounts

  select sum (*amount*)
  from *loan*

  - **Above statement ignores null amounts**
  - **Result is *null* if there is no non-null amount**

- **All aggregate operations except count(*) ignore tuples with null values on the aggregated attributes.**

Joined Relations**

- **Join operations** take two relations and return as a result another relation.
- **These additional operations are typically used as subquery expressions in the from clause**
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Joined Relations – Datasets for Examples

- **Relation *borrower***

  - Relation *loan*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- **Note: borrower information missing for L-260 and loan information missing for L-155**

- ***Select S.sid, R.bid from Sailors S natural left outer join Reserves R***

Sid	Bid
22	101
31	Null
58	103

Joined Relations – Examples

- *loan* **inner join** *borrower* **on**
  loan.loan_number = borrower.loan_number

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- *loan* **left outer join** *borrower* **on**
  loan.loan_number = borrower.loan_number

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Joined Relations – Examples

- *loan* **natural inner join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- *loan* **natural right outer join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

customers who have either an account or a loan (but not both) at the bank.

select ***customer_name***
from (*depositor* natural full outer join *borrower*)
where *account_number* is null or *loan_number*

Joined Relations – Examples

- Natural join can get into trouble if two relations have an attribute with same name that should not affect the join condition
  - e.g. an attribute such as *remarks* may be present in many tables
- *Solution:*
  - *loan* **full outer join** *borrower* **using** (*loan_number*)

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- **Find the average account balance of those branches where the average account balance is greater than $1200.**

> **select *branch_name, avg_balance***
> **from (select *branch_name,* avg (*balance*)**
> **from *account***
> **group by *branch_name*)**
> **as *branch_avg* (*branch_name, avg_balance*)**
> **where *avg_balance* > 1200**

Note that we do not need to use the having clause, since we compute the temporary (view) relation *branch_avg* in the from clause, and the attributes of *branch_avg* can be used directly in the where clause.

Integrity Constraints (Review)

- **An IC describes conditions that every *legal instance* of a relation must satisfy.**
  - **Inserts/deletes/updates that violate IC's are disallowed.**
  - **Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)**
- ***Types of IC's*: Domain constraints, primary key constraints, foreign key constraints, general constraints.**
  - ***Domain constraints*: Field values must be of right type. Always enforced.**
  - **EX:Create domain ratingval integer default 1 check(value>=1 and value<=10)**
  - **Rating ratingval**

General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

```
CREATE TABLE
Sailors
( sid  INTEGER,
sname  CHAR(10),
rating  INTEGER,
age  REAL,
PRIMARY KEY  (sid),
CHECK  ( rating >= 1
        AND rating <= 10
```

```
CREATE TABLE  Reserves
        ( sname  CHAR(10),
        bid  INTEGER,
        day  DATE,
        PRIMARY KEY  (bid,day),
        CONSTRAINT  noInterlakeRes
        CHECK  (`Interlake' <>
                    ( SELECT  B.bname
                    FROM  Boats B
                    WHERE  B.bid=bid)))
```

Constraints Over Multiple Relations

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.

```
CREATE TABLE  Sailors
        ( sid  INTEGER,
        sname  CHAR(10),
        rating  INTEGER,
        age  REAL,
        PRIMARY KEY  (sid),
        CHECK
        ( (SELECT COUNT (S.sid) FROM Sailors S)
        + (SELECT COUNT (B.bid) FROM Boats B) < 100
```

Number of boats plus number of sailors is < 100

```
CREATE ASSERTION  smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

Triggers

- **Trigger: procedure that starts automatically if specified changes occur to the DBMS**
- **Three parts:**
  - **Event (activates the trigger)**
  - **Condition (tests whether the triggers should run)**
  - **Action (what happens if the trigger runs)**
  - **Types of triggers**
  - **Row level triggers:triggering event should be defined to occur for each modified record. For each row clause is used.**
  - **Statement-level triggers: trigger is executed just once for each(insert) statement. For each statement clause is used.**

Examples

```
Create Trigger init_count before insert on students /*event*/
  Declare
  Count Integer;
  Begin
   count:=0;                    /*action*/
 End

Create Trigger incr_count after insert on students /*event*/
When(new.age<18) /*condition*/
For each row
Begin
count:=count+1; /*action*/
end
```

Triggers: Example (SQL:1999)

