

Computers and Data Organization

CS281

Department of Computer Engineering, Bilkent University

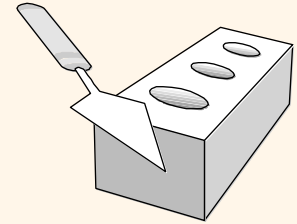
Dr. Mustafa Değerli



Bilkent University

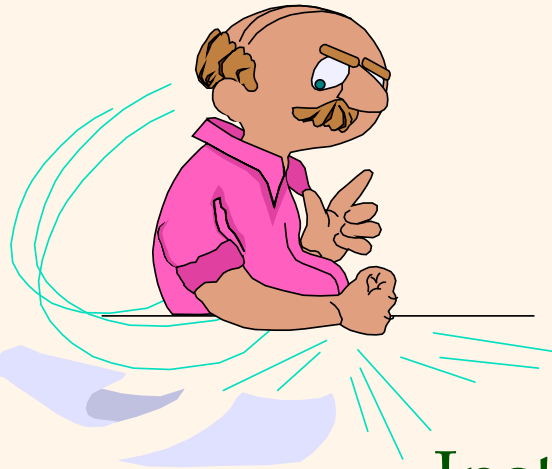
1

- Introduction to Database Design (**Ch.1**)



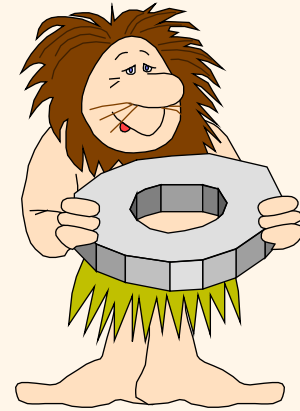
Database Management Systems

Chapter 1

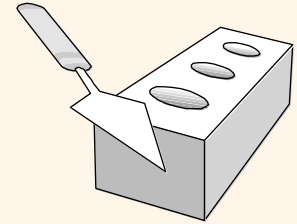


Instructor: Raghu Ramakrishnan
raghu@cs.wisc.edu

What Is a DBMS?



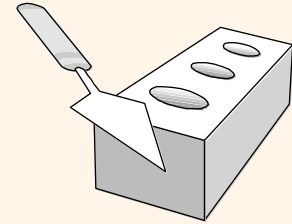
- ❖ A very large, integrated collection of data.
- ❖ Models real-world enterprise.
 - Entities (e.g., students, courses)
 - Relationships (e.g., Madonna is taking CS564)
- ❖ A Database Management System (DBMS) is a software package designed to store and manage databases.



Files vs. DBMS

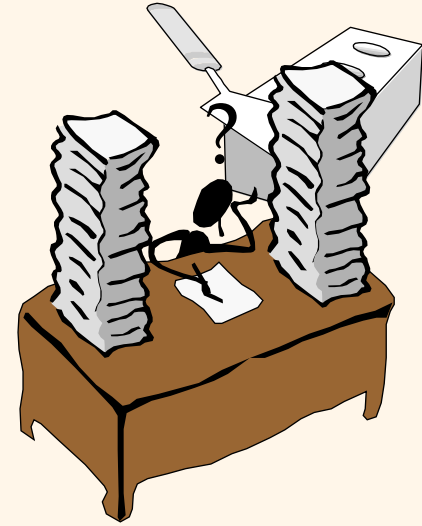
- ❖ Application must stage large datasets between main memory and secondary storage (e.g., buffering, page-oriented access, 32-bit addressing, etc.)
- ❖ Special code for different queries
- ❖ Must protect data from inconsistency due to multiple concurrent users
- ❖ Crash recovery
- ❖ Security and access control

Why Use a DBMS?

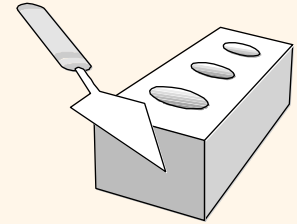


- ❖ Data independence and efficient access.
- ❖ Reduced application development time.
- ❖ Data integrity and security.
- ❖ Uniform data administration.
- ❖ Concurrent access, recovery from crashes.

Why Study Databases??

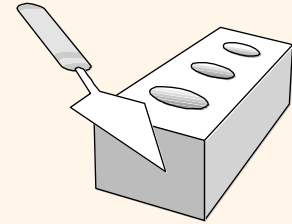


- ❖ Shift from computation to information
 - at the “low end”: scramble to webspace (a mess!)
 - at the “high end”: scientific applications
- ❖ Datasets increasing in diversity and volume.
 - Digital libraries, interactive video, Human Genome project, EOS project
 - ... need for DBMS exploding
- ❖ DBMS encompasses most of CS
 - OS, languages, theory, “A”I, multimedia, logic



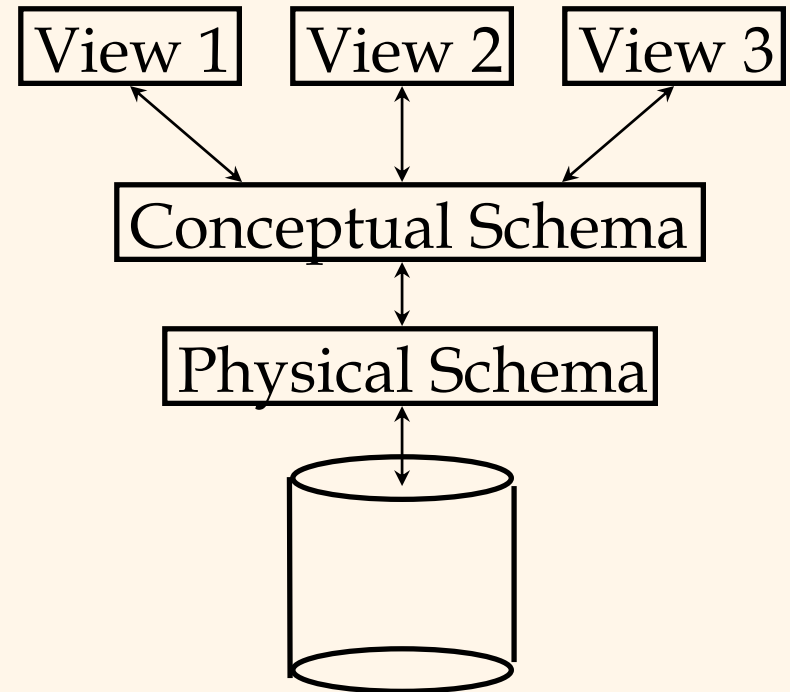
Data Models

- ❖ A *data model* is a collection of concepts for describing data.
- ❖ A *schema* is a description of a particular collection of data, using the a given data model.
- ❖ The *relational model of data* is the most widely used model today.
 - Main concept: *relation*, basically a table with rows and columns.
 - Every relation has a *schema*, which describes the columns, or fields.

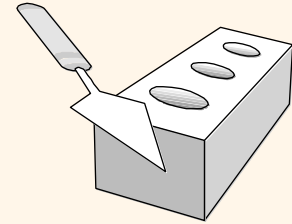


Levels of Abstraction

- ❖ Many views, single conceptual (logical) schema and physical schema.
 - Views describe how users see the data.
 - Conceptual schema defines logical structure
 - Physical schema describes the files and indexes used.



* *Schemas are defined using DDL; data is modified/queried using DML.*



Example: University Database

❖ Conceptual schema:

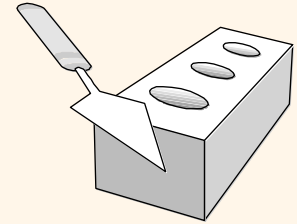
- *Students*(*sid*: string, *name*: string, *login*: string, *age*: integer, *gpa*:real)
- *Courses*(*cid*: string, *cname*:string, *credits*:integer)
- *Enrolled*(*sid*:string, *cid*:string, *grade*:string)

❖ Physical schema:

- Relations stored as unordered files.
- Index on first column of Students.

❖ External Schema (View):

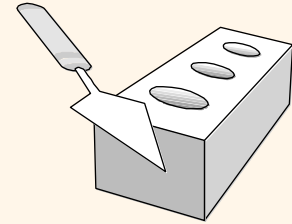
- *Course_info*(*cid*:string,*enrollment*:integer)



*Data Independence **

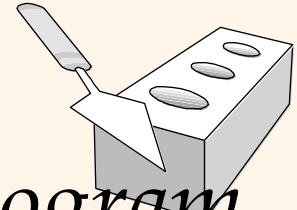
- ❖ Applications insulated from how data is structured and stored.
- ❖ *Logical data independence*: Protection from changes in *logical* structure of data.
- ❖ *Physical data independence*: Protection from changes in *physical* structure of data.

** One of the most important benefits of using a DBMS!*



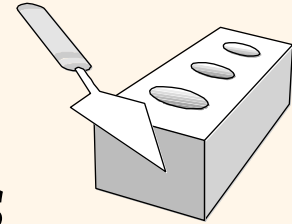
Concurrency Control

- ❖ Concurrent execution of user programs is essential for good DBMS performance.
 - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- ❖ Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed.
- ❖ DBMS ensures such problems don't arise: users can pretend they are using a single-user system.



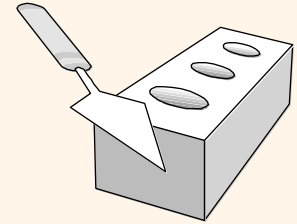
Transaction: An Execution of a DB Program

- ❖ Key concept is transaction, which is an *atomic* sequence of database actions (reads/writes).
- ❖ Each transaction, executed completely, must leave the DB in a consistent state if DB is consistent when the transaction begins.
 - Users can specify some simple integrity constraints on the data, and the DBMS will enforce these constraints.
 - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
 - Thus, ensuring that a transaction (run alone) preserves consistency is ultimately the **user's** responsibility!



Scheduling Concurrent Transactions

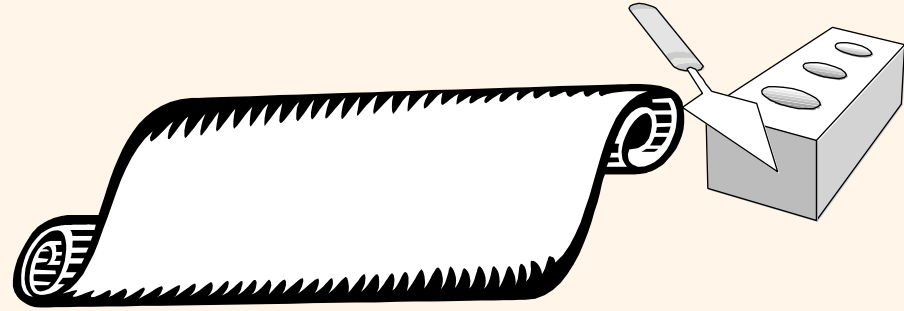
- ❖ DBMS ensures that execution of $\{T_1, \dots, T_n\}$ is equivalent to some serial execution $T_1' \dots T_n'$.
 - Before reading/writing an object, a transaction requests a lock on the object, and waits till the DBMS gives it the lock. All locks are released at the end of the transaction. (Strict 2PL locking protocol.)
 - **Idea:** If an action of T_i (say, writing X) affects T_j (which perhaps reads X), one of them, say T_i , will obtain the lock on X first and T_j is forced to wait until T_i completes; this effectively orders the transactions.
 - What if T_j already has a lock on Y and T_i later requests a lock on Y ? (Deadlock!) T_i or T_j is aborted and restarted!



Ensuring Atomicity

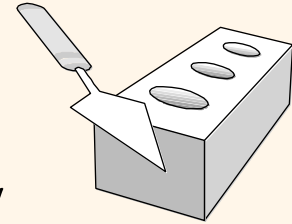
- ❖ DBMS ensures *atomicity* (all-or-nothing property) even if system crashes in the middle of a Xact.
- ❖ **Idea:** Keep a log (history) of all actions carried out by the DBMS while executing a set of Xacts:
 - **Before** a change is made to the database, the corresponding log entry is forced to a safe location. (WAL protocol; OS support for this is often inadequate.)
 - After a crash, the effects of partially executed transactions are undone using the log. (Thanks to WAL, if log entry wasn't saved before the crash, corresponding change was not applied to database!)

The Log

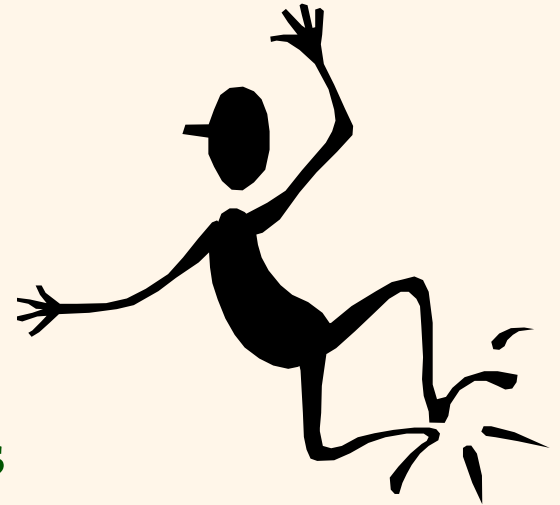


- ❖ The following actions are recorded in the log:
 - *Ti writes an object*: the old value and the new value.
 - Log record must go to disk before the changed page!
 - *Ti commits/aborts*: a log record indicating this action.
- ❖ Log records chained together by Xact id, so it's easy to undo a specific Xact (e.g., to resolve a deadlock).
- ❖ Log is often *duplexed* and *archived* on “stable” storage.
- ❖ All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Databases make these folks happy ...



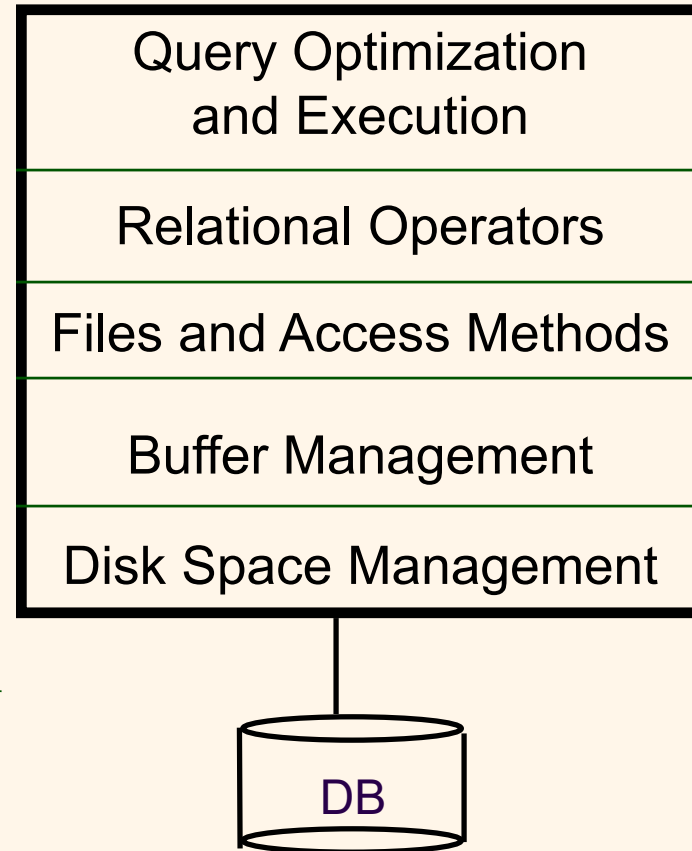
- ❖ End users and DBMS vendors
- ❖ DB application programmers
 - E.g. smart webmasters
- ❖ Database administrator (DBA)
 - Designs logical / physical schemas
 - Handles security and authorization
 - Data availability, crash recovery
 - Database tuning as needs evolve



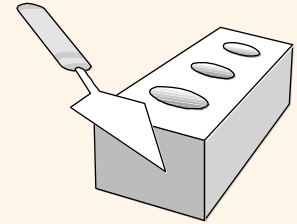
Must understand how a DBMS works!

Structure of a DBMS

- ❖ A typical DBMS has a layered architecture.
- ❖ The figure does not show the concurrency control and recovery components.
- ❖ This is one of several possible architectures; each system has its own variations.

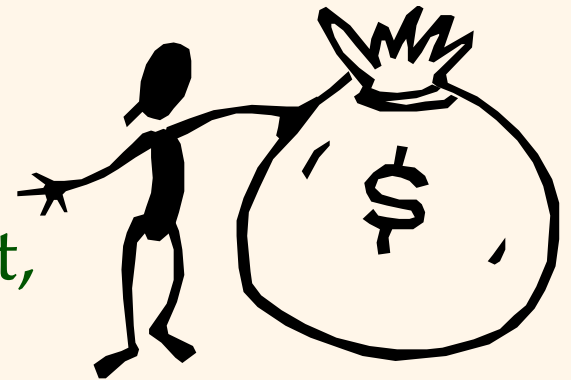


These layers must consider concurrency control and recovery



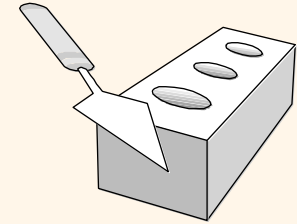
Summary

- ❖ DBMS used to maintain, query large datasets.
- ❖ Benefits include recovery from system crashes, concurrent access, quick application development, data integrity and security.
- ❖ Levels of abstraction give data independence.
- ❖ A DBMS typically has a layered architecture.
- ❖ DBAs hold responsible jobs and are **well-paid!**
- ❖ DBMS R&D is one of the broadest, most exciting areas in CS.



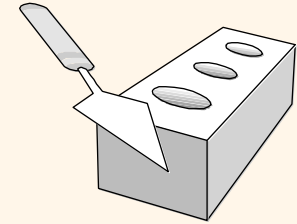
2

- Entity-Relationship (ER) Model (**Ch.2**)



The Entity-Relationship Model

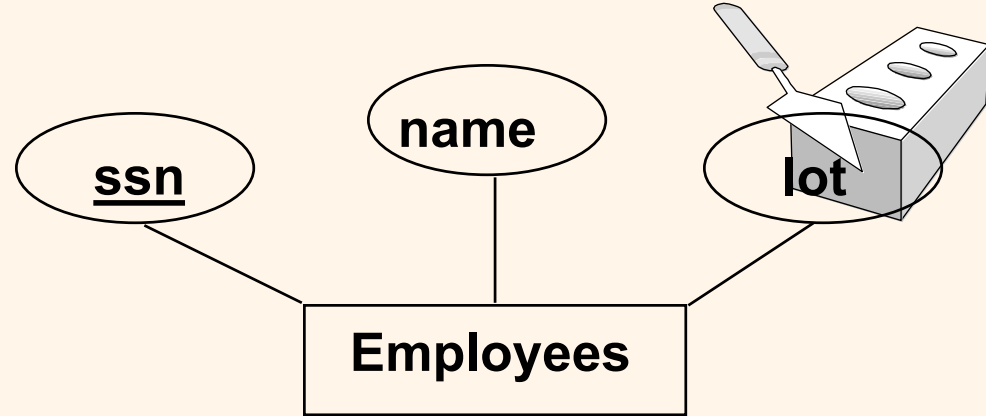
Chapter 2



Overview of Database Design

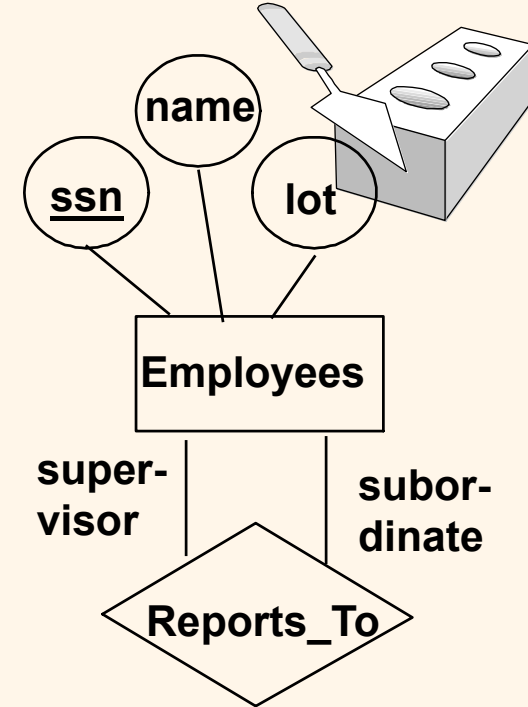
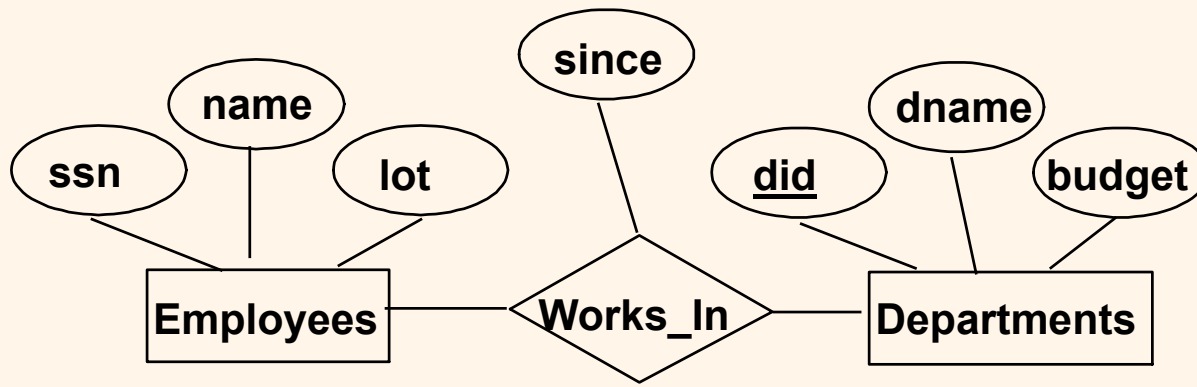
- ❖ Conceptual design: (*ER Model is used at this stage.*)
 - What are the *entities* and *relationships* in the enterprise?
 - What information about these entities and relationships should we store in the database?
 - What are the *integrity constraints* or *business rules* that hold?
 - A database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

ER Model Basics



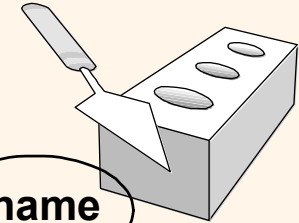
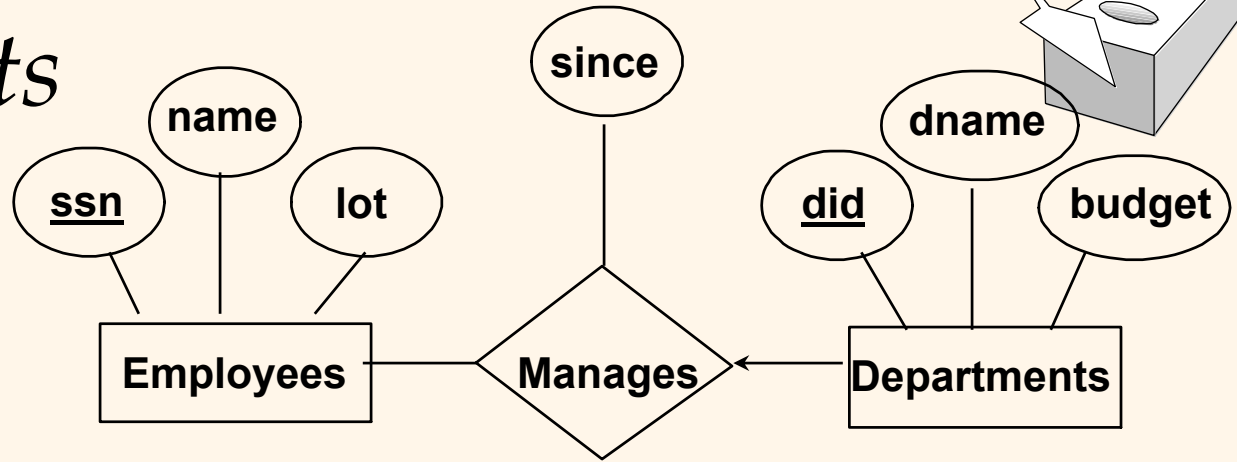
- ❖ Entity: Real-world object distinguishable from other objects. An entity is described (in DB) using a set of attributes.
- ❖ Entity Set: A collection of similar entities. E.g., all employees.
 - All entities in an entity set have the same set of attributes. (Until we consider ISA hierarchies, anyway!)
 - Each entity set has a *key*.
 - Each attribute has a *domain*.

ER Model Basics (Contd.)



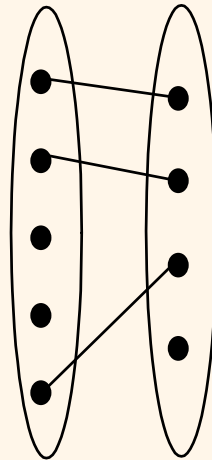
- ❖ **Relationship**: Association among two or more entities. E.g., Attishoo works in Pharmacy department.
- ❖ **Relationship Set**: Collection of similar relationships.
 - An n-ary relationship set R relates n entity sets $E_1 \dots E_n$; each relationship in R involves entities $e_1 \in E_1, \dots, e_n \in E_n$
 - Same entity set could participate in different relationship sets, or in different “roles” in same set.

Key Constraints

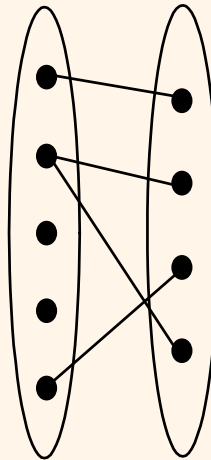


❖ Consider Works_In:
An employee can work in many departments; a dept can have many employees.

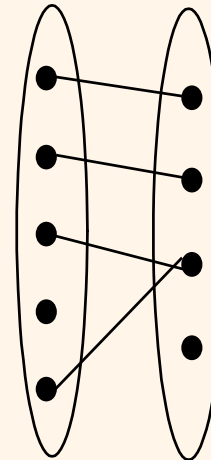
❖ In contrast, each dept has at most one manager, according to the key constraint on Manages.



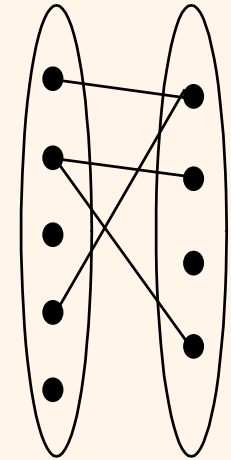
1-to-1



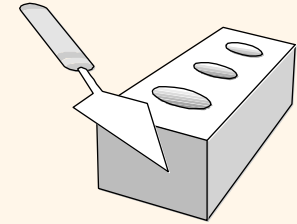
1-to Many



Many-to-1

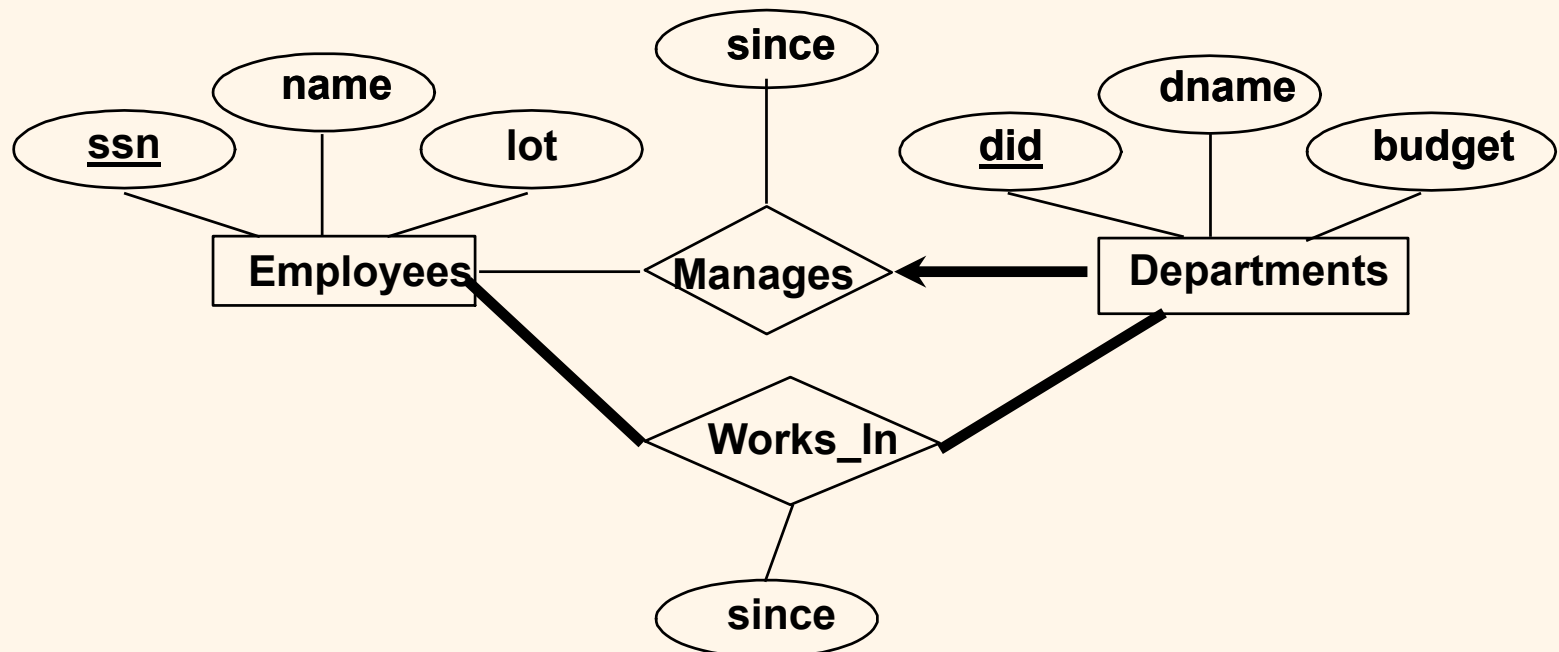


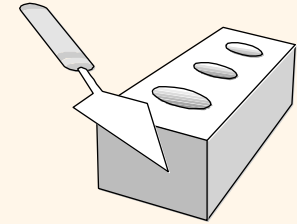
Many-to-Many



Participation Constraints

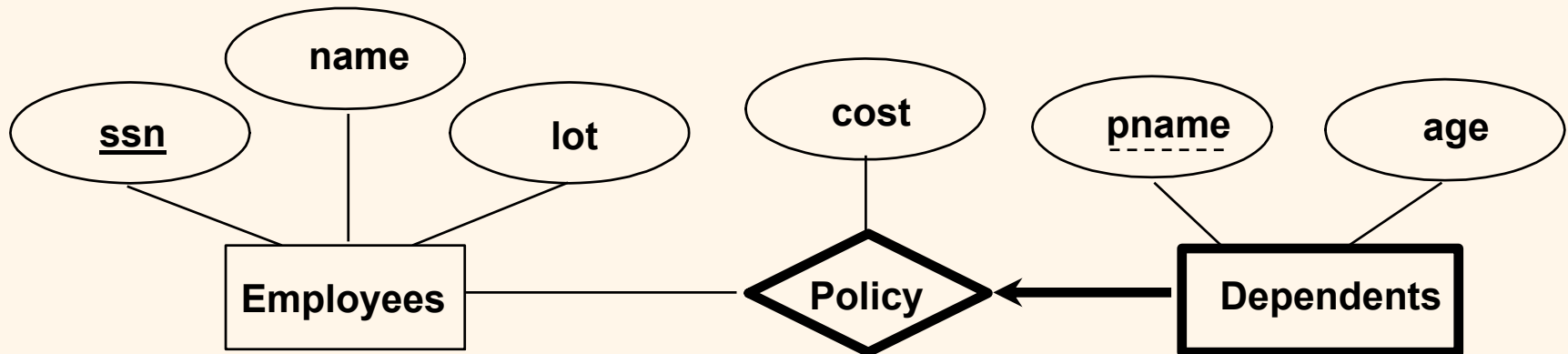
- ❖ Does every department have a manager?
 - If so, this is a participation constraint: the participation of Departments in Manages is said to be *total* (vs. *partial*).
 - Every *did* value in Departments table must appear in a row of the Manages table (with a non-null *ssn* value!)





Weak Entities

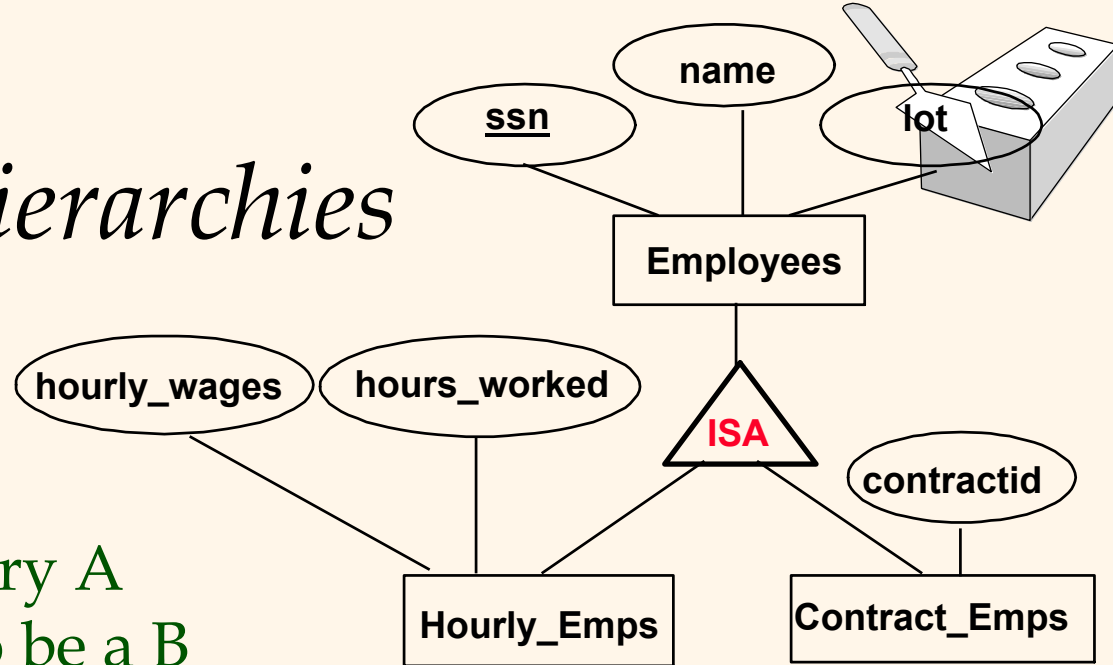
- ❖ A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.
 - Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).
 - Weak entity set must have total participation in this *identifying*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



ISA ('is a') Hierarchies

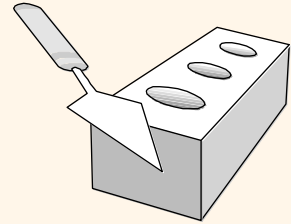
✓ As in C++, or other PLs, attributes are inherited.

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



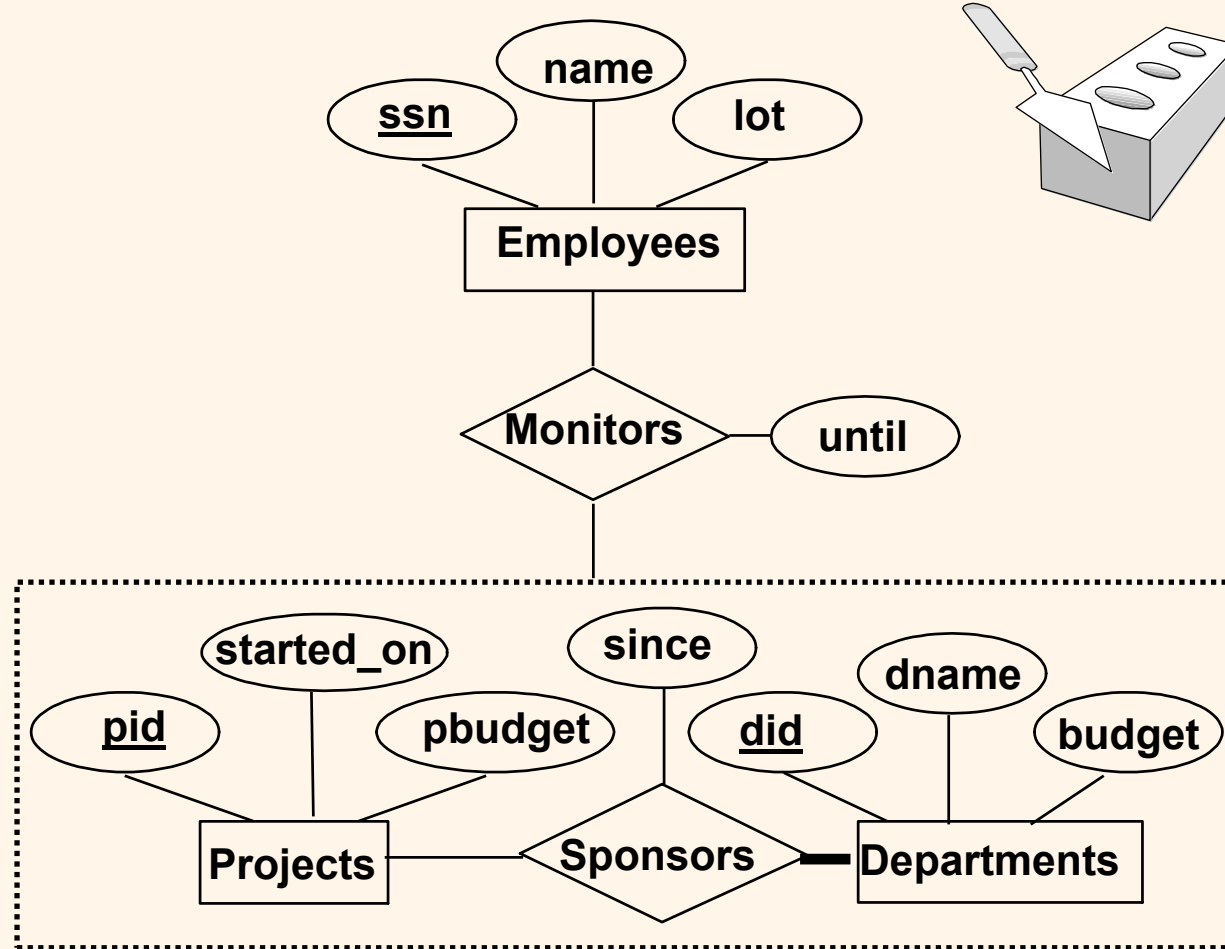
- ❖ *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? (*Allowed/disallowed*)
- ❖ *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? (*Yes/no*)
- ❖ Reasons for using ISA:
 - To add descriptive attributes specific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

Aggregation

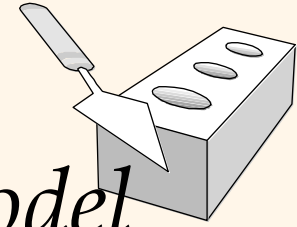


❖ Used when we have to model a relationship involving (entity sets and) a *relationship set*.

- Aggregation allows us to treat a relationship set as an entity set for purposes of participation in (other) relationships.



- * *Aggregation vs. ternary relationship:*
 - ✓ Monitors is a distinct relationship, with a descriptive attribute.
 - ✓ Also, can say that each sponsorship is monitored by at most one employee.



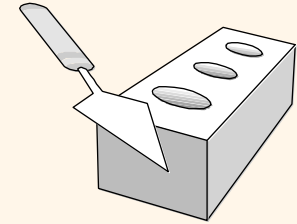
Conceptual Design Using the ER Model

❖ Design choices:

- 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?

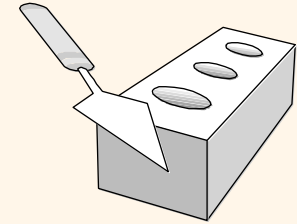
❖ Constraints in the ER Model:

- A lot of data semantics can (and should) be captured.
- But some constraints cannot be captured in ER diagrams.



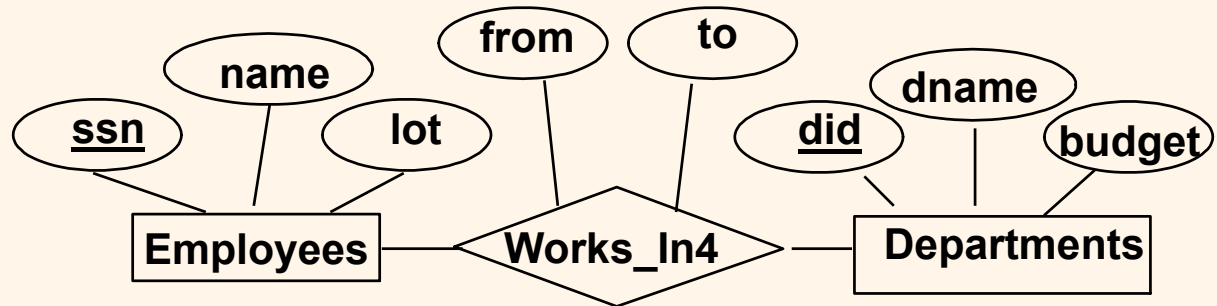
Entity vs. Attribute

- ❖ 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).

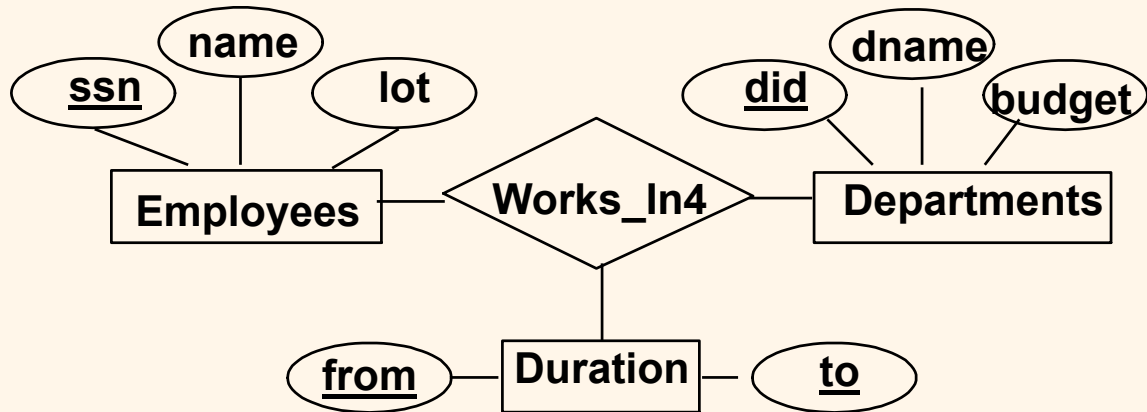


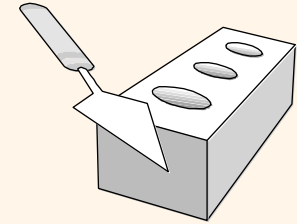
Entity vs. Attribute (Contd.)

❖ Works_In4 does not allow an employee to work in a department for two or more periods.



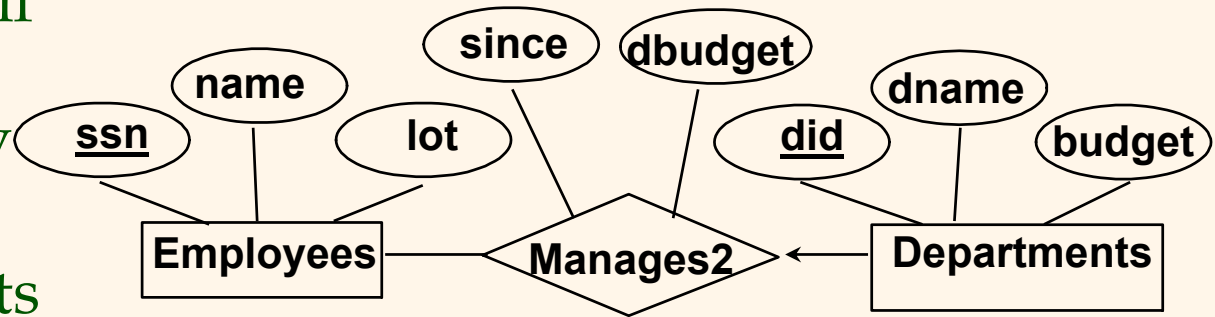
❖ Similar to the problem of wanting to record several addresses for an employee: We want to record *several values of the descriptive attributes for each instance of this relationship*. Accomplished by introducing new entity set, Duration.





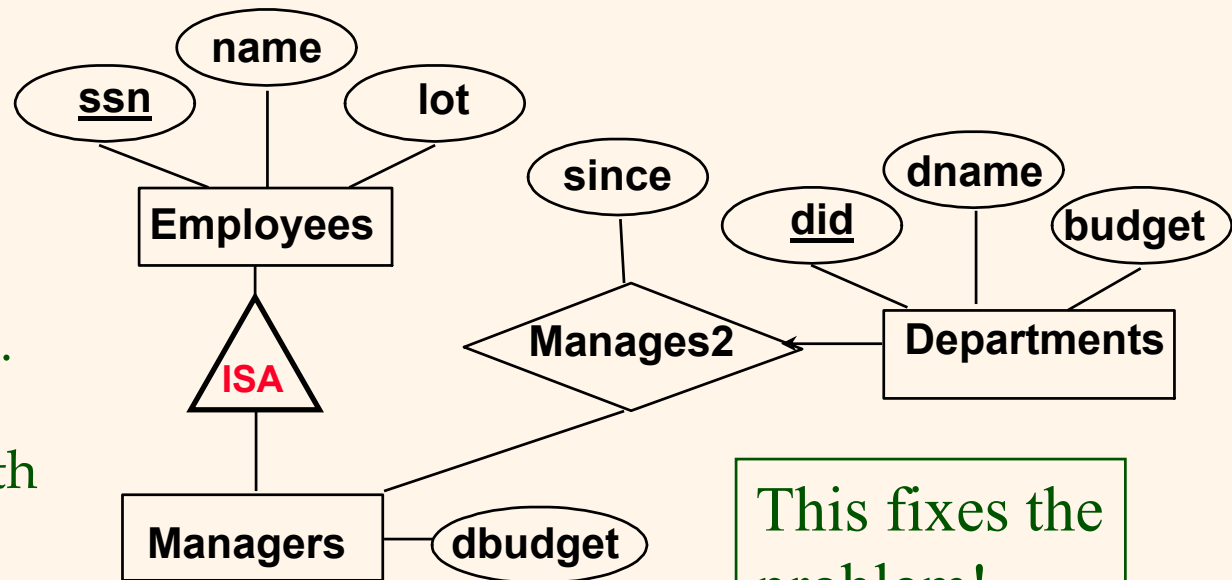
Entity vs. Relationship

❖ First ER diagram OK if a manager gets a separate discretionary budget for each dept.

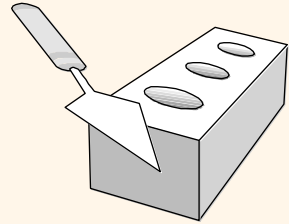


❖ What if a manager gets a discretionary budget that covers *all* managed depts?

- **Redundancy:** *dbudget* stored for each dept managed by manager.
- **Misleading:** Suggests *dbudget* associated with department-mgr combination.

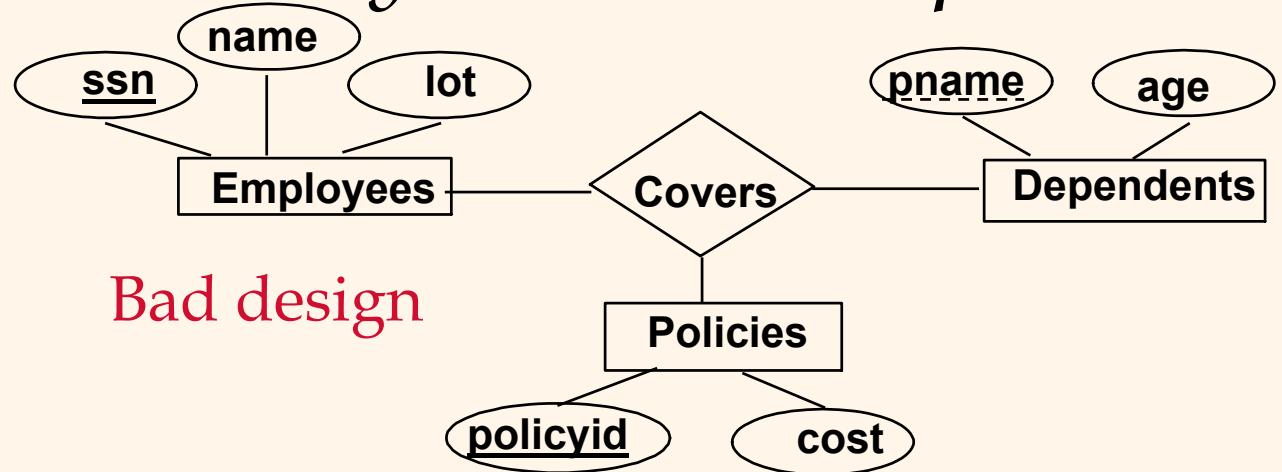


This fixes the problem!