```
CREATE TRIGGER youngSailorUpdate
    AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT
    INSERT
        INTO YoungSailors(sid, name, age, rating)
        SELECT sid, name, age, rating
        FROM NewSailors N
        WHERE N.age <= 18
```

DATABASE MANAGEMENT SYSTEMS

UNIT-III

INTRODUCTION TO SCHEMA REFINEMENT

Problems Caused by Redundancy

- **Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems:**

- **Redundant storage: Some information is stored repeatedly.**
- **Update anomalies: If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.**
- **Insertion anomalies: It may not be possible to store some information unless some other information is stored as well.**
- **Deletion anomalies: It may not be possible to delete some information without losing some other information as well.**

- **Consider a relation obtained by translating a variant of the Hourly Emps entity set**

Ex: Hourly Emps(*ssn*, *name*, *lot*, *rating*, *hourly wages*, *hours worked*)

- **The key for Hourly Emps is *ssn*. In addition, suppose that the *hourly wages* attribute**

  is determined by the *rating* attribute.

  That is, for a given *rating* value, there is only one permissible *hourly wages* value. This IC is an example of a *functional dependency*.
- **It leads to possible redundancy in the relation Hourly Emps**

Example: Constraints on Entity Set

- Consider relation obtained from Hourly_Emps:
  - **Hourly_Emps (*<u>ssn</u>*, *name*, *lot*, *rating*, *hrly_wages*, *hrs_worked*)**
- *<u>Notation</u>*: We will denote this relation schema by listing the attributes: SNLRWH
  - This is really the *set* of attributes {S,N,L,R,W,H}.
  - Sometimes, we will refer to all attributes of a relation by using the relation name. (e.g., Hourly_Emps for SNLRWH)
- Some FDs on Hourly_Emps:
  - *ssn* is the key: $S \rightarrow SNLRWH$
  - *rating* determines *hrly_wages*: $R \rightarrow W$

Example (Contd.)

R	W
8	10
5	7

- Problems due to R $\longrightarrow$ W :
  - ***Update anomaly***: **Can we change W in just the 1st tuple of SNLRWH?**
  - ***Insertion anomaly***: **What if we want to insert an employee and don't know the hourly wage for his rating?**
  - ***Deletion anomaly***: **If we delete all employees with rating 5, we lose the information about the wage for rating 5!**

Wages Hourly_Emps2

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

ssn	*name*	*lot*	*rating*	*Hourly wages*	*hours worked*
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Decomposition

- ***Redundancy*** **is at the root of several problems associated with relational schemas:**
  - **redundant storage, insert/delete/update anomalies**

- **Main refinement technique: *decomposition* (replacing ABCD with, say, AB and BCD, or ACD and ABD).**
- **Decomposition should be used judiciously:**
  - **Is there reason to decompose a relation?**
  - **What problems (if any) does the decomposition cause?**

Use of Decompositions

- **Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural.**
- **Functional dependencies (ICs) can be used to identify such situations and to suggest revetments to the schema.**
- **The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of smaller relations.**
- **Each of the smaller relations contains a subset of the attributes of the original relation.**
- **We refer to this process as decomposition of the larger relation into the smaller relations**

- We can deal with the redundancy in Hourly Emps by decomposing it into two relations:

Hourly Emps2(*ssn, name, lot, rating, hours worked*)

Wages(*rating, hourly wages*)

rating	*hourly wages*
8	10
5	7

ssn	*name*	*lot*	*rating*	*hours worked*
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

Problems Related todecomposition

- **Unless we are careful, decomposing a relation schema can create more problems than it solves.**
- **Two important questions must be asked repeatedly:**
- **1. Do we need to decompose a relation?**
- **2. What problems (if any) does a given decomposition cause?**
- **To help with the rst question, several *normal forms* have been proposed for relations.**
- **If a relation schema is in one of these normal forms, we know that certain kinds of**

problems cannot arise.

Functional Dependencies (FDs)

- A <u>functional dependency</u> X → Y holds over relation R if, for every allowable instance *r* of R:
  - $t1 \in r$, $t2 \in r$, $\pi_X(t1) = \pi_X(t2)$ implies $\pi_Y(t1) = \pi_Y(t2)$
  - **If t1.X=t2.X then t1.Y=t2.Y**
  - i.e., given two tuples in *r*, if the X values agree, then the Y values must a
    attributes.)
  - FD: AB → C (

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

REASONING ABOUT FD'S

- **Workers(<u>ssn,</u>name,lot,did,since)**
- **We know ssn-did holds and FD did->lot is given to hold. Therefore FD ssn->lot holds**
- **We say that an FD f is implied by a given set F of FD's if f holds on every relation instance that satisfies all dependencies in F.i.e f holds whenever all FD's hold.**
- **<u>Closure of set of FD's:</u>**
- **The set of all FD's implied by a given set F of FD's is called closure of F denoted as F+.**
- **How can we infer or compute the closure of given set F of FD's. Sol:Armstrong axioms can be applied repeatedly to infer all FD's implied by set of F of FD's**

- We use X,Y,Z to denote sets of attributes over a relation schema R

- Relexivity: If X subset of Y then X->Y
- Augmentation: If X->Y then XZ->YZ for any Z
- Transitivity: If X->Y and Y-> Z then X->Z

- Union:If X->Y ,X->Z then X->YZ
- Decomposition:If X->YZ then X->Y and X->Z

Constraints on a Relationship Set

- **Suppose that we have entity sets Parts, Suppliers, and Departments, as well as a relationship set Contracts that involves all of them.**
- We refer to the schema for Contracts as *CQPSD*. A contract with contract id *C* species that a supplier *S* will supply some quantity *Q* of a part *P* to a department *D*.
- **We might have a policy that a department purchases at most one part from any given supplier**.
- **Thus, if there are several contracts between the same supplier and department, we know that the same part must be involved in all of them. This constraint is an FD, *DS ! P*.**

- Consider relation schema ABC with FD's

A->B and B->C.

Using reflexivity

X->Y where Y C X,X C ABC and Y C ABC

From transitivity we get A->C

From augmentation we get nontrivial dependencies

AC->BC,AB->AC,AB->CB

Reasoning About FDs (Contd.)

- Couple of additional rules (that follow from AA):
  - *Union*: If X → Y and X → Z, then X → YZ
  - *Decomposition*: If X → YZ, then X → Y and X → Z
- Example: Contracts(*cid,sid,jid,did,pid,qty,value*), and:
  - C is the key: C → CSJDPQV
  - Project purchases each part using single contract:
  - JP → C
  - Dept purchases at most one part from a supplier: S
  - SD → P
- JP → C, C → CSJDPQV imply JP → CSJDPQV
- SD → P implies SDJ → JP
- SDJ → JP, JP → CSJDPQV imply SDJ → CSJDPQV

Closure of a Set of FDs

- The set of all FDs implied by a given set *F* of FDs is called the **closure of F** and is denoted as *F*+.
- **An important question is how we can infer, or compute, the closure of a given set *F* of FDs.**
- The following three rules, called **Armstrong's Axioms**, can be applied repeatedly to infer all FDs implied by a set *F* of FDs.
- We use *X*, *Y*, and *Z* to denote *sets* of attributes over a relation schema *R*:

Attribute Closure

- **If we just want to check whether a given dependency, say, *X* → *Y*, is in the closure of a set *F* of FDs, we can do so effciently without computing *F*+.**
- **We first compute the attribute closure *X*+ with respect to *F*, which is the set of attributes *A* such that *X* → *A* can be inferred using the Armstrong Axioms.**
- **The algorithm for computing the attribute closure of a set *X* of attributes is**
- *closure* = *X*;
  repeat until there is no change: *{*
  if there is an FD *U* → *V* in *F* such that *U subset of closure*,
  then set *closure* = *closure union of V*
  }

NORMAL FORMS

- **The normal forms based on FDs are fi*rst normal form (1NF), second normal form (2NF), third normal form (3NF)*, and *Boyce-Codd normal form (BCNF)*.**
- **These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF.**
- **A relation is in first normal form if every field contains only atomic values, that is, not lists or sets.**
- **This requirement is implicit in our defition of the relational model.**
- **Although some of the newer database systems are relaxing this requirement 2NF is mainly of historical interest.**
- **3NF and BCNF are important from a database design standpoint.**

Normal Forms

- **Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed!**
- **If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help.**
- Role of FDs in detecting redundancy:
  - **Consider a relation R with 3 attributes, ABC.**
    - No FDs hold: **There is no redundancy here.**
    - Given A $\rightarrow$ B: **Several tuples could have the same A value, and if so, they'll all have the same B value!**

Normal Forms

- Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed!
- If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help.
- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC.
    - No FDs hold: There is no redundancy here.
    - Given A B: Several tuples could have the same A value, and if so, they'll all have the same B value!

First Normal Form

- 1NF (First Normal Form)
  - a relation R is in 1NF if and only if it has only single-valued attributes (atomic values)
  - EMP_PROJ (SSN, PNO, HOURS, ENAME, PNAME, PLOCATION)

    PLOCATION is not in 1NF (multi-valued attrib.)
  - solution: decompose the relation

    EMP_PROJ2 (SSN, PNO, HOURS, ENAME, PNAME)

    LOC (PNO, PLOCATION)

Second Normal Form

- **2NF (Second Normal Form)**
  - **a relation R in 2NF if and only if it is in 1NF and every nonkey column depends on a key not a subset of a key**
  - **all nonprime attributes of R must be fully functionally dependent on a whole key(s) of the relation, not a part of the key**
  - **no violation: single-attribute key or no nonprime attribute**

Second Normal Form (Contd)

- 2NF (Second Normal Form)
  - violation: part of a key $\rightarrow$ nonkey

    EMP_PROJ2 (<u>SSN</u>, <u>PNO</u>, HOURS, ENAME, PNAME)

    SSN $\rightarrow$ ENAME

    PNO $\rightarrow$ PNAME
  - solution: decompose the relation

    EMP_PROJ3 (<u>SSN</u>, <u>PNO</u>, HOURS)

    EMP (<u>SSN</u>, ENAME)

    PROJ (<u>PNO</u>, PNAME)

Third Normal Form

- a relation R in 3NF if and only if it is in 2NF and every nonkey column does not depend on another nonkey column
- all nonprime attributes of R must be non-transitively functionally dependent on a key of the relation

Third Normal Form (Contd)

- 3NF (Third Normal Form)
- violation: nonkey $\rightarrow$ nonkey

  - SUPPLIER (<u>SNAME</u>, STREET, CITY, STATE, TAX)

    SNAME $\rightarrow$ STREET, CITY, STATE

    STATE $\rightarrow$ TAX (nonkey $\rightarrow$ nonkey)

    SNAME $\rightarrow$ STATE $\rightarrow$ TAX (transitive FD)
  - solution: decompose the relation

    SUPPLIER2 (<u>SNAME</u>, STREET, CITY, STATE)

    TAXINFO (<u>STATE</u>, TAX)

Boyce-Codd Normal Form (BCNF)

- Reln R with FDs *F* is in BCNF if, for all X $\rightarrow$ A in F^+
  - A $\in$ X (called a *trivial* FD), or
  - X contains a key for R.
- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.
  - No dependency in R that can be predicted using FDs alone.
  - If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.
  - If example relation is in BCNF, the 2 tuples identical (since X is a key).

X	Y	A
x	y 1	a
x	y 2	?

Decomposition of a Relation Scheme

- Suppose that relation R contains attributes *A1 ... An.* A *decomposition* of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
  - Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- E.g., Can decompose SNLRWH into SNLRH and RW.

Example Decomposition

- Decompositions should be used only when needed.
  - SNLRWH has FDs $S \rightarrow SNLRWH$ and $R \rightarrow W$
  - Second FD causes violation of 3NF; W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations, and to remove W from the main schema:
    - i.e., we decompose SNLRWH into SNLRH and RW
- The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?

Problems with Decompositions

- There are three potential problems to consider:
  - Some queries become more expensive.
    - e.g., How much did sailor Joe earn? (salary = W*H)
  - Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!
    - Fortunately, not in the SNLRWH example.
  - Checking some dependencies may require joining the instances of the decomposed relations.
    - Fortunately, not in the SNLRWH example.
- *Tradeoff*: Must consider these issues vs. redundancy.

Lossless Join Decompositions

- Decomposition of R into X and Y is <u>*lossless-join*</u> w.r.t. a set of FDs F if, for every instance *r* that satisfies F:
- $\pi_X(r) \bowtie \pi_Y(r) = r$

- It is always true that $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$
  - In general, the other direction does not hold! If it does, the decomposition is lossless-join.
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- *It is essential that all decompositions used to deal with redundancy be lossless!*
- *Consider Hourly emps relation.It has attributes SNLRWH and FD R->W causes a violation of 3NF.We dealt this violation by decomposing into SNLRH and RW.*
- *Since R is common to both decomposed relation and*

  R->W holds,this decomposition is lossles-join

More on Lossless Join

- The decomposition of R into X and Y is lossless-join wrt F if and only if the closure of F contains:
  - $X \cap Y \rightarrow X$, or
  - $X \cap Y \rightarrow Y$
- **In** particular,if an fd X->Y holds over relation R and X∩ Y is empty, the decomposition of R into R-Y and XY is lossless.
- Imp observation is repeated decompositions

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

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

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

Dependency Preserving Decomposition

- Consider CSJDPQV, C is key, JP $\to$ C and SD $\to$ P.
  - Bcoz SD->P is not a key,it causes violation
  - BCNF decomposition: CSJDQV and SDP
  - Problem: Checking JP $\to$ C requires a join!
- Dependency preserving decomposition (Intuitive):
  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold.
  - <u>*Projection of set of FDs F*</u>: If R is decomposed into X, ... projection of F onto X (denoted F_X) is the set of FDs U $\to$ V in F^+ (*closure of F*) such that U, V are in X**.**

Dependency Preserving Decompositions (Contd.)

- Decomposition of R into X and Y is ***dependency preserving*** if $(F_X \text{ union } F_Y)^+ = F^+$
  - i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.
- Important to consider F^+, not F, in this definition:
  - ABC, $A \to B$, $B \to C$, $C \to A$, decomposed into AB and BC.
  - Is this dependency preserving? Is $C \to A$ preserved?????
- Dependency preserving does not imply lossless join:
  - ABC, $A \to B$, decomposed into AB and BC.
- And vice-versa! (Example?)

Decomposition into BCNF

- Consider relation R with FDs F. If $X \rightarrow Y$ violates BCNF, decompose R into R - Y and XY.
  - Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.
  - e.g., CSJDPQV, key C, $JP \rightarrow C$, $SD \rightarrow P$, $J \rightarrow S$
  - To deal with $SD \rightarrow P$, decompose into SDP, CSJDQV.
  - To deal with $J \rightarrow S$, decompose CSJDQV into JS and CJDQV
- In general, several dependencies may cause violation of BCNF. The order in which we ``deal with'' them could lead to very different sets of relations!

BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.
  - e.g., CSZ, CS $\rightarrow$ Z, Z $\rightarrow$ C
  - Can't decompose while preserving 1st FD; not in BCNF.
- Similarly, decomposition of CSJDQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs JP $\rightarrow$ C, SD $\rightarrow$ P and J $\rightarrow$ S).
  - However, it is a lossless join decomposition.
  - In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.
    - JPC tuples stored only for checking FD! (*Redundancy!*)

Decomposition into 3NF

- Obviously, the algorithm for lossless join decomp into BCNF can be used to obtain a lossless join decomp into 3NF (typically, can stop earlier).
- To ensure dependency preservation, one idea:
  - If $X \rightarrow Y$ is not preserved, add relation XY.
  - Problem is that XY may violate 3NF! e.g., consider the addition of CJP to `preserve' $JP \rightarrow C$. What if we also have $J \rightarrow C$?
- Refinement: Instead of the given set of FDs F, use a *minimal cover for F*.

SCHEMA REFINEMENT
Constraints

on an Entity Set

- Consider the Hourly Emps relation again. The constraint that attribute *ssn* is a key can be expressed as an FD:
- *{ ssn }-> { ssn, name, lot, rating, hourly wages, hours worked}*
- For brevity, we will write this FD as *S -> SNLRWH*, using a single letter to denote each attribute
- In addition, the constraint that the *hourly wages* attribute is determined by the *rating* attribute is an

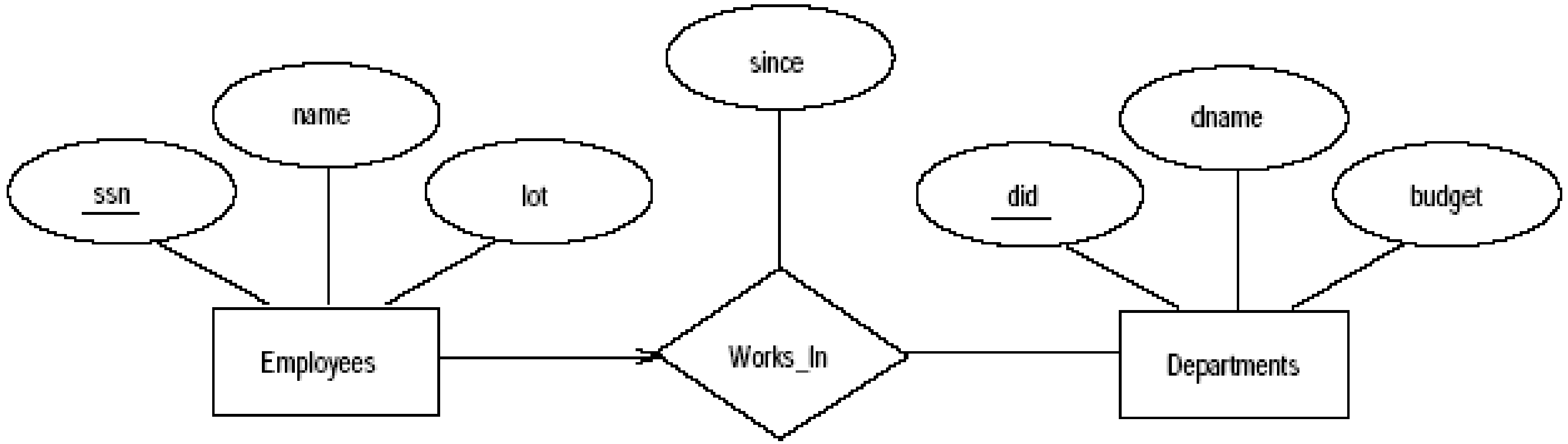
FD: *R -> W*.

Constraints on a Relationship Set

- The previous example illustrated how FDs can help to rene the subjective decisions made during ER design,
- but one could argue that the best possible ER diagram would have led to the same nal set of relations.
- Our next example shows how FD information can lead to a set of relations that eliminates some redundancy problems and is unlikely to be arrived at solely through ER design.

Identifying Attributes of Entities

- in particular, it shows that attributes can easily be associated with the `wrong' entity set during ER design.
- The ER diagram shows a relationship set called Works In that is similar to the Works In relationship set
- Using the key constraint, we can translate this ER diagram into two relations:
- Workers(*ssn*, *name*, *lot*, *did*, *since*)



Identifying Entity Sets

- Let Reserves contain attributes *S*, *B*, and *D* as before, indicating that sailor *S* has a reservation for boat *B* on day *D*.
- In addition, let there be an attribute *C* denoting the credit card to which the reservation is charged.
- Suppose that every sailor uses a unique credit card for reservations. This constraint is expressed by the FD *S* -> *C*. This constraint indicates that in relation Reserves, we store the credit card number for a sailor as often as we have reservations for that
- sailor, and we have redundancy and potential update anomalies.

Multivalued Dependencies

- Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*.
- The meaning of a tuple is that teacher *T* can teach course *C*, and book *B* is a recommended text for the course.
- There are no FDs; the key is *CTB*. However, the recommended texts for a course are independent of the instructor.

course	*teacher*	*book*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

There are three points to note here:

- The relation schema *CTB* is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over *CTB*.
- There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB*.
- Let *R* be a relation schema and let *X* and *Y* be subsets of the attributes of *R*. Intuitively,
- the **multivalued dependency** *X !! Y* is said to hold over *R* if, in every legal

- The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be epressed in terms of FDs.
- This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation using two binary relationship sets, Instructors with attributes *CT* and Text with attributes *CB*.
- Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate.

- Three of the additional rules involve only MVDs:
- **MVD Complementation:** If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow R - XY$
- **MVD Augmentation:** If $X \twoheadrightarrow Y$ and $W > \ Z$, then

  $WX \twoheadrightarrow YZ$.
- **MVD Transitivity:** If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then

  $X \twoheadrightarrow (Z - Y)$.
- **Fourth Normal Form**
- *R* is said to be in **fourth normal form (4NF)** if for every MVD $X \twoheadrightarrow Y$ that holds over *R*, one of the following statements is true:
- *Y subset of X* or $XY = R$, or
- *X* is a superkey.

Join Dependencies

- A join dependency is a further generalization of MVDs. A **join dependency** (JD) ∞{ *R*1,….. *Rn }* is said to hold over a relation *R* if *R*1*,…. Rn* is a lossless-join decomposition of *R*.
- An MVD *X ->-> Y* over a relation *R* can be expressed as the join dependency ∞ { XY,X(R−Y)*}*
- As an example, in the *CTB* relation, the MVD *C ->->T* can be expressed as the join dependency ∞{ CT, CB*}*
- Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

Fifth Normal Form

- A relation schema *R* is said to be in **fifth normal form (5NF)** if for every JD ∞{ *R*1,…. *Rn* } that holds over *R*, one of the following statements is true:
- *Ri* = *R* for some *i*, or
- The JD is implied by the set of those FDs over *R* in which the left side is a key for *R*.
- The following result, also due to Date and Fagin, identies conditions|again, detected using only FD information|under which we can safely ignore JD information.
- If a relation schema is in 3NF and each of its keys consists of a single attribute,it is also in 5NF.

Inclusion Dependencies

- MVDs and JDs can be used to guide database design, as we have seen, although they are less common than FDs and harder to recognize and reason about.
- In contrast, inclusion dependencies are very intuitive and quite common. However, they typically have little influence on database design
- The main point to bear in mind is that we should not split groups of attributes that participate in an inclusion dependency.
- Most inclusion dependencies in practice are *key-based*, that is, involve only keys.

Recovery System

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers \$50 from account *A* to account *B*; goal is either to perform all database modifications made by T_i or none at all.
- Several output operations may be required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.

Recovery and Atomicity (Cont.)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - log-based recovery, and
  - shadow-paging
- We assume (initially) that transactions run serially, that is, one after the other.

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
  - Focus of this chapter
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Log-Based Recovery

- A log is kept on stable storage.
  - The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a <T_i start>log record
- *Before* T_i executes write(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record <T_i commit> is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- The deferred database modification scheme records all modifications to the log, but defers all the writes to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing <T_i *start*> record to log.
- A write(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, <T_i commit> is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

- **During recovery after a crash, a transaction needs to be redone if and only if both <T_i start> and<T_i commit> are there in the log.**
- **Redoing a transaction T_i (redoT_i) sets the value of all data items updated by the transaction to the new values.**
- **Crashes can occur while**
  - **the transaction is executing the original updates, or**
  - **while recovery action is being taken**
- example transactionsT_0 and T_1 (T_0 executes before T_1):T_0: **read** (A)
  T_1 : **read** (C)

A: - A - 50	C:- C- 100
Write (A)	**write** (C)
read (B)	
B:- B + 50	
write (B)	

LOG DATABASE

<t0 start>

<t0,A,950>

<t0,B,2050>

<t0,commit>

<t1,start>

<t1,c,600>

<t1,commit>

<u>Portion of log</u>

<t0 start>

<t0,A,950>

<t0,B,2050>

<t0,commit>

A=950

B=2050

<t1,start>

<t1,c,600>

<t1,commit>

C=600

Deferred Database Modification (Cont.)

- Below we show the log as it appears at three instances of time.

$<T_0$ start$>$	$<T_0$ start$>$	$<T_0$ start$>$
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ commit$>$	$<T_0$ commit$>$
	$<T_1$ start$>$	$<T_1$ start$>$
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ commit$>$
(a)	(b)	(c)

Immediate Database Modification Example

Log	**Write**	**Output**
<T_0 **start**>		
<T_0, A, 1000, 950>		
T_o, B, 2000, 2050		
	A = 950	
	B = 2050	
<T_0 **commit**> X_1		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

$<T_0$ start$>$	$<T_0$ start$>$	$<T_0$ start$>$
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit$>$	$<T_0$ commit$>$
	$<T_1$ start$>$	$<T_1$ start$>$
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit$>$
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

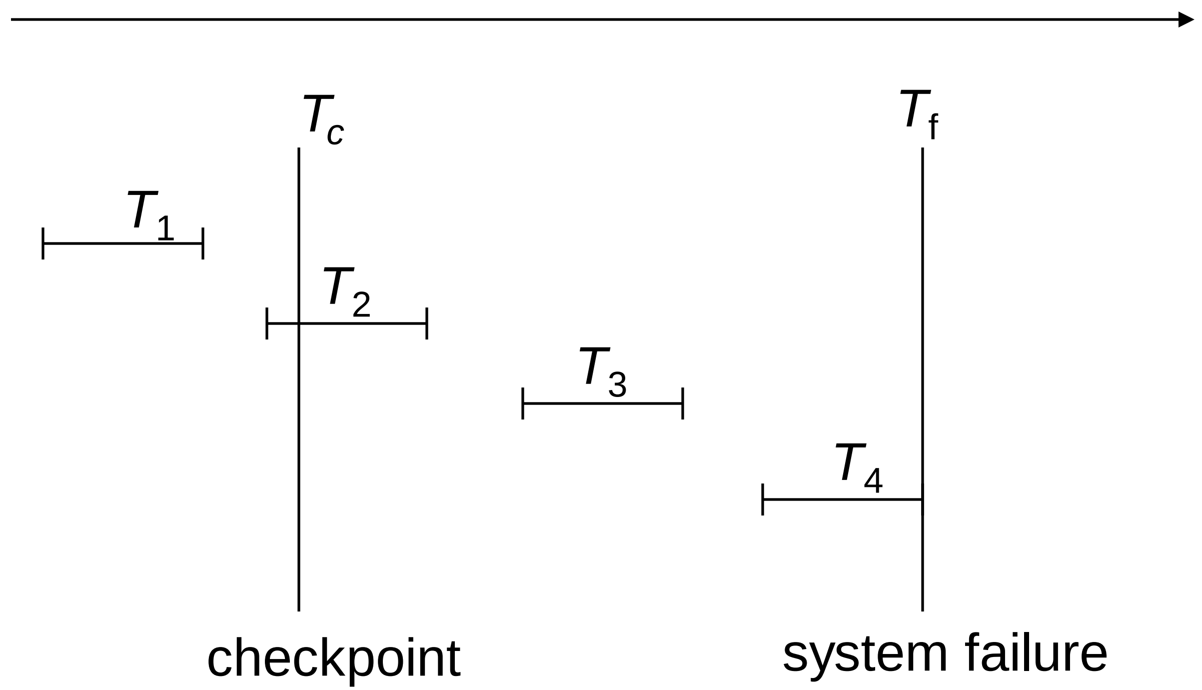
(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Checkpoints

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing checkpointing
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < checkpoint> onto stable storage.

Example of Checkpoints



- T_1 **can be ignored (updates already output to disk due to checkpoint)**
- T_2 **and** T_3 **redone.**
- T_4 **undone**

Recovery With Concurrent Transactions

- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.
  - All transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions

  <u>1)Interaction with concurrency control</u>
- We assume concurrency control using strict two-phase locking;
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
- Logging is done as described earlier.
  - Log records of different transactions may be interspersed in the log.

Recovery With Concurrent Transactions

- 2)Transaction Rollback

- We rollback a failed transaction ,Ti by using log
- System scans log backward.
- Scanning terminates when system finds<ti,start>
- Ex:<Ti,A,10,20>
-     <Tj,A,20,30>
- Backward scanning correct. result:10
- Forward scanning incorrect. result:20

Recovery With Concurrent Transactions

- <u>3)checkpoints</u>
- The checkpointing technique and actions taken on recovery have to be changed
  – since several transactions may be active when a checkpoint is performed.
- Checkpoints are performed as before, except that the checkpoint log record is now of the form

< checkpoint *L*> where *L* is the list of transactions active at the time of the checkpoint

  – We assume no updates are in progress either on biffer blocks or on log while the checkpoint is carried out (will relax this later)
  – A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.

Recovery With Concurrent Transactions (Cont.)

- 4)restart recovery
- When the system recovers from a crash, it first does the following:
  1. Initialize *undo-list* and *redo-list* to empty
  2. Scan the log backwards from the end, stopping when the first <checkpoint *L*> record is found.
     For each record found during the backward scan:
     - if the record is <T_i commit>, add T_i to *redo-list*
     - if the record is <T_i start>, then if T_i is not in *redo-list*, add T_i to *undo-list*
  3. For every T_i in *L*, if T_i is not in *redo-list*, add T_i to *undo-list*

Recovery With Concurrent Transactions

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  1. Scan log backwards from most recent record, stopping when <T_i start> records have been encountered for every T_i in *undo-list*.
     - During the scan, perform undo for each log record that belongs to a transaction in *undo-list*.
  2. Locate the most recent <checkpoint *L*> record.
  3. Scan log forwards from the <checkpoint *L*> record till the end of the log.
     - During the scan, perform redo for each log record that belongs to a transaction on *redo-list*

BUFFER MANAGEMENT

- 1.Log Record Buffering
- Log record buffering: log records are buffered in main memory, instead of of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a log force operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.

3.Operating system role in buffer management

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.
- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.

4.Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically dump the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <dump> to log on stable storage.

Recovering from Failure of Non-Volatile Storage

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as fuzzy dump or online dump

Advanced Recovery: Operation Logging (Cont.)

- If crash/rollback occurs before operation completes:
  - the operation-end log record is not found, and
  - the physical undo information is used to undo operation.
- If crash/rollback occurs after the operation completes:
  - the operation-end log record is found, and in this case
  - logical undo is performed using *U*; the physical undo information for the operation is ignored.
- <u>Redo of operation (after crash) still uses physical redo information</u>.

Advanced Recovery: Txn Rollback (Cont.)

- Scan the log backwards (cont.):
  3. If a redo-only record is found ignore it
  4. If a <T_i, O_j, operation-abort> record is found:
     - skip all preceding log records for T_i until the record <T_i, O_j, operation-begin> is found.
  5. Stop the scan when the record <T_i, start> is found
  6. Add a <T_i, abort> record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

Advanced Recovery: Crash Recovery

The following actions are taken when recovering from system crash

1. (Redo phase): Scan log forward from last < checkpoint *L*> record till end of log
   1. Repeat history by physically redoing all updates of all transactions,
   2. Create an undo-list during the scan as follows
      - *undo-list* is set to *L* initially
      - Whenever <T_i start> is found T_i is added to *undo-list*
      - Whenever <T_i commit> or <T_i abort> is found, T_i is deleted from *undo-list*

   This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

Now *undo-list* contains transactions that are incomplete, that is, have neither committed nor been fully rolled back.

Advanced Recovery: Checkpointing

- Check pointing is done as follows:
  1. Output all log records in memory to stable storage
  2. Output to disk all modified buffer blocks
  3. put to log on stable storage a < checkpoint *L*> record.

  Transactions are not allowed to perform any actions while checkpointing is in progress.

Advanced Rec: Fuzzy Checkpointing (Cont.)

- When recovering using a fuzzy checkpoint, start scan from the checkpoint record pointed to by last_checkpoint
  - Log records before last_checkpoint have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely

ARIES

- **ARIES is a state of the art recovery method**
  - **Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery**
  - **The “advanced recovery algorithm” we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations**
- **Unlike the advanced recovery algorithm, ARIES**
  1. **Uses log sequence number (LSN) to identify log records**
     - **Stores LSNs in pages to identify what updates have already been applied to a database page**
  2. **Physiological redo**
  3. **Dirty page table to avoid unnecessary redos during recovery**
  4. **Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time**

ARIES Optimizations

- Physiological redo
  - Affected page is physically identified, action within page can be logical
    - Used to reduce logging overheads
      - e.g. when a record is deleted and all other records have to be moved to fill hole
        - » Physiological redo can log just the record deletion
        - » Physical redo would require logging of old and new values for much of the page
    - Requires page to be output to disk atomically
      - Easy to achieve with hardware RAID, also supported by some disk systems
      - Incomplete page output can be detected by checksum techniques,
        - » But extra actions are required for recovery
        - » Treated as a media failure

ARIES Data Structures

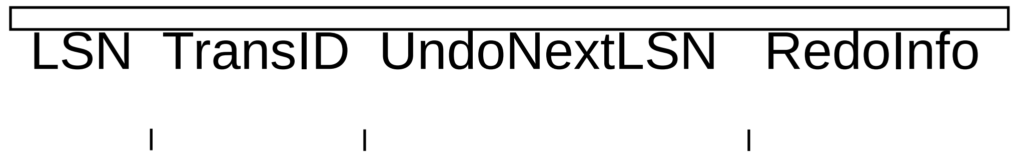
- ARIES uses several data structures
  - Log sequence number (LSN) identifies each log record
    - Must be sequentially increasing
    - Typically an offset from beginning of log file to allow fast access
      - Easily extended to handle multiple log files
  - Page LSN
  - Log records of several different types
  - Dirty page table

ARIES Data Structures: Page LSN

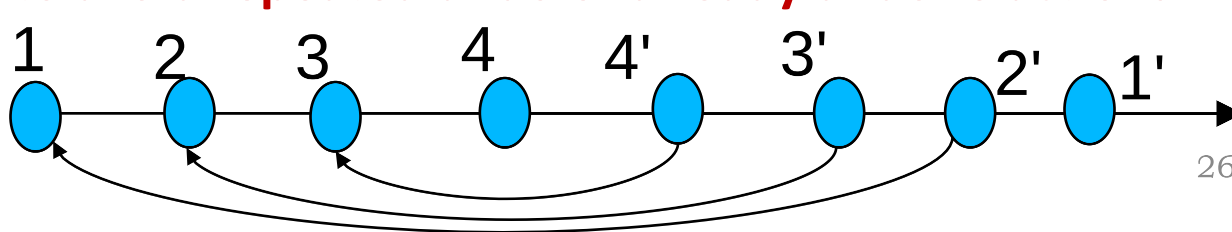
- Each page contains a PageLSN which is the LSN of the last log record whose effects are reflected on the page
  - To update a page:
    - X-latch the page, and write the log record
    - Update the page
    - Record the LSN of the log record in PageLSN
    - Unlock page
  - To flush page to disk, must first S-latch page
    - Thus page state on disk is operation consistent
      - Required to support physiological redo
  - PageLSN is used during recovery to prevent repeated redo

ARIES Data Structures: Log Record

- Each log record contains LSN of previous log record of the same transaction

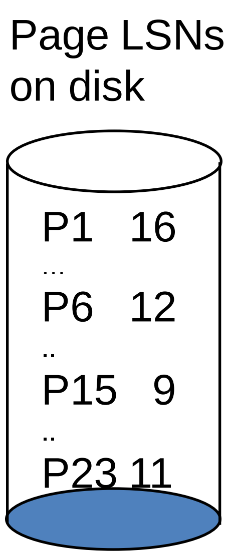


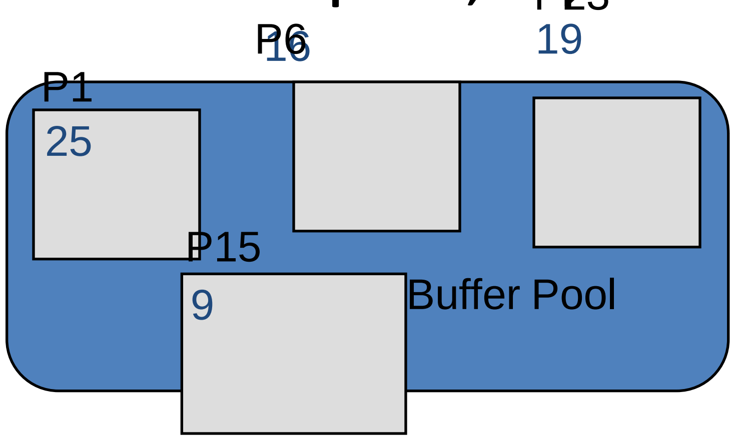
  - LSN in log record may be implicit
- Special redo-only log record called compensation log record (CLR) used to log actions taken during recovery that never need to be undone
  - Serves the role of operation-abort log records used in advanced recovery algorithm
  - Has a field UndoNextLSN to note next (earlier) record to be undone
    - Records in between would have already been undone
    - Required to avoid repeated undo of already undone actions



ARIES Data Structures: DirtyPage Table

- **DirtyPageTable**
  - **List of pages in the buffer that have been updated**
  - **Contains, for each such page**
    - **PageLSN of the page**
    - **RecLSN is an LSN such that log records before this LSN have already been applied to the page version on disk**
      - **Set to current end of log when a page is inserted into dirty page table (just before being updated)**
      - **Recorded in checkpoints, helps to minimize redo work**



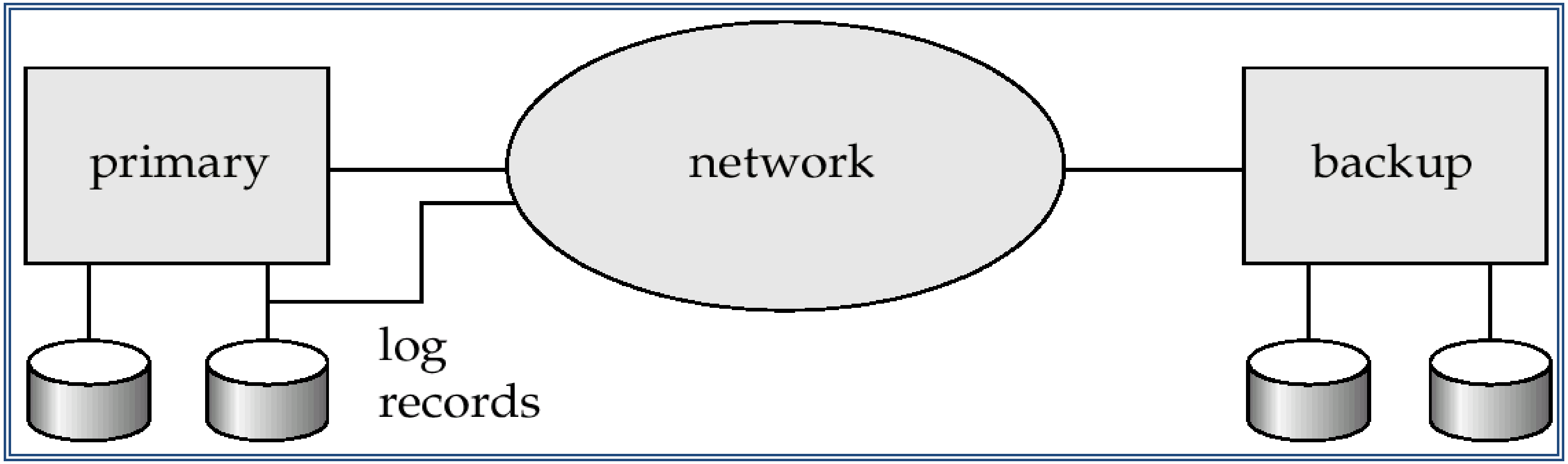


Page	PLSN	RLSN
P1	25	17
P6	16	15
P23	19	18

DirtyPage Table

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.



Remote Backup Systems (Cont.)

- **Detection of failure: Backup site must detect when primary site has failed**
  - **to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.**
  - **Heart-beat messages**
- **Transfer of control:**
  - **To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.**
    - **Thus, completed transactions are redone and incomplete transactions are rolled back.**
  - **When the backup site takes over processing it becomes the new primary**
  - **To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.**

Remote Backup Systems (Cont.)

- Time to recover: To reduce delay in takeover, backup site periodically process the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- Hot-Spare configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
- Alternative to remote backup: distributed database with replicated data

Remote Backup Systems (Cont.)

- **Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.**
- **One-safe: commit as soon as transaction's commit log record is written at primary**
  - **Problem: updates may not arrive at backup before it takes over.**
- **Two-very-safe: commit when transaction's commit log record is written at primary and backup**
  - **Reduces availability since transactions cannot commit if either site fails.**

DATABASE MANAGEMENT SYSTEMS

UNIT-V

Data on External Storage

- **<u>Disks:</u> Can retrieve random page at fixed cost**
  - **But reading several consecutive pages is much cheaper than reading them in random order**
- **<u>Tapes:</u> Can only read pages in sequence**
  - **Cheaper than disks; used for archival storage**
- **<u>File organization:</u> Method of arranging a file of records on external storage.**
  - **Record id (rid) is sufficient to physically locate record**
  - **Indexes are data structures that allow us to find the record ids of records with given values in index search key fields**
- **<u>Architecture:</u> Buffer manager stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager**.

Alternative File Organizations

Many alternatives exist, *each ideal for some situations, and not so good in others:*

- **<u>Heap (random order) files:</u> Suitable when typical access is a file scan retrieving all records.**
- **<u>Sorted Files:</u> Best if records must be retrieved in some order, or only a `range' of records is needed.**
- **<u>Indexes:</u> Data structures to organize records via trees or hashing.**
  - **Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields**
  - **Updates are much faster than in sorted files.**

Alternatives for Data Entry k* in Index

- In a data entry k* we can store:
  - Data record with key value **k,** or
  - <**k**, rid of data record with search key value **k**>, or
  - <**k**, list of rids of data records with search key **k**>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - **Examples of indexing techniques: B+ trees, hash-based structures**
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)

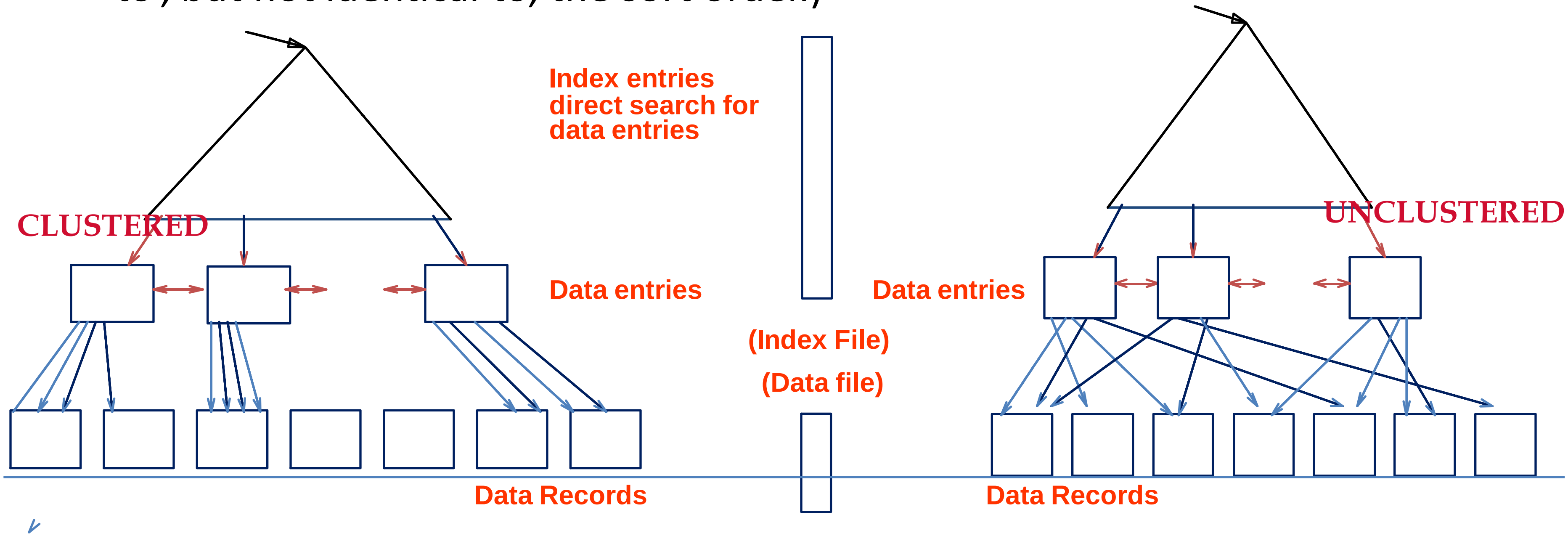
- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
  - If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternatives for Data Entries (Contd.)

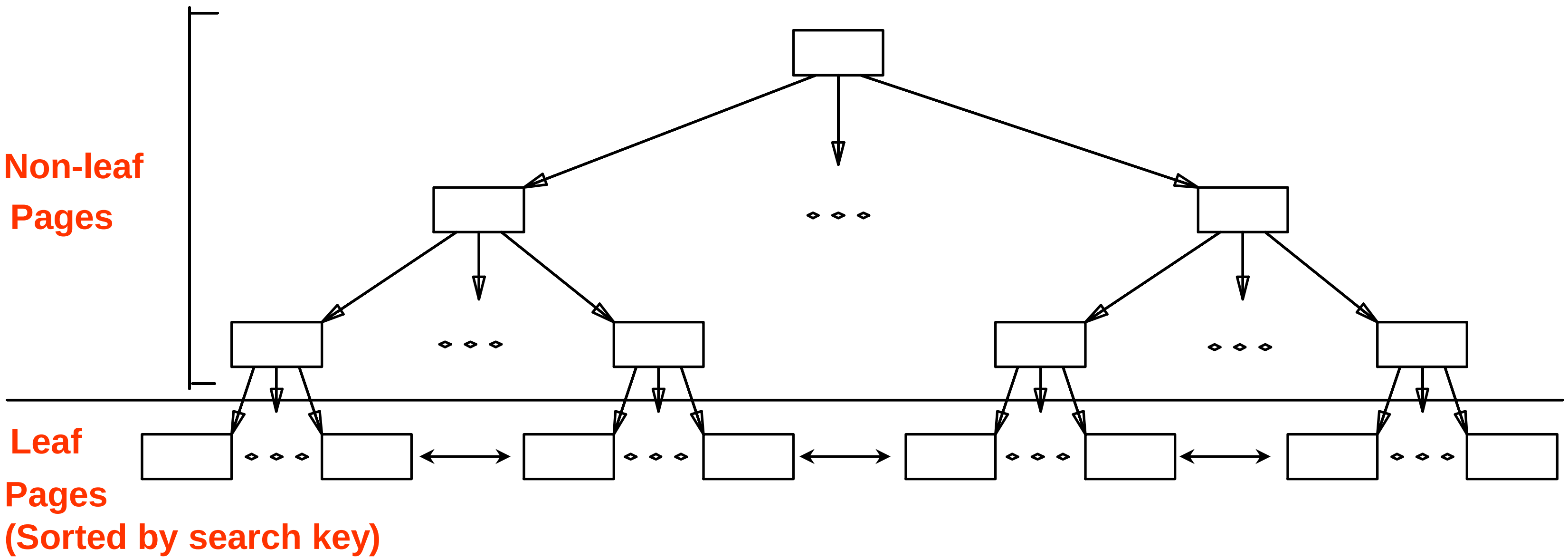
- Alternatives 2 and 3:
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Clustered vs. Unclustered Index

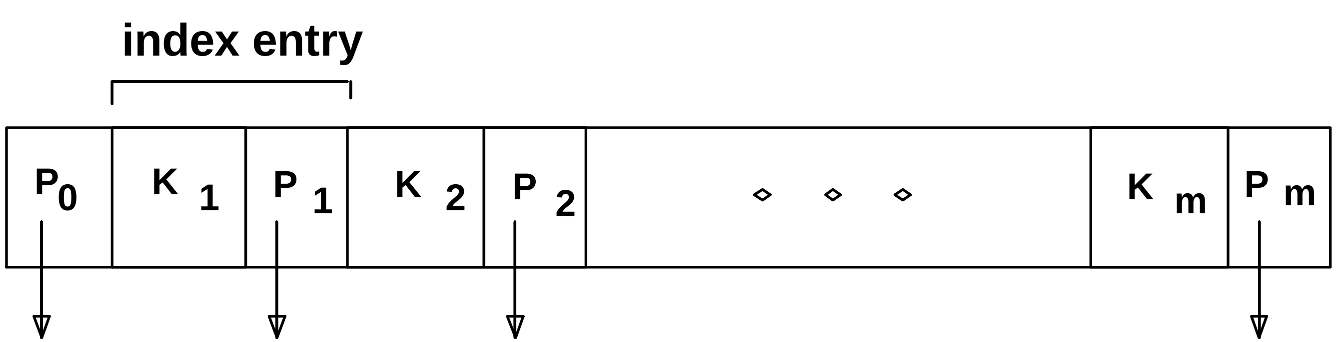
- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



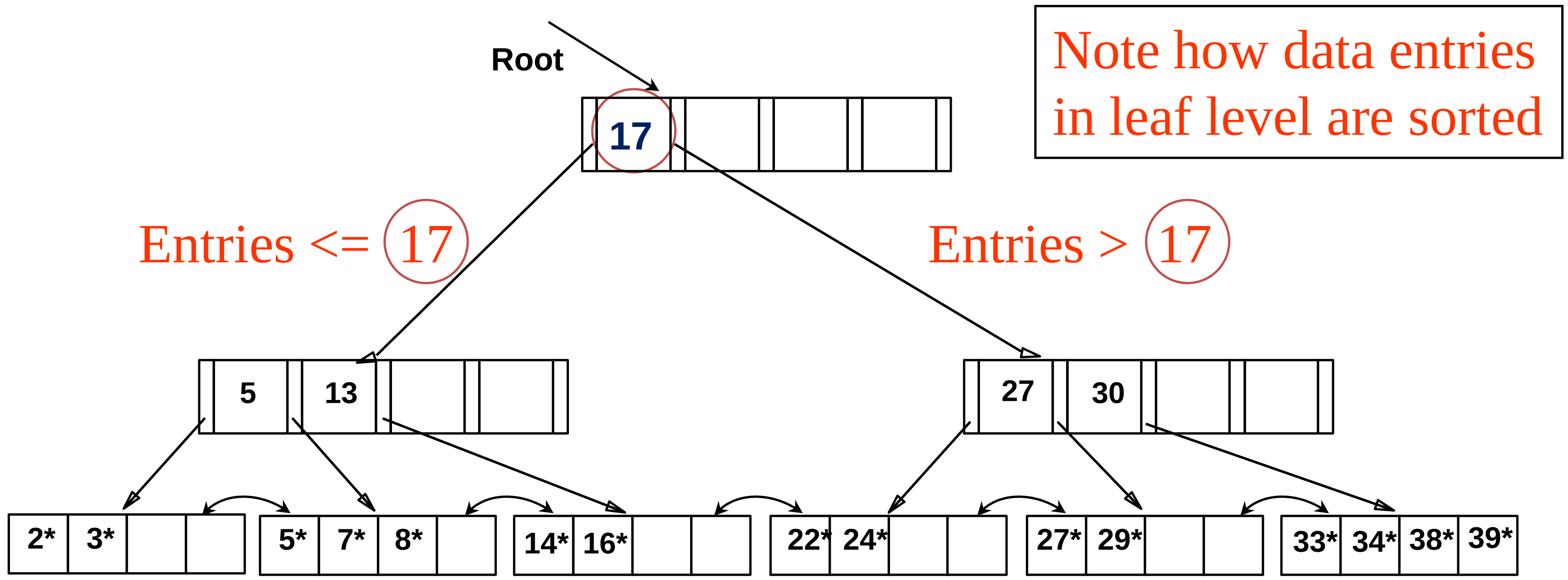
B+ Tree Indexes



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages have *index entries;* only used to direct searches:



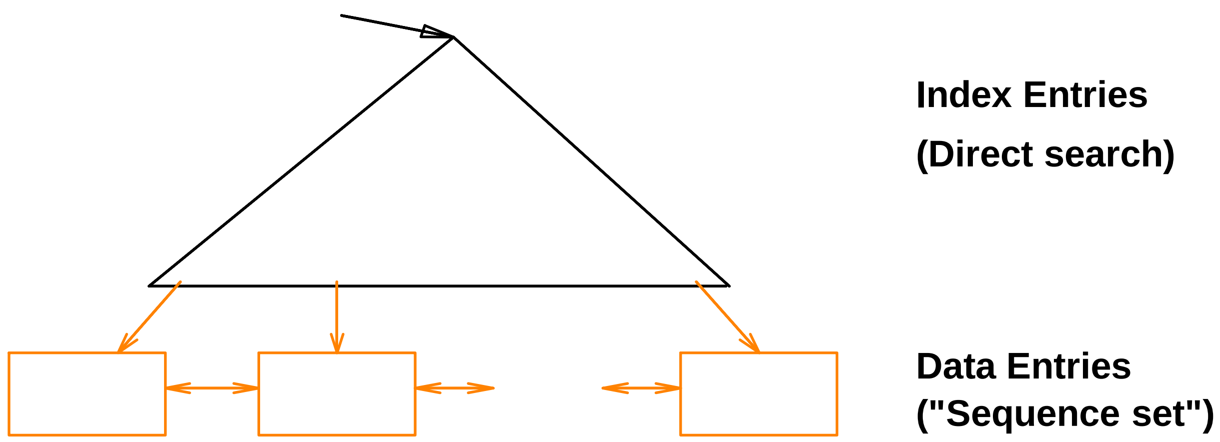
Example B+ Tree



- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

B+ Tree: Most Widely Used Index

- Insert/delete at log $_F$ N cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.



B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MByteS

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.

Comparing File Organizations

- Heap files (random order; insert at eof)
- Sorted files, sorted on *<age, sal>*
- Clustered B+ tree file, Alternative (1), search key *<age, sal>*
- Heap file with unclustered B + tree index on search key *<age, sal>*
- Heap file with unclustered hash index on search key *<age, sal>*

Operations to Compare

- Scan: Fetch all records from disk
- Equality search
- Range selection
- Insert a record
- Delete a record

Assumptions (contd.)

- Scans:
  - Leaf levels of a tree-index are chained.
  - Index data-entries plus actual file scanned for unclustered indexes.
- Range searches:
  - We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	D(log 2 B + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	D(log F 1.5B + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	D(1 + log F 0.15B)	D(log F 0.15B + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

Clustered index organization

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!

Examples of Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?
- Consider the GROUP BY query.
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!
- Equality queries and duplicates:
  - Clustering on *E.hobby* helps!

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

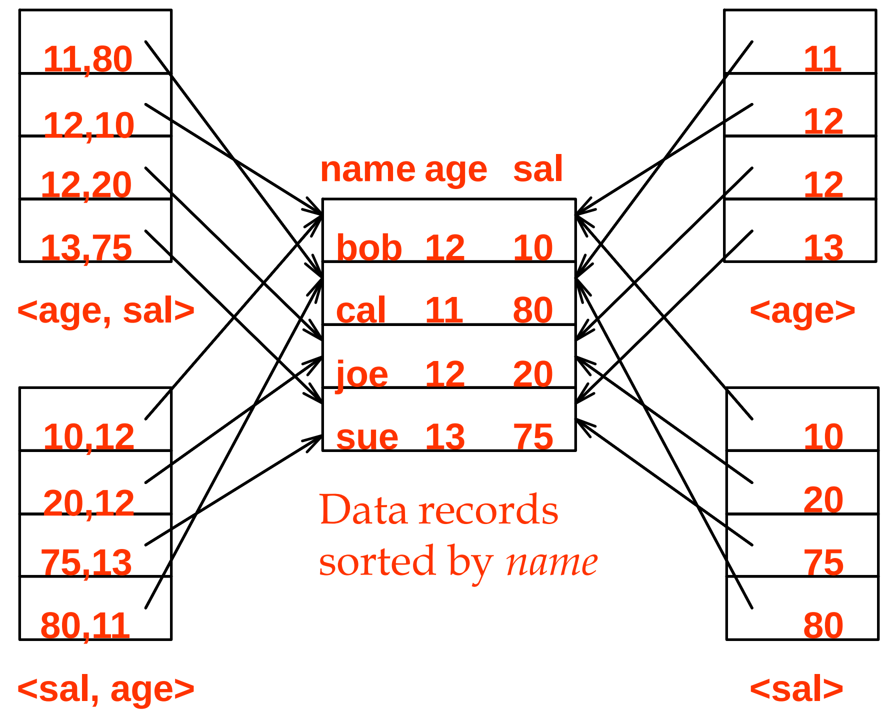
```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