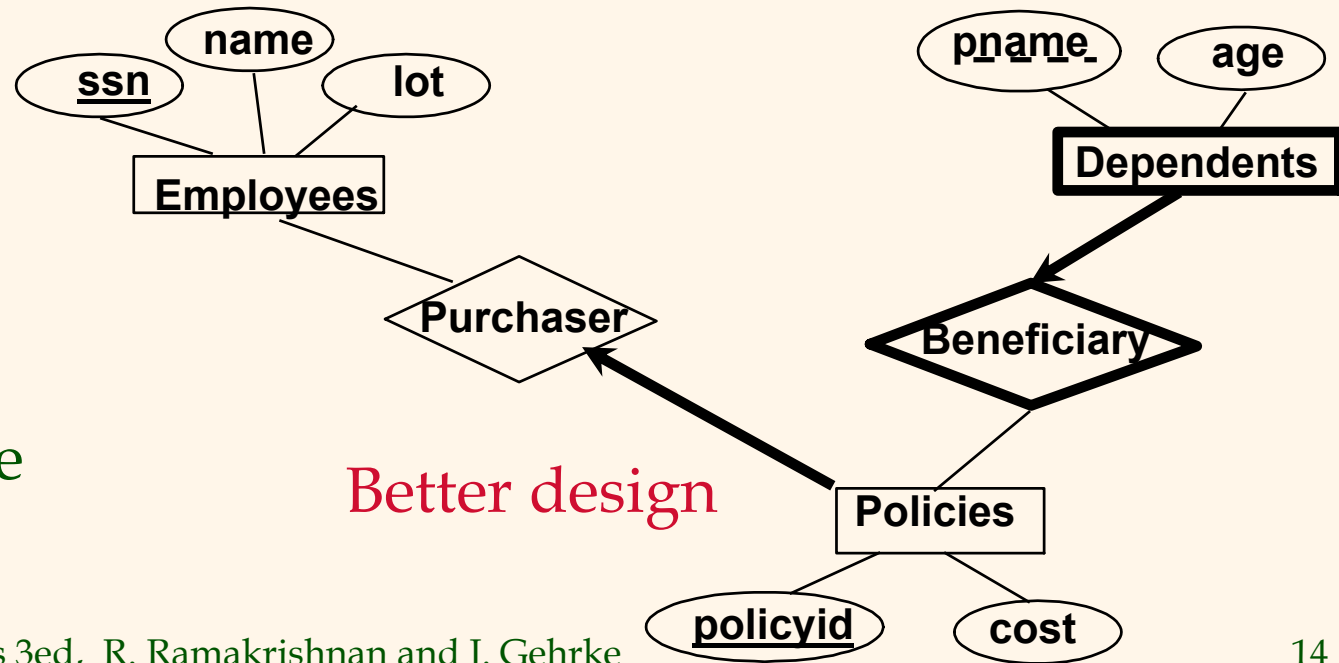


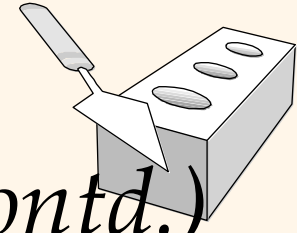
Binary vs. Ternary Relationships

❖ If each policy is owned by just 1 employee, and each dependent is tied to the covering policy, first diagram is inaccurate.



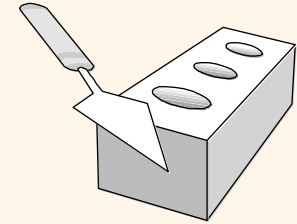
❖ What are the additional constraints in the 2nd diagram?





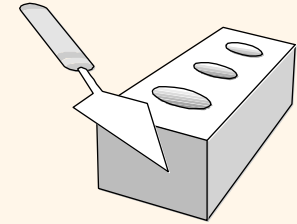
Binary vs. Ternary Relationships (Contd.)

- ❖ 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*. No combination of binary relationships is an adequate substitute:
 - 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*?



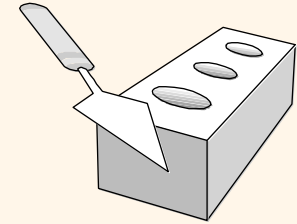
Summary of Conceptual Design

- ❖ *Conceptual design follows requirements analysis,*
 - Yields a high-level description of data to be stored
- ❖ ER model popular for conceptual design
 - Constructs are expressive, close to the way people think about their applications.
- ❖ Basic constructs: *entities, relationships, and attributes* (of entities and relationships).
- ❖ Some additional constructs: *weak entities, ISA hierarchies, and aggregation.*
- ❖ Note: There are many variations on ER model.



Summary of ER (Contd.)

- ❖ Several kinds of integrity constraints can be expressed in the ER model: *key constraints, participation constraints, and overlap/covering constraints* for ISA hierarchies. Some *foreign key constraints* are also implicit in the definition of a relationship set.
 - Some constraints (notably, *functional dependencies*) cannot be expressed in the ER model.
 - Constraints play an important role in determining the best database design for an enterprise.

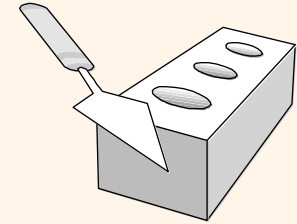


Summary of ER (Contd.)

- ❖ ER design is *subjective*. There are often many ways to model a given scenario! Analyzing alternatives can be tricky, especially for a large enterprise. Common choices include:
 - Entity vs. attribute, entity vs. relationship, binary or n-ary relationship, whether or not to use ISA hierarchies, and whether or not to use aggregation.
- ❖ Ensuring good database design: resulting relational schema should be analyzed and refined further. FD information and normalization techniques are especially useful.

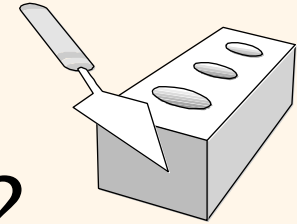
3

- Relational Data Model (**Ch.3**)



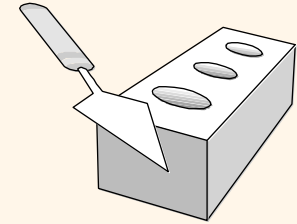
The Relational Model

Chapter 3



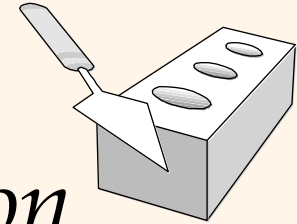
Why Study the Relational Model?

- ❖ Most widely used model.
 - Vendors: IBM, Informix, Microsoft, Oracle, Sybase, etc.
- ❖ “Legacy systems” in older models
 - E.G., IBM’s IMS
- ❖ Recent competitor: object-oriented model
 - ObjectStore, Versant, Ontos
 - A synthesis emerging: *object-relational model*
 - Informix Universal Server, UniSQL, O2, Oracle, DB2



Relational Database: Definitions

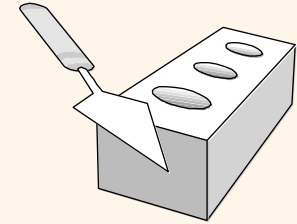
- ❖ *Relational database*: a set of *relations*
- ❖ *Relation*: made up of 2 parts:
 - *Instance* : a *table*, with rows and columns.
#Rows = *cardinality*, #fields = *degree / arity*.
 - *Schema* : specifies name of relation, plus name and type of each column.
 - E.G. Students(*sid*: string, *name*: 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

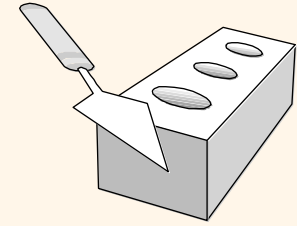
sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

- ❖ Cardinality = 3, degree = 5, all rows distinct
- ❖ Do all columns in a relation instance have to be distinct?



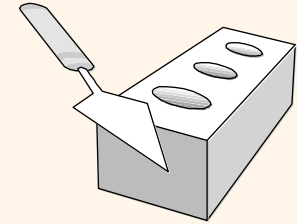
Relational Query Languages

- ❖ A major strength of the relational model: supports simple, powerful *querying* of data.
- ❖ Queries can be written intuitively, and the DBMS is responsible for efficient evaluation.
 - The key: precise semantics for relational queries.
 - Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.



The SQL Query Language

- ❖ Developed by IBM (system R) in the 1970s
- ❖ Need for a standard since it is used by many vendors
- ❖ Standards:
 - SQL-86
 - SQL-89 (minor revision)
 - SQL-92 (major revision)
 - SQL-99 (major extensions, current standard)



The SQL Query Language

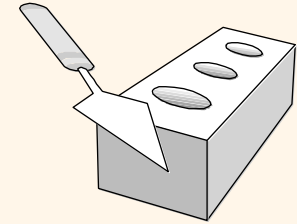
- ❖ To find all 18 year old students, we can write:

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

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

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

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



Querying Multiple Relations

❖ What does the following query compute?

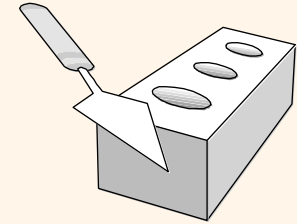
```
SELECT S.name, E.cid  
FROM Students S, Enrolled E  
WHERE S.sid=E.sid AND E.grade="A"
```

Given the following instance of Enrolled (is this possible if the DBMS ensures referential integrity?):

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

we get:

S.name	E.cid
Smith	Topology112



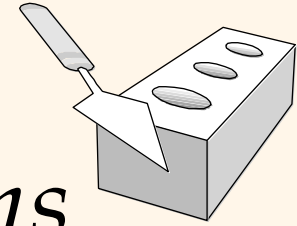
Creating Relations in SQL

- ❖ Creates the Students relation. Observe that the type (**domain**) 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



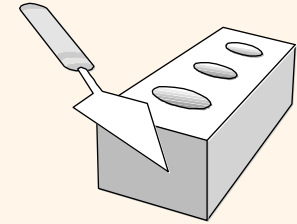
DROP TABLE Students

- ❖ 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

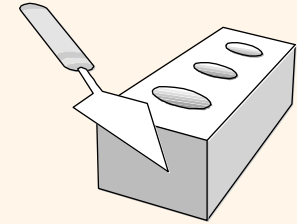
- ❖ Can insert a single tuple using:

```
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):

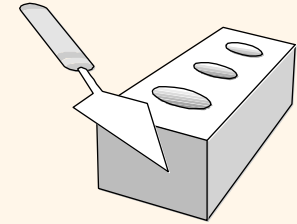
```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```

** Powerful variants of these commands are available; more later!*



Integrity Constraints (ICs)

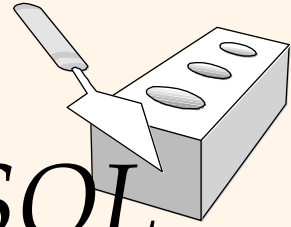
- ❖ **IC:** condition that must be true for *any* instance of the database; e.g., *domain constraints*.
 - ICs are specified when schema is defined.
 - ICs are checked when relations are modified.
- ❖ A *legal* instance of a relation is one that satisfies all specified ICs.
 - DBMS should not allow illegal 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*.

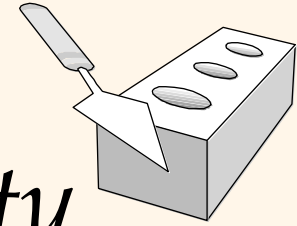
❖ “For a given student and course, there is a single grade.” **vs.**
“Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”

❖ Used carelessly, an IC can prevent the storage of database instances that arise in practice!

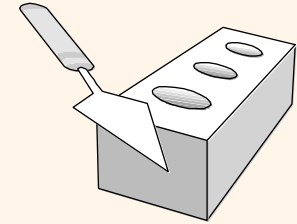
```
CREATE TABLE Enrolled
(sid CHAR(20)
 cid CHAR(20),
 grade CHAR(2),
 PRIMARY KEY (sid,cid) )
```

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

Foreign Keys, Referential Integrity



- ❖ Foreign key: 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`.
- ❖ E.g. *sid* is a foreign key referring to **Students**:
 - Enrolled(*sid*: string, *cid*: string, *grade*: string)
 - If all foreign key constraints are enforced, referential integrity is achieved, i.e., no dangling references.
 - Can you name a data model w/o referential integrity?
 - Links in HTML!



Foreign Keys in SQL

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

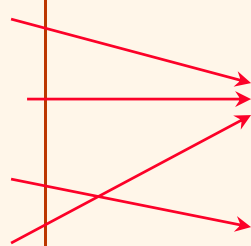
```
CREATE TABLE Enrolled
  (sid CHAR(20), cid CHAR(20), grade CHAR(2),
   PRIMARY KEY (sid,cid),
   FOREIGN KEY (sid) REFERENCES Students )
```

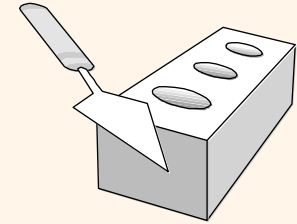
Enrolled

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

Students

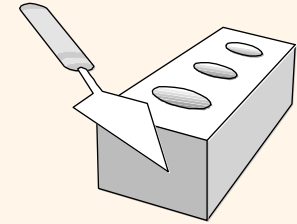
sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8





Enforcing Referential Integrity

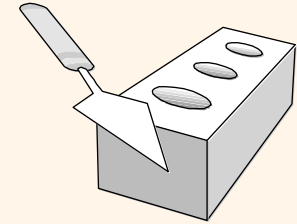
- ❖ Consider Students and Enrolled; *sid* in Enrolled is a foreign key that references Students.
- ❖ What should be done if an Enrolled tuple with a non-existent student id is inserted? (*Reject it!*)
- ❖ What should be done if a Students tuple is deleted?
 - Also delete all Enrolled tuples that refer 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)

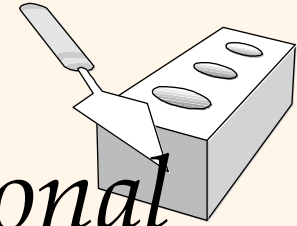
```
CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid)
REFERENCES Students
ON DELETE CASCADE
ON UPDATE SET DEFAULT )
```



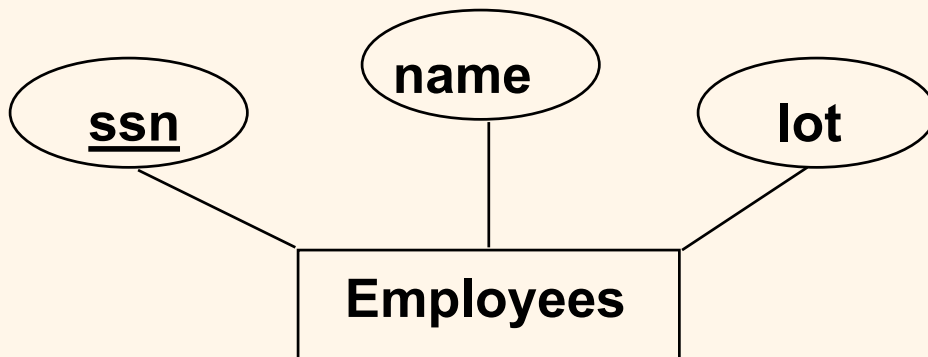
Where do ICs Come From?

- ❖ ICs are based upon the semantics of the real-world enterprise that is being described in the database relations.
- ❖ We can check a database instance to see if an IC is violated, but we can **NEVER** infer that an IC is true by looking at an instance.
 - An IC is a statement about *all possible* instances!
 - From example, we know *name* is not a key, but the assertion that *sid* is a key is given to us.
- ❖ Key and foreign key ICs are the most common; more general ICs supported too.

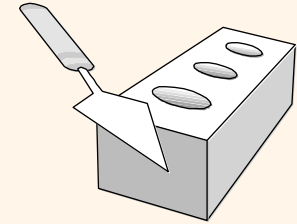
Logical DB Design: ER to Relational



❖ Entity sets to tables:



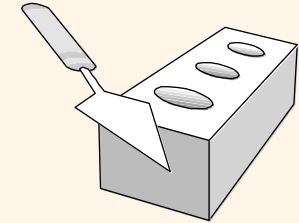
```
CREATE TABLE Employees  
  (ssn CHAR(11),  
   name CHAR(20),  
   lot INTEGER,  
   PRIMARY KEY (ssn))
```



Relationship Sets to Tables

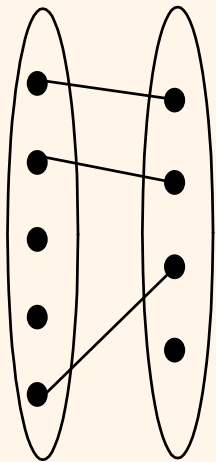
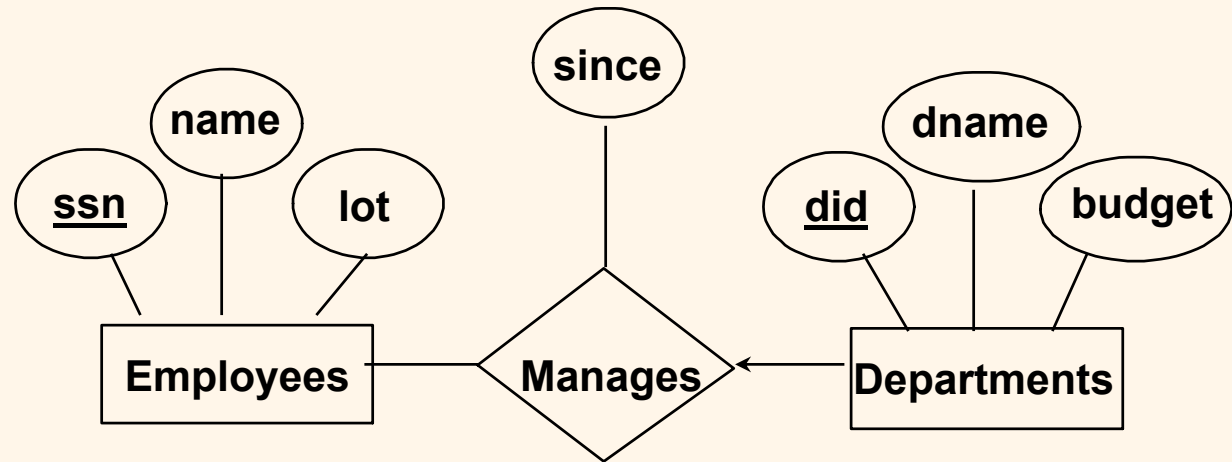
- ❖ In translating a relationship set to a relation, attributes of the relation must include:
 - Keys for each participating entity set (as foreign keys).
 - This set of attributes forms a *superkey* for the relation.
 - All descriptive attributes.

```
CREATE TABLE Works_In(  
    ssn CHAR(1),  
    did INTEGER,  
    since DATE,  
    PRIMARY KEY (ssn, did),  
    FOREIGN KEY (ssn)  
        REFERENCES Employees,  
    FOREIGN KEY (did)  
        REFERENCES Departments)
```

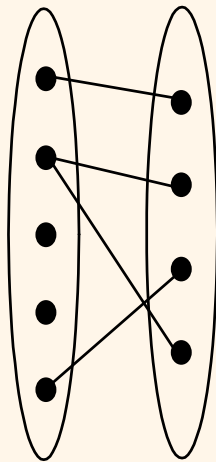


Review: Key Constraints

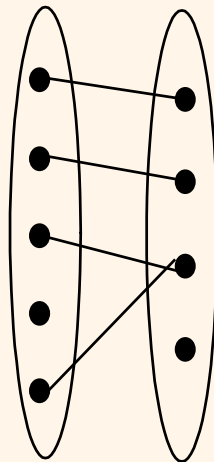
- ❖ Each dept has at most one manager, according to the key constraint on Manages.



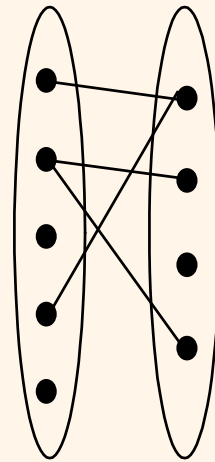
1-to-1



1-to Many

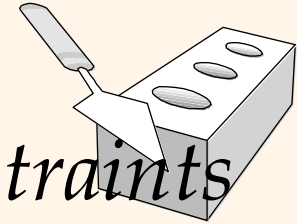


Many-to-1



Many-to-Many

Translation to relational model?

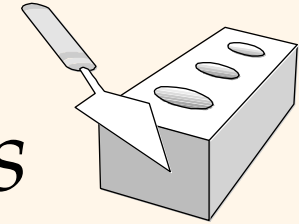


Translating ER Diagrams with Key Constraints

- ❖ 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.

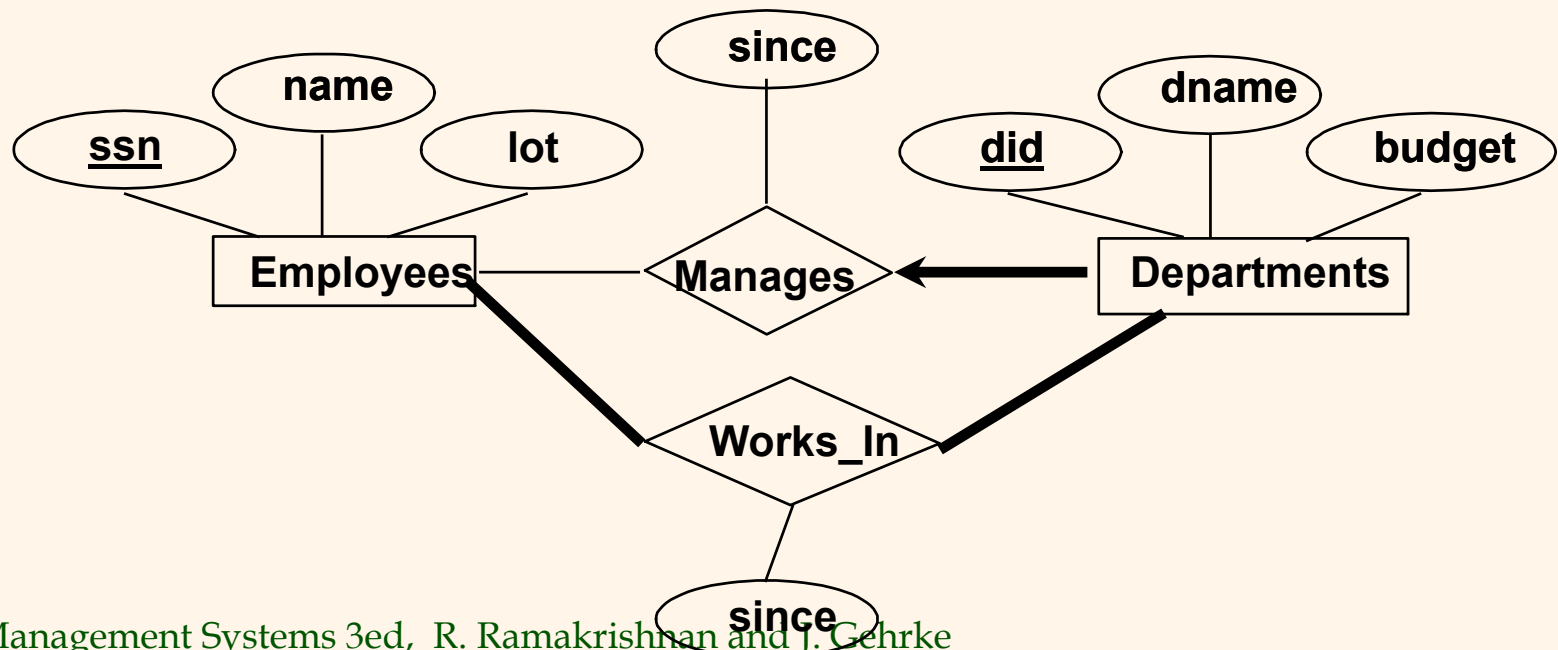
```
CREATE TABLE Manages(  
  ssn CHAR(11),  
  did INTEGER,  
  since DATE,  
  PRIMARY KEY (did),  
  FOREIGN KEY (ssn) REFERENCES Employees,  
  FOREIGN KEY (did) REFERENCES Departments)
```

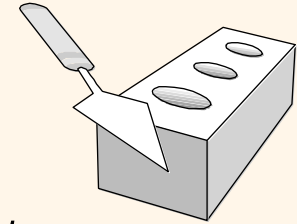
```
CREATE TABLE Dept_Mgr(  
  did INTEGER,  
  dname CHAR(20),  
  budget REAL,  
  ssn CHAR(11),  
  since DATE,  
  PRIMARY KEY (did),  
  FOREIGN KEY (ssn) REFERENCES Employees)
```



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!)

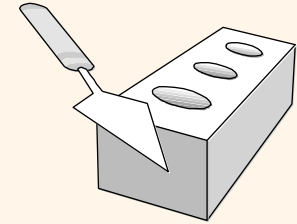




Participation Constraints in SQL

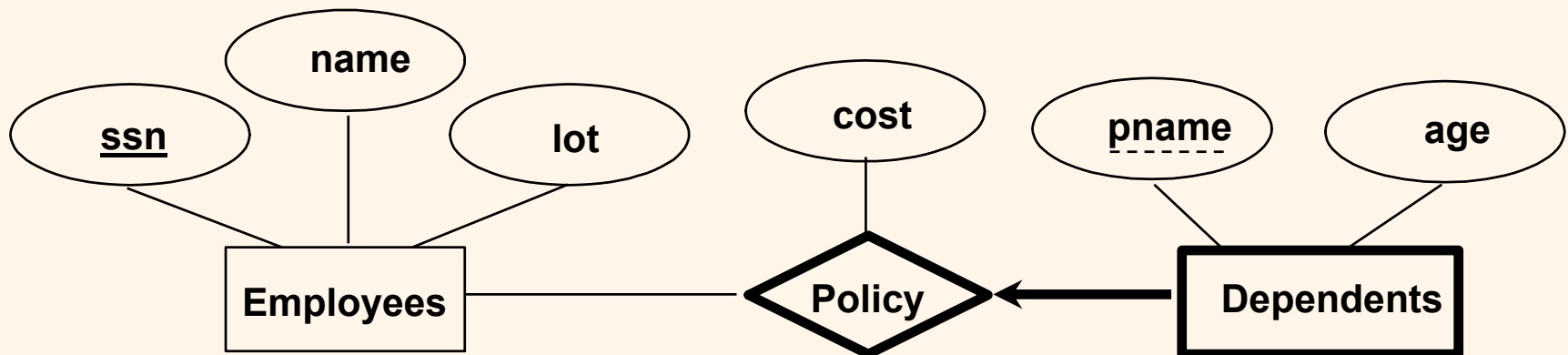
- ❖ We can capture participation constraints involving one entity set in a binary relationship, but little else (without resorting to CHECK constraints).

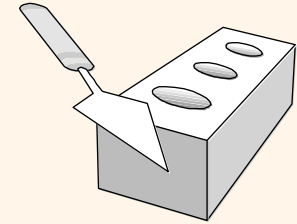
```
CREATE TABLE Dept_Mgr(  
  did INTEGER,  
  dname CHAR(20),  
  budget REAL,  
  ssn CHAR(11) NOT NULL,  
  since DATE,  
  PRIMARY KEY (did),  
  FOREIGN KEY (ssn) REFERENCES Employees,  
  ON DELETE NO ACTION)
```



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.



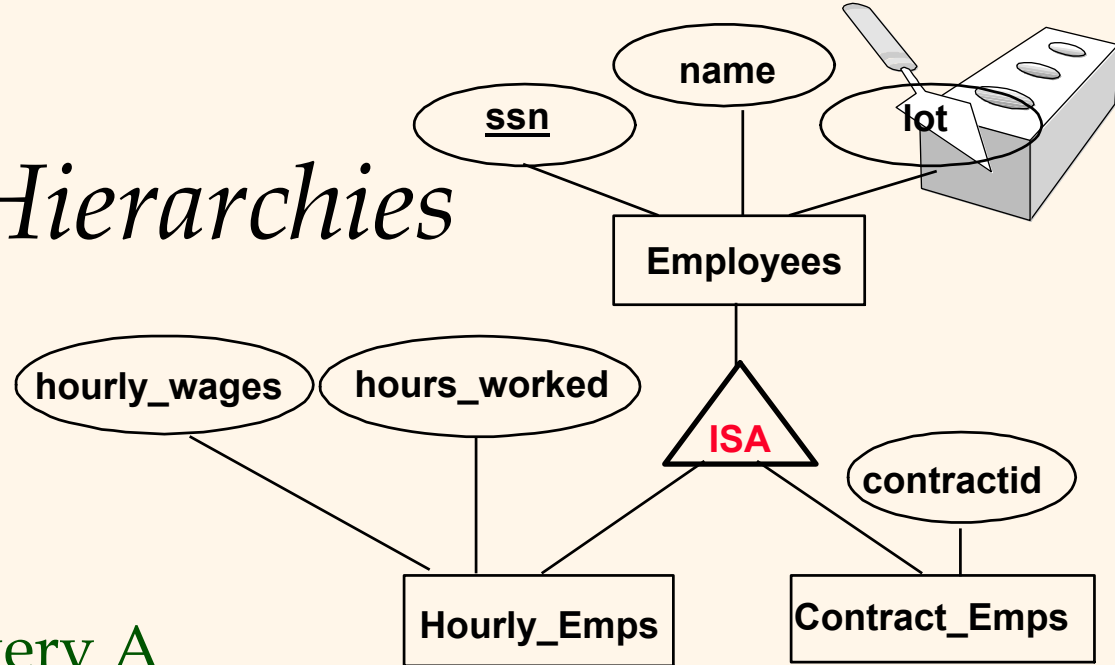


Translating Weak Entity Sets

- ❖ 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)
```

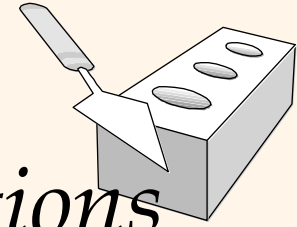
Review: ISA Hierarchies



❖ 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

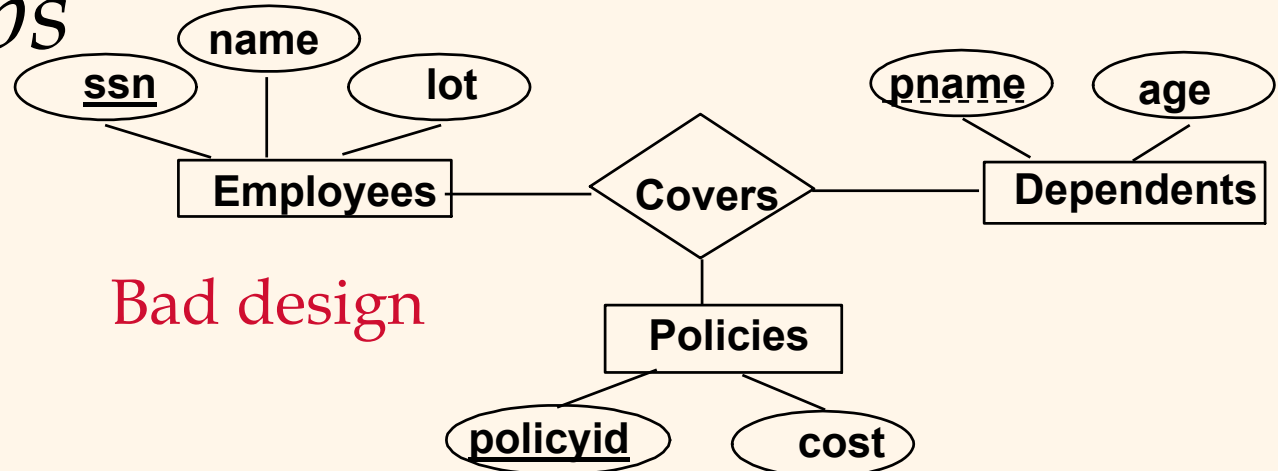
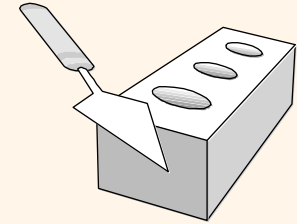
❖ *General approach:*

- 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.

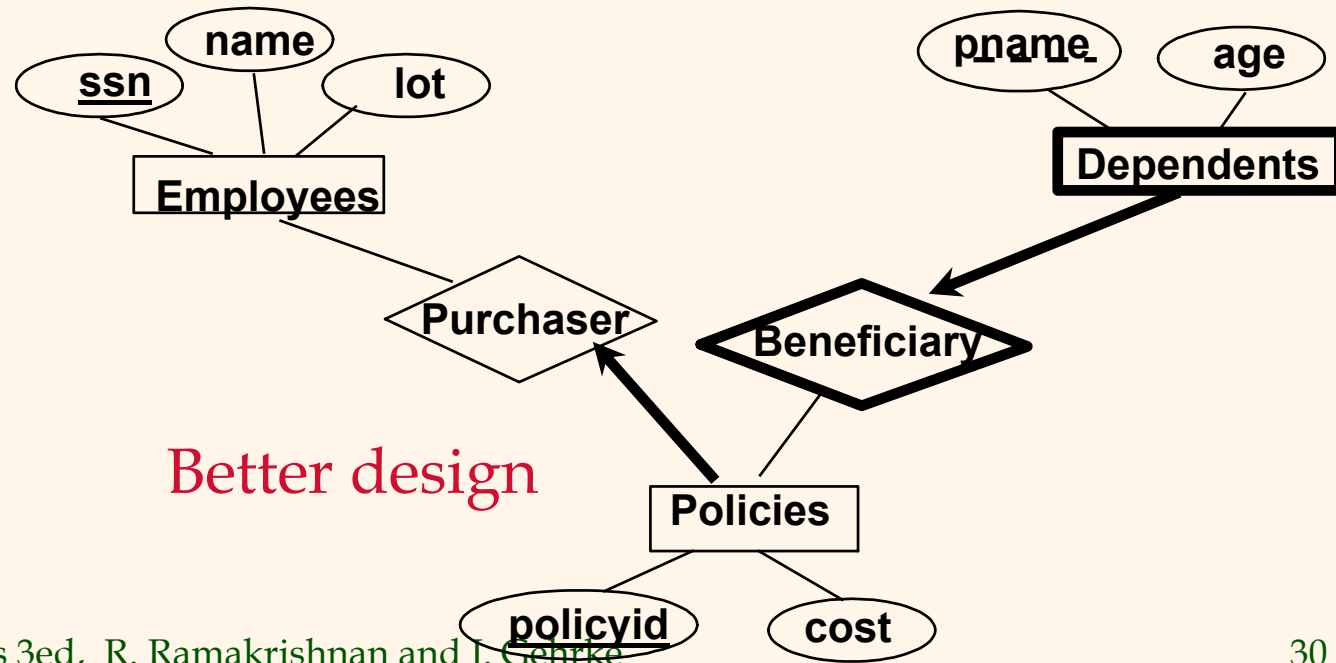
❖ *Alternative: Just Hourly_Emps and Contract_Emps.*

- *Hourly_Emps*: *ssn*, *name*, *lot*, *hourly_wages*, *hours_worked*.
- Each employee must be in one of these two subclasses.

Review: Binary vs. Ternary Relationships

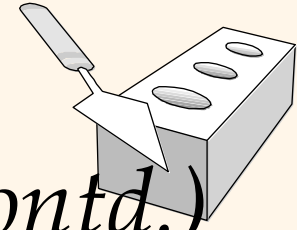


Bad design



Better design

❖ 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.

```
CREATE TABLE Policies (
```

```
  policyid INTEGER,
```

```
  cost REAL,
```

```
  ssn CHAR(11) NOT NULL,
```

```
  PRIMARY KEY (policyid).
```

```
  FOREIGN KEY (ssn) REFERENCES Employees,  
  ON DELETE CASCADE)
```

❖ Participation constraints lead to **NOT NULL** constraints.

```
CREATE TABLE Dependents (
```

```
  pname CHAR(20),
```

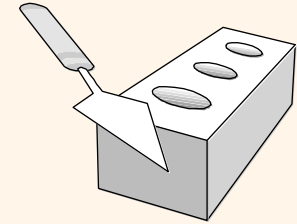
```
  age INTEGER,
```

```
  policyid INTEGER,
```

```
  PRIMARY KEY (pname, policyid).
```

```
  FOREIGN KEY (policyid) REFERENCES Policies,  
  ON DELETE CASCADE)
```

❖ What if Policies is a weak entity set?

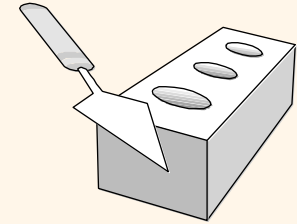


Views

- ❖ A view is just a relation, but we store a *definition*, rather than a set of tuples.

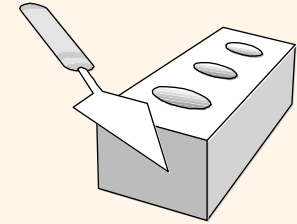
```
CREATE VIEW YoungActiveStudents (name, grade)
AS SELECT S.name, E.grade
FROM Students S, Enrolled E
WHERE S.sid = E.sid and S.age < 21
```

- ❖ Views can be dropped using the **DROP VIEW** command.
 - How to handle **DROP TABLE** if there's a view on the table?
 - DROP TABLE command has options to let the user specify this.



Views and Security

- ❖ Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
 - 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.

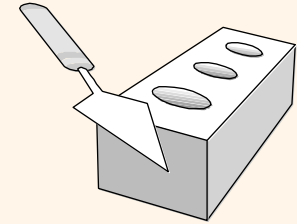


Relational Model: Summary

- ❖ A tabular representation of data.
- ❖ Simple and intuitive, currently the most widely used.
- ❖ Integrity constraints can be specified by the DBA, based on application semantics. DBMS checks for violations.
 - Two important ICs: primary and foreign keys
 - In addition, we *always* have domain constraints.
- ❖ Powerful and natural query languages exist.
- ❖ Rules to translate ER to relational model

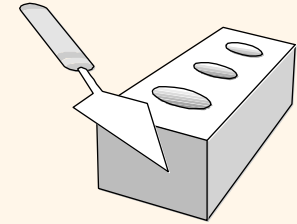
4

- Relational Algebra (**Ch.4**)



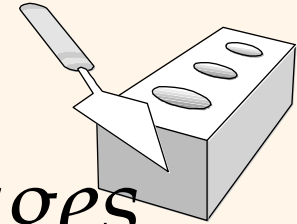
Relational Algebra

Chapter 4, Part A



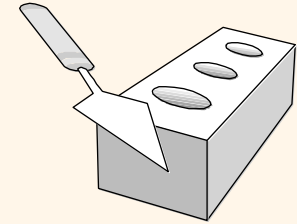
Relational Query Languages

- ❖ Query languages: Allow manipulation and **retrieval of data** from a database.
- ❖ Relational model supports simple, powerful QLs:
 - Strong formal foundation based on logic.
 - Allows for much optimization.
- ❖ Query Languages **!=** programming languages!
 - QLs not expected to be “Turing complete”.
 - QLs not intended to be used for complex calculations.
 - QLs support easy, efficient access to large data sets.



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.)



Preliminaries

- ❖ A query is applied to *relation instances*, and the result of a query is also a relation instance.
 - *Schemas of input* relations for a query are **fixed** (but query will run regardless of instance!)
 - The **schema for the result** of a given query is also **fixed!** Determined by definition of query language constructs.
- ❖ Positional vs. named-field notation:
 - Positional notation easier for formal definitions, named-field notation more readable.
 - Both used in SQL

Example Instances

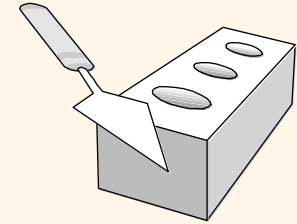


<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

- ❖ “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.

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

<u>sid</u>	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 (\cup) Tuples in reln. 1 and in reln. 2.

❖ Additional operations:

- Intersection, join, division, renaming: Not essential, but (very!) useful.

❖ Since each operation returns a relation, **operations can be composed!** (Algebra is “closed”.)

Projection

- ❖ Deletes attributes that are not in *projection list*.
- ❖ *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. (Why not?)



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*.)

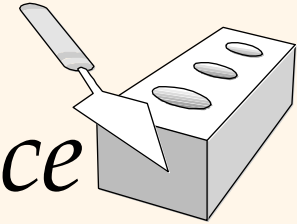


sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$\sigma_{rating > 8}(S_2)$$

sname	rating
yuppy	9
rusty	10

$$\pi_{sname, rating}(\sigma_{rating > 8}(S_2))$$



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

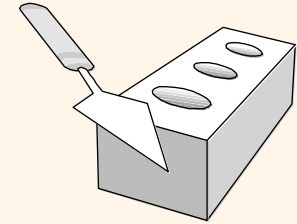
$S1 \cup S2$

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

$S1 \cap S2$

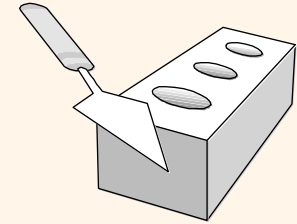


Cross-Product

- ❖ 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*.

(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

- Renaming operator: $\rho (C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$



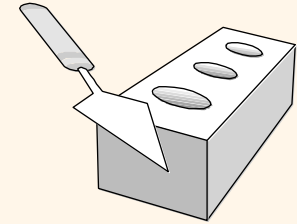
Joins

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

(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

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

- ❖ *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*.



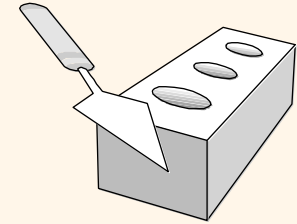
Joins

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

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

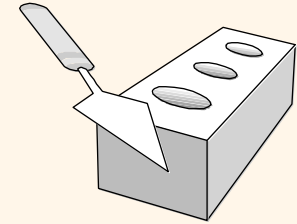
$$S1 \bowtie_{sid} R1$$

- ❖ Result schema similar to cross-product, but only one copy of fields for which equality is specified.
- ❖ Natural Join: Equijoin on *all* common fields.



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 :
 - $A/B = \{ \langle x \rangle \mid \exists \langle x, y \rangle \in A \ \forall \langle y \rangle \in B \}$
 - i.e., **A/B contains all x tuples (sailors) such that for every 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 $x \cup y$ 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

sno
s1
s2
s3
s4

A/B1

pno
p2
p4

B2

sno
s1
s4

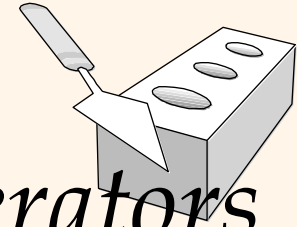
A/B2

pno
p1
p2
p4

B3

sno
s1

A/B3

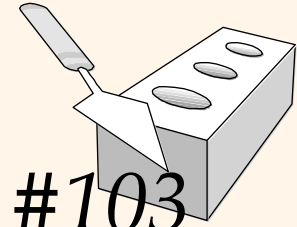


Expressing A/B Using Basic Operators

- ❖ Division is not essential op; just a useful shorthand.
 - (Also true of joins, but joins are so common that systems implement joins specially.)
- ❖ *Idea*: For A/B , compute all x values that are not 'disqualified' by some y value in B .
 - x value is *disqualified* if by attaching y value from B , we obtain an xy tuple that is not in A .

Disqualified x values: $\pi_x ((\pi_x(A) \times B) - A)$

A/B : $\pi_x(A)$ - all disqualified tuples



Find names of sailors who've reserved boat #103

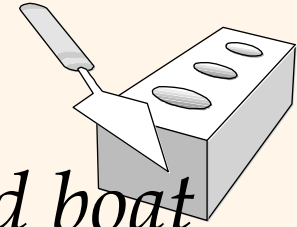
❖ **Solution 1:** $\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$

❖ **Solution 2:** $\rho(Temp1, \sigma_{bid=103} Reserves)$

$\rho(Temp2, Temp1 \bowtie Sailors)$

$\pi_{sname}(Temp2)$

❖ **Solution 3:** $\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$



Find names of sailors who've reserved a red boat

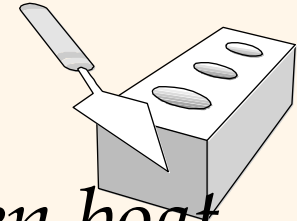
- ❖ Information about boat color only available in Boats; so need an extra join:

$$\pi_{sname}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$$

- ❖ A more efficient solution:

$$\pi_{sname}(\pi_{sid}((\pi_{bid} \sigma_{color='red'} Boats) \bowtie Res) \bowtie Sailors)$$

A query optimizer can find this, given the first solution!

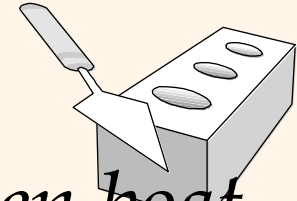


Find sailors who've reserved a red or a green boat

- ❖ Can identify all red or green boats, then find sailors who've reserved one of these boats:

$$\rho (\text{Tempboats}, (\sigma_{\text{color}='red' \vee \text{color}='green'} \text{Boats}))$$
$$\pi_{\text{sname}}(\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors})$$

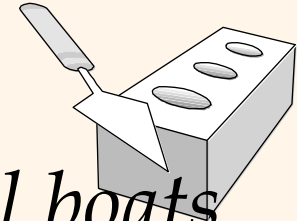
- ❖ Can also define Tempboats using union! (How?)
- ❖ What happens if \vee is replaced by \wedge in this query?



Find sailors who've reserved a red and a green boat

- ❖ Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for *Sailors*):

$$\rho (Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves))$$
$$\rho (Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie Reserves))$$
$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$



Find the names of sailors who've reserved all boats

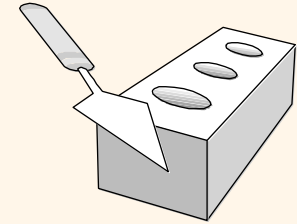
- ❖ Uses division; schemas of the input relations to / must be carefully chosen:

$$\rho (Tempsids, (\pi_{sid,bid} Reserves) / (\pi_{bid} Boats))$$

$$\pi_{sname} (Tempsids \bowtie Sailors)$$

- ❖ To find sailors who've reserved all 'Interlake' boats:

$$\dots / \pi_{bid} (\sigma_{bname='Interlake'} Boats)$$

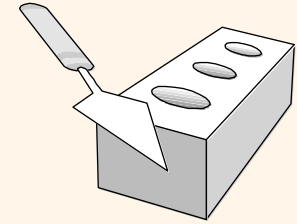


Summary

- ❖ The relational model has rigorously defined query languages that are simple and powerful.
- ❖ Relational algebra is more operational; useful as internal representation for query evaluation plans.
- ❖ Several ways of expressing a given query; a query optimizer should choose the most efficient version.

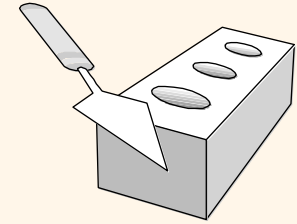
5

- Relational Algebra (**Ch.4**)



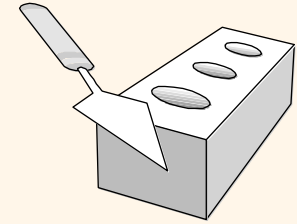
Relational Calculus

Chapter 4, Part B



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*.

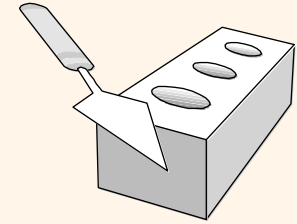


Domain Relational Calculus

- ❖ *Query* has the form:

$$\left\{ \langle x_1, x_2, \dots, x_n \rangle \mid p(\langle x_1, x_2, \dots, x_n \rangle) \right\}$$

- ❖ *Answer* includes all tuples $\langle x_1, x_2, \dots, x_n \rangle$ that make the *formula* $p(\langle x_1, x_2, \dots, x_n \rangle)$ be *true*.
- ❖ *Formula* is recursively defined, starting with simple *atomic formulas* (getting tuples from relations or making comparisons of values), and building bigger and better formulas using the *logical connectives*.



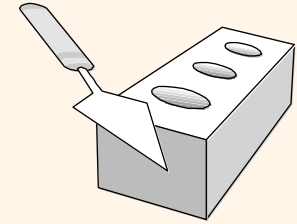
DRC Formulas

❖ Atomic formula:

- $\langle x_1, x_2, \dots, x_n \rangle \in Rname$, or $X \text{ op } Y$, or $X \text{ op } \text{constant}$
- op is one of $<, >, =, \leq, \geq, \neq$

❖ Formula:

- an atomic formula, or
 - $\neg p, p \wedge q, p \vee q$, where p and q are formulas, or
 - $\exists X(p(X))$, where variable X is *free* in $p(X)$, or
 - $\forall X(p(X))$, where variable X is *free* in $p(X)$
- ❖ The use of **quantifiers** $\exists X$ and $\forall X$ is said to **bind** X .
- A variable that is **not bound** is **free**.



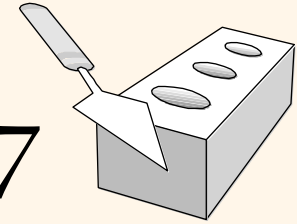
Free and Bound Variables

- ❖ 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**:

$$\left\{ \langle x_1, x_2, \dots, x_n \rangle \mid p(\langle x_1, x_2, \dots, x_n \rangle) \right\}$$

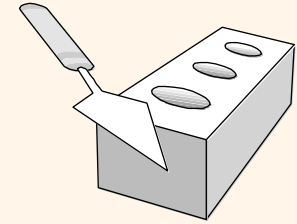
- ❖ There is an important restriction: the variables **x_1, \dots, x_n** that appear to the left of `|` must be the ***only*** free variables in the formula $p(\dots)$.

Find all sailors with a rating above 7



$$\{ \langle I, N, T, A \rangle \mid \langle I, N, T, A \rangle \in \text{Sailors} \wedge T > 7 \}$$

- ❖ The condition $\langle I, N, T, A \rangle \in \text{Sailors}$ ensures that the domain variables I , N , T and A are bound to fields of the same Sailors tuple.
- ❖ The term $\langle I, N, T, A \rangle$ to the left of `|` (which should be read as *such that*) says that every tuple $\langle I, N, T, A \rangle$ that satisfies $T > 7$ is in the answer.
- ❖ Modify this query to answer:
 - Find sailors who are older than 18 or have a rating under 9, and are called 'Joe'.

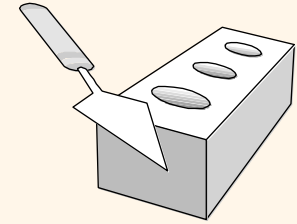


*Find sailors rated > 7 who have reserved
boat #103*

$$\left\{ \langle I, N, T, A \rangle \mid \langle I, N, T, A \rangle \in \text{Sailors} \wedge T > 7 \wedge \right. \\ \left. \exists Ir, Br, D \left(\langle Ir, Br, D \rangle \in \text{Reserves} \wedge Ir = I \wedge Br = 103 \right) \right\}$$

- ❖ We have used $\exists Ir, Br, D (\dots)$ as a shorthand for $\exists Ir (\exists Br (\exists D (\dots)))$
- ❖ Note the use of \exists to find a tuple in Reserves that ‘joins with’ the Sailors tuple under consideration.

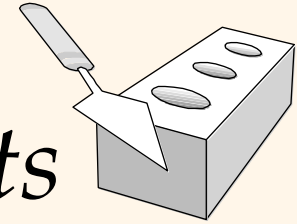
Find sailors rated > 7 who've reserved a red boat



$$\left\{ \left\langle I, N, T, A \right\rangle \mid \left\langle I, N, T, A \right\rangle \in \text{Sailors} \wedge T > 7 \wedge \right. \\ \left. \exists Ir, Br, D \left(\left\langle Ir, Br, D \right\rangle \in \text{Reserves} \wedge Ir = I \wedge \right. \right. \\ \left. \left. \exists B, BN, C \left(\left\langle B, BN, C \right\rangle \in \text{Boats} \wedge B = Br \wedge C = 'red' \right) \right) \right\}$$

- ❖ Observe how the parentheses control the scope of each quantifier's binding.
- ❖ This may look cumbersome, but with a good user interface, it is very intuitive. (MS Access, QBE)

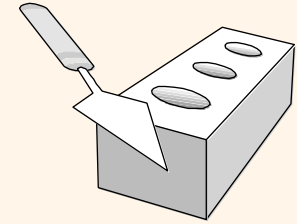
Find sailors who've reserved all boats



$$\left\{ \langle I, N, T, A \rangle \mid \langle I, N, T, A \rangle \in \text{Sailors} \wedge \right. \\ \left. \forall B, BN, C \left(\neg \left(\langle B, BN, C \rangle \in \text{Boats} \right) \vee \right. \right. \\ \left. \left. \left(\exists Ir, Br, D \left(\langle Ir, Br, D \rangle \in \text{Reserves} \wedge I = Ir \wedge Br = B \right) \right) \right) \right\}$$

- ❖ Find all sailors I such that for each 3-tuple $\langle B, BN, C \rangle$ either it is not a tuple in Boats or there is a tuple in Reserves showing that sailor I has reserved it.

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

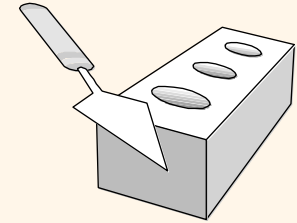


$$\left\{ \langle I, N, T, A \rangle \mid \langle I, N, T, A \rangle \in \text{Sailors} \wedge \right. \\ \left. \forall \langle B, BN, C \rangle \in \text{Boats} \right. \\ \left. \left(\exists \langle Ir, Br, D \rangle \in \text{Reserves} (I = Ir \wedge Br = B) \right) \right\}$$

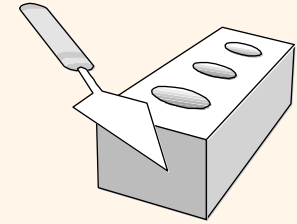
- ❖ Simpler notation, same query. (Much clearer!)
- ❖ To find sailors who've reserved all red boats:

$$\dots \left\{ C \neq 'red' \vee \exists \langle Ir, Br, D \rangle \in \text{Reserves} (I = Ir \wedge Br = B) \right\}$$

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., $\{S \mid \neg(S \in Sailors)\}$
- ❖ 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.



Summary

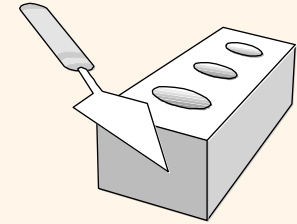
- ❖ Relational calculus is non-operational, and users define queries in terms of what they want, not in terms of how to compute it. (Declarativeness.)
- ❖ Algebra and safe calculus have same expressive power, leading to the notion of relational completeness.

6

- SQL Query Language (**Ch.5**)

7

- SQL Query Language (**Ch.5**)



SQL: Queries, Programming, Triggers

Chapter 5

Example Instances

R1

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

- ❖ We will use these instances of the Sailors and Reserves relations in our examples.

S1

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

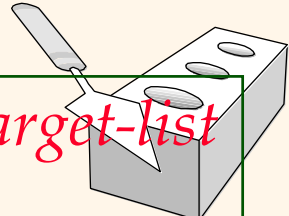
- ❖ If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

S2

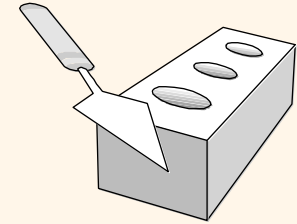
<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Basic SQL Query

SELECT	[DISTINCT] <i>target-list</i>
FROM	<i>relation-list</i>
WHERE	<i>qualification</i>

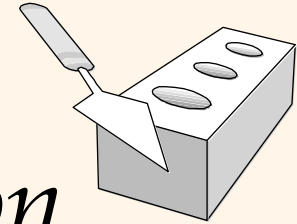


- ❖ *relation-list* A list of relation names (possibly with a *range-variable* after each name).
- ❖ *target-list* A list of attributes of relations in *relation-list*
- ❖ *qualification* Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of $<$, $>$, $=$, \leq , \geq , \neq) combined using AND, OR and NOT.
- ❖ **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are not eliminated!



Conceptual Evaluation Strategy

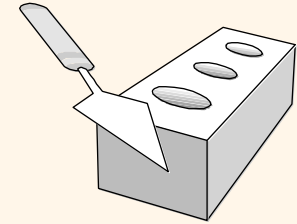
- ❖ Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
 - Compute the cross-product of *relation-list*.
 - Discard resulting tuples if they fail *qualifications*.
 - Delete attributes that are not in *target-list*.
 - If **DISTINCT** is specified, eliminate duplicate rows.
- ❖ This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.



Example of Conceptual Evaluation

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid AND R.bid=103
```