Indexes with Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.
  - Lexicographic order, or
  - Spatial order.

Examples of composite key indexes using lexicographic order.



Data records sorted by *name*

Data entries in index sorted by <*sal,age*>

Data entries sorted by <*sal*>

tradeoffs

- A composite key index can support a broader range of queries bcoz it matches more selection conditions
- Index only evaluation strategies are increased
- Disadv:a composite index must be updated in response to any operation(insert,delete or update) that modifies any field in search key.
- A composite index is also likely to be larger than single attribute search key
- For B+ tree index this increases no. of levels

Composite Search Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
  - Choice of index key orthogonal to clustering etc.
- If condition is: 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often.

Composite keys-Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

<E.dno>

```
SELECT E.dno, COUNT(*)
FROM Emp E
GROUP BY E.dno
```

<E.dno,E.sal>

Tree index!

```
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno
```

<E. age,E.sal>
or
<E.sal, E.age>

Tree index!

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
 E.sal BETWEEN 3000 AND 5000
```

Creating index in sql

<u>Syntax</u>

Create index indexname on tablename

with structure=Btree,

key=(age,sal);

Summary

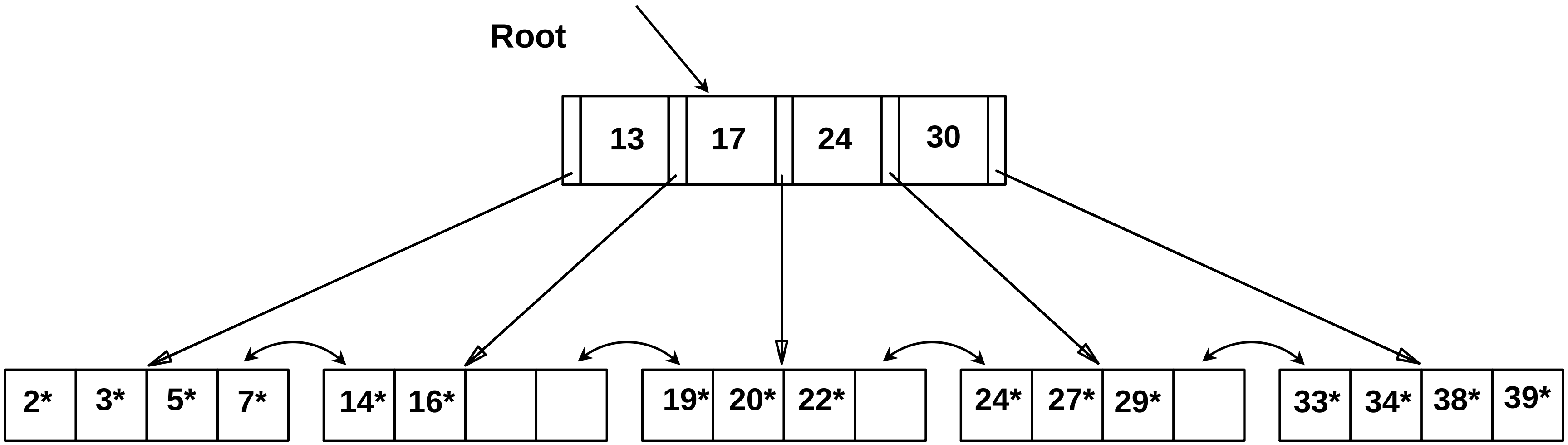
- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values.

Summary (Contd.)

- Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.
  - Choice orthogonal to *indexing technique* used to locate data entries with a given key value.
- Can have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.

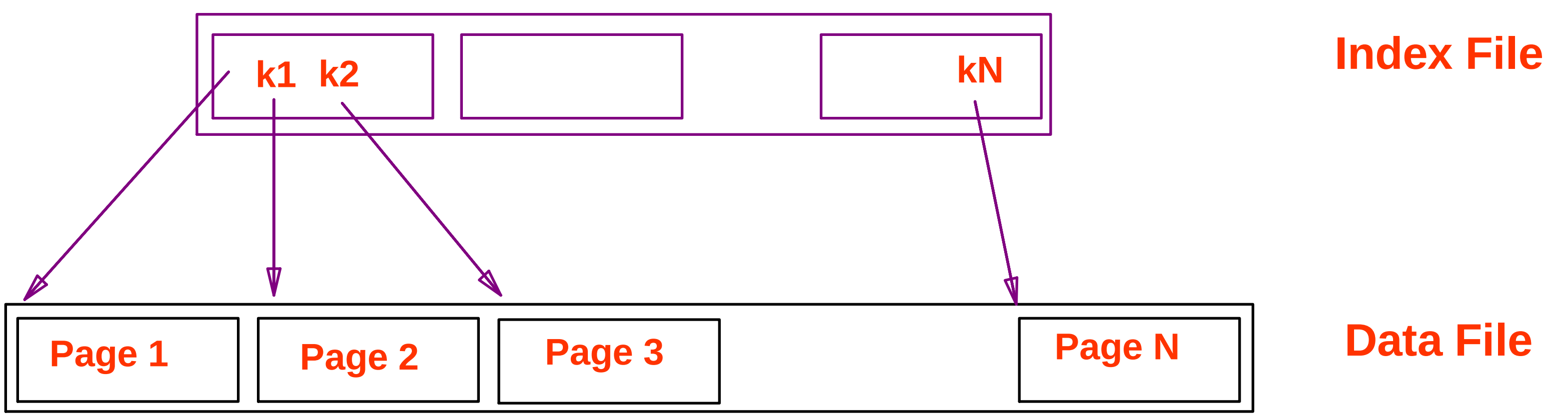
Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...

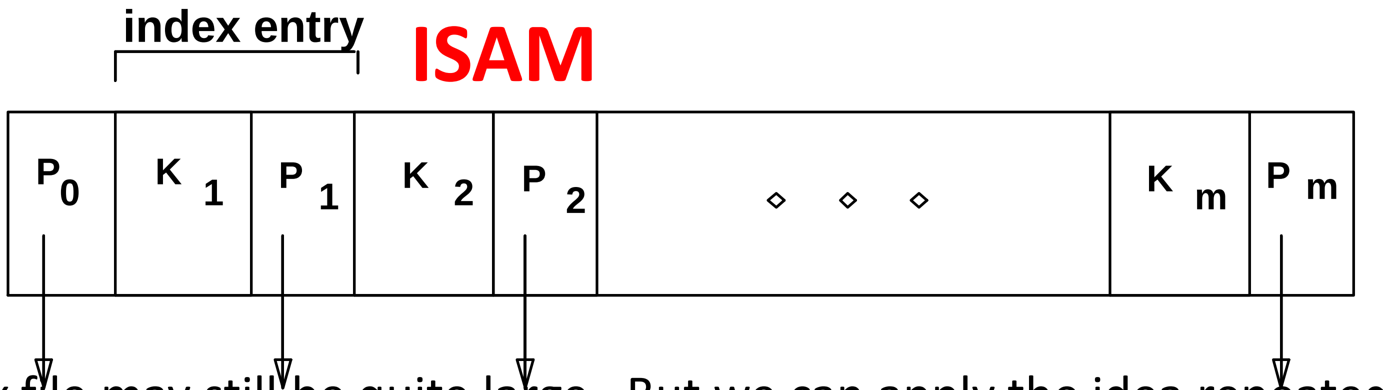


Range Searches

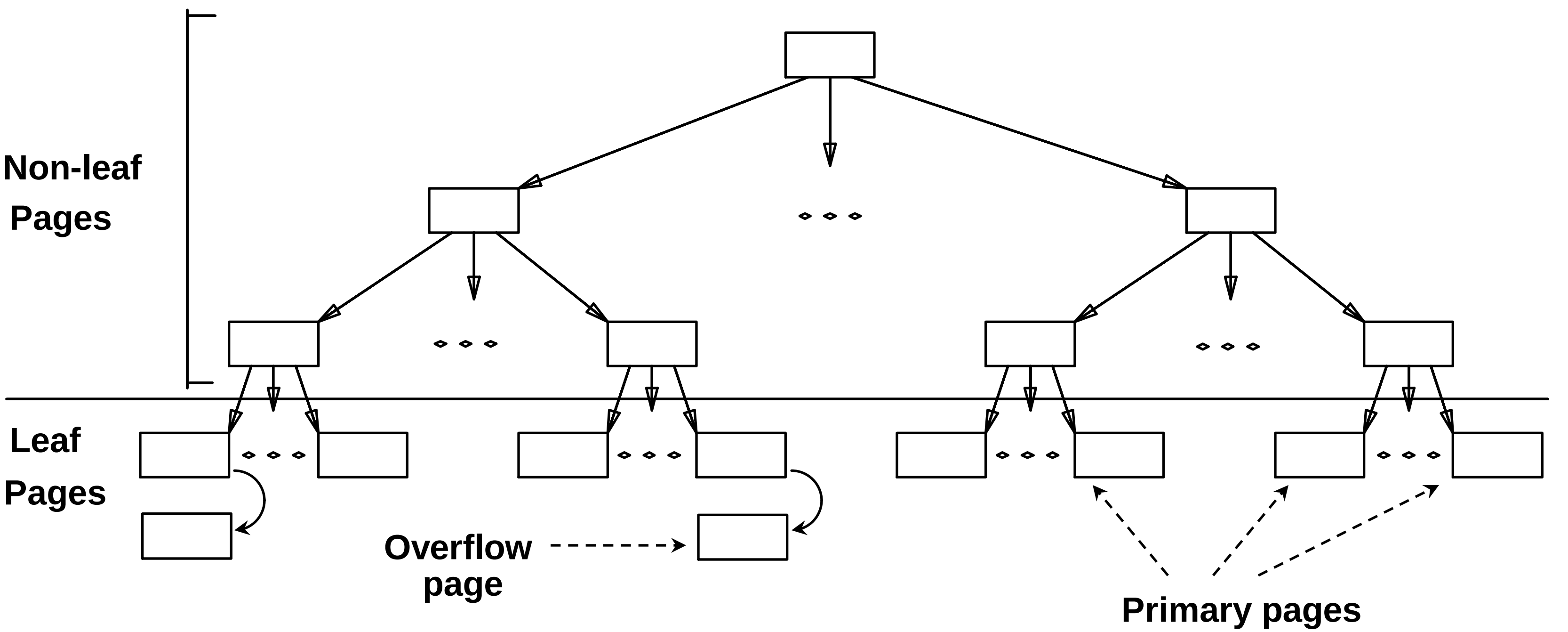
- ``*Find all students with gpa > 3.0*''
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.



ISAM

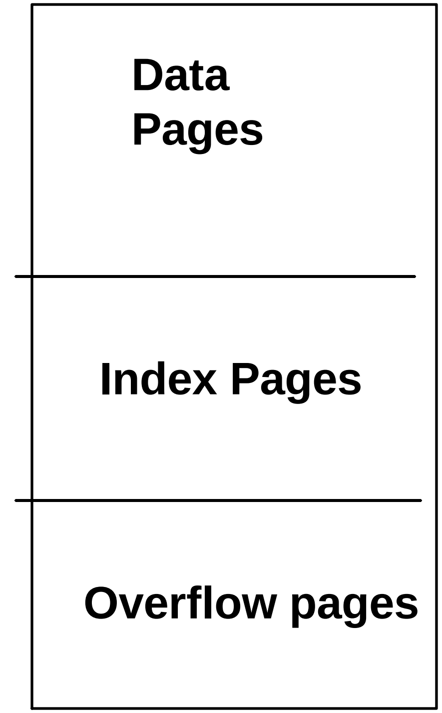


- Index file may still be quite large. But we can apply the idea repeatedly!



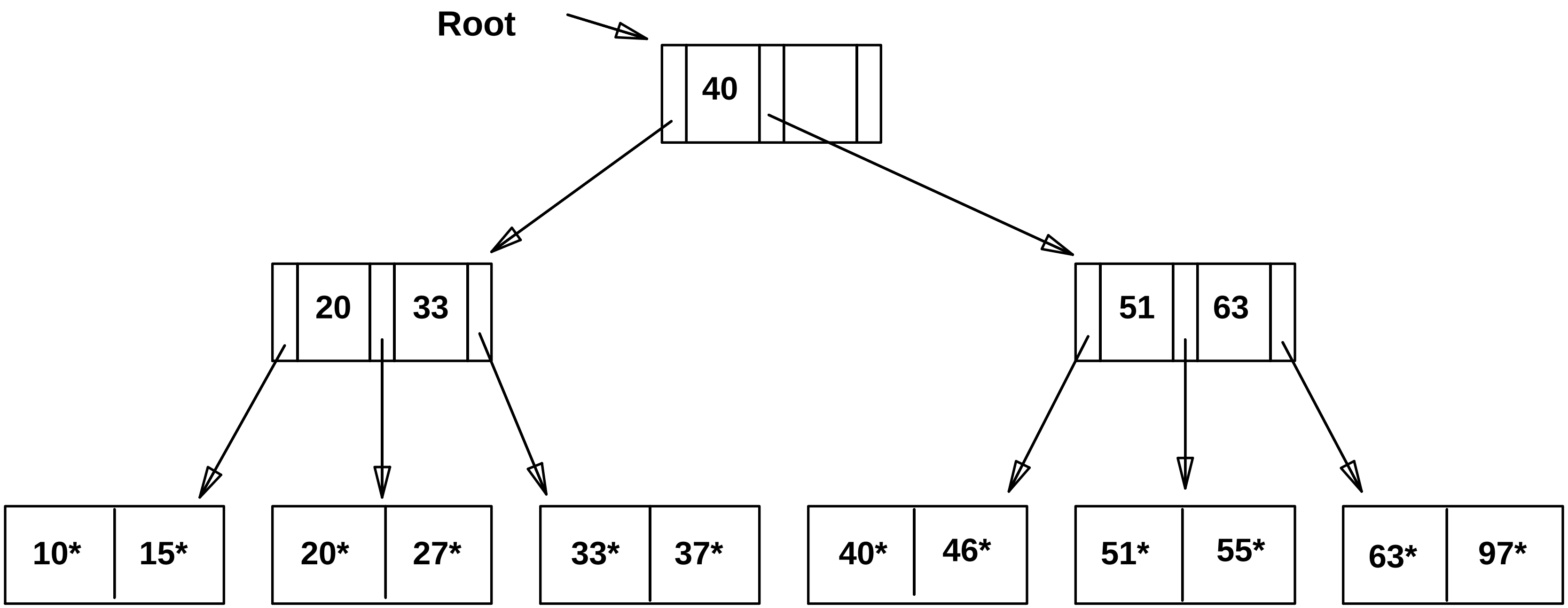
Comments on ISAM

- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\log_F N$; F = # entries/index pg, N = # leaf pgs
- *Insert*: Find leaf data entry belongs to, and put it there.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

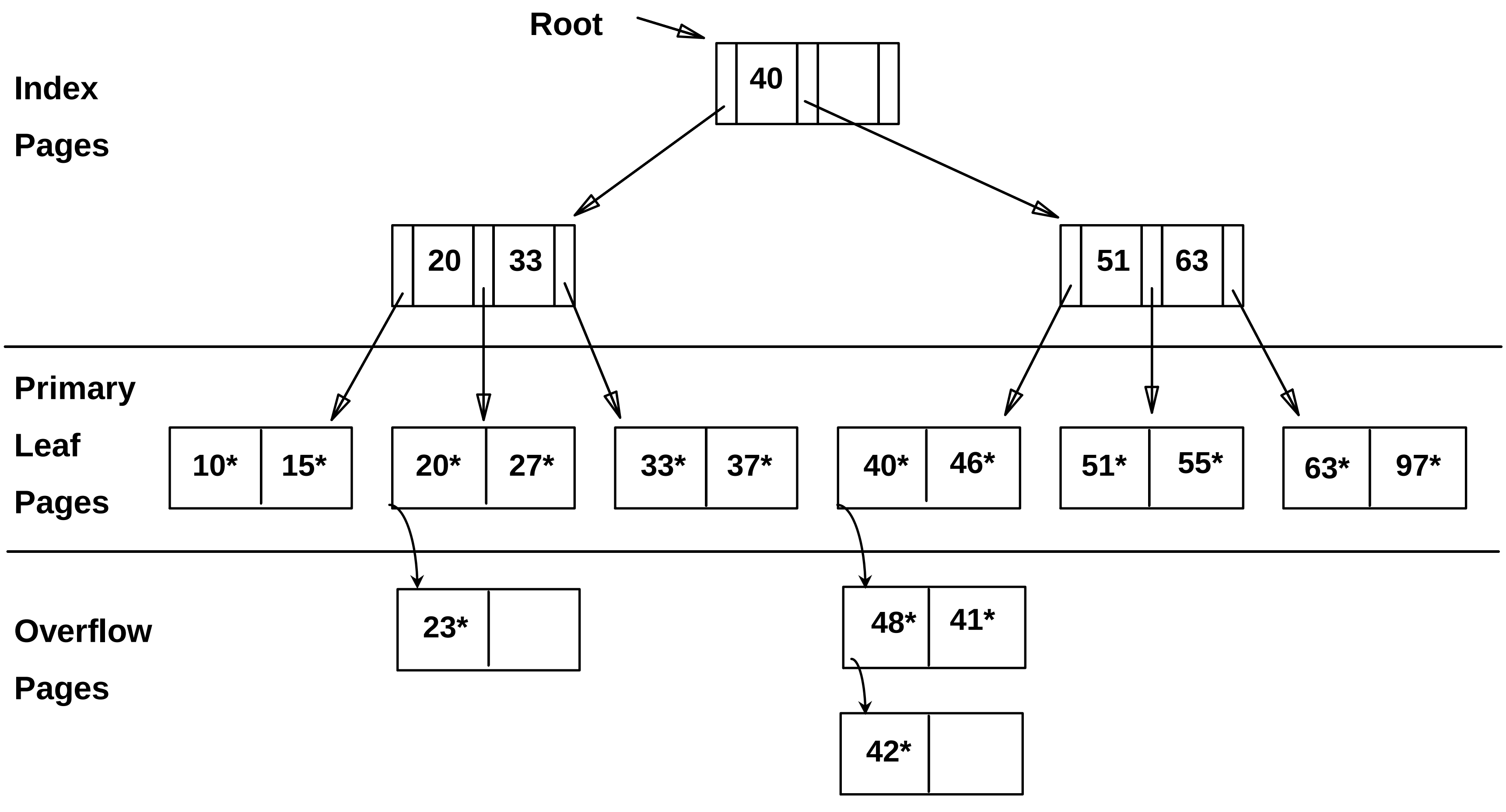


Example ISAM Tree

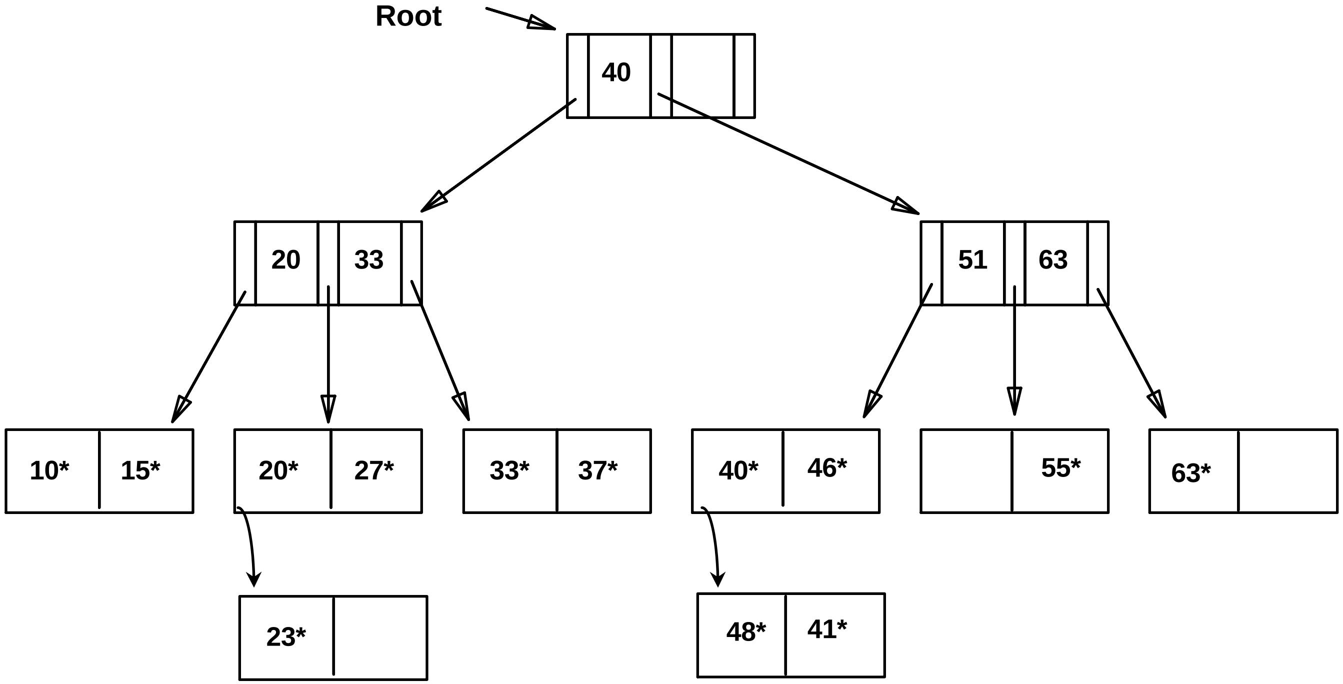
- Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)



After Inserting 23*, 48*, 41*, 42* ...

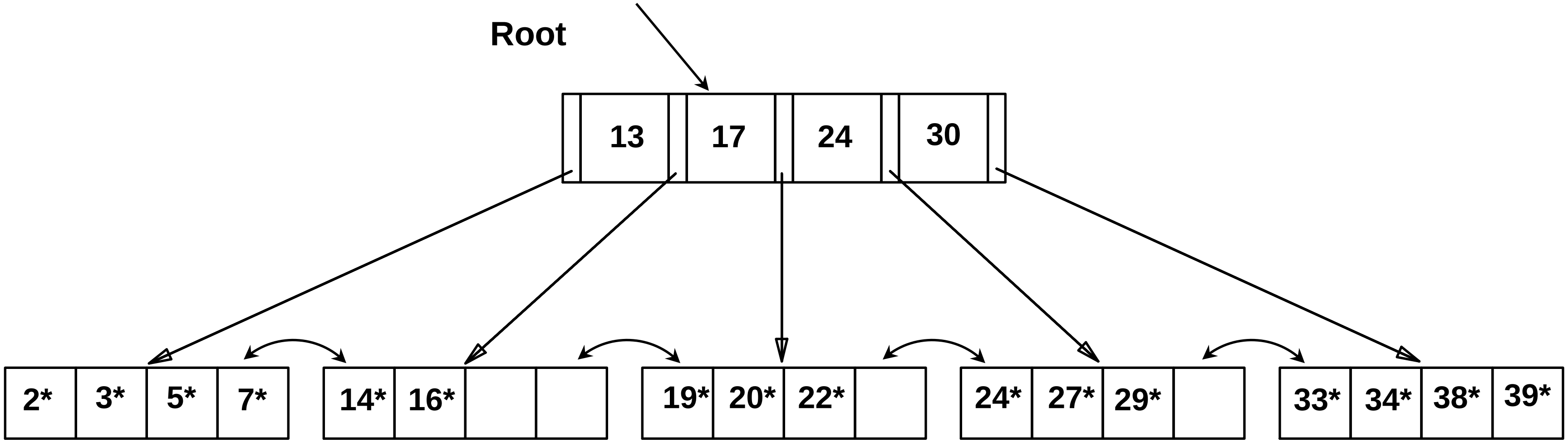


Then Deleting 42*, 51*, 97*



Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...

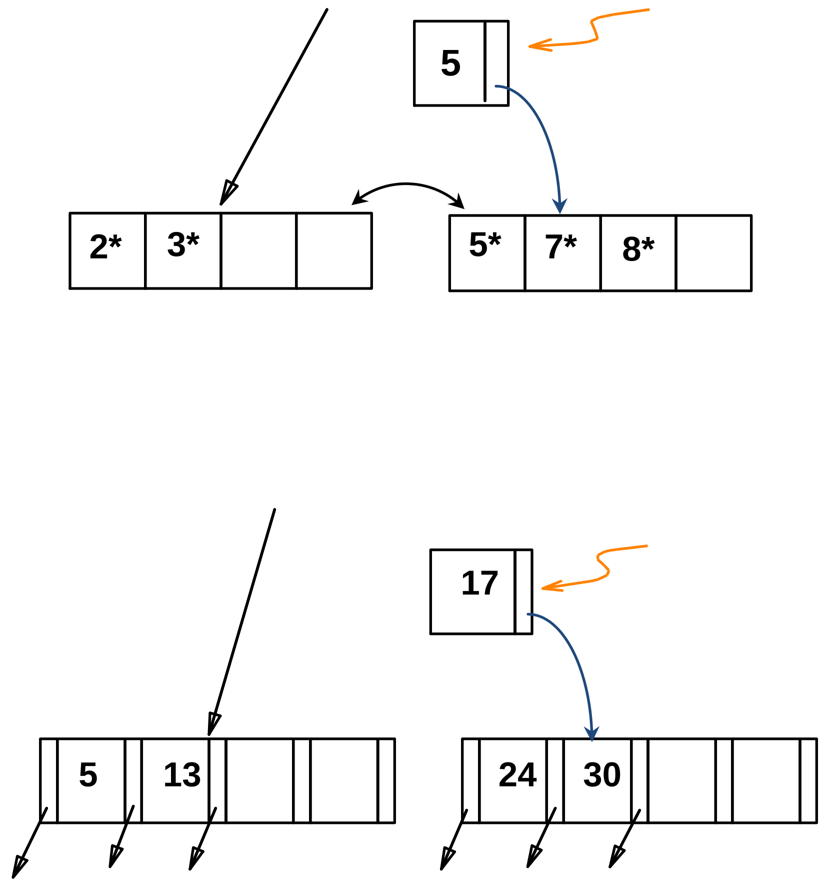


Inserting a Data Entry into a B+ Tree

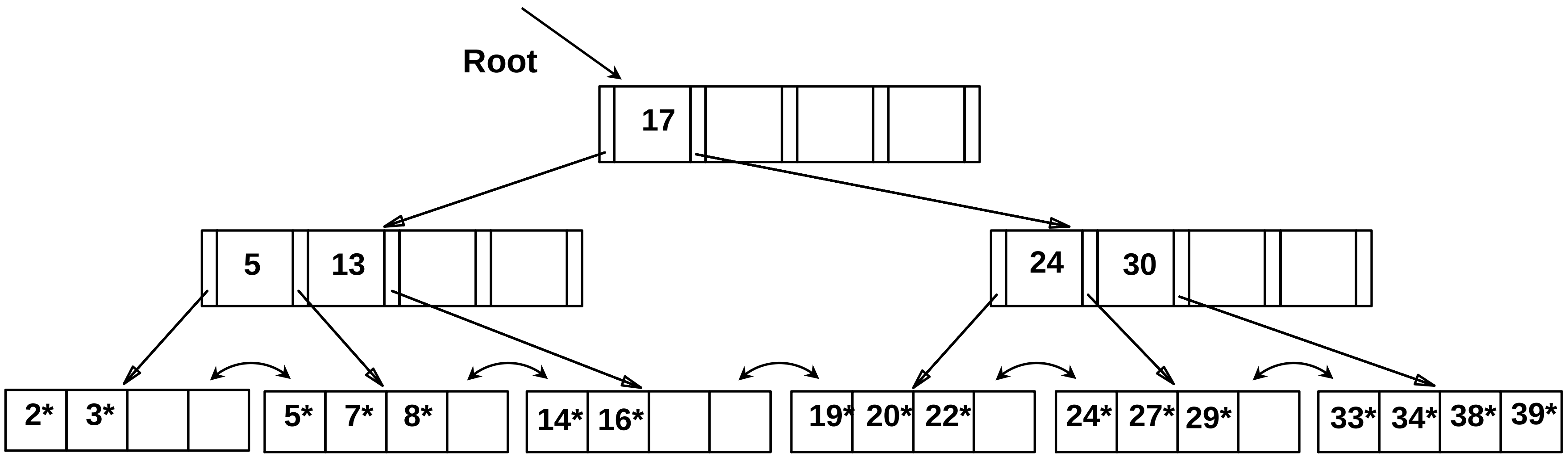
- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *<u>split</u>* *L (into L and a new node L2)*
    - Redistribute entries evenly, <u>copy up</u> middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but <u>push up</u> middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *<u>wider</u>* or *<u>one level taller at top.</u>*

Inserting 8* into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.



Example B+ Tree After Inserting 8*



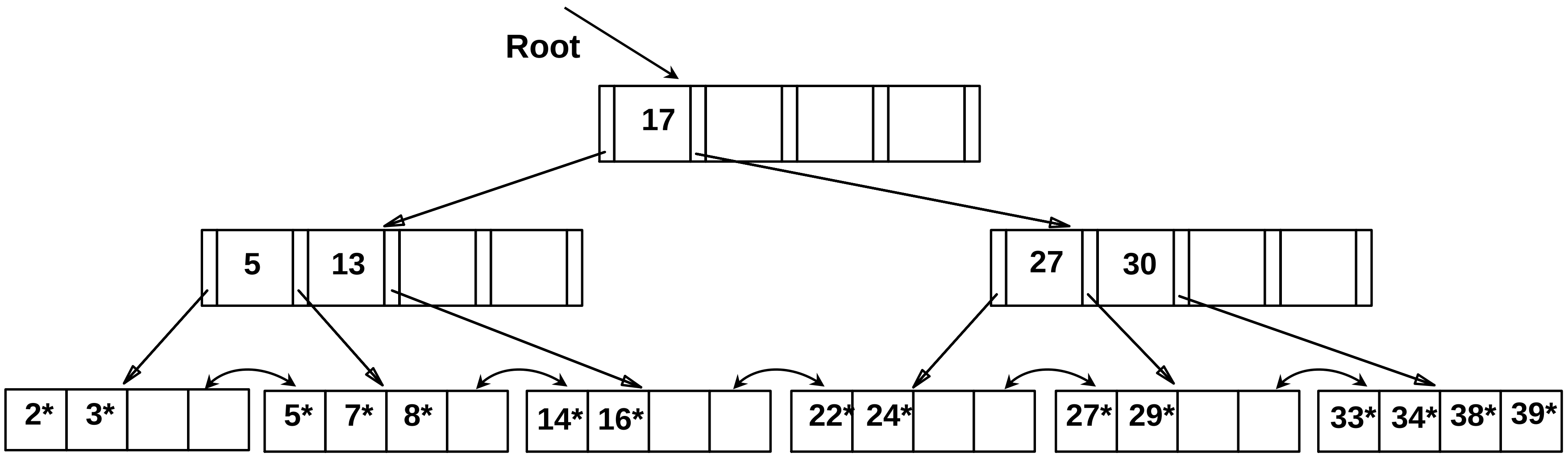
❖ Notice that root was split, leading to increase in height.

❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only d-1 entries,
    - Try to re-distribute, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

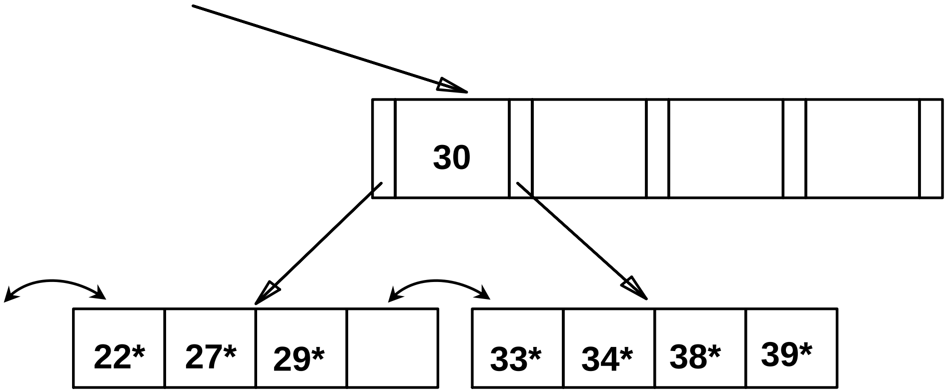
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

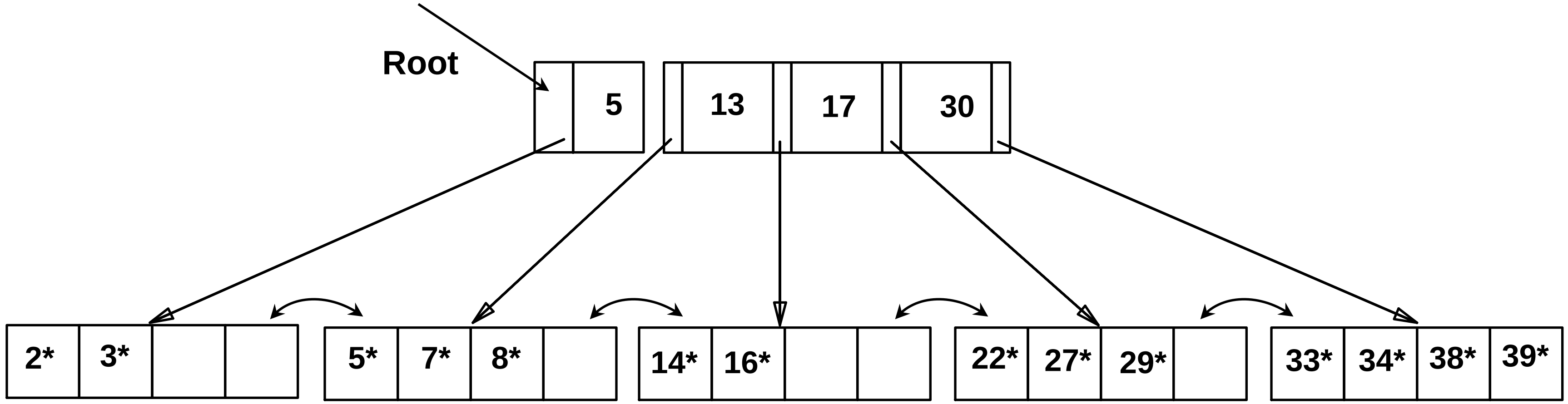


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

And Then Deleting 24*

- Must merge.
- Observe `toss' of index entry (on right), and `pull down' of index entry (below)





First Normal Form

- **1NF (First Normal Form)**
  - **a relation R is in 1NF if and only if it has only single-valued attributes (atomic values)**
  - **EMP_PROJ (<u>SSN</u>, <u>PNO</u>, HOURS, ENAME, PNAME, PLOCATION)**

    PLOCATION is not in 1NF (multi-valued attrib.)
  - **solution: decompose the relation**

    EMP_PROJ2 (<u>SSN</u>, <u>PNO</u>, HOURS, ENAME, PNAME)

    LOC (<u>PNO</u>, <u>PLOCATION</u>)

Second Normal Form

- **2NF (Second Normal Form)**
  - **a relation R in 2NF if and only if it is in 1NF and every nonkey column depends on a key not a subset of a key**
  - **all nonprime attributes of R must be fully functionally dependent on a whole key(s) of the relation, not a part of the key**
  - **no violation: single-attribute key or no nonprime attribute**

Second Normal Form (Contd)

- 2NF (Second Normal Form)
  - violation: part of a key $\rightarrow$ nonkey

    EMP_PROJ2 (<u>SSN</u>, <u>PNO</u>, HOURS, ENAME, PNAME)

    SSN $\rightarrow$ ENAME

    PNO $\rightarrow$ PNAME
  - solution: decompose the relation

    EMP_PROJ3 (<u>SSN</u>, <u>PNO</u>, HOURS)

    EMP (<u>SSN</u>, ENAME)

    PROJ (<u>PNO</u>, PNAME)

Third Normal Form

- **a relation R in 3NF if and only if it is in 2NF and every nonkey column does not depend on another nonkey column**
- **all nonprime attributes of R must be non-transitively functionally dependent on a key of the relation**

Third Normal Form (Contd)

- 3NF (Third Normal Form)
  - **violation: nonkey → nonkey**

- **SUPPLIER (<u>SNAME</u>, STREET, CITY, STATE, TAX)**

  SNAME → STREET, CITY, STATE

  STATE → TAX (nonkey → nonkey)

  SNAME → STATE → TAX (transitive FD)

- solution: decompose the relation

  SUPPLIER2 (<u>SNAME</u>, STREET, CITY, STATE)

  TAXINFO (<u>STATE</u>, TAX)

Boyce-Codd Normal Form (BCNF)

- **Reln R with FDs *F* is in BCNF if, for all $X \rightarrow A$ in F^+**
  - A $\in$ X (called a *trivial* FD), or
  - X contains a key for R.
- **In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.**
  - **No dependency in R that can be predicted using FDs alone.**
  - **If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.**
  - **If example relation is in BCNF, the 2 tuples**

X	Y	A
x	y1	a
x	y2	?

Decomposition of a Relation Scheme

- **Suppose that relation R contains attributes *A1 ... An.* A *decomposition* of R consists of replacing R by two or more relations such that:**
  - **Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and**
  - **Every attribute of R appears as an attribute of one of the new relations.**
- **Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.**
- **E.g., Can decompose SNLRWH into SNLRH and RW.**

Example Decomposition

- **Decompositions should be used only when needed.**
  - **SNLRWH has FDs S → SNLRWH and R → W**
  - **Second FD causes violation of 3NF; W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations, and to remove W from the main schema:**
    - **i.e., we decompose SNLRWH into SNLRH and RW**
- **The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?**

Problems with Decompositions

- **There are three potential problems to consider:**
  - **Some queries become more expensive.**
    - **e.g., How much did sailor Joe earn? (salary = W*H)**
  - **Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!**
    - **Fortunately, not in the SNLRWH example.**
  - **Checking some dependencies may require joining the instances of the decomposed relations.**
    - **Fortunately, not in the SNLRWH example.**
- ***Tradeoff*: Must consider these issues vs.**

Lossless Join Decompositions

- Decomposition of R into X and Y is ***lossless-join*** w.r.t. a set of FDs F if, for every instance *r* that satisfies F:
- $\pi_X(r) \bowtie \pi_Y(r) = r$
- It is always true that $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$
  - In general, the other direction does not hold! If it does, the decomposition is lossless-join.
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- ***It is essential that all decompositions used to deal with redundancy be lossless!***
- ***Consider Hourly emps relation.It has attributes SNLRWH and FD R->W causes a violation of 3NF.We dealt this violation by decomposing into SNLRH and RW.***
- ***Since R is common to both decomposed relation and R->W holds,this decomposition is lossles-join***

More on Lossless Join

- The decomposition of R into X and Y is lossless-join wrt F if and only if the closure of F contains: →
  - $X \cap Y \rightarrow X$, or
  - X Y Y

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

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

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

- **In particular,if an fd X->Y holds over relation R and X∩ Y is empty, the decomposition of R into R-Y and XY is lossless.**
- **Imp observation is repeated decompositions**

Dependency Preserving Decomposition

- Consider CSJDPQV, C is key, JP $\rightarrow$ C and SD $\rightarrow$ P.
  - **Bcoz SD->P is not a key,it causes violation**
  - BCNF decomposition: CSJDQV and SDP
  - Problem: Checking JP $\rightarrow$ C requires a join!
- Dependency preserving decomposition (Intuitive):
  - **If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold.**
  - ***Projection of set of FDs F***: **If R is decomposed into X, ... projection of F onto X (denoted F_X) is the set of FDs U $\rightarrow$ V in F^+ *(closure of F)* such that U, V are in X.**

Dependency Preserving Decompositions (Contd.)

- **Decomposition of R into X and Y is *dependency preserving* if $(F_X \text{ union } F_Y)^+ = F^+$**
  - i.e., **if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.**
- Important to consider F^+, not F, in this definition:
  - **ABC, $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, decomposed into AB and BC.**
  - **Is this dependency preserving? Is $C \rightarrow A$ preserved?????**
- **Dependency preserving does not imply lossless join:**
  - **ABC, $A \rightarrow B$, decomposed into AB and BC.**
- And vice-versa! (Example?)

Decomposition into BCNF

- Consider relation R with FDs F. If X $\rightarrow$ Y violates BCNF, decompose R into R - Y and XY.
  - Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.
  - e.g., CSJDPQV, key C, JP $\rightarrow$ C, SD $\rightarrow$ P, J $\rightarrow$ S
  - To deal with SD $\rightarrow$ P, decompose into SDP, CSJDQV.
  - To deal with J $\rightarrow$ S, decompose CSJDQV into JS and CJDQV
- In general, several dependencies may cause violation of BCNF. The order in which we ``deal with'' them could lead to very different sets of relations!

BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.
  - e.g., CSZ, $CS \rightarrow Z$, $Z \rightarrow C$
  - Can't decompose while preserving 1st FD; not in BCNF.
- Similarly, decomposition of CSJDQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs $JP \rightarrow C$, $SD \rightarrow P$ and $J \rightarrow S$).
  - However, it is a lossless join decomposition.
  - In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.

Decomposition into 3NF

- Obviously, the algorithm for lossless join decomp into BCNF can be used to obtain a lossless join decomp into 3NF (typically, can stop earlier).
- To ensure dependency preservation, one idea:
  - If X → Y is not preserved, add relation XY.
  - Problem is that XY may violate 3NF! e.g., consider the addition of CJP to `preserve' JP → C. What if we also have J → C ?
- Refinement: Instead of the given set of FDs F, use a *minimal cover for F.*

Constraints on an Entity Set

- Consider the Hourly Emps relation again. The constraint that attribute *ssn* is a key can be expressed as an FD:
- *{ ssn }-> { ssn, name, lot, rating, hourly wages, hours worked}*
- For brevity, we will write this FD as *S -> SNLRWH*, using a single letter to denote each attribute
- In addition, the constraint that the *hourly wages* attribute is determined by the *rating* attribute is an

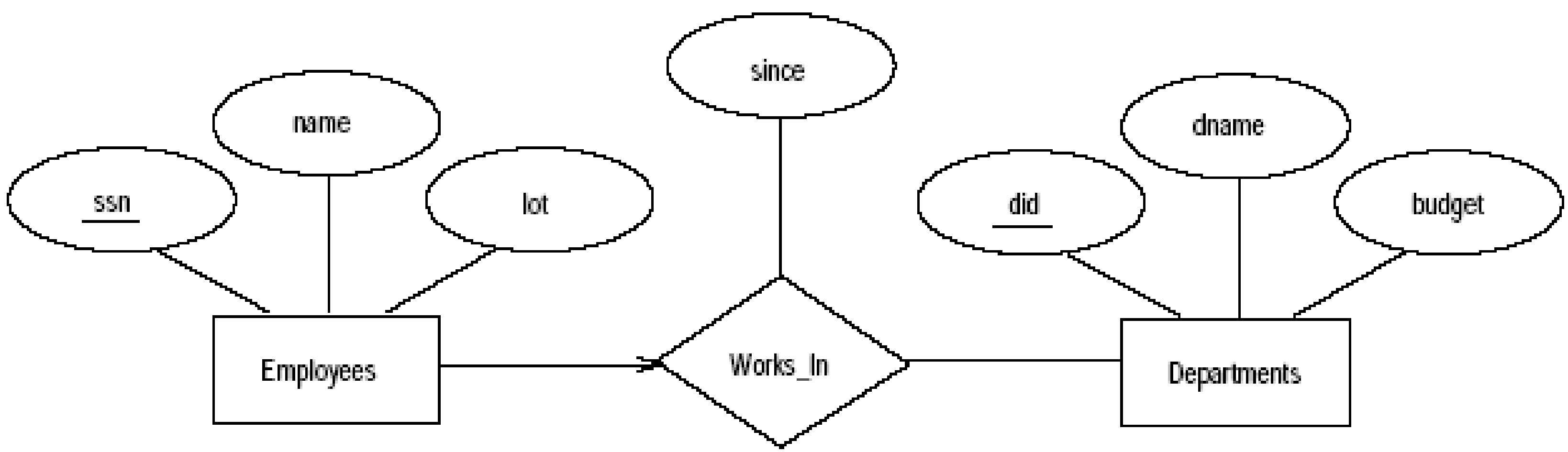
  FD: *R -> W*.

Constraints on a Relationship Set

- The previous example illustrated how FDs can help to rene the subjective decisions made during ER design,
- but one could argue that the best possible ER diagram would have led to the same nal set of relations.
- Our next example shows how FD information can lead to a set of relations that eliminates some redundancy problems and is unlikely to be arrived at solely through ER design.

Identifying Attributes of Entities

- in particular, it shows that attributes can easily be associated with the `wrong' entity set during ER design.
- The ER diagram shows a relationship set called Works In that is similar to the Works In
-
-



Identifying Entity Sets

- Let Reserves contain attributes *S*, *B*, and *D* as before, indicating that sailor *S* has a reservation for boat *B* on day *D*.
- In addition, let there be an attribute *C* denoting the credit card to which the reservation is charged.
- Suppose that every sailor uses a unique credit card for reservations. This constraint is expressed by the FD *S* -> *C*. This constraint indicates that in relation Reserves, we store the credit card number for a sailor as often as we have reservations for that
- sailor, and we have redundancy and potential update anomalies.

Multivalued Dependencies

- Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*.
- The meaning of a tuple is that teacher *T* can teach course *C*, and book *B* is a recommended text for the course.
- There are no FDs; the key is *CTB*. However, the recommer
independe

course	*teacher*	*book*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

There are three points to note here:

- The relation schema *CTB* is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over *CTB*.
- There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB*.
- Let *R* be a relation schema and let *X* and *Y* be subsets of the attributes of *R*. Intuitively,
- the **multivalued dependency** *X !! Y* is said to hold over *R* if, in every legal

- The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be epressed in terms of FDs.
- This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation using two binary relationship sets, Instructors with attributes *CT* and Text with attributes *CB*.
- Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate.

- Three of the additional rules involve only MVDs:
- **MVD Complementation:** If $X \to\to Y$, then $X \to\to R - XY$
- **MVD Augmentation:** If $X \to\to Y$ and $W > Z$, then $WX \to\to YZ$.
- **MVD Transitivity:** If $X \to\to Y$ and $Y \to\to Z$, then $X \to\to (Z - Y)$.
- **Fourth Normal Form**
- R is said to be in **fourth normal form (4NF)** if for every MVD $X \to\to Y$ that holds over R, one of the following statements is true:
- *Y subset of* X or $XY = R$, or
- X is a superkey.

Join Dependencies