(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



A Note on Range Variables

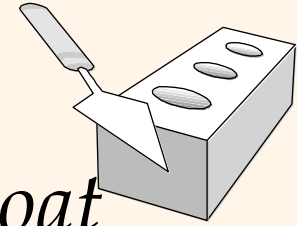
- ❖ Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT S.sname  
FROM Sailors S, Reserves R  
WHERE S.sid=R.sid AND bid=103
```

OR

```
SELECT sname  
FROM Sailors, Reserves  
WHERE Sailors.sid=Reserves.sid  
AND bid=103
```

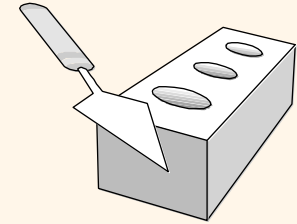
*It is good style,
however, to use
range variables
always!*



Find sailors who've reserved at least one boat

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

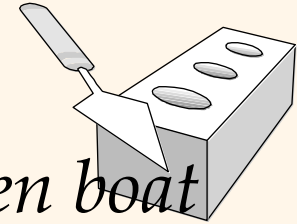
- ❖ Would adding DISTINCT to this query make a difference?
- ❖ What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?



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.



Find sid's of sailors who've reserved a red or a green boat

- ❖ **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- ❖ If we replace **OR** by **AND** in the first version, what do we get?
- ❖ Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

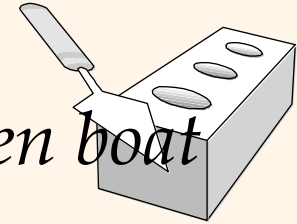
```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND (B.color='red' OR B.color='green')
```

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='red'
```

UNION

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='green'
```

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



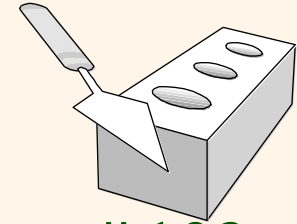
- ❖ **INTERSECT**: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- ❖ Included in the SQL/92 standard, but some systems don't support it.
- ❖ Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```
SELECT S.sid
FROM Sailors S, Boats B1, Reserves R1,
      Boats B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
      AND S.sid=R2.sid AND R2.bid=B2.bid
      AND (B1.color='red' AND B2.color='green')
```

```
SELECT S.sid Key field!
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='red'
```

INTERSECT

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='green'
```

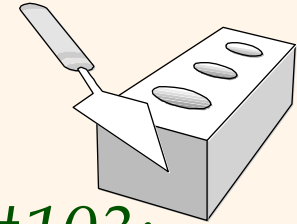


Nested Queries

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 nested loops evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

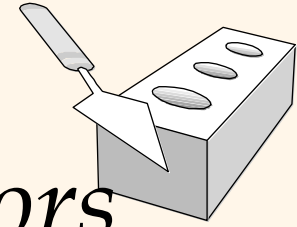


Nested Queries with Correlation

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.

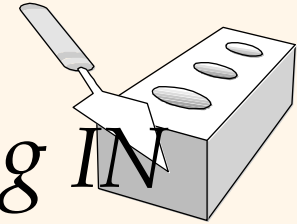


More on Set-Comparison Operators

- ❖ 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

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

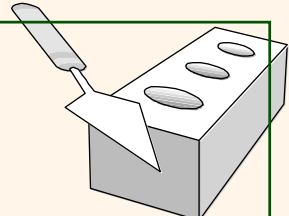
Sailors S such that ... WHERE NOT EXISTS (SELECT R.bid
FROM Reserves R
WHERE R.bid=B.bid
AND R.sid=S.sid))

there is no boat B without ...

a Reserves tuple showing S reserved B

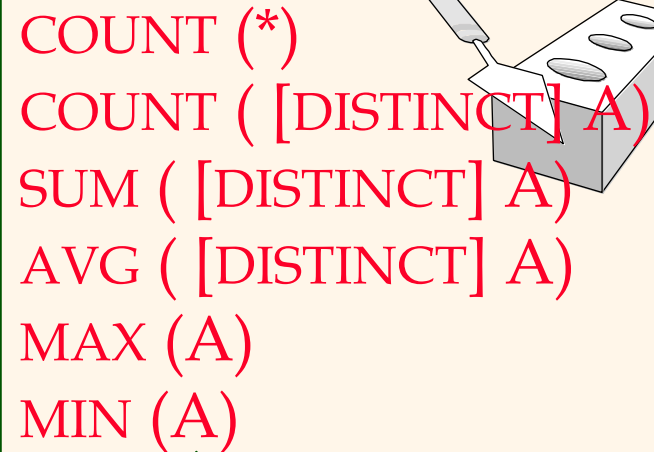
(1)

```
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))
```



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
```

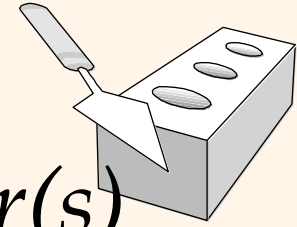
```
SELECT AVG (S.age)  
FROM Sailors S  
WHERE S.rating=10
```

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

```
SELECT S.sname  
FROM Sailors S
```

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

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



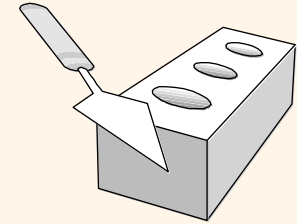
Find name and age of the oldest sailor(s)

- ❖ 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
```



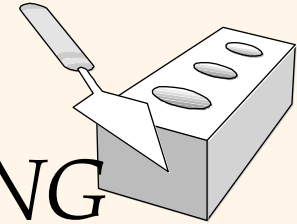
GROUP BY and HAVING

- ❖ 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 $i = 1, 2, \dots, 10$:

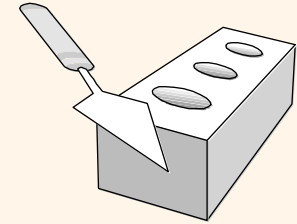
```
SELECT MIN (S.age)
FROM Sailors S
WHERE S.rating = i
```

Queries With GROUP BY and HAVING



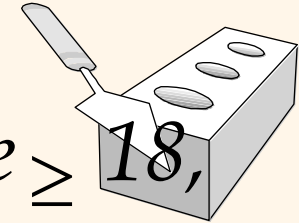
SELECT	[DISTINCT] <i>target-list</i>
FROM	<i>relation-list</i>
WHERE	<i>qualification</i>
GROUP BY	<i>grouping-list</i>
HAVING	<i>group-qualification</i>

- ❖ 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*.)



Conceptual Evaluation

- ❖ The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, 'unnecessary' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- ❖ The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a *single value per group!*
 - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- ❖ One answer tuple is generated per qualifying group.



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

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

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

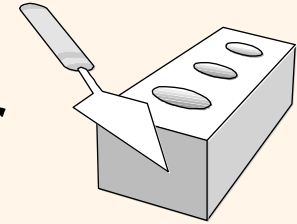
rating	age
1	33.0
7	45.0
7	35.0
8	55.5
10	35.0

rating	
7	35.0

Answer relation

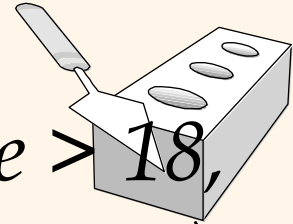
- ❖ Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes `unnecessary`.
- ❖ 2nd column of result is unnamed. (Use AS to name it.)

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?

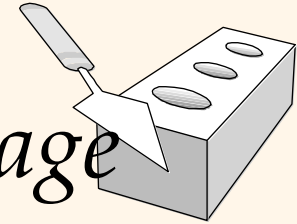


*Find the age of the youngest sailor with age > 18,
for each rating with at least 2 sailors (of any age)*

```
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

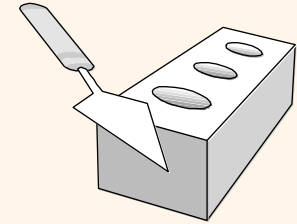


❖ Aggregate operations cannot be nested! **WRONG:**

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

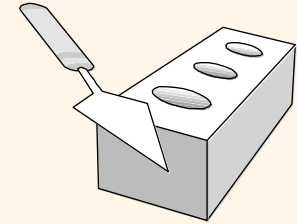
✓ Correct solution (in SQL/92):

```
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.



Integrity Constraints (Review)

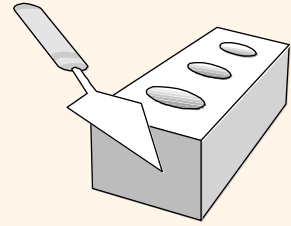
- ❖ An IC describes conditions that every *legal instance* of a relation must satisfy.
 - Inserts/deletes/updates that violate IC's are disallowed.
 - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- ❖ *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.

General Constraints

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

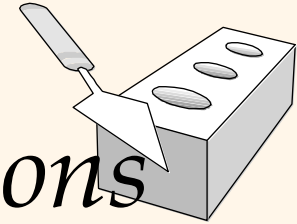
```
CREATE TABLE Sailors
```

```
( 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

```
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 )
```

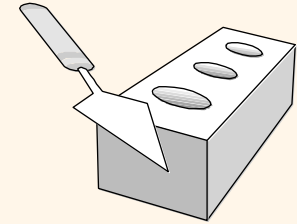
*Number of boats
plus number of
sailors is < 100*

- ❖ 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 ASSERTION smallClub
```

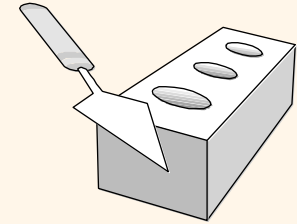
```
CHECK
```

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



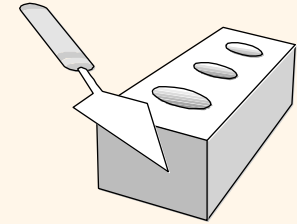
Triggers

- ❖ 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)



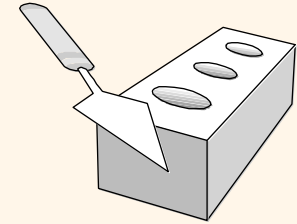
Triggers: Example (SQL:1999)