- A join dependency is a further generalization of MVDs. A **join dependency** (JD) ∞{ *R*1,….. *Rn }* is said to hold over a relation *R* if *R*1*,….* *Rn* is a lossless-join decomposition of *R*.
- An MVD *X ->-> Y* over a relation *R* can be expressed as the join dependency ∞ { XY,X(R−Y)*}*
- As an example, in the *CTB* relation, the MVD *C ->->T* can be expressed as the join dependency ∞{ CT, CB*}*
- Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

Fifth Normal Form

- A relation schema *R* is said to be in **fth normal form (5NF)** if for every JD ∞{ *R*1,…. *Rn }* that holds over *R*, one of the following statements is true:
- *Ri* = *R* for some *i*, or
- The JD is implied by the set of those FDs over *R* in which the left side is a key for *R*.
- The following result, also due to Date and Fagin, identies conditions|again, detected using only FD information|under which we can safely ignore JD information.
- If a relation schema is in 3NF and each of its keys consists of a single attribute,it is also in 5NF.

Inclusion Dependencies

- MVDs and JDs can be used to guide database design, as we have seen, although they are less common than FDs and harder to recognize and reason about.
- In contrast, inclusion dependencies are very intuitive and quite common. However, they typically have little influence on database design
- The main point to bear in mind is that we should not split groups of attributes that participate in an inclusion dependency.
- Most inclusion dependencies in practice are *key-based*, that is, involve only keys.

Recovery System

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers \$50 from account *A* to account *B*; goal is either to perform all database modifications made by T_i or none at all.
- Several output operations may be required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are

Recovery and Atomicity (Cont.)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - log-based recovery, and
  - shadow-paging
- We assume (initially) that transactions run serially, that is, one after the other.

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
  - Focus of this chapter
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Log-Based Recovery

- A log is kept on stable storage.
  - The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a
  <T_i start>log record
- *Before* T_i executes write(X), a log record <T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record <T_i commit> is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- **Transaction starts by writing <T_i *start*> record to log.**
- **When T_i partially commits, <T_i commit> is written to the log**
- **Finally, the log records are read and used to actually execute the previously deferred writes.**

Deferred Database Modification (Cont.)

- **During recovery after a crash, a transaction needs to be redone if and only if both <T_i start> and<T_i commit> are there in the log.**
- **Redoing a transaction T_i (redoT_i) sets the value of all data items updated by the transaction to the new values.**
- **Crashes can occur while**
  - **the transaction is executing the original updates, or**
  - **while recovery action is being taken**

<t0 start>

<t0,A,950>

<t0,B,2050>

<t0,commit>

<t1,start>

<t1,c,600>

<t1,commit>

<u>Portion of log</u>

Log	database
<t0 start>	
<t0,A,950>	
<t0,B,2050>	
<t0,commit>	
	A=950
	B=2050
<t1,start>	
<t1,c,600>	
<t1,commit>	
	C=600

Deferred Database Modification (Cont.)

- **Below we show the log as it appears at three instances of time**.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>
(a)	(b)	(c)

Immediate Database Modification

- The immediate database modification scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an output(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage

Immediate Database Modification

- Output of updated blocks can take place at any time before or after transaction commit

<to start>

<t0,A,1000,950>

<t0,B,2000,2050>

<t0 commit>

<t1 start>

<t1 start>

<t1,C,700,600>

<t1 commit>

- Recovery procedure has two operations instead of one:
  - undo(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - redo(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i

Immediate Database Modification (Cont.)

- Both operations must be idempotent
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i start>, but does not contain the record <T_i commit>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i start> and the record <T_i commit>.
- Undo operations are performed first, then redo operations.

Immediate Database Modification Example

Log	**Write**	**Output**
<T_0 **start**>		
<T_0, A, 1000, 950>		
T_0, B, 2000, 2050		
	A = 950	
	B = 2050	
<T_0 **commit**> X_1		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.

Below we show the log as it appears at three instances of time.

$<T_0$ start$>$	$<T_0$ start$>$	$<T_0$ start$>$
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit$>$	$<T_0$ commit$>$
	$<T_1$ start$>$	$<T_1$ start$>$
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit$>$
(a)	(b)	(c)

Reco

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Checkpoints

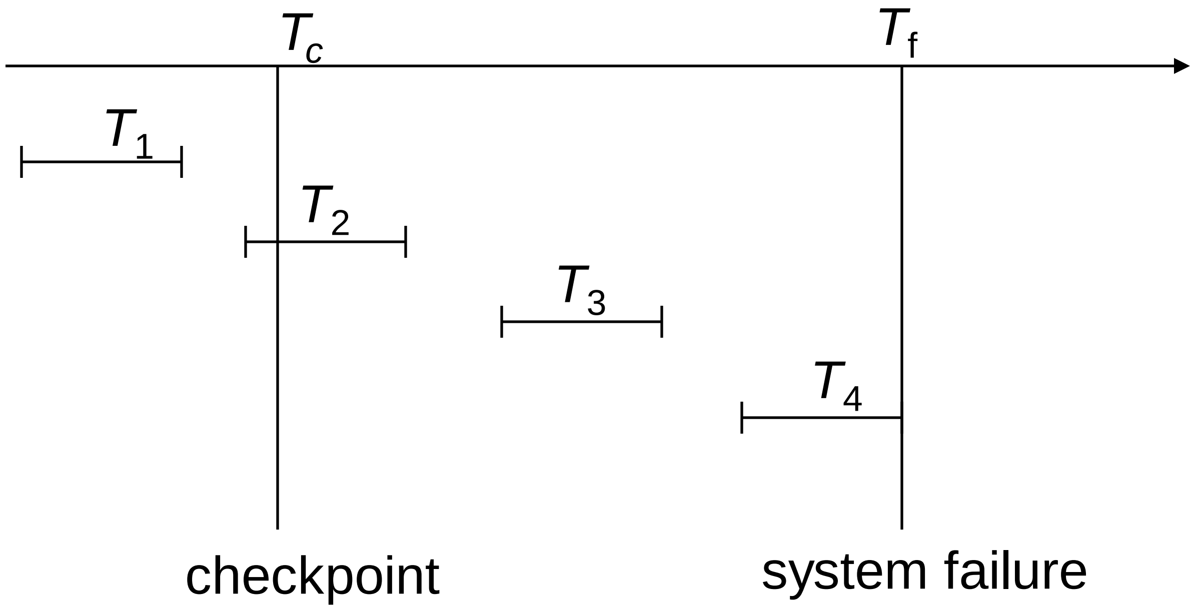
- **Problems in recovery procedure as discussed earlier :**
  1. **searching the entire log is time-consuming**
  2. **we might unnecessarily redo transactions which have already output their updates to the database.**
- **Streamline recovery procedure by periodically performing checkpointing**
  1. **Output all log records currently residing in main memory onto stable storage.**
  2. **Output all modified buffer blocks to the disk**

Checkpoints (Cont.)

- **During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.**

1. **Scan backwards from end of log to find the most recent <checkpoint> record**
2. **Continue scanning backwards till a record <T_i start> is found.**
3. **Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.**
4. **For all transactions (starting from T_i or later) with no <T_i commit>, execute undo(T_i). (Done only in case of immediate modification.)**
5. **Scanning forward in the log, for all transactions starting from T_i or later with a <T_i commit>, execute redo(T_i).**

Example of Checkpoints



- ***T_1*** **can be ignored (updates already output to disk due to checkpoint)**

Recovery With Concurrent Transactions

- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.
  - All transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions

  1)Interaction with concurrency control
- We assume concurrency control using strict two-phase locking;
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
- Logging is done as described earlier.
  - Log records of different transactions may be interspersed in the log.

Recovery With Concurrent Transactions

- <u>2)Transaction Rollback</u>

- We rollback a failed transaction ,Ti by using log
- System scans log backward.
- Scanning terminates when system finds<ti,start>
- Ex:<Ti,A,10,20>
- <Tj,A,20,30>

Recovery With Concurrent Transactions

- 3)checkpoints
- The checkpointing technique and actions taken on recovery have to be changed
  - since several transactions may be active when a checkpoint is performed.
- Checkpoints are performed as before, except that the checkpoint log record is now of the form

< checkpoint *L*> where *L* is the list of transactions active at the time of the checkpoint

  - We assume no updates are in progress either on biffer blocks or on log while the checkpoint is carried out (will relax this later)
  - A fuzzy checkpoint is a checkpoint where transactions are allowed to perform updates even while buffer blocks are being written out.

Recovery With Concurrent Transactions (Cont.)

- **4)restart recovery**
- **When the system recovers from a crash, it first does the following:**
  1. **Initialize *undo-list* and *redo-list* to empty**
  2. **Scan the log backwards from the end, stopping when the first <checkpoint *L*> record is found.**
     For each record found during the backward scan:
     - **if the record is <T_i commit>, add T_i to *redo-list***
     - **if the record is <T_i start>, then if T_i is not in *redo-list*, add T_i to *undo-list***
  3. **For every T_i in *L*, if T_i is not in *redo-list*, add T_i to *undo-list***

Recovery With Concurrent Transactions (Cont.)

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:

1. Scan log backwards from most recent record, stopping when <T_i start> records have been encountered for every T_i in *undo-list*.
   - During the scan, perform undo for each log record that belongs to a transaction in *undo-list*.
2. Locate the most recent <checkpoint *L*> record.

Example of Recovery

- **Go over the steps of the recovery algorithm on the following log:**

<T_0 **start**>
<T_0, A, 0, 10>
<T_0 **commit**>
<T_1 **start**> /* Scan at step 1 comes up to here */
<T_1, B, 0, 10>
<T_2 **start**>
<T_2, C, 0, 10>
<T_2, C, 10, 20>
<checkpoint {T_1, T_2}>
<T_3 **start**>
<T_3, A, 10, 20>
<T_3, D, 0, 10>
<T_3 **commit**>

BUFFER MANAGEMENT

- **1.Log Record Buffering**
- **Log record buffering: log records are buffered in main memory, instead of of being output directly to stable storage.**
  - **Log records are output to stable storage when a block of log records in the buffer is full, or a log force operation is executed.**
- **Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.**
- **Several log records can thus be output using**

Log Record Buffering (Cont.)

- **The rules below must be followed if log records are buffered:**
  - **Log records are output to stable storage in the order in which they are created.**
  - **Transaction T_i enters the commit state only when the log record <T_i commit> has been output to stable storage.**
  - **Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.**

2.Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.

3.Operating system role in buffer management

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.
- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.

Buffer Management (Cont.)

- When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
  - Known as dual paging problem.
- Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
  1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
  2. Release the page from the buffer, for the OS to use

Dual paging can thus be avoided, but common operating systems do not support such functionality.

4.Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically dump the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <dump> to log on stable storage.

Recovering from Failure of Non-Volatile Storage

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as fuzzy dump or online dump

Advanced Recovery: Key Features

- Support for high-concurrency locking techniques, such as those used for B^+-tree concurrency control, which release locks early
  – Supports "logical undo"
- Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing
  – including redo of log records of incomplete transactions, followed by subsequent undo
  – Key benefits
    - supports logical undo
    - easier to understand/show correctness

Advanced Recovery: Logical Undo Logging

- Operations like B^+-tree insertions and deletions release locks early.
  - They cannot be undone by restoring old values (physical undo), since once a lock is released, other transactions may have updated the B^+-tree.
  - Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as logical undo).
- For such operations, undo log records should contain the undo operation to be executed
  - Such logging is called logical undo logging, in contrast to physical undo logging
    - Operations are called logical operations

Advanced Recovery: Physical Redo

- Redo information is logged physically (that is, new value for each write) even for operations with logical undo
  - Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts
  - Physical redo logging does not conflict with early lock release

Advanced Recovery: Operation Logging

- **Operation logging is done as follows:**
  1. **When operation starts, log <T_i, O_j, operation-begin>. Here O_j is a unique identifier of the operation instance.**
  2. **While operation is executing, normal log records with physical redo and physical undo information are logged.**
  3. **When operation completes, <T_i, O_j, operation-end, U> is logged, where U contains information needed to perform a logical undo information.**

Example: insert of (key, record-id) pair (K5, RID7) into index I9

<T1, O1, operation-begin>

….

<T1, X, 10, K5>

<T1, Y, 45, RID7>

Physical redo of steps in insert

<T1, O1, operation-end, (delete I9, K5, RID7)>

Advanced Recovery: Operation Logging (Cont.)

- **If crash/rollback occurs before operation completes:**
  - **the operation-end log record is not found, and**
  - **the physical undo information is used to undo operation.**
- **If crash/rollback occurs after the operation completes:**
  - **the operation-end log record is found, and in this case**
  - **logical undo is performed using *U*; the physical undo information for the operation is ignored.**
- **Redo of operation (after crash) still uses physical redo information.**

Advanced Recovery: Txn Rollback

Rollback of transaction T_i is done as follows:

- **Scan the log backwards**
  1. **If a log record <T_i, X, V_1, V_2> is found, perform the undo and log a special redo-only log record <T_i, X, V_1>.**
  2. **If a <T_i, O_j, operation-end, U> record is found**
     - **Rollback the operation logically using the undo information U.**
       - **Updates performed during roll back are logged just like during normal operation execution.**
       - **At the end of the operation rollback, instead of logging an operation-end record, generate a record**

         <T_i, O_j, operation-abort>.
     - **Skip all preceding log records for T_i until the record <T_i, O_j operation-begin> is found**

Advanced Recovery: Txn Rollback (Cont.)

- **Scan the log backwards (cont.):**

  3. If a redo-only record is found ignore it

  4. If a <T_i, O_j, operation-abort> record is found:

  - **skip all preceding log records for T_i until the record <T_i, O_j, operation-begin> is found.**

  5. Stop the scan when the record <T_i, start> is found

  6. Add a <T_i, abort> record to the log

Some points to note:

- **Cases 3 and 4 above can occur only if the database crashes while a transaction is being**

Advanced Recovery: Txn Rollback

- Example with a complete and an incomplete

<T1, start>

<T1, O1, operation-begin>

….

<T1, X, 10, K5>

<T1, Y, 45, RID7>

<T1, O1, operation-end, (delete I9, K5, RID7)>

<T1, O2, operation-begin>

<T1, Z, 45, 70>

← T1 Rollback begins here

<T1, Z, 45> ← redo-only log record during physical undo (of incomplete O2)