```
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
```



Summary

- ❖ SQL was an important factor in the early acceptance of the relational model; more natural than earlier, procedural query languages.
- ❖ Relationally complete; in fact, significantly more expressive power than relational algebra.
- ❖ Even queries that can be expressed in RA can often be expressed more naturally in SQL.
- ❖ Many alternative ways to write a query; optimizer should look for most efficient evaluation plan.
 - In practice, users need to be aware of how queries are optimized and evaluated for best results.



Summary (Contd.)

- ❖ NULL for unknown field values brings many complications
- ❖ SQL allows specification of rich integrity constraints
- ❖ Triggers respond to changes in the database

8

- Schema Refinement and Normal Forms (**Ch.19**)

9

- Schema Refinement and Normal Forms (**Ch.19**)



Schema Refinement and Normal Forms

Chapter 19



The Evils of Redundancy

- ❖ *Redundancy* is at the root of several problems associated with relational schemas:
 - *redundant storage, insert/delete/update anomalies*
- ❖ Integrity constraints, in particular *functional dependencies*, can be used to identify schemas with such problems and to suggest refinements.
- ❖ 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?

Functional Dependencies (FDs)



- ❖ A functional dependency $X \rightarrow Y$ 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 $\pi_Y(t1) = \pi_Y(t2)$
 - i.e., given two tuples in r , if the X values agree, then the Y values must also agree. (X and Y are *sets* of attributes.)
- ❖ An FD is a statement about *all* allowable relations.
 - Must be identified based on semantics of application.
 - Given some allowable instance $r1$ of R, we can check if it violates some FD f , but we cannot tell if f holds over R!
- ❖ K is a candidate key for R means that $K \rightarrow R$
 - However, $K \rightarrow R$ does not require K to be *minimal*!

Example: Constraints on Entity Set



- ❖ Consider relation obtained from Hourly_Emps:
 - Hourly_Emps (ssn, name, lot, rating, hrly_wages, hrs_worked)
- ❖ Notation: 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*: $S \rightarrow \text{SNLRWH}$
 - *rating determines hrly_wages*: $R \rightarrow W$

Example (Contd.)



Wages

R	W
8	10
5	7

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

- ❖ Problems due to R → 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!

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

Will 2 smaller tables be better?



Reasoning About FDs

- ❖ Given some FDs, we can usually infer additional FDs:
 - $ssn \rightarrow did, did \rightarrow lot$ implies $ssn \rightarrow lot$
- ❖ An FD f is *implied by* a set of FDs F if f holds whenever all FDs in F hold.
 - F^+ = *closure of F* is the set of all FDs that are implied by F .
- ❖ Armstrong's Axioms (X, Y, Z are sets of attributes):
 - Reflexivity: If $X \subseteq Y$, then $Y \rightarrow X$
 - Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
 - Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- ❖ These are *sound* and *complete* inference rules for FDs!

Reasoning About FDs (Contd.)



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

Reasoning About FDs (Contd.)



- ❖ Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)
- ❖ Typically, we just want to check if a given FD $X \rightarrow Y$ is in the closure of a set of FDs F . An efficient check:
 - Compute attribute closure of X (denoted X^+) wrt F :
 - Set of all attributes A such that $X \rightarrow A$ is in F^+
 - There is a linear time algorithm to compute this.
 - Check if Y is in X^+
- ❖ Does $F = \{A \rightarrow B, B \rightarrow C, C D \rightarrow E\}$ imply $A \rightarrow E$?
 - i.e, is $A \rightarrow E$ in the closure F^+ ? Equivalently, is E in A^+ ?

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!

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 must be identical (since X is a key).

X	Y	A
x	y1	a
x	y2	?

Third Normal Form (3NF)



- ❖ Reln R with FDs F is in **3NF** if, for all $X \rightarrow A$ in F^+
 - $A \in X$ (called a *trivial* FD), or
 - X contains a key for R, or
 - A is part of some key for R.
- ❖ *Minimality* of a key is crucial in third condition above!
- ❖ If R is in BCNF, obviously in 3NF.
- ❖ If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no “good” decomp, or performance considerations).
 - *Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.*



What Does 3NF Achieve?

- ❖ If 3NF violated by $X \rightarrow A$, one of the following holds:
 - X is a subset of some key K
 - We store (X, A) pairs redundantly.
 - X is not a proper subset of any key.
 - There is a chain of FDs $K \rightarrow X \rightarrow A$, which means that we cannot associate an X value with a K value unless we also associate an A value with an X value.
- ❖ **But:** even if reln is in 3NF, these problems could arise.
 - e.g., Reserves SBDC, $S \rightarrow C$, $C \rightarrow S$ is in 3NF, but for each reservation of sailor S , same (S, C) pair is stored.
- ❖ Thus, 3NF is indeed a compromise relative to BCNF.

Decomposition of a Relation Scheme



- ❖ Suppose that relation R contains attributes $A_1 \dots A_n$. 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 $S \rightarrow \text{SNLRWH}$ 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



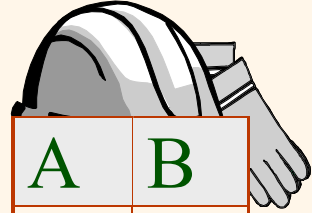
- ❖ There are three potential problems to consider:
 - Some queries become more expensive.
 - e.g., How much did sailor Joe earn? (salary = $W \cdot H$)
 - Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!
 - Fortunately, 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 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! Avoids Problem (2).*

More on Lossless Join



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



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



❖ 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, the decomposition of R into UV and R - V is lossless-join if $U \rightarrow V$ holds over R.

Dependency Preserving Decomposition



- ❖ Consider CSJDPQV, C is key, $JP \rightarrow C$ and $SD \rightarrow P$.
 - 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. (Avoids Problem (3).)
- ❖ 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.
 - 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* .



Minimal Cover for a Set of FDs

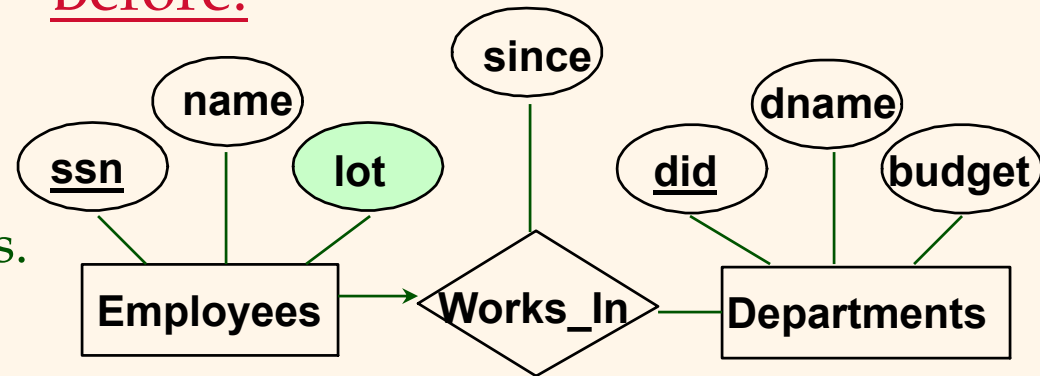
- ❖ Minimal cover G for a set of FDs F :
 - Closure of F = closure of G .
 - Right hand side of each FD in G is a single attribute.
 - If we modify G by deleting an FD or by deleting attributes from an FD in G , the closure changes.
- ❖ Intuitively, every FD in G is needed, and “*as small as possible*” in order to get the same closure as F .
- ❖ e.g., $A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$ has the following minimal cover:
 - $A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$ and $EF \rightarrow H$
- ❖ M.C. \rightarrow Lossless-Join, Dep. Pres. Decomp!!! (in book)



Refining an ER Diagram

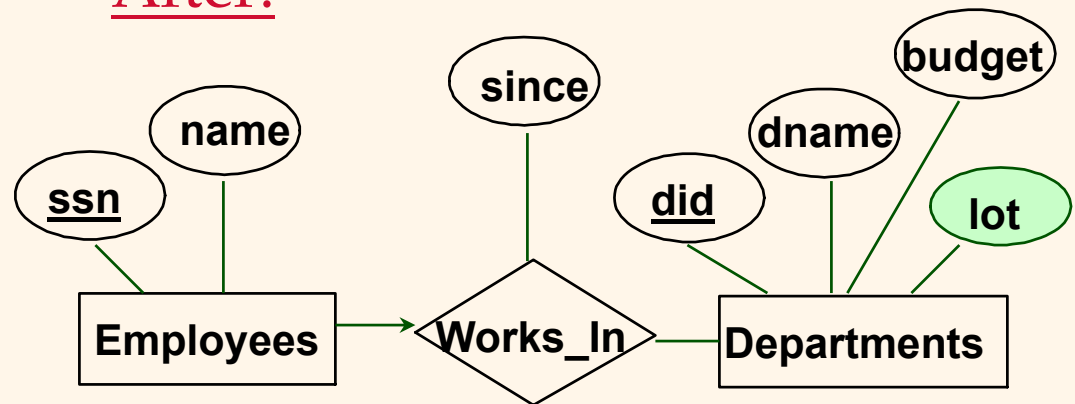
- ❖ 1st diagram translated:
Workers(S,N,L,D,S)
Departments(D,M,B)
 - Lots associated with workers.
- ❖ Suppose all workers in a dept are assigned the same lot: $D \rightarrow L$

Before:



- ❖ Redundancy; fixed by:
Workers2(S,N,D,S)
Dept_Lots(D,L)
- ❖ Can fine-tune this:
Workers2(S,N,D,S)
Departments(D,M,B,L)

After:



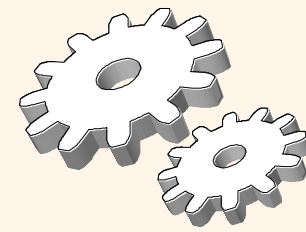
Summary of Schema Refinement



- ❖ If a relation is in BCNF, it is free of redundancies that can be detected using FDs. Thus, trying to ensure that all relations are in BCNF is a good heuristic.
- ❖ If a relation is not in BCNF, we can try to decompose it into a collection of BCNF relations.
 - Must consider whether all FDs are preserved. If a lossless-join, dependency preserving decomposition into BCNF is not possible (or unsuitable, given typical queries), should consider decomposition into 3NF.
 - Decompositions should be carried out and/or re-examined while keeping *performance requirements* in mind.

10

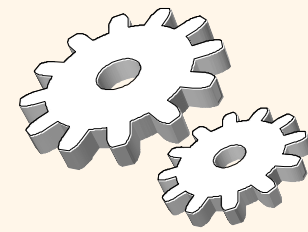
- Overview of Storage and Indexing (**Ch.8**)



Overview of Storage and Indexing

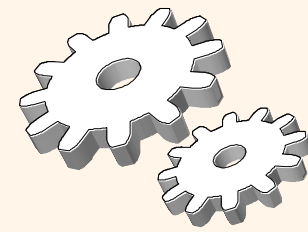
Chapter 8

*“How index-learning turns no student pale
Yet holds the eel of science by the tail.”
-- Alexander Pope (1688-1744)*



Data on External Storage

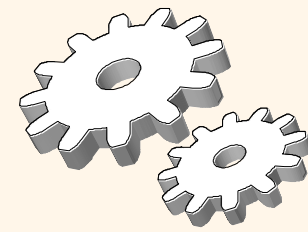
- ❖ Disks: Can retrieve random page at fixed cost
 - But reading several consecutive pages is much cheaper than reading them in random order
- ❖ Tapes: Can only read pages in sequence
 - Cheaper than disks; used for archival storage
- ❖ File organization: Method of arranging a file of records on external 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

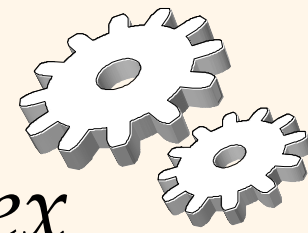
Many alternatives exist, *each ideal for some situations, and not so good in others:*

- 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

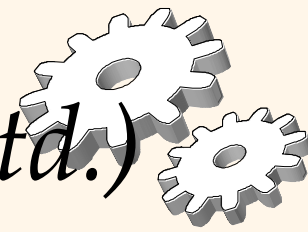
- ❖ An index on a file speeds up selections on the *search key fields* for the index.
 - Any subset of the fields of a relation can be the search key for an index on the relation.
 - *Search key* is **not** the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- ❖ An index contains a collection of *data entries*, and supports efficient retrieval of all data entries \mathbf{k}^* with a given key value \mathbf{k} .



Alternatives for Data Entry k^ in Index*

- ❖ Three alternatives:
 - Data record with key value k
 - $\langle k, \text{rid of data record with search key value } k \rangle$
 - $\langle k, \text{list of rids of data records with search key } k \rangle$
- ❖ 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

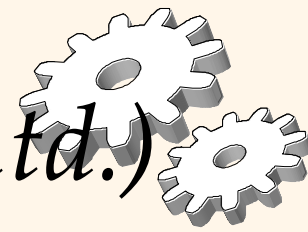
Alternatives for Data Entries (Contd.)



❖ **Alternative 1:**

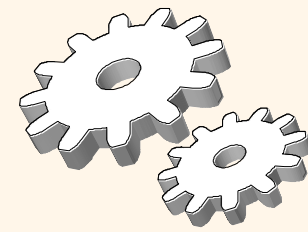
- If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
- At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
- If 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.

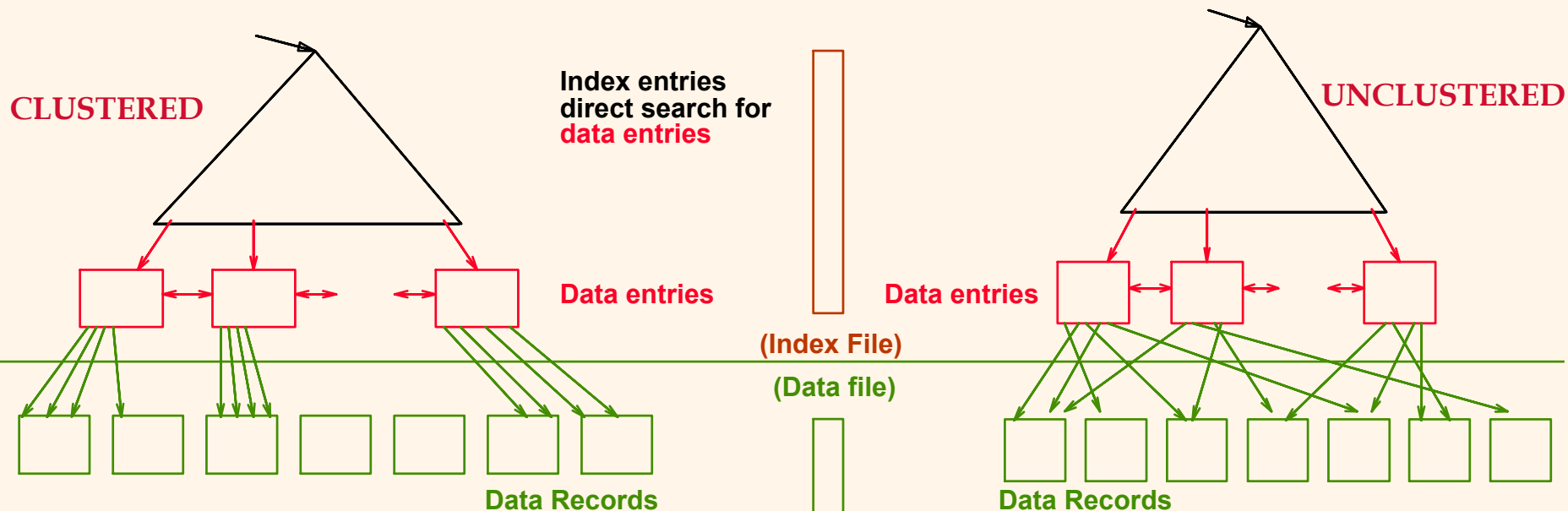


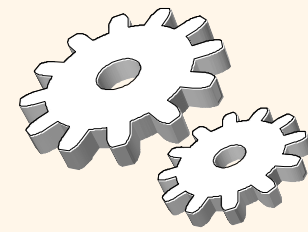
Index Classification

- ❖ *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 or not!

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.)

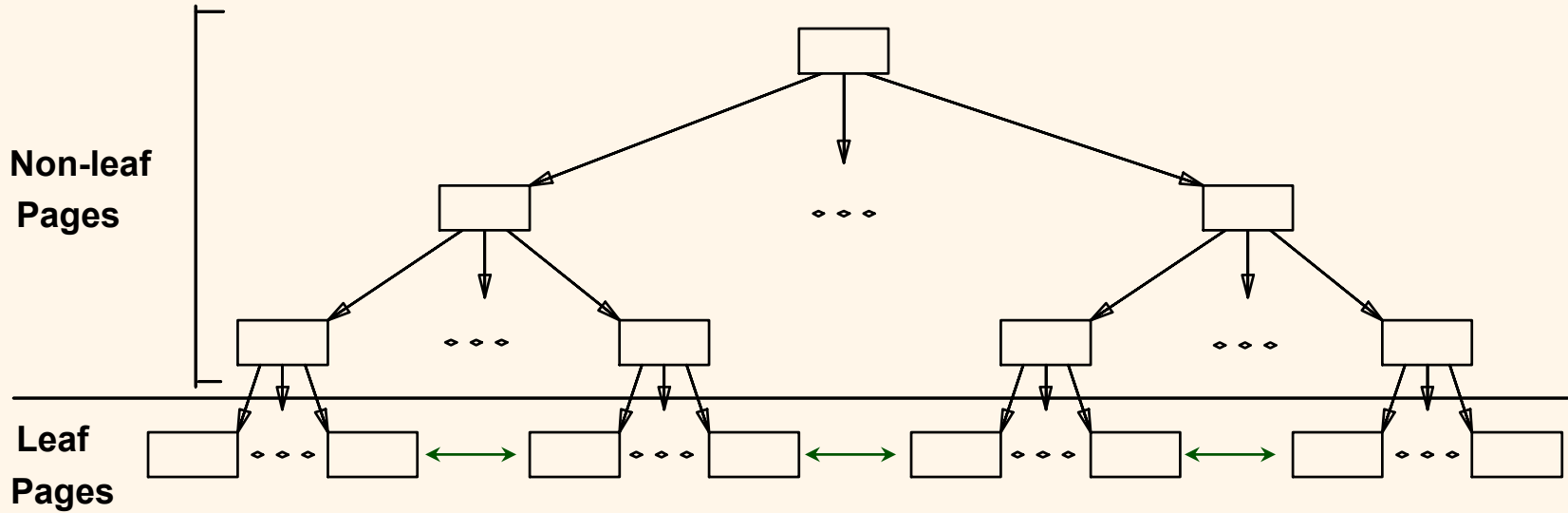
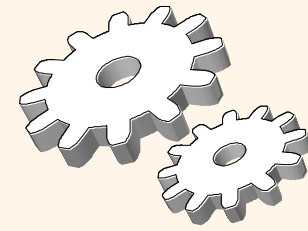




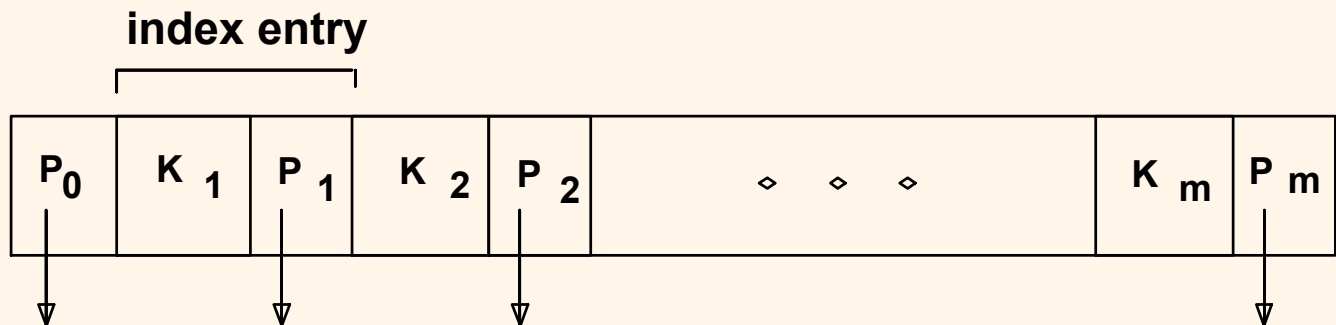
Hash-Based Indexes

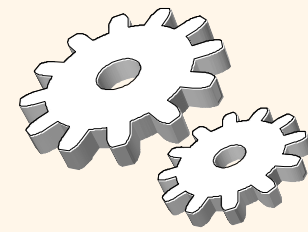
- ❖ Good for equality selections.
 - Index is a collection of buckets. Bucket = *primary page* plus zero or more *overflow pages*.
 - *Hashing function h*: $h(r)$ = bucket in which record r belongs. h looks at the *search key* fields of r .
- ❖ If Alternative (1) is used, the buckets contain the data records; otherwise, they contain <key, rid> or <key, rid-list> pairs.

B+ Tree Indexes

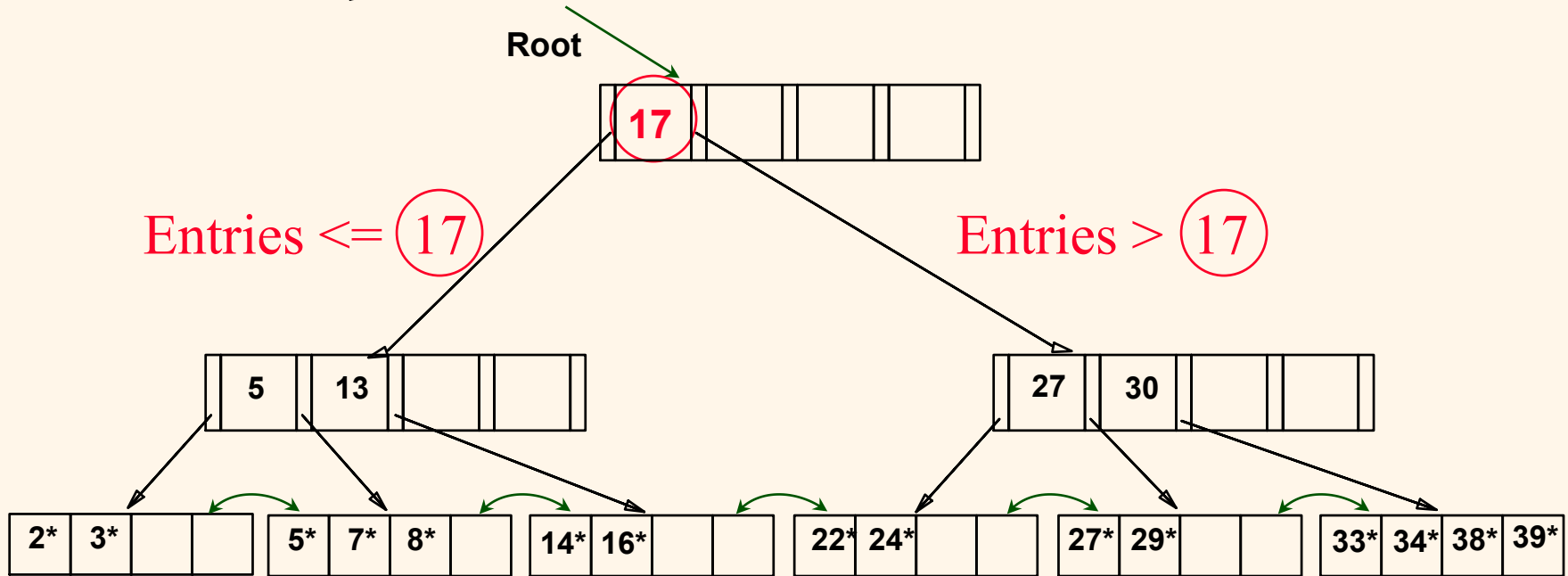


- ❖ Leaf pages contain *data entries*, and are chained (prev & next)
- ❖ Non-leaf pages contain *index entries* and direct searches:

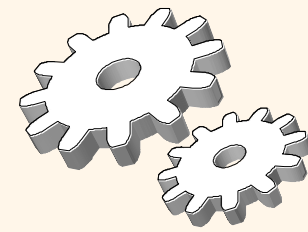




Example B+ Tree



- ❖ 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

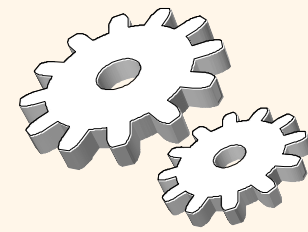


Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

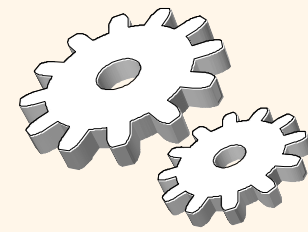
- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.

** Good enough to show the overall trends!*



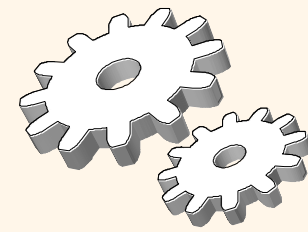
Comparing File Organizations

- ❖ Heap files (random order; insert at eof)
- ❖ Sorted files, sorted on $\langle age, sal \rangle$
- ❖ Clustered B+ tree file, Alternative (1), search key $\langle age, sal \rangle$
- ❖ Heap file with unclustered B + tree index on search key $\langle age, sal \rangle$
- ❖ Heap file with unclustered hash index on search key $\langle age, sal \rangle$



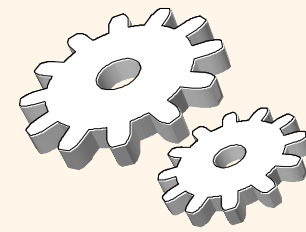
Operations to Compare

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



Assumptions in Our Analysis

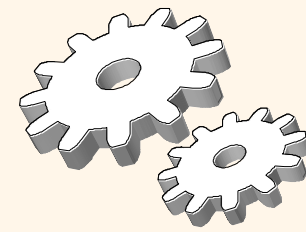
- ❖ **Heap Files:**
 - Equality selection on key; exactly one match.
- ❖ **Sorted Files:**
 - Files compacted after deletions.
- ❖ **Indexes:**
 - Alt (2), (3): data entry size = 10% size of record
 - Hash: No overflow buckets.
 - 80% page occupancy => File size = 1.25 data size
 - Tree: 67% occupancy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



Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap					
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

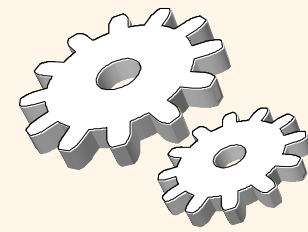
** Several assumptions underlie these (rough) estimates!*



Cost of Operations

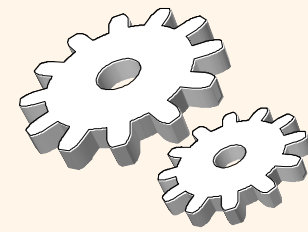
	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	$D \log_2 B$	$D \log_2 B + \# \text{ matches}$	Search + BD	Search +BD
(3) Clustered	1.5BD	$D \log_F 1.5B$	$D \log_F 1.5B + \# \text{ matches}$	Search + D	Search +D
(4) Unclustered Tree index	$BD(R+0.15)$	$D(1 + \log_F 0.15B)$	$D \log_F 0.15B + \# \text{ matches}$	$D(3 + \log_F 0.15B)$	Search + 2D
(5) Unclustered Hash index	$BD(R+0.125)$	2D	BD	4D	Search + 2D

** Several assumptions underlie these (rough) estimates!*



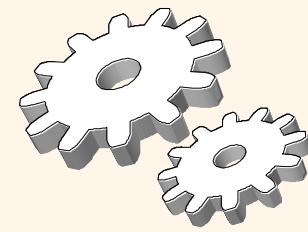
Understanding the Workload

- ❖ For each query in the workload:
 - Which relations does it access?
 - Which attributes are retrieved?
 - Which attributes are involved in selection/join conditions?
How selective are these conditions likely to be?
- ❖ For each update in the workload:
 - Which attributes are involved in selection/join conditions?
How selective are these conditions likely to be?
 - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.



Choice of Indexes

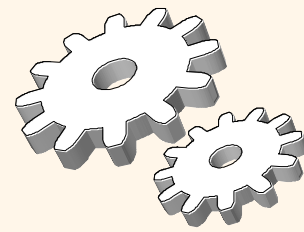
- ❖ What indexes should we create?
 - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- ❖ For each index, what kind of an index should it be?
 - Clustered? Hash/tree?



Choice of Indexes (Contd.)

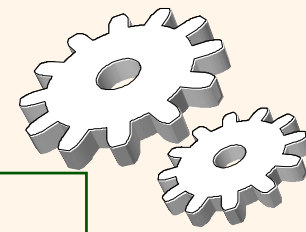
- ❖ **One approach:** Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
 - Obviously, this implies that we must understand how a DBMS evaluates queries and creates **query evaluation plans!**
 - For now, we discuss simple 1-table queries.
- ❖ Before creating an index, must also consider the impact on updates in the workload!
 - **Trade-off:** Indexes can make queries go faster, updates slower. Require disk space, too.

Index Selection Guidelines



- ❖ 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!
- ❖ Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

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:

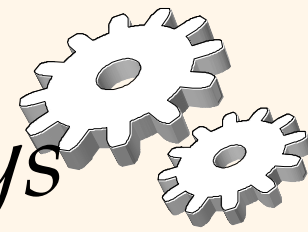
- Clustering on *E.hobby* helps!

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

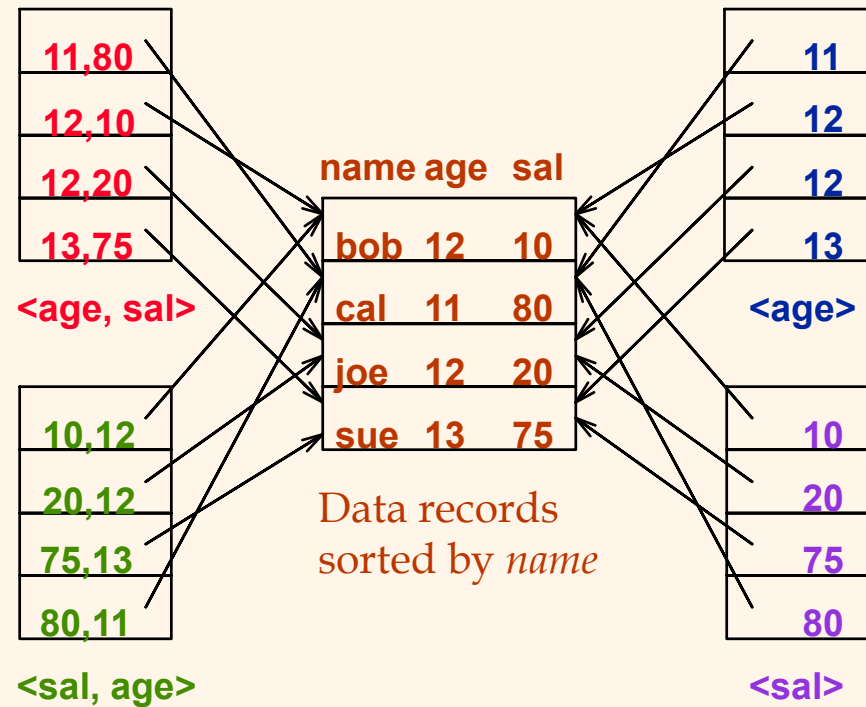
```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age>10
GROUP BY E.dno
```

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

Indexes with Composite Search Keys



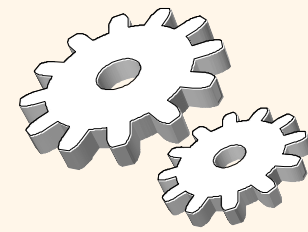
Examples of composite key indexes using lexicographic order.



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

Data entries sorted by **<sal>**

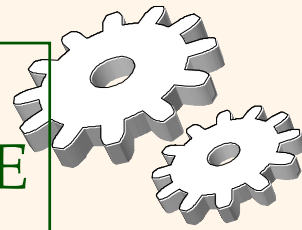
- ❖ **Composite Search Keys:** Search on a combination of fields.
 - **Equality query:** Every field value is equal to a constant value. E.g. wrt **<sal, age>** index:
 - age=20 and sal =75
 - **Range query:** Some field value is not a constant. E.g.:
 - age =20; or age=20 and sal > 10
- ❖ Data entries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



Composite Search Keys

- ❖ To retrieve Emp records with $age=30$ AND $sal=4000$, an index on $\langle age, sal \rangle$ 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 $\langle age, sal \rangle$ or $\langle sal, age \rangle$ is best.
- ❖ If condition is: $age=30$ AND $3000 < sal < 5000$:
 - Clustered $\langle age, sal \rangle$ index much better than $\langle sal, age \rangle$ index!
- ❖ Composite indexes are larger, updated more often.

Index-Only Plans



<E.dno>

```
SELECT D.mgr
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<E.dno,E.eid>

Tree index!

```
SELECT D.mgr, E.eid
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<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>

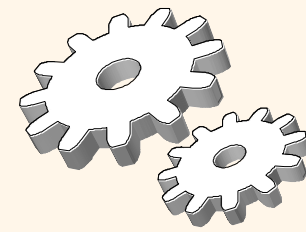
or

<E.sal, E.age>

Tree!

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

- ❖ 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.



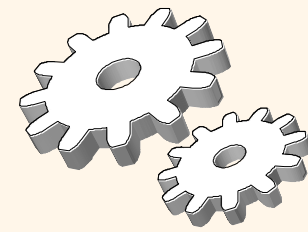
Index-Only Plans (Contd.)

❖ Index-only plans are possible if the key is $\langle \text{dno}, \text{age} \rangle$ or we have a tree index with key $\langle \text{age}, \text{dno} \rangle$

- Which is better?
- What if we consider the second query?

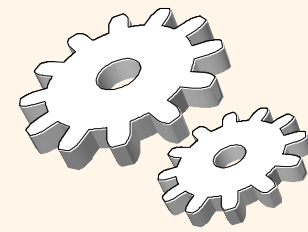
```
SELECT E.dno, COUNT (*)  
FROM Emp E  
WHERE E.age=30  
GROUP BY E.dno
```