<T1, Y, .., ..> ← Normal redo records for logical undo of O1

…

<T1, O1, operation-abort> ← What if crash occurred immediately after this?

<T1, abort>

Advanced Recovery: Crash Recovery

The following actions are taken when recovering from system crash

1. **(Redo phase): Scan log forward from last < checkpoint *L*> record till end of log**
   1. **Repeat history by physically redoing all updates of all transactions,**
   2. **Create an undo-list during the scan as follows**
      - ***undo-list* is set to *L* initially**
      - **Whenever <T_i start> is found T_i is added to *undo-list***
      - **Whenever <T_i commit> or <T_i abort> is found, T_i is deleted from *undo-list***

   This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

Now *undo-list* contains transactions that are incomplete, that is, have neither committed nor been fully rolled back.

Advanced Recovery: Crash Recovery (Cont.)

Recovery from system crash (cont)

2. (**Undo phase**): **Scan log backwards, performing undo on log records of transactions found in *undo-list*.**
   - **Log records of transactions being rolled back are processed as described earlier, as they are found**
     - **Single shared scan for all transactions being undone**
   - **When <T_i start> is found for a transaction T_i in *undo-list*, write a <T_i abort> log record.**
   - **Stop scan when <T_i start> records have been found for all T_i in *undo-list***

- **This undoes the effects of incomplete**

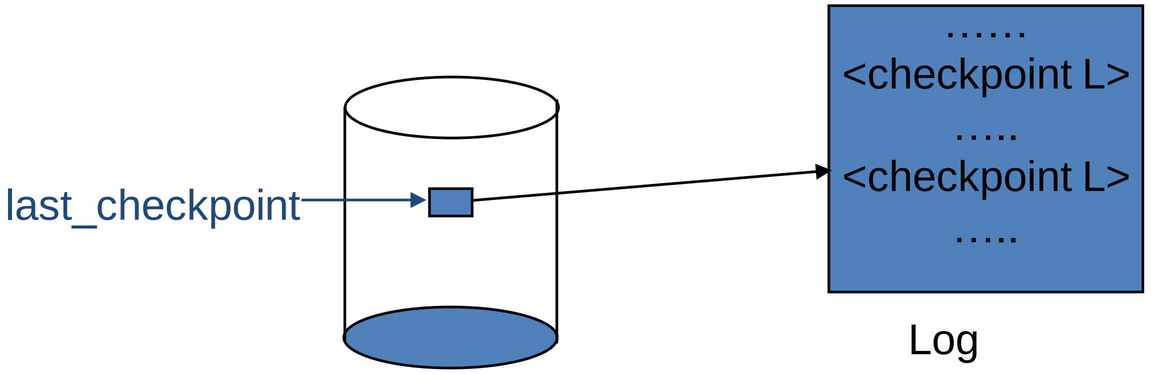
Advanced Recovery: Checkpointing

- **Checkpointing is done as follows:**
  1. **Output all log records in memory to stable storage**
  2. **Output to disk all modified buffer blocks**
  3. **put to log on stable storage a < checkpoint *L*> record.**

  Transactions are not allowed to perform any actions while checkpointing is in progress.
- **Fuzzy checkpointing allows transactions to progress while the most time consuming**

Advanced Recovery: Fuzzy Checkpointing

- Fuzzy checkpointing is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a <checkpoint *L*> log record and force log to stable storage
  3. Note list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list *M*
     - blocks should not be updated while being output
     - Follow WAL: all log records pertaining to a block must be output before the block is output
  6. Store a pointer to the checkpoint record in a fixed position last_checkpoint on disk



Advanced Rec: Fuzzy Checkpointing (Cont.)

- When recovering using a fuzzy checkpoint, start scan from the checkpoint record pointed to by last_checkpoint
  - Log records before last_checkpoint have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely

ARIES

- ARIES is a state of the art recovery method
  - Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
  - The "advanced recovery algorithm" we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations
- Unlike the advanced recovery algorithm, ARIES
  1. Uses log sequence number (LSN) to identify log records
     - Stores LSNs in pages to identify what updates have already been applied to a database page
  2. Physiological redo
  3. Dirty page table to avoid unnecessary redos during recovery
  4. Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time

ARIES Optimizations

- Physiological redo
  - Affected page is physically identified, action within page can be logical
    - Used to reduce logging overheads
      - e.g. when a record is deleted and all other records have to be moved to fill hole
        - » Physiological redo can log just the record deletion
        - » Physical redo would require logging of old and new values for much of the page
    - Requires page to be output to disk atomically
      - Easy to achieve with hardware RAID, also supported by some disk systems
      - Incomplete page output can be detected by checksum techniques,
        - » But extra actions are required for recovery
        - » Treated as a media failure

ARIES Data Structures

- ARIES uses several data structures
  - Log sequence number (LSN) identifies each log record
    - Must be sequentially increasing
    - Typically an offset from beginning of log file to allow fast access
      - Easily extended to handle multiple log files
  - Page LSN
  - Log records of several different types
  - Dirty page table

ARIES Data Structures: Page LSN

- Each page contains a **PageLSN** which is the LSN of the last log record whose effects are reflected on the page
  - To update a page:
    - X-latch the page, and write the log record
    - Update the page
    - Record the LSN of the log record in PageLSN
    - Unlock page
  - To flush page to disk, must first S-latch page
    - Thus page state on disk is operation consistent
      - Required to support physiological redo
  - PageLSN is used during recovery to prevent repeated redo
    - Thus ensuring idempotence

ARIES Data Structures: Log Record

- Each log record contains LSN of previous log record of the same transaction

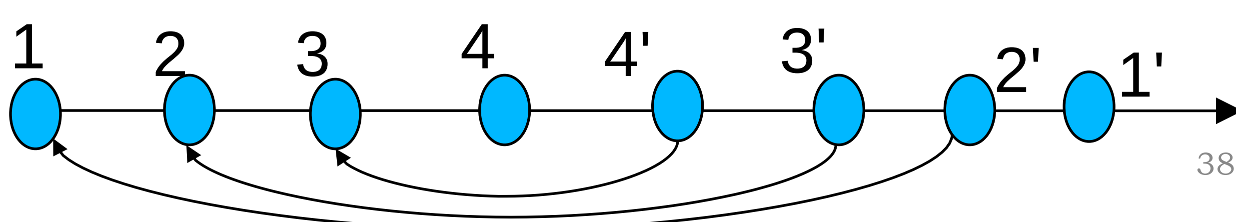
LSN	TransID	PrevLSN	RedoInfo	UndoInfo

  – LSN in log record may be implicit

- Special redo-only log record called compensation log record (CLR) used to log actions taken during recovery that never need to be undone
  – Serves the role of operation-abort log records used in advanced recovery algorithm
  – Has a field UndoNextLSN to note next (earlier) record to be undone
    - Records in between would have already been undone

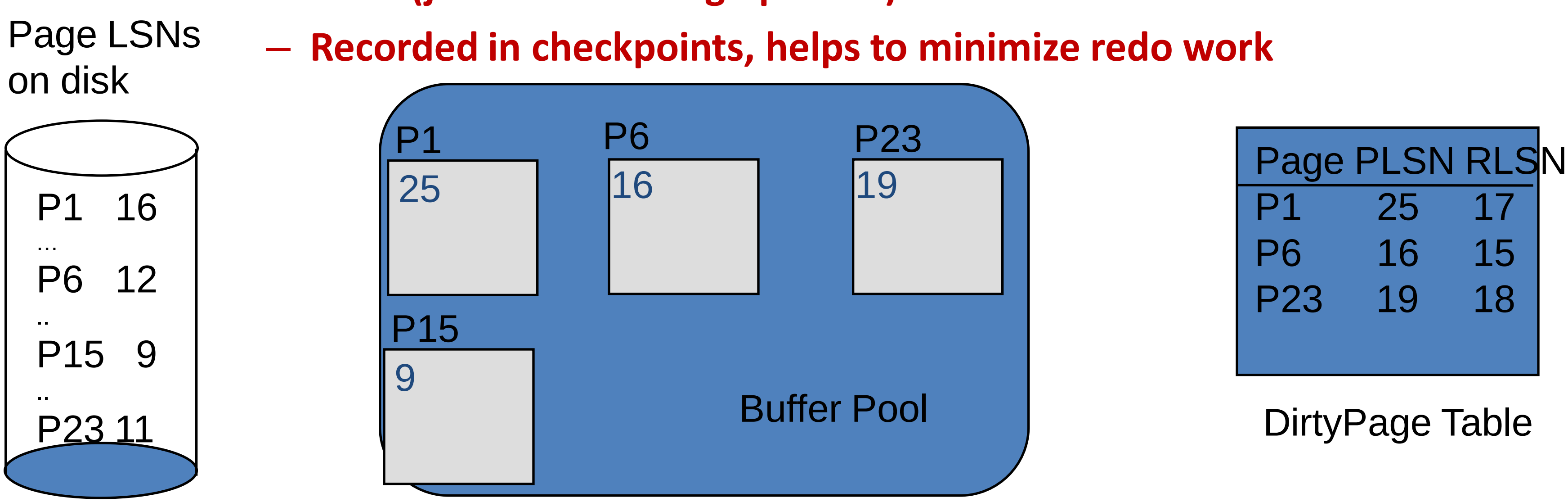
LSN	TransID	UndoNextLSN	RedoInfo

    - Required to avoid repeated undo of already undone actions



ARIES Data Structures: DirtyPage Table

- **DirtyPageTable**
  - **List of pages in the buffer that have been updated**
  - **Contains, for each such page**
    - **PageLSN of the page**
    - **RecLSN is an LSN such that log records before this LSN have already been applied to the page version on disk**
      - **Set to current end of log when a page is inserted into dirty page table (just before being updated)**
      - **Recorded in checkpoints, helps to minimize redo work**



ARIES Data Structures: Checkpoint Log

- Checkpoint log record
  - Contains:
    - DirtyPageTable and list of active transactions
    - For each active transaction, LastLSN, the LSN of the last log record written by the transaction
  - Fixed position on disk notes LSN of last completed checkpoint log record
- Dirty pages are not written out at checkpoint time
    - Instead, they are flushed out continuously, in the background
- Checkpoint is thus very low overhead
  - can be done frequently

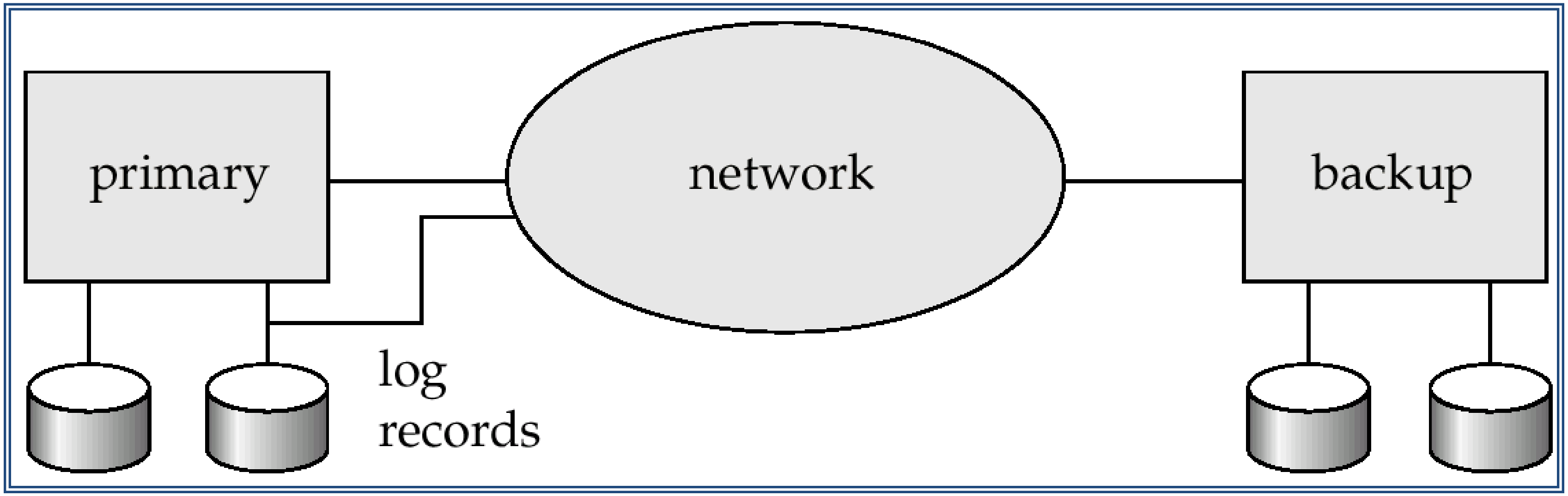
ARIES Recovery Algorithm

ARIES recovery involves three passes

- **Analysis pass**: Determines
  - Which transactions to undo
  - Which pages were dirty (disk version not up to date) at time of crash
  - RedoLSN: LSN from which redo should start
- **Redo pass:**
  - Repeats history, redoing all actions from RedoLSN
    - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
- **Undo pass:**
  - Rolls back all incomplete transactions

Remote Backup Systems

- **Remote backup systems provide high availability by allowing transaction processing**



Remote Backup Systems (Cont.)

- **Detection of failure: Backup site must detect when primary site has failed**
  - **to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.**
  - **Heart-beat messages**
- **Transfer of control:**
  - **To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.**
    - **Thus, completed transactions are redone and incomplete transactions are rolled back.**
  - **When the backup site takes over processing it becomes the new primary**
  - **To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.**

Remote Backup Systems (Cont.)

- **Time to recover**: **To reduce delay in takeover, backup site periodically proceses the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.**
- **Hot-Spare configuration permits very fast takeover**:
  - **Backup continually processes redo log record as they arrive, applying the updates locally.**
  - **When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions**

Remote Backup Systems (Cont.)

- **Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.**
- **One-safe: commit as soon as transaction's commit log record is written at primary**
  - **Problem: updates may not arrive at backup before it takes over.**
- **Two-very-safe: commit when transaction's commit log record is written at primary and backup**
  - **Reduces availability since transactions cannot commit if either site fails.**

- **Two-safe: proceed as in two-very-safe if both**

DATABASE MANAGEMENT SYSTEMS

UNIT-V

Data on External Storage

- **Disks:** Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order
- **Tapes:** Can only read pages in sequence
  - Cheaper than disks; used for archival storage
- **File organization:** Method of arranging a file of records on external storage.
  - Record id (rid) is sufficient to physically locate record
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields
- **Architecture:** Buffer manager stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

Alternative File Organizations

Many alternatives exist, *each ideal for some situations, and not so good in others:*

- **Heap (random order) files:** **Suitable when typical access is a file scan retrieving all records.**
- **Sorted Files:** **Best if records must be retrieved in some order, or only a `range' of records is needed.**
- **Indexes:** **Data structures to organize records via trees or hashing.**
  - **Like sorted files, they speed up searches for a subset of records, based on values in certain (“search key”) fields**
  - **Updates are much faster than in sorted files.**

Indexes

- An ***index*** on a file speeds up selections on the ***search key fields*** for the index.
  - **Any subset of the fields of a relation can be the search key for an index on the relation.**
  - ***Search key*** **is not the same as** ***key*** **(minimal set of fields that uniquely identify a record in a relation).**
- **An index contains a collection of *data entries*, and supports efficient retrieval of all data entries k* with a given key value k.**
  - **Given data entry k*, we can find record with key k in at most one disk I/O**

Alternatives for Data Entry k* in Index

- In a data entry k* we can store:
  - Data record with key value **k,** or
  - <**k**, rid of data record with search key value **k**>, or
  - <**k**, list of rids of data records with search key **k**>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - **Examples of indexing techniques: B+ trees, hash-based structures**
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries (Contd.)

- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
  - If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternatives for Data Entries (Contd.)

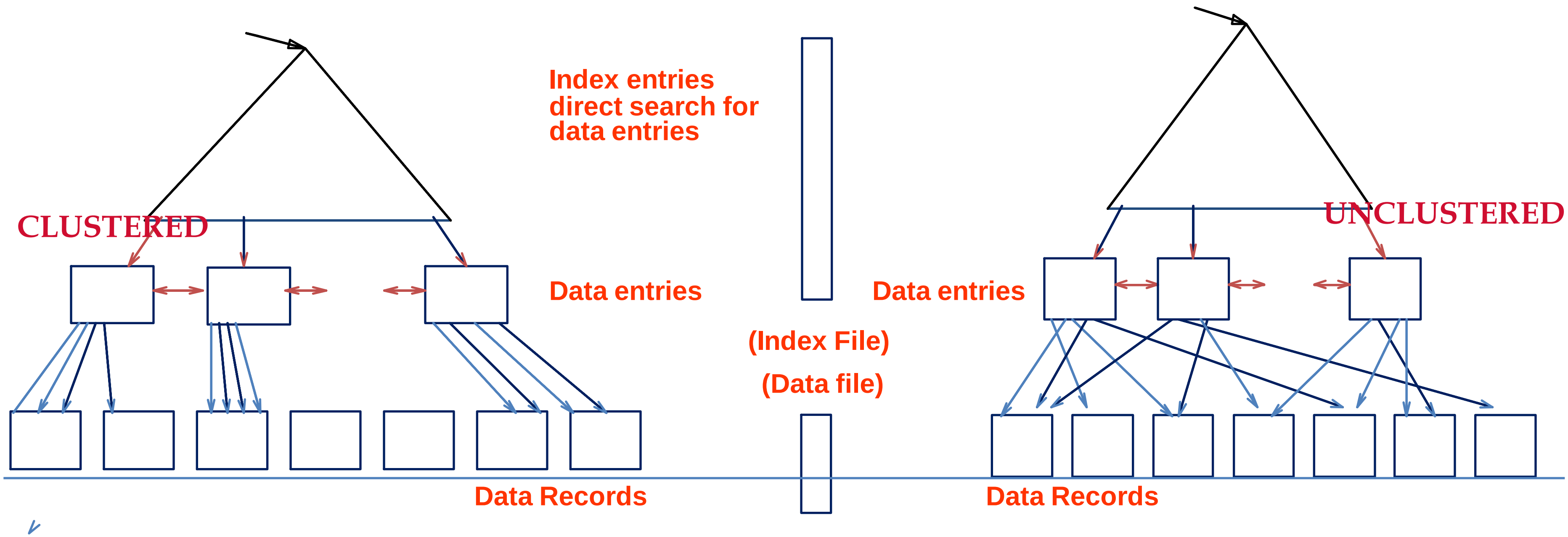
- Alternatives 2 and 3:
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Index Classification

- ***Primary* vs. *secondary*:** **If search key contains primary key, then called primary index.**
  - ***Unique* index: Search key contains a candidate key.**
- ***Clustered* vs. *unclustered*:** **If order of data records is the same as, or `close to', order of data entries, then called clustered index.**
  - **Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).**
  - **A file can be clustered on at most one search key.**
  - **Cost of retrieving data records through index varies *greatly* based on whether index is clustered**

Clustered vs. Unclustered Index

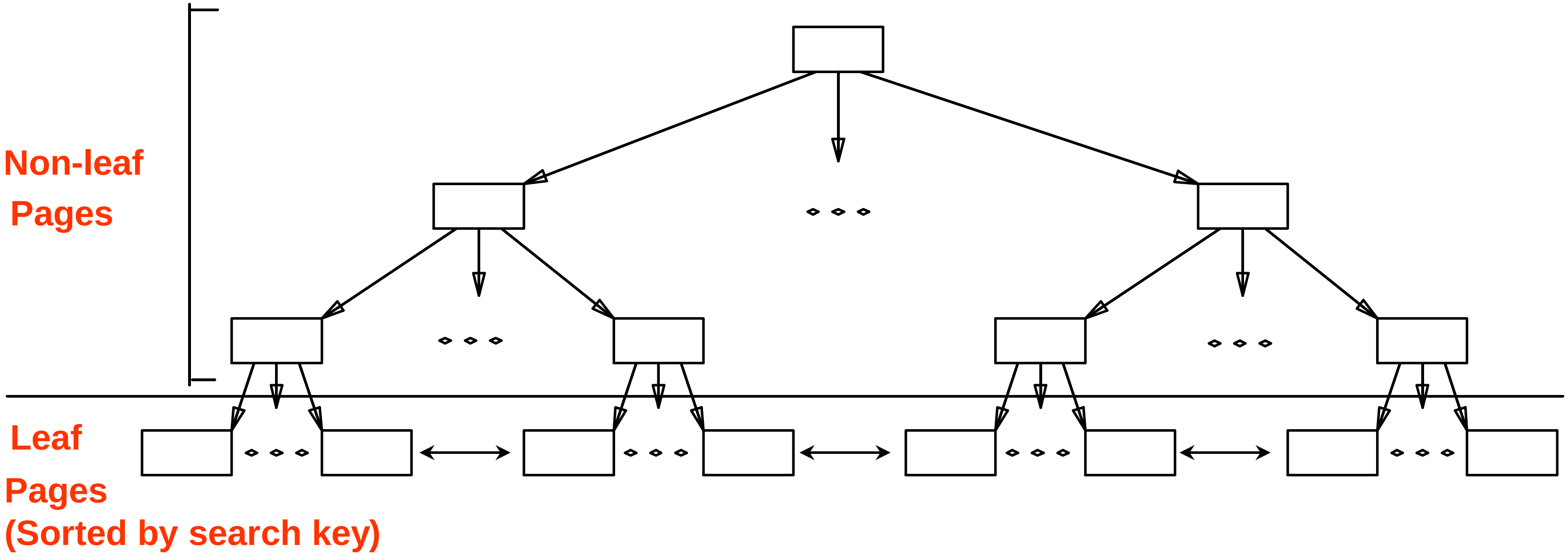
- **Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.**
  - **To build clustered index, first sort the Heap file (with some free space on each page for future inserts).**
  - **Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)**



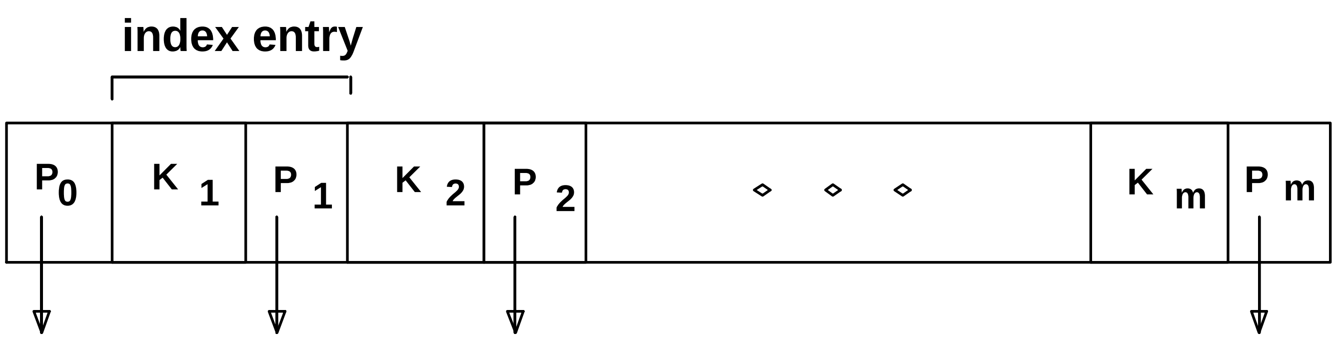
Hash-Based Indexes

- **Good for equality selections.**
- **Index is a collection of *buckets*.**
  - **Bucket = *primary* page plus zero or more *overflow* pages.**
  - **Buckets contain data entries.**
- ***Hashing function* h: h(*r*) = bucket in which (data entry for) record *r* belongs. h looks at the *search key* fields of *r*.**
  - ***No need for "index entries" in this scheme.***

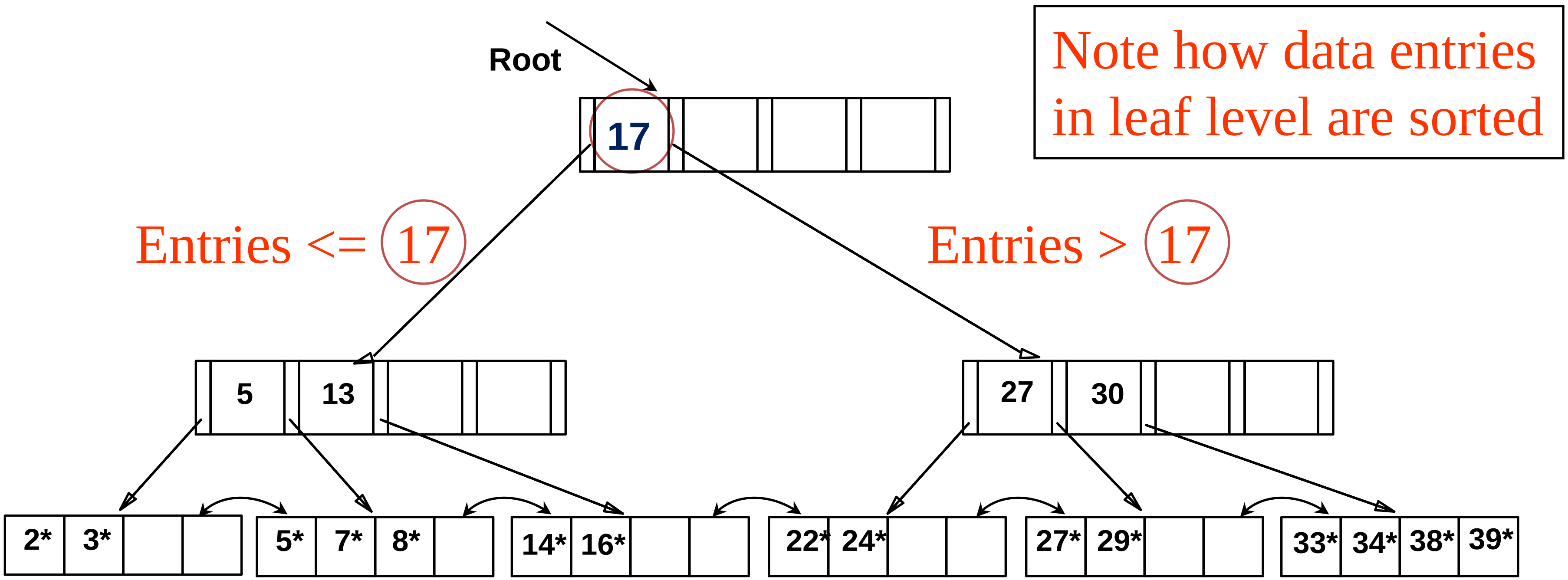
B+ Tree Indexes



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages have *index entries;* only used to direct searches:



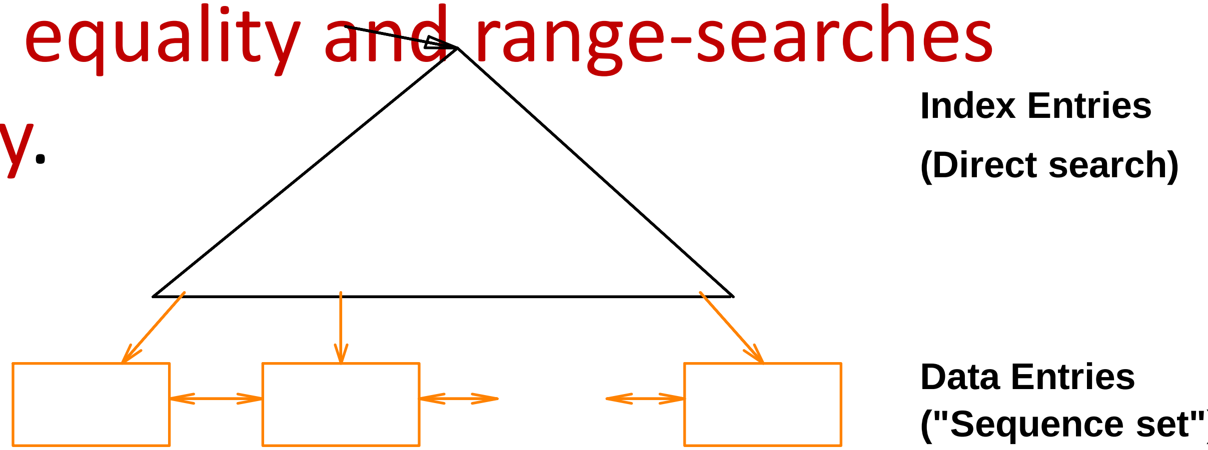
Example B+ Tree



- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

B+ Tree: Most Widely Used Index

- Insert/delete at log $_F$ N cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.



B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.

Comparing File Organizations

- **Heap files (random order; insert at eof)**
- **Sorted files, sorted on <*age, sal*>**
- **Clustered B+ tree file, Alternative (1), search key <*age, sal*>**
- **Heap file with unclustered B + tree index on search key <*age, sal*>**
- **Heap file with unclustered hash index on search key <*age, sal*>**

Operations to Compare

- **Scan: Fetch all records from disk**
- **Equality search**
- **Range selection**
- **Insert a record**
- **Delete a record**

Assumptions in Our Analysis

- **Heap Files:**
  - **Equality selection on key; exactly one match.**
- **Sorted Files:**
  - **Files compacted after deletions.**
- **Indexes:**
  - **Alt (2), (3): data entry size = 10% size of record**
  - **Hash: No overflow buckets.**
    - **80% page occupancy => File size = 1.25 data size**
  - **Tree: 67% occupancy (this is typical).**
    - **Implies file size = 1.5 data size**

Assumptions (contd.)

- **Scans:**
  - **Leaf levels of a tree-index are chained.**
  - **Index data-entries plus actual file scanned for unclustered indexes.**
- **Range searches:**
  - **We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.**

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	D log $_2$B	D(log $_2$ B + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	D log $_F$ 1.5B	D(log $_F$ 1.5B + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	D(1 + log $_F$ 0.15B)	D(log $_F$ 0.15B + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

Clustered index organization

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!

Examples of Clustered Indexes

- **B+ tree ndex on E.age can be used to get qualifying tuples.**
  - **How selective is the condition?**
  - **Is the index clustered?**

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

- Consider the GROUP BY query.
  - **If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.**
  - **Clustered *E.dno* index may be better!**

```
SELECT  E.dno, COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

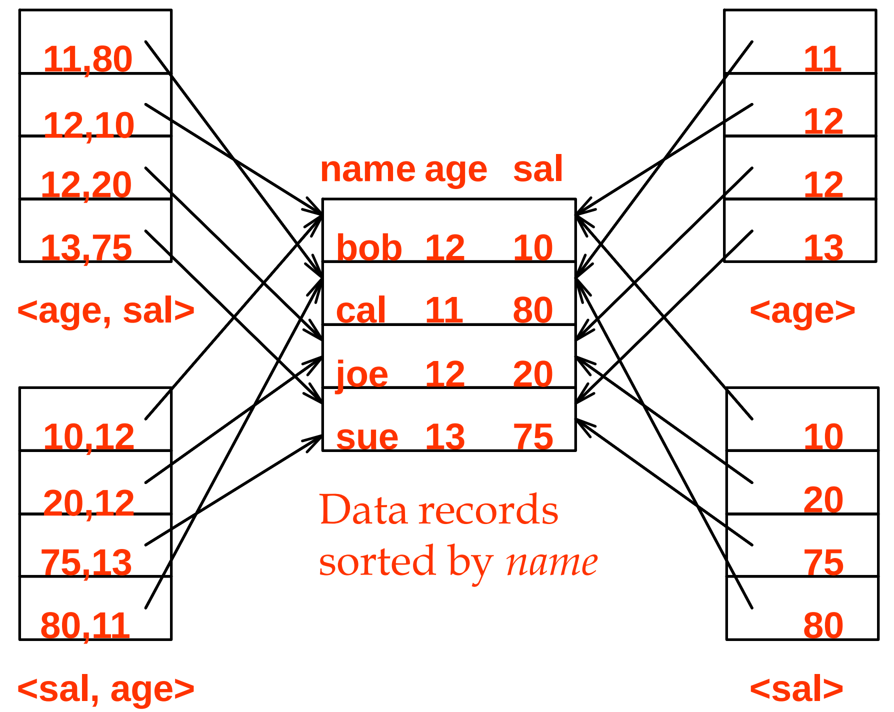
- **Equality queries and duplicates:**
  - **Clustering on *E.hobby* helps!**

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

Indexes with Composite Search Keys

- *Composite Search Keys*: **Search on a combination of fields.**
  - Equality query: **Every field value is equal to a constant value. E.g. wrt <sal,age> index:**
    - **age=20 and sal =75**
  - Range query: **Some field value is not a constant. E.g.:**
    - **age =20; or age=20 and sal > 10**
- **Data entries in index sorted by search key to support range queries.**
  - Lexicographic order, or
  - Spatial order.

Examples of composite key indexes using lexicographic order.



Data entries in index sorted by <*sal,age*>

Data entries sorted by <*sal*>

tradeoffs

- A composite key index can support a broader range of queries bcoz it matches more selection conditions
- Index only evaluation strategies are increased
- Disadv:a composite index must be updated in response to any operation(insert,delete or update) that modifies any field in search key.
- A composite index is also likely to be larger than single attribute search key
- For B+ tree index this increases no. of levels

Composite Search Keys

- **To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.**
  - **Choice of index key orthogonal to clustering etc.**
- **If condition is: 20<*age*<30 AND 3000<*sal*<5000:**
  - **Clustered tree index on <*age,sal*> or <*sal,age*> is best.**
- **If condition is: *age*=30 AND 3000<*sal*<5000:**
  - **Clustered <*age,sal*> index much better than <*sal,age*> index!**
- **Composite indexes are larger, updated more often.**

Composite keys-Index-Only Plans

- **A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.**

<E.dno>

```
SELECT E.dno, COUNT(*)
FROM Emp E
GROUP BY E.dno
```

<E.dno,E.sal>

Tree index!

```
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno
```

<E. age,E.sal>

or

<E.sal, E.age>

Tree index!