```
SELECT E.dno, COUNT (*)  
FROM Emp E  
WHERE E.age>30  
GROUP BY E.dno
```



Summary

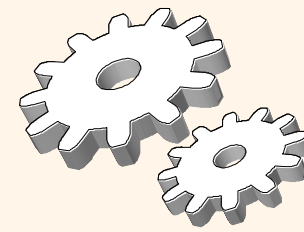
- ❖ Many alternative file organizations exist, each appropriate in some situation.
- ❖ If selection queries are frequent, sorting the file or building an *index* is important.
 - Hash-based indexes only good for equality search.
 - Sorted files and tree-based indexes best for range search; also good 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, $\langle \text{key}, \text{rid} \rangle$ pairs, or $\langle \text{key}, \text{rid-list} \rangle$ 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.

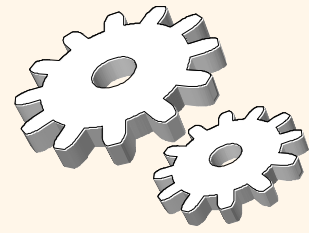
Summary (Contd.)



- ❖ Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
 - What are the important queries and updates? What attributes/relations are involved?
- ❖ Indexes must be chosen to speed up important queries (and perhaps some updates!).
 - Index maintenance overhead on updates to key fields.
 - Choose indexes that can help many queries, if possible.
 - Build indexes to support index-only strategies.
 - Clustering is an important decision; only one index on a given relation can be clustered!
 - Order of fields in composite index key can be important.

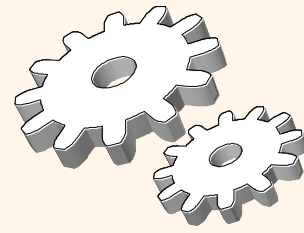
11

- Tree-Structured Indexing (**Ch.10**)



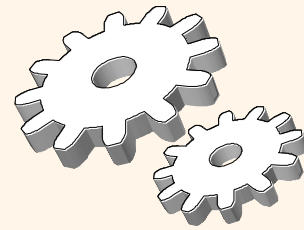
Tree-Structured Indexes

Chapter 9



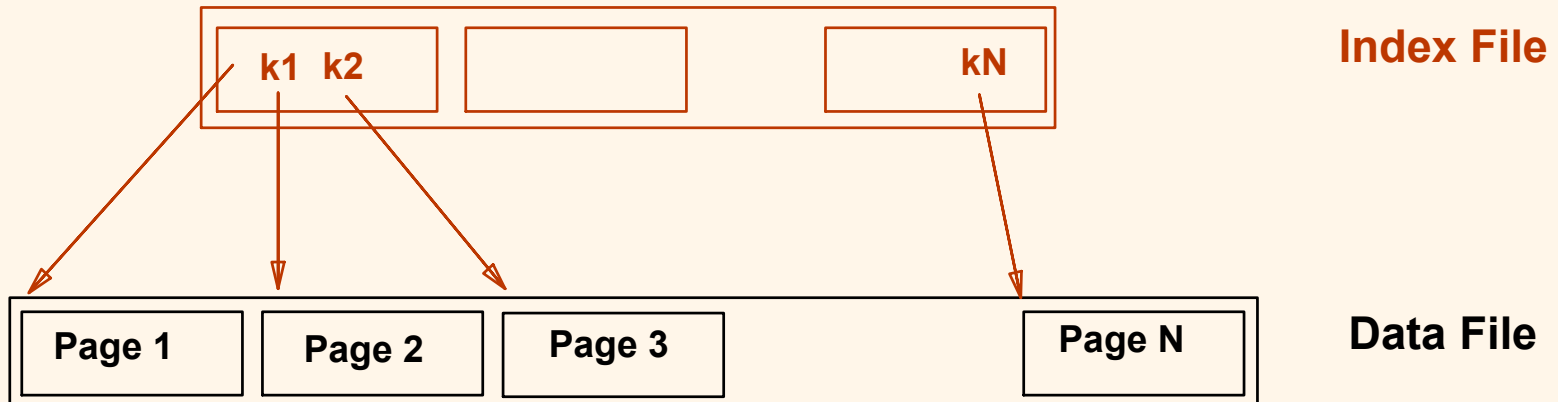
Introduction

- ❖ As for any index, 3 alternatives for data entries \mathbf{k}^* :
 - Data record with key value \mathbf{k}
 - $\langle \mathbf{k}, \text{rid of data record with search key value } \mathbf{k} \rangle$
 - $\langle \mathbf{k}, \text{list of rids of data records with search key } \mathbf{k} \rangle$
- ❖ Choice is orthogonal to the *indexing technique* used to locate data entries \mathbf{k}^* .
- ❖ Tree-structured indexing techniques support both *range searches* and *equality searches*.
- ❖ ISAM: static structure; B+ tree: dynamic, adjusts gracefully under inserts and deletes.



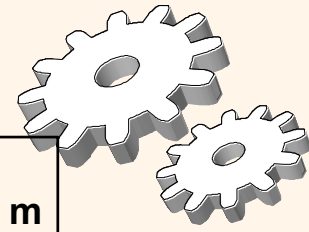
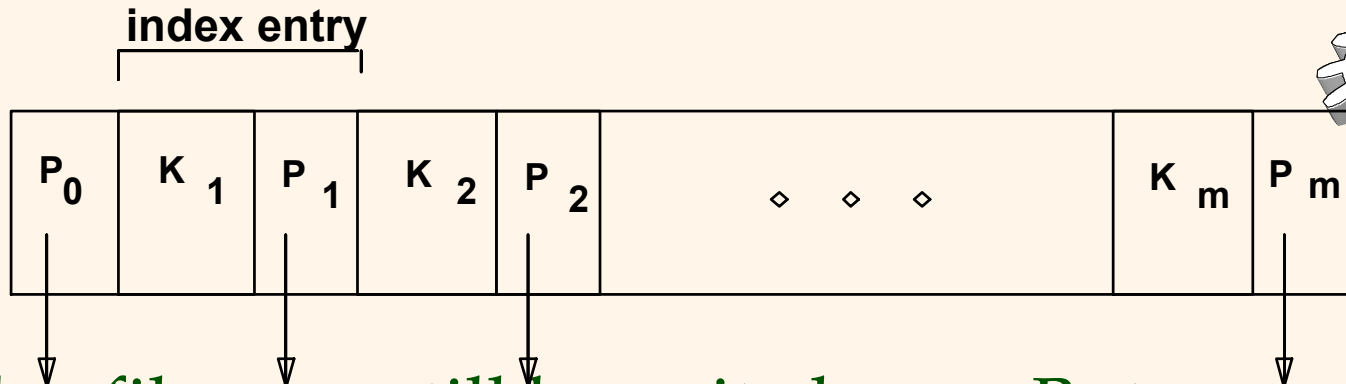
Range Searches

- ❖ *“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.

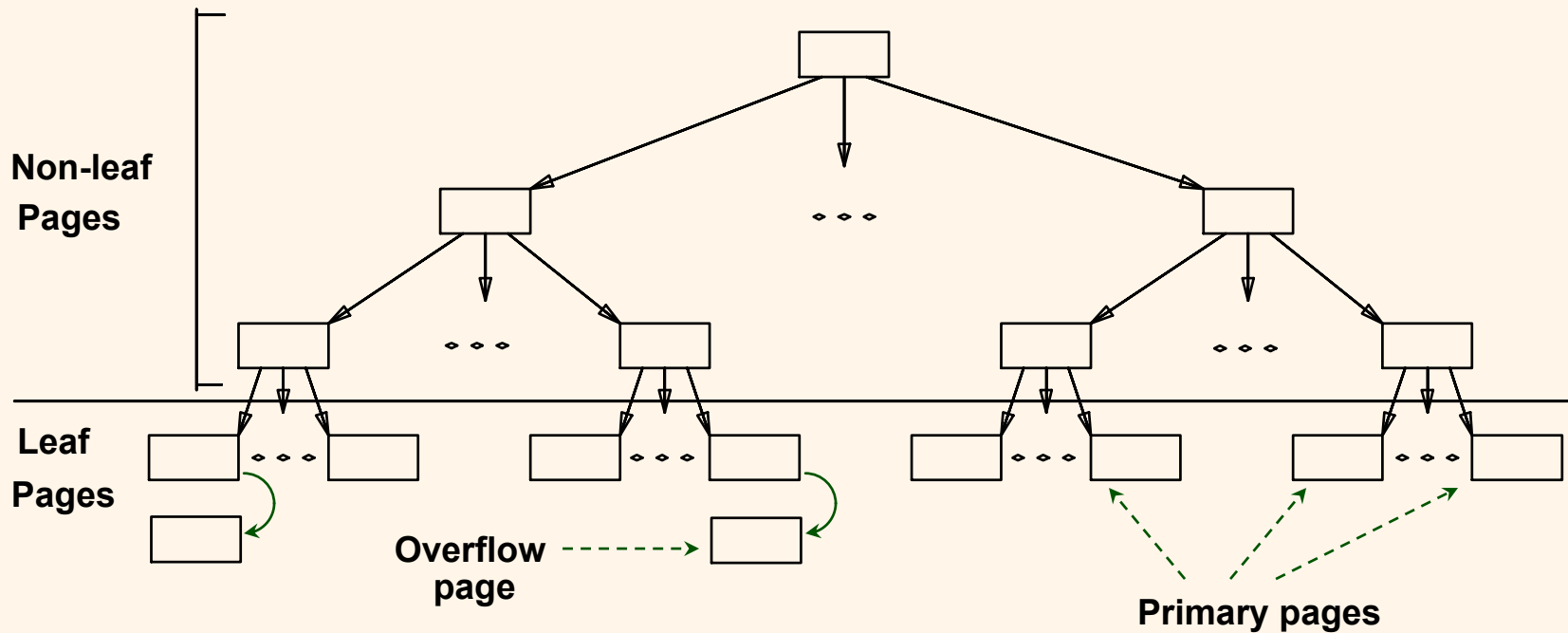


** Can do binary search on (smaller) index file!*

ISAM



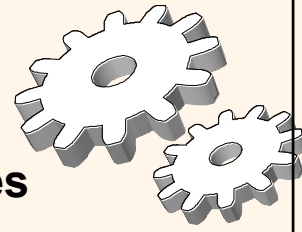
❖ Index file may still be quite large. But we can apply the idea repeatedly!



* Leaf pages contain *data entries*.

Comments on ISAM

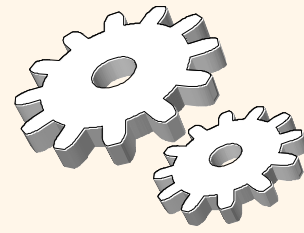
Data
Pages



Index Pages

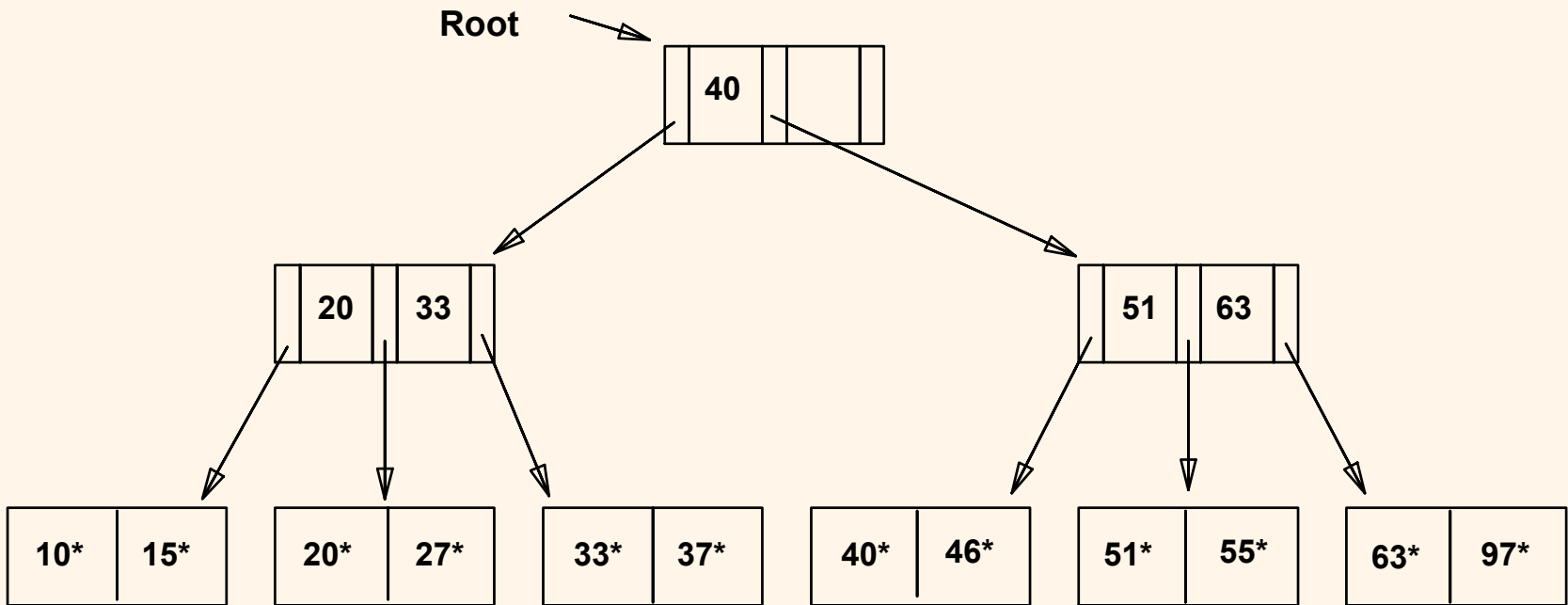
Overflow pages

- ❖ *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 $\propto \log_F N$; $F = \# \text{ entries/index pg}$, $N = \# \text{ leaf pgs}$
 - ❖ *Insert*: Find leaf data entry belongs to, and put it there.
 - ❖ *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.
- * **Static tree structure**: *inserts/deletes affect only leaf pages.*

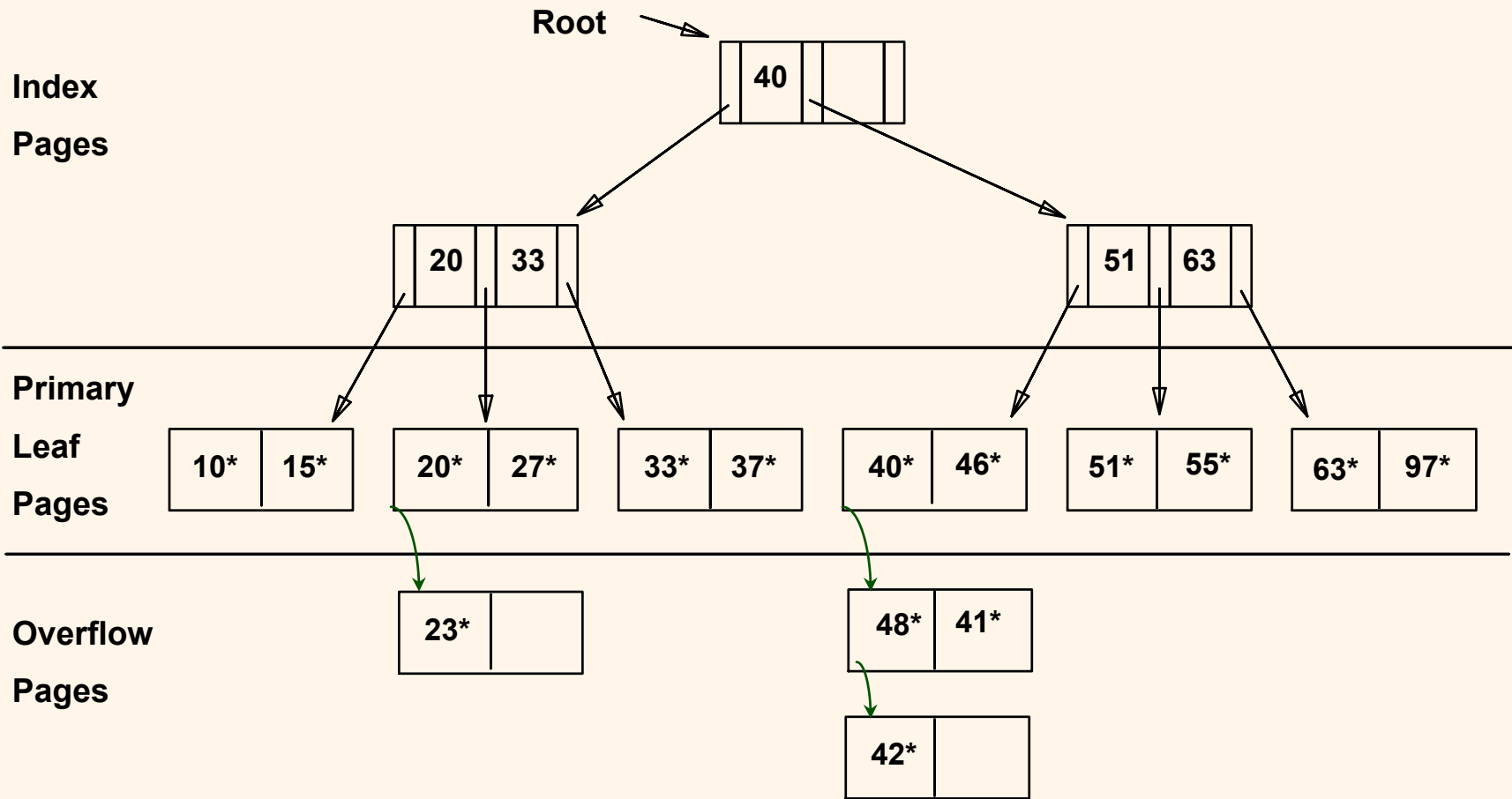
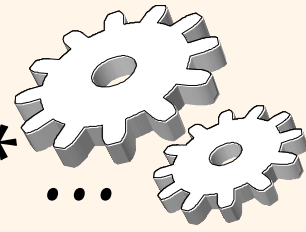


Example ISAM Tree

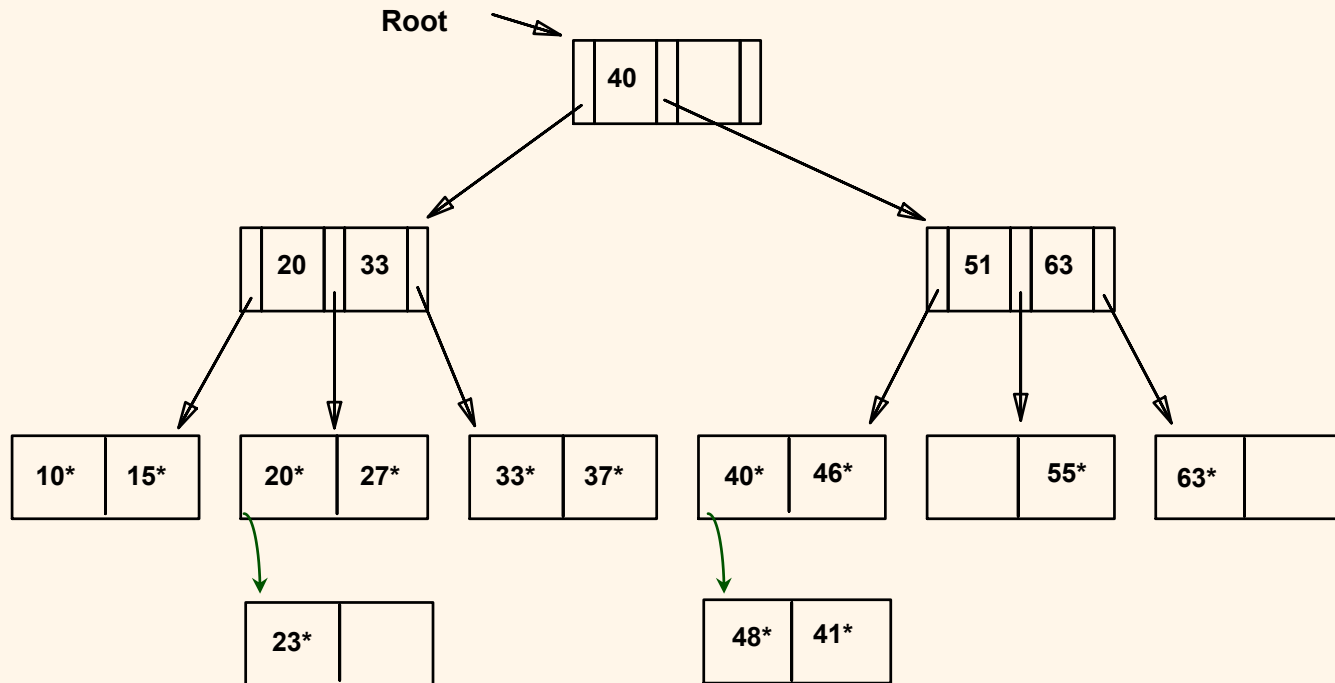
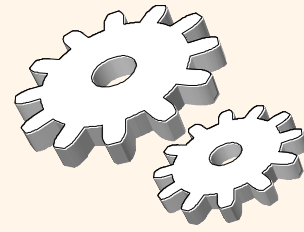
- ❖ Each node can hold 2 entries; no need for 'next-leaf-page' pointers. (Why?)



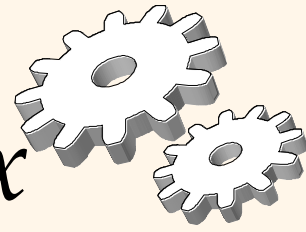
After Inserting 23, 48*, 41*, 42* ...*



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

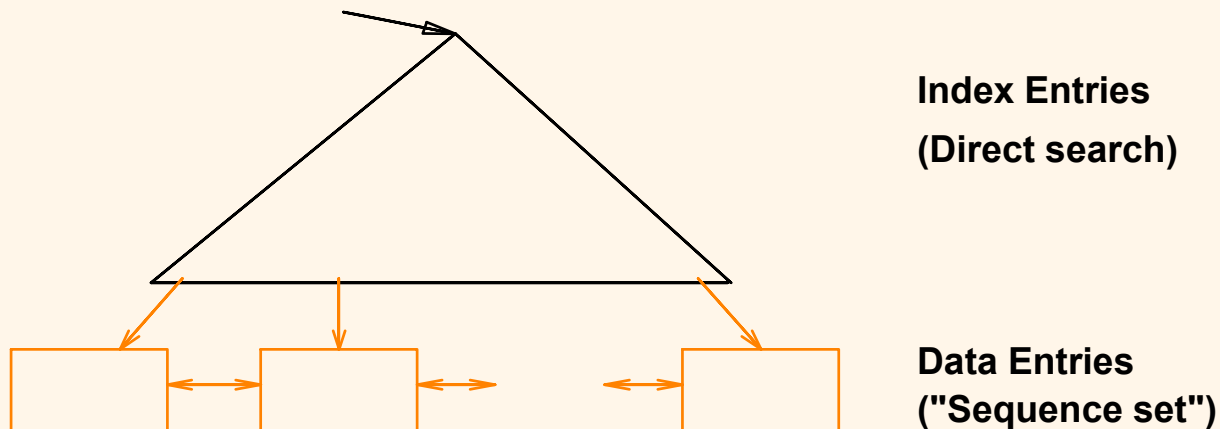


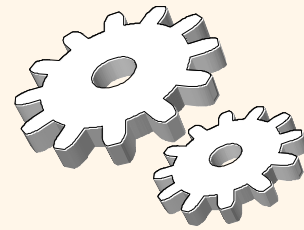
** Note that 51* appears in index levels, but not in leaf!*



B+ Tree: Most Widely Used Index

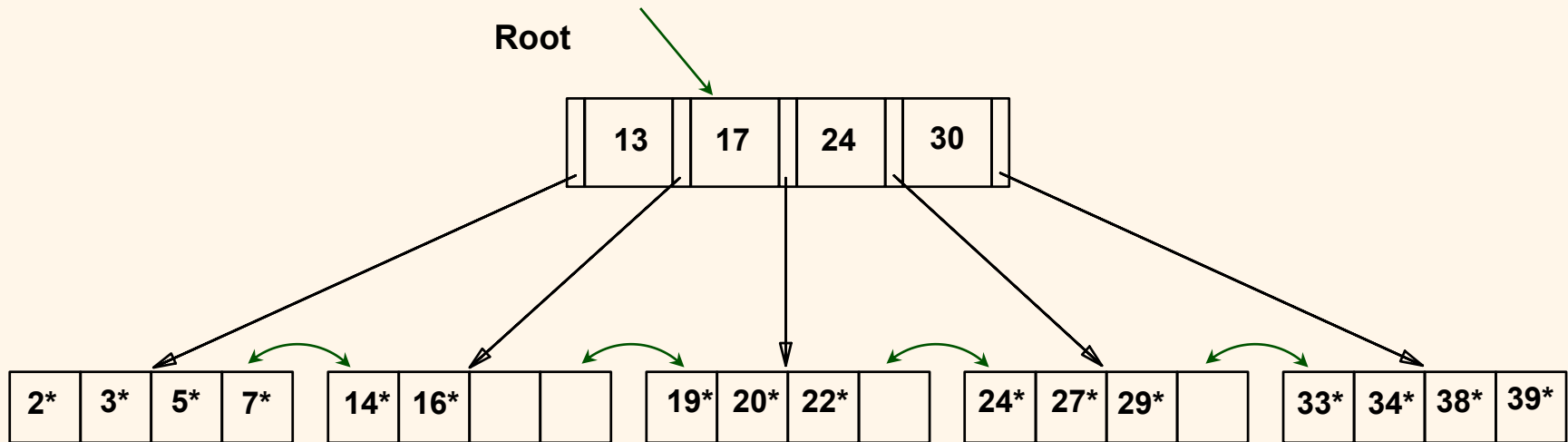
- ❖ Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- ❖ Minimum 50% occupancy (except for root). Each node contains $d \leq m \leq 2d$ entries. The parameter d is called the *order* of the tree.
- ❖ Supports equality and range-searches efficiently.



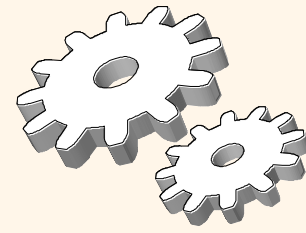


Example B+ Tree

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



** Based on the search for 15*, we know it is not in the tree!*



B+ Trees in Practice

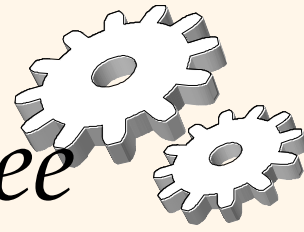
- ❖ Typical order: 100. Typical fill-factor: 67%.
 - average fanout = 133
- ❖ Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- ❖ Can often hold top levels in buffer pool:
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level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



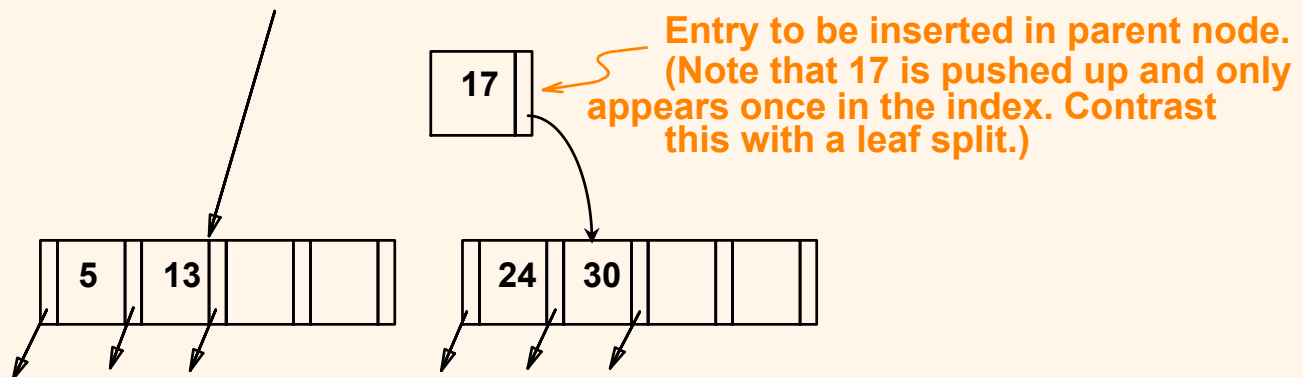
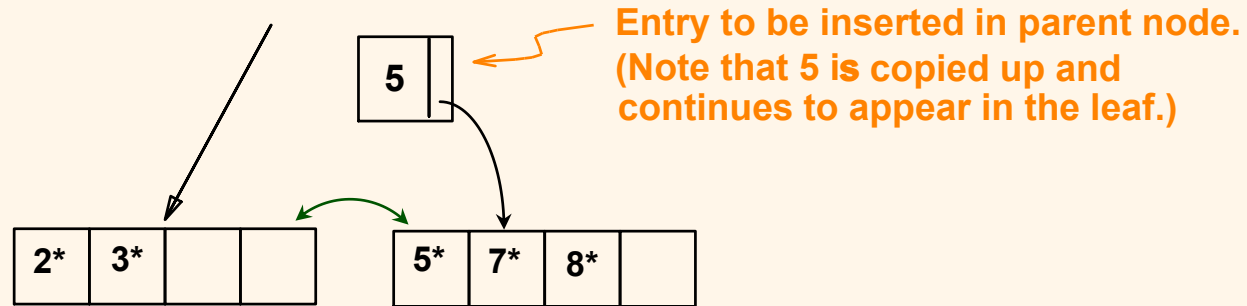
Inserting a Data Entry into a B+ Tree