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
 E.sal BETWEEN 3000 AND 5000
```

Creating index in sql

Syntax

Create index indexname on tablename

with structure=Btree,

key=(age,sal);

Summary

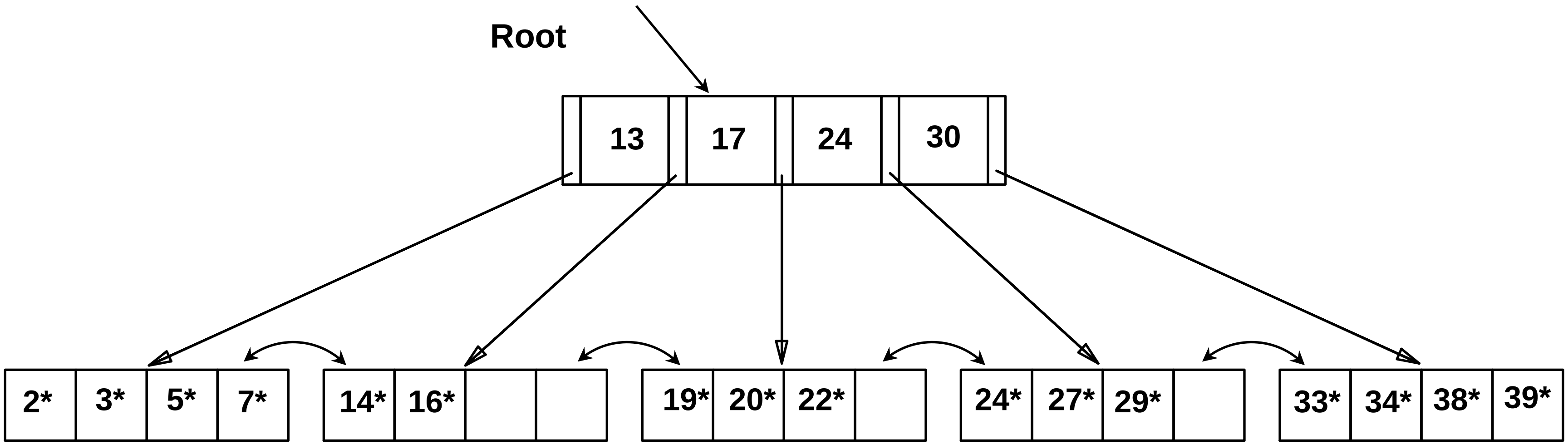
- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values.

Summary (Contd.)

- **Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.**
  - **Choice orthogonal to *indexing technique* used to locate data entries with a given key value.**
- **Can have several indexes on a given file of data records, each with a different search key.**
- **Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.**

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...

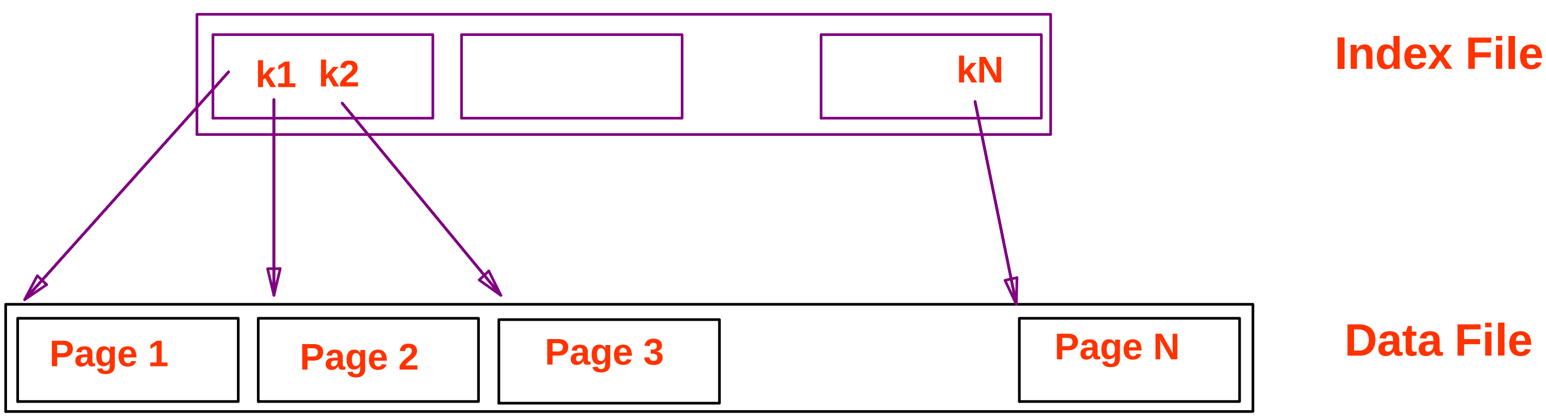


Introduction

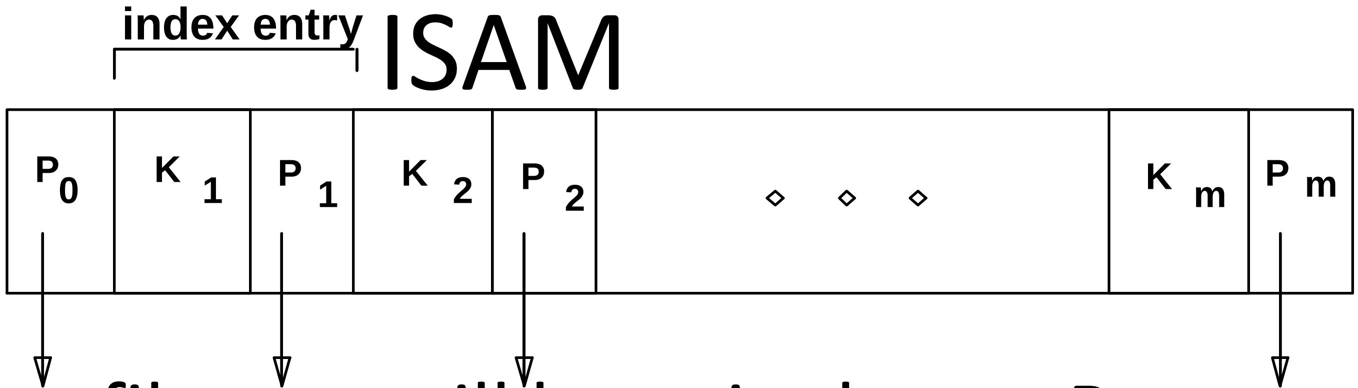
- ***As for any index, 3 alternatives for data entries*** **k*:**
  - **Data record with key value k**
  - **<k, rid of data record with search key value k>**
  - **<k, list of rids of data records with search key k>**
- **Choice is orthogonal to the *indexing technique* used to locate data entries k*.**
- **Tree-structured indexing techniques support both *range searches* and *equality searches*.**
- ***ISAM*: static structure; *B+ tree*: dynamic,**

Range Searches

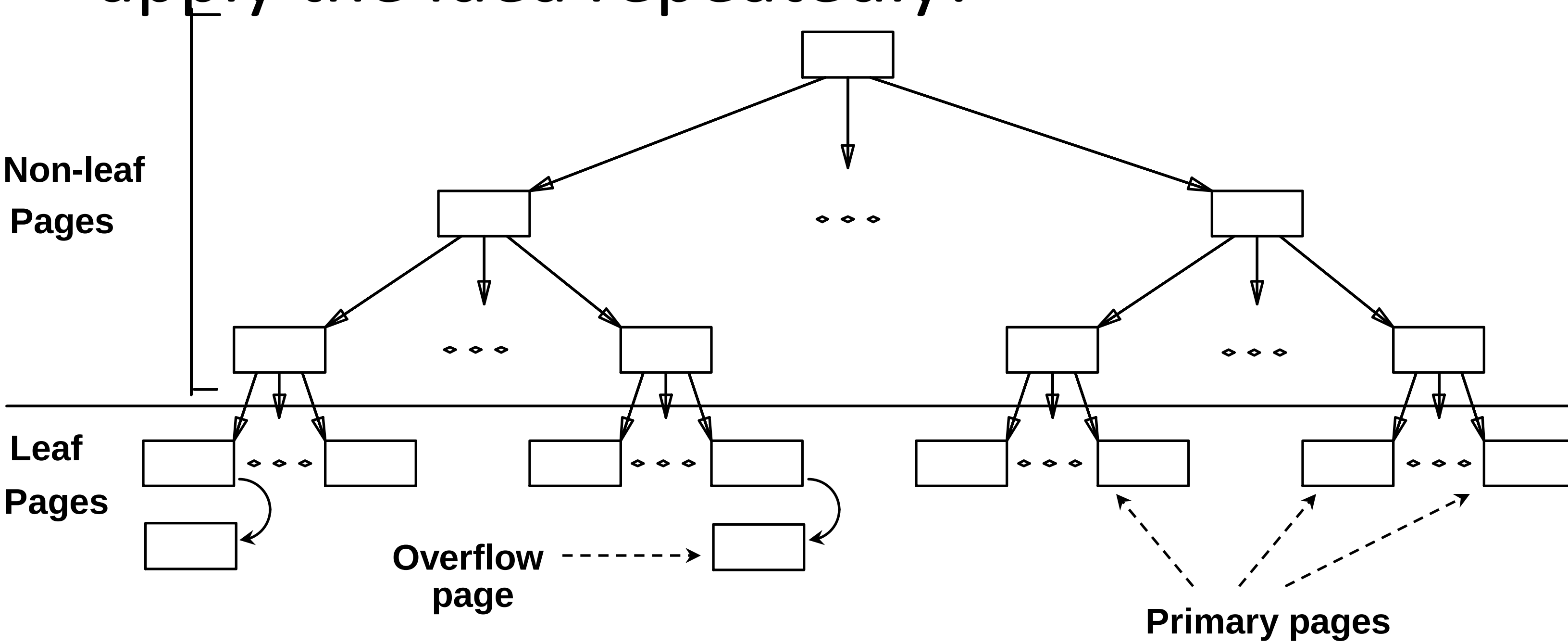
- ``*Find all students with gpa > 3.0*''
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- **Simple idea: Create an `index' file.**



ISAM

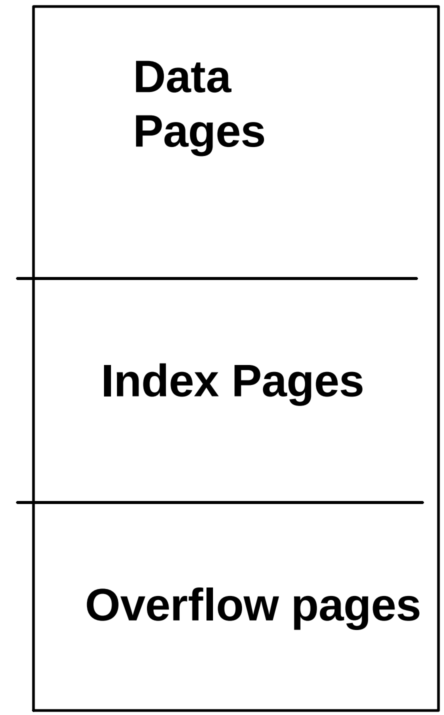


- Index file may still be quite large. But we can apply the idea repeatedly!



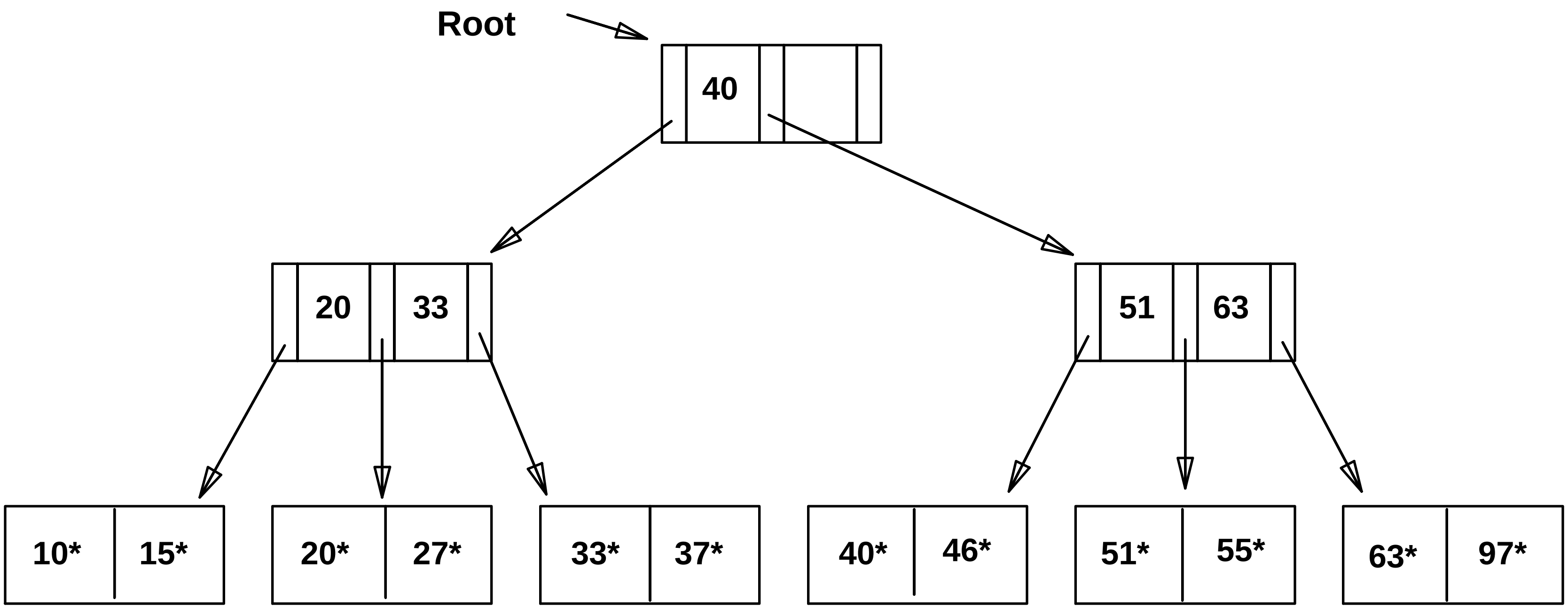
Comments on ISAM

- ***File creation***: **Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.**
- ***Index entries***: **<search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.**
- ***Search***: **Start at root; use key comparisons to go to leaf. Cost $\propto$ $\log_F N$; F = # entries/index pg, N = # leaf pgs**
- ***Insert***: **Find leaf data entry belongs to, and put it there.**
- ***Delete***: **Find and remove from leaf; if empty overflow page, de-allocate.**

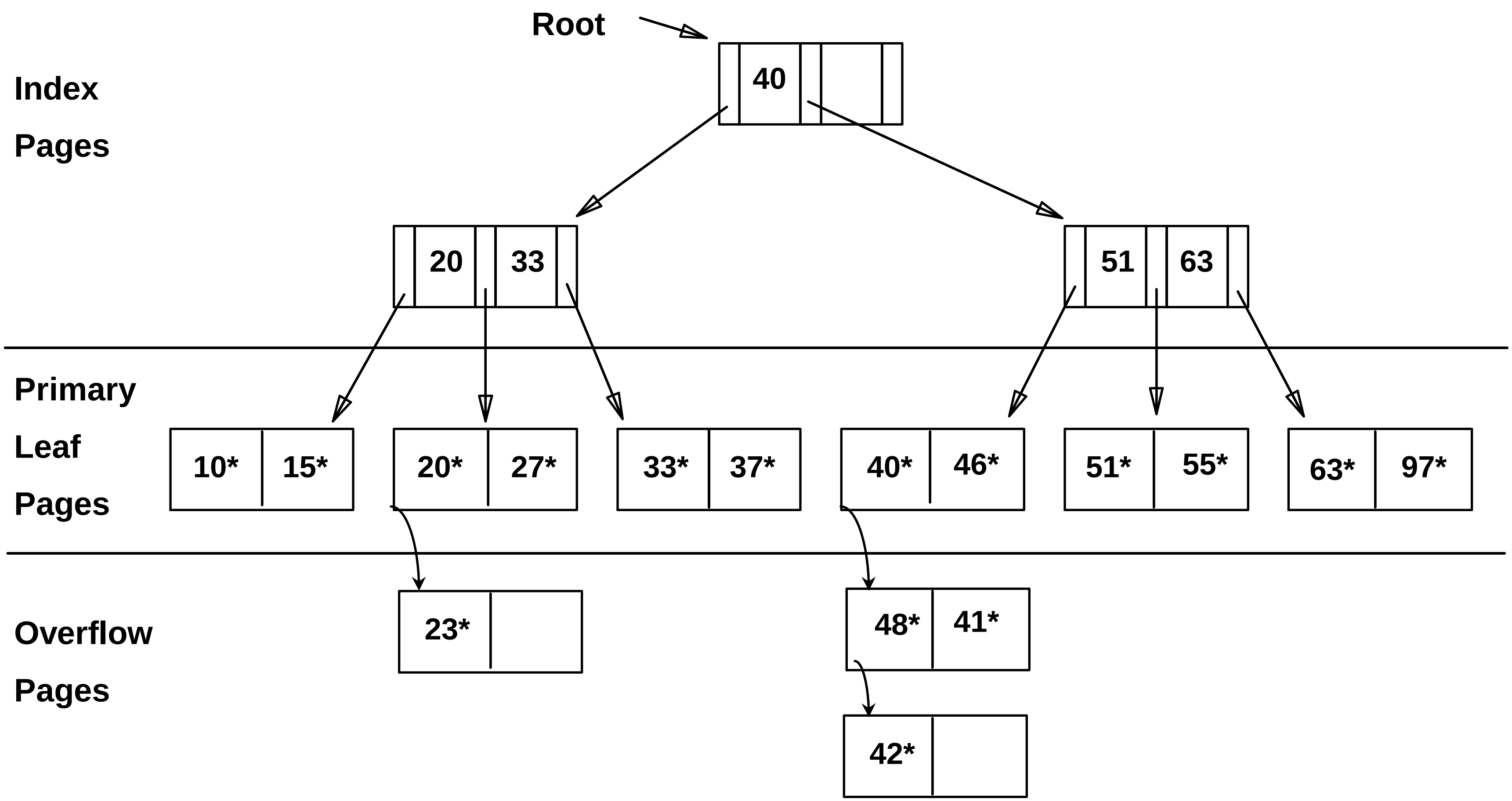


Example ISAM Tree

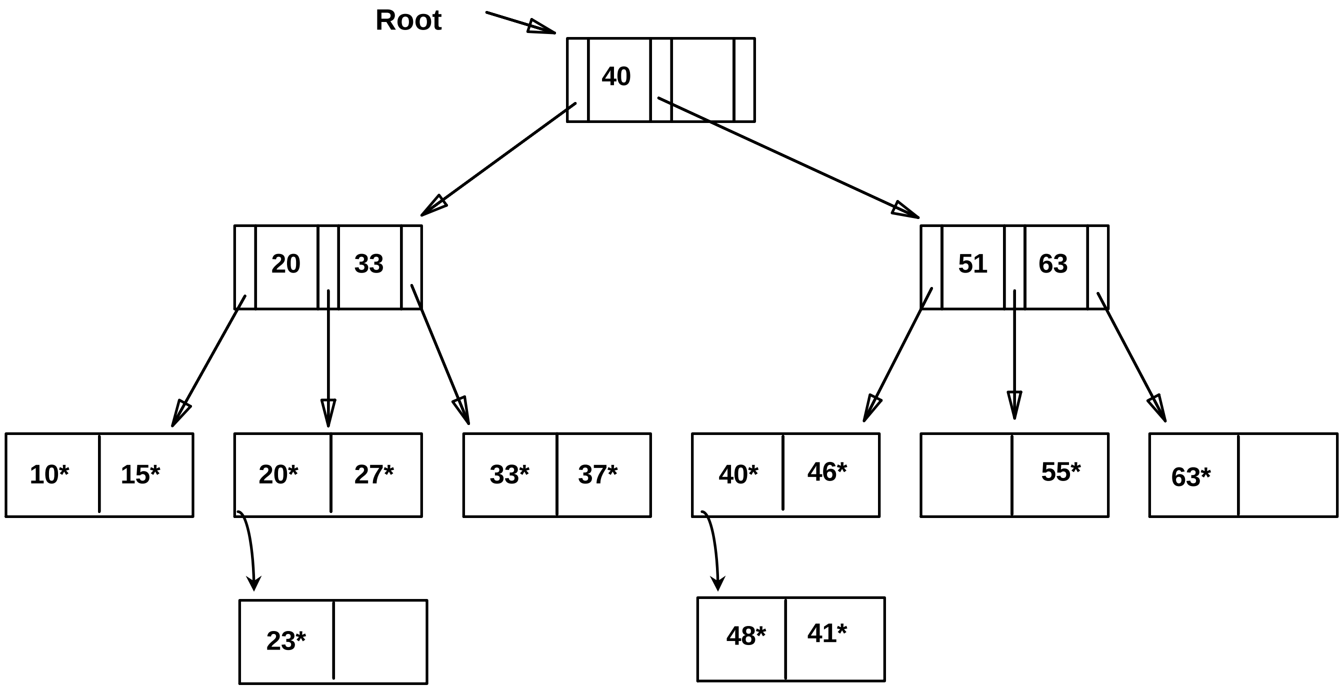
- **Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)**



After Inserting 23*, 48*, 41*, 42* ...



... Then Deleting 42*, 51*, 97*

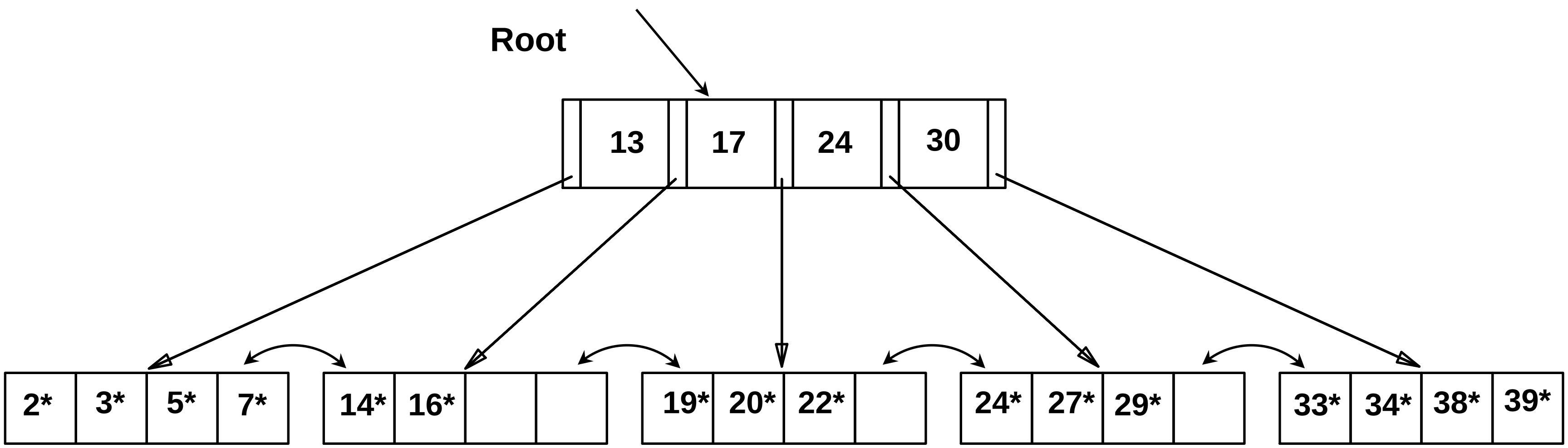


Overflow pages,locking considerations

- Once ISAM file is created,inserts and deletes affect only contents of leaf pages.so as a result for more number of insertions overflow pages increase.
- Solution:20% of pages sholud be left free when initially tree is created
- The fact that only leaf pages can be modified has advantage with respect to concurrent access.
- When a page is accessed it is typically locked by the requestor to ensure that it is not concurrently modified by other users
- ADV:Since we know that indexlevel pages are never modifiedwe can safely omit locking step.

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...

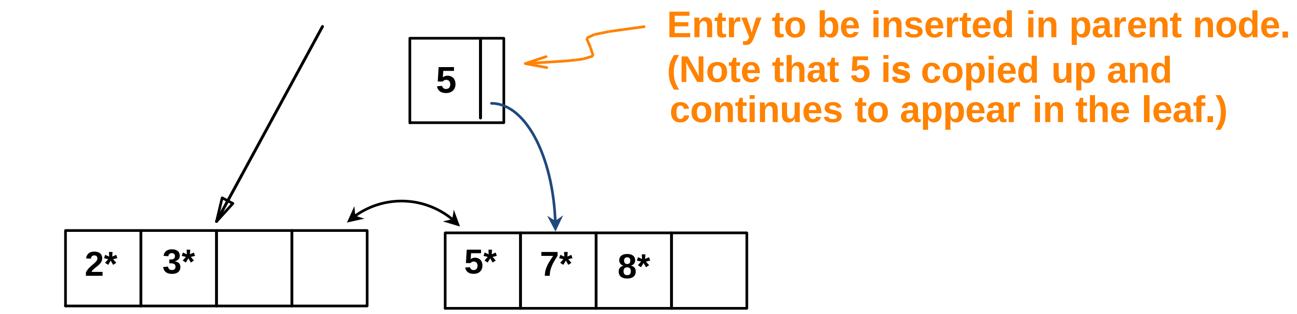


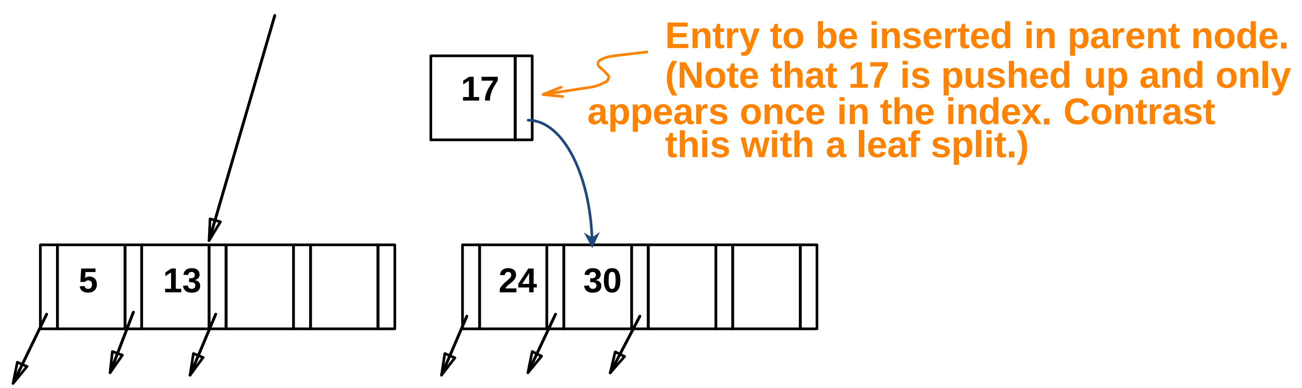
Inserting a Data Entry into a B+ Tree

- **Find correct leaf *L.***
- **Put data entry onto *L*.**
  - **If *L* has enough space, *done*!**
  - **Else, must *split* *L (into L and a new node L2)***
    - **Redistribute entries evenly, copy up middle key.**
    - **Insert index entry pointing to *L2* into parent of *L*.**
- **This can happen recursively**
  - **To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)**
- **Splits "grow" tree; root split increases height.**
  - **Tree growth: gets *wider* or *one level taller at top.***

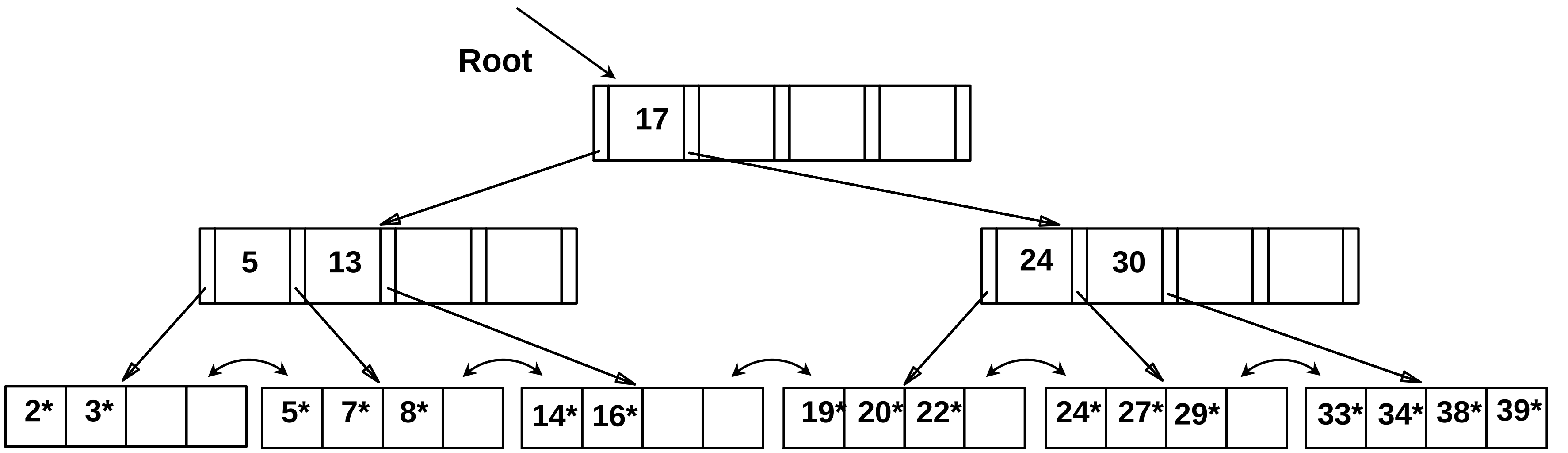
Inserting 8* into Example B+ Tree

- **Observe how minimum occupancy is guaranteed in both leaf and index pg splits.**
- **Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.**





Example B+ Tree After Inserting 8*



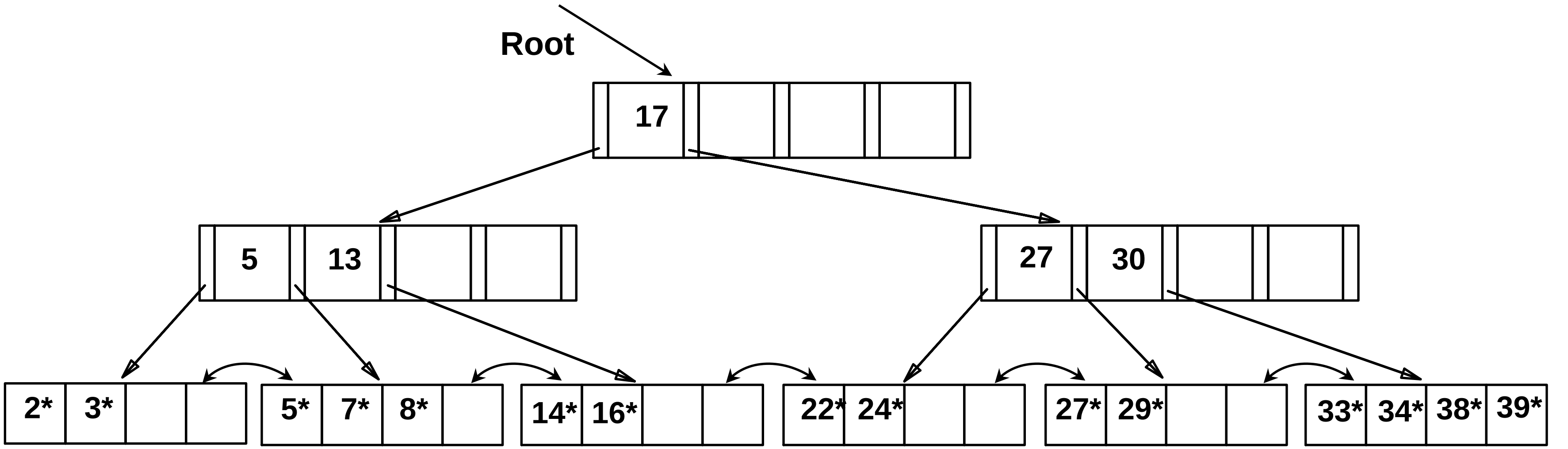
❖ Notice that root was split, leading to increase in height.

❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Deleting a Data Entry from a B+ Tree

- **Start at root, find leaf *L* where entry belongs.**
- **Remove the entry.**
  - **If L is at least half-full, *done!***
  - **If L has only d-1 entries,**
    - **Try to re-distribute, borrowing from *sibling* *(adjacent node with same parent as L)*.**
    - **If re-distribution fails, *merge* *L* and sibling.**
- **If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.**
- **Merge could propagate to root, decreasing height.**

Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...



- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

- **Must merge.**
- **Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).**