- ❖ 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.

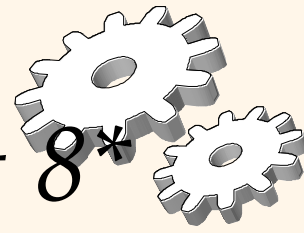
Inserting 8* into Example B+ Tree



- ❖ 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*



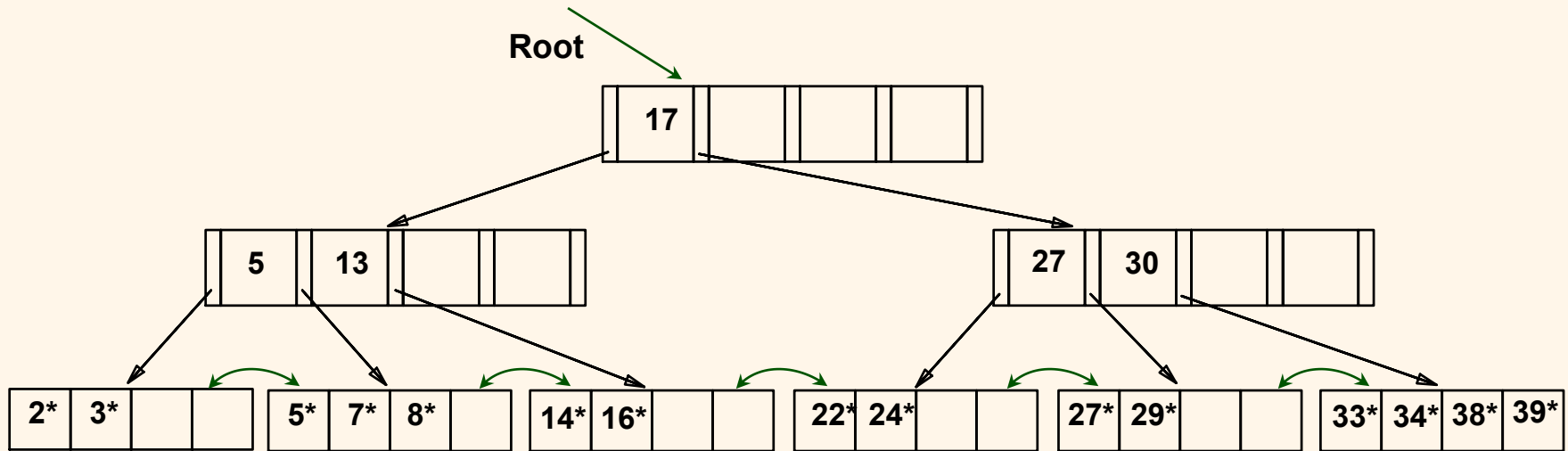
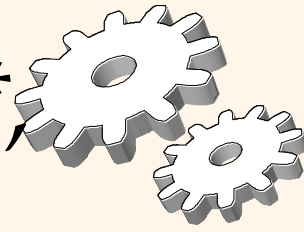
- v Notice that root was split, leading to increase in height.
- v In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.



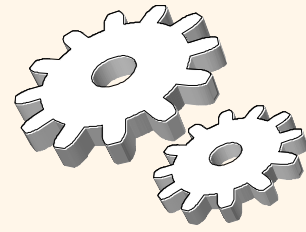
Deleting a Data Entry from a B+ Tree

- ❖ 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* ...

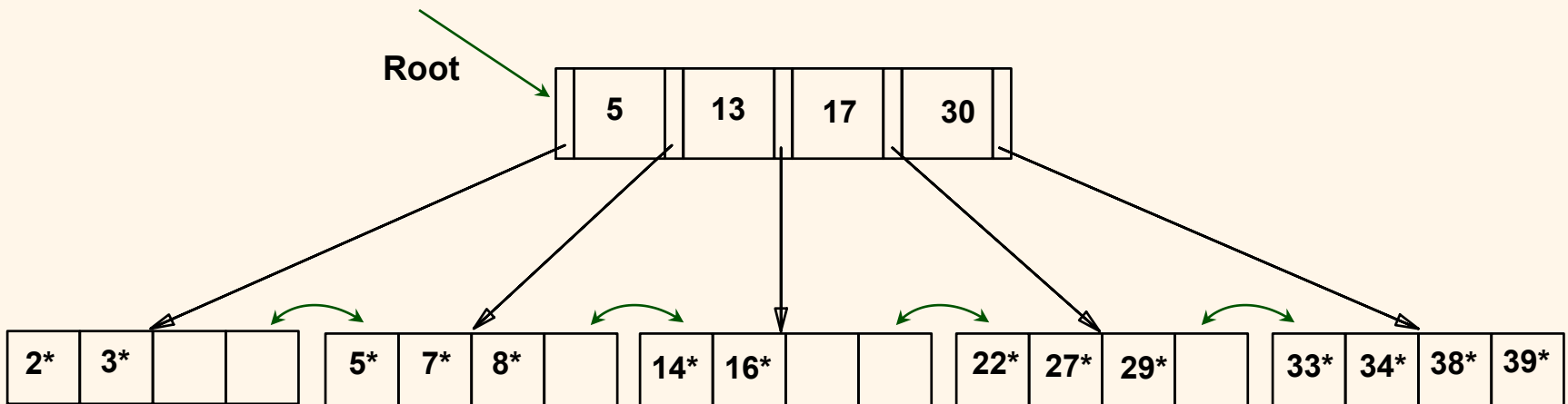
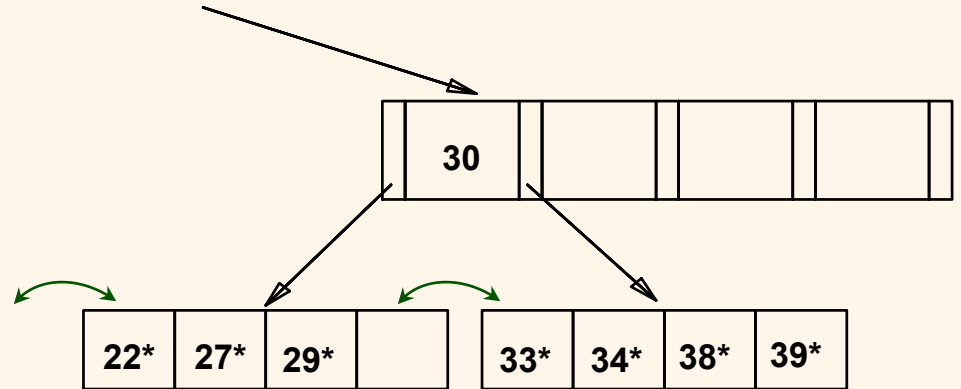


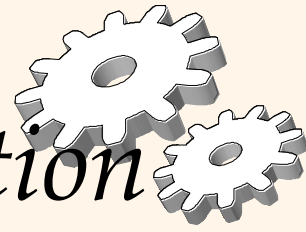
- ❖ Deleting 19* is easy.
- ❖ Deleting 20* is done with re-distribution.
Notice how middle key is *copied up*.



... And Then Deleting 24*

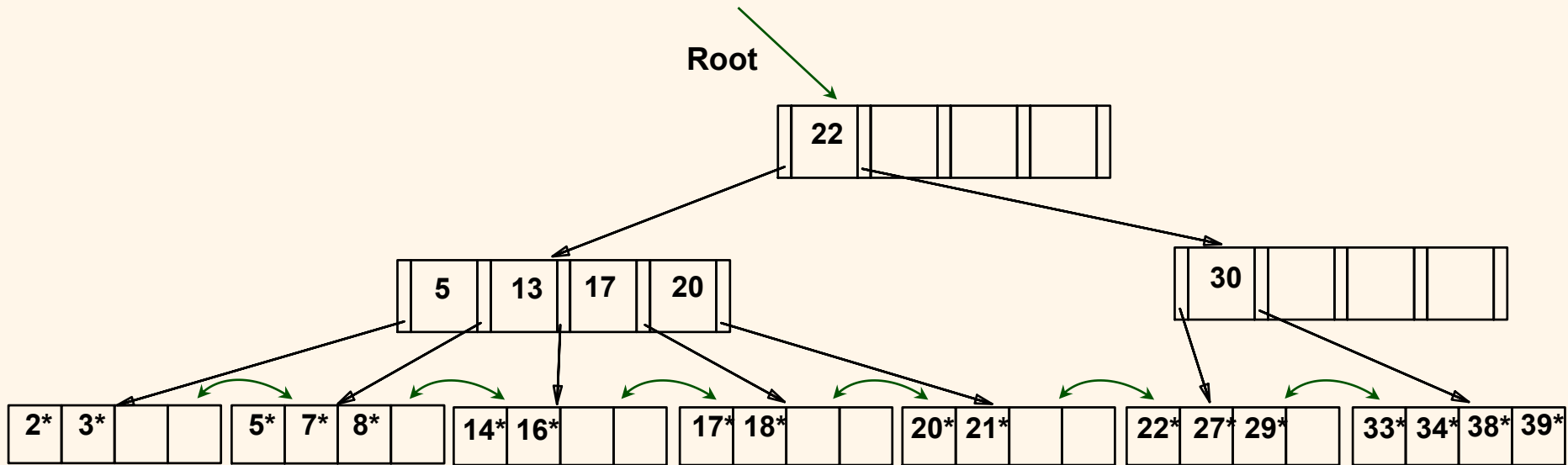
- ❖ Must merge.
- ❖ Observe *'toss'* of index entry (on right), and *'pull down'* of index entry (below).

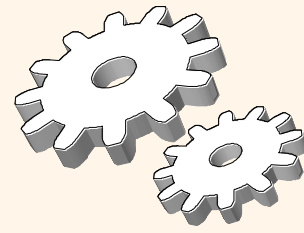




Example of Non-leaf Re-distribution

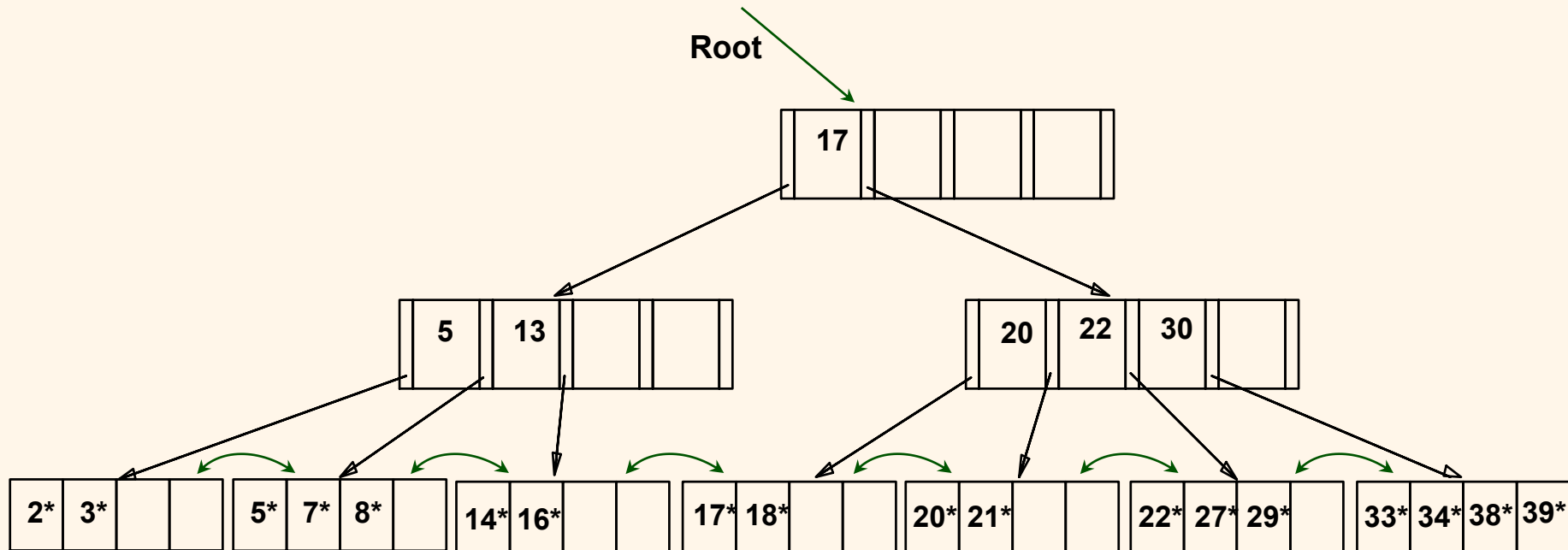
- ❖ Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- ❖ In contrast to previous example, can re-distribute entry from left child of root to right child.

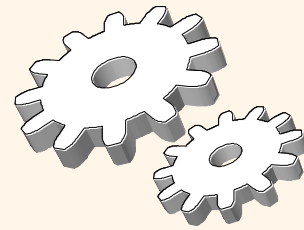




After Re-distribution

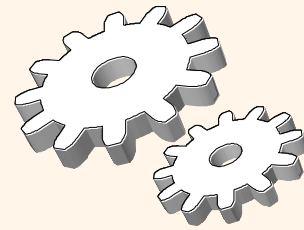
- ❖ Intuitively, entries are **re-distributed by 'pushing through'** the splitting entry in the parent node.
- ❖ It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.





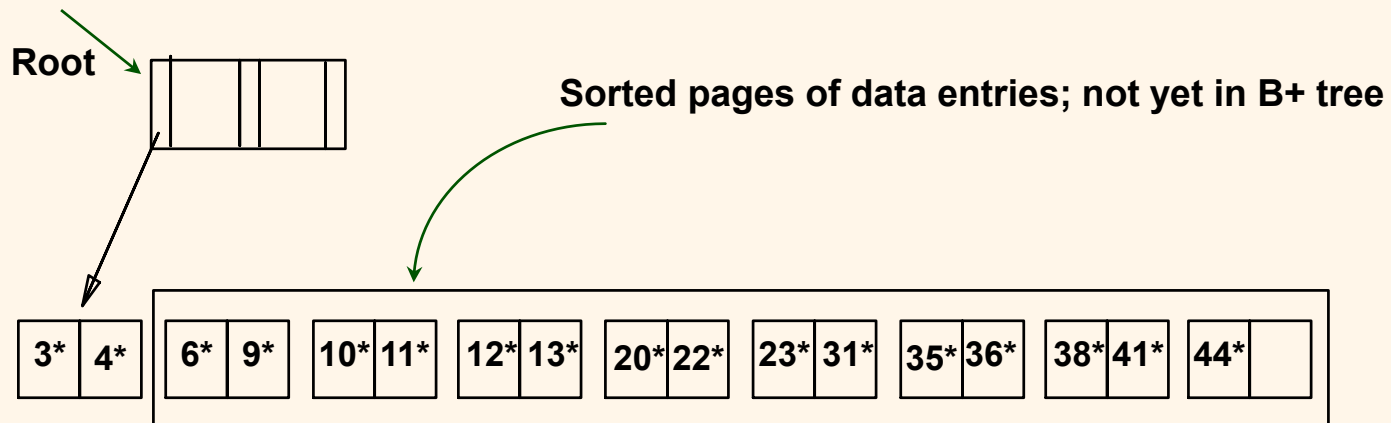
Prefix Key Compression

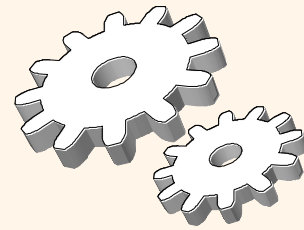
- ❖ Important to increase fan-out. (Why?)
- ❖ Key values in index entries only ‘direct traffic’; can often compress them.
 - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
 - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
 - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- ❖ Insert/delete must be suitably modified.



Bulk Loading of a B+ Tree

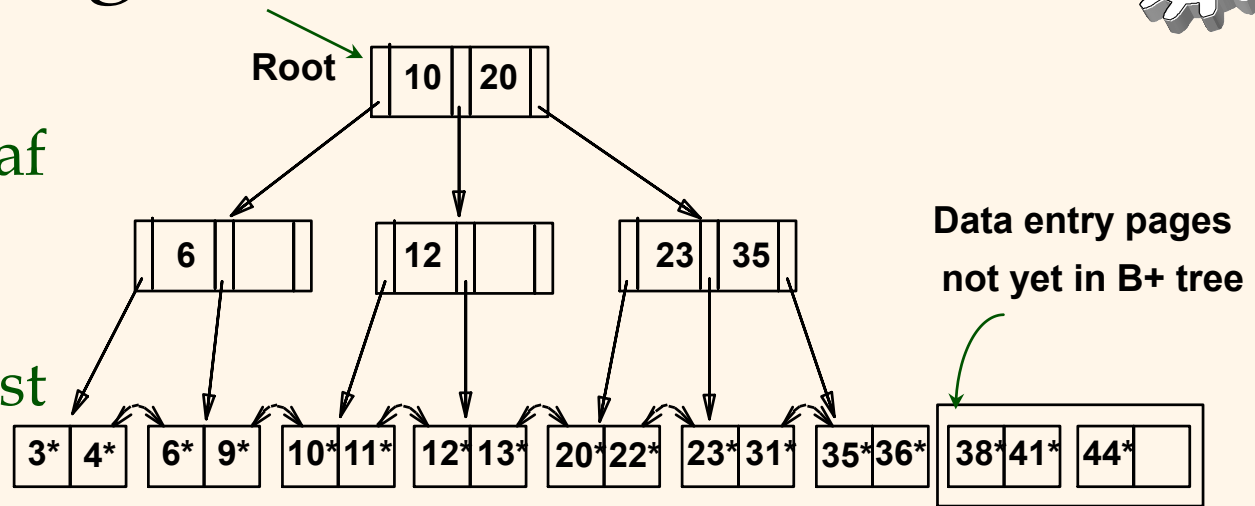
- ❖ If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- ❖ Bulk Loading can be done much more efficiently.
- ❖ *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.





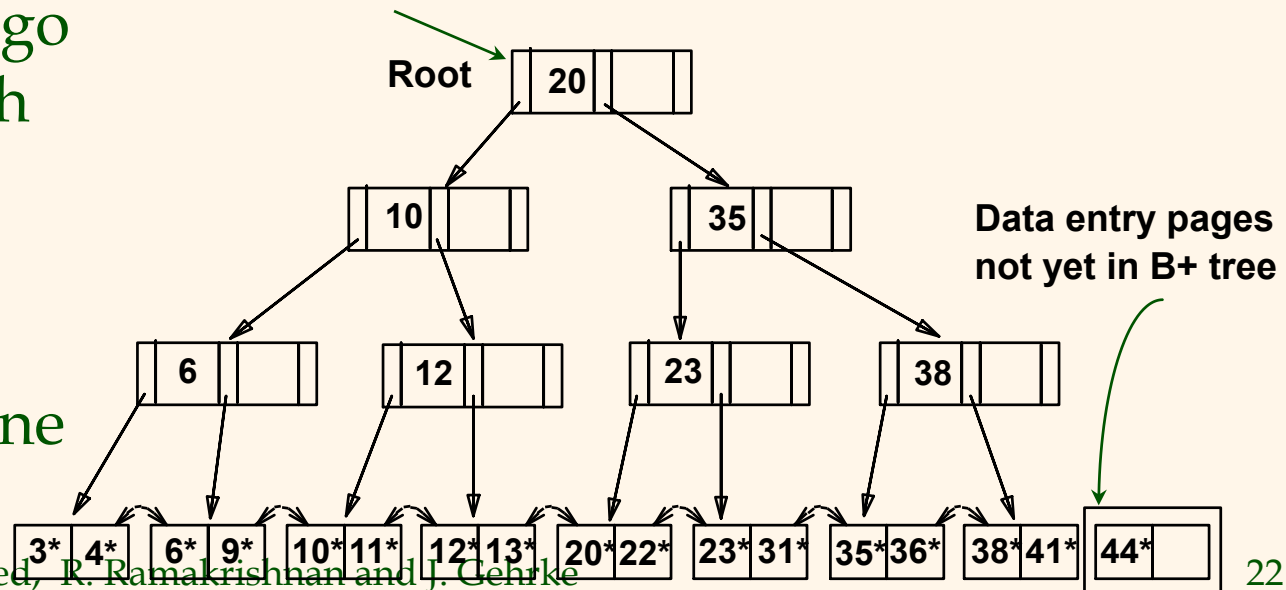
Bulk Loading (Contd.)

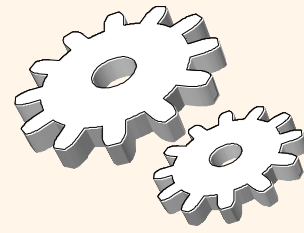
- ❖ Index entries for leaf pages always entered into right-most index page just above leaf level.



When this fills up, it splits. (Split may go up right-most path to the root.)

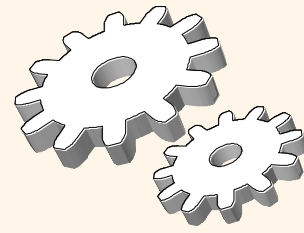
- ❖ Much faster than repeated inserts, especially when one considers locking!





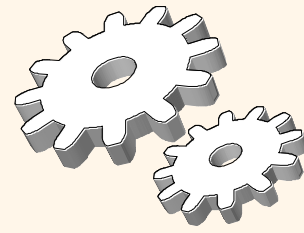
Summary of Bulk Loading

- ❖ Option 1: multiple inserts.
 - Slow.
 - Does not give sequential storage of leaves.
- ❖ Option 2: Bulk Loading
 - Has advantages for concurrency control.
 - Fewer I/Os during build.
 - Leaves will be stored sequentially (and linked, of course).
 - Can control “fill factor” on pages.



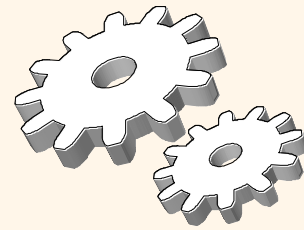
A Note on `Order`

- ❖ *Order (d)* concept replaced by physical space criterion in practice (*`at least half-full`*).
 - Index pages can typically hold many more entries than leaf pages.
 - Variable sized records and search keys mean different nodes will contain different numbers of entries.
 - Even with fixed length fields, multiple records with the same search key value (*duplicates*) can lead to variable-sized data entries (if we use Alternative (3)).



Summary

- ❖ Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ❖ ISAM is a static structure.
 - Only leaf pages modified; overflow pages needed.
 - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- ❖ B+ tree is a dynamic structure.
 - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
 - High fanout (**F**) means depth rarely more than 3 or 4.
 - Almost always better than maintaining a sorted file.

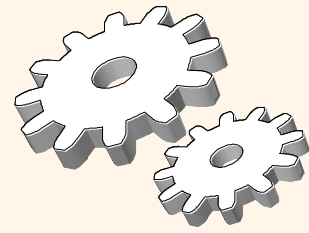


Summary (Contd.)

- Typically, **67%** occupancy on average.
- Usually preferable to ISAM, modulo *locking* considerations; adjusts to growth gracefully.
- If data entries are data records, splits can change rids!
- ❖ Key compression increases fanout, reduces height.
- ❖ Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- ❖ Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.

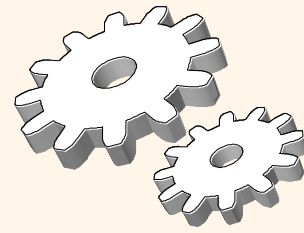
12

- Hash-Based Indexing (**Ch.11**)



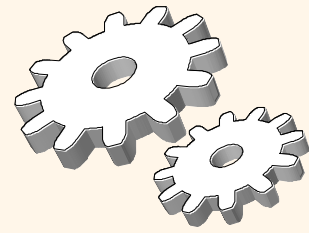
Hash-Based Indexes

Chapter 11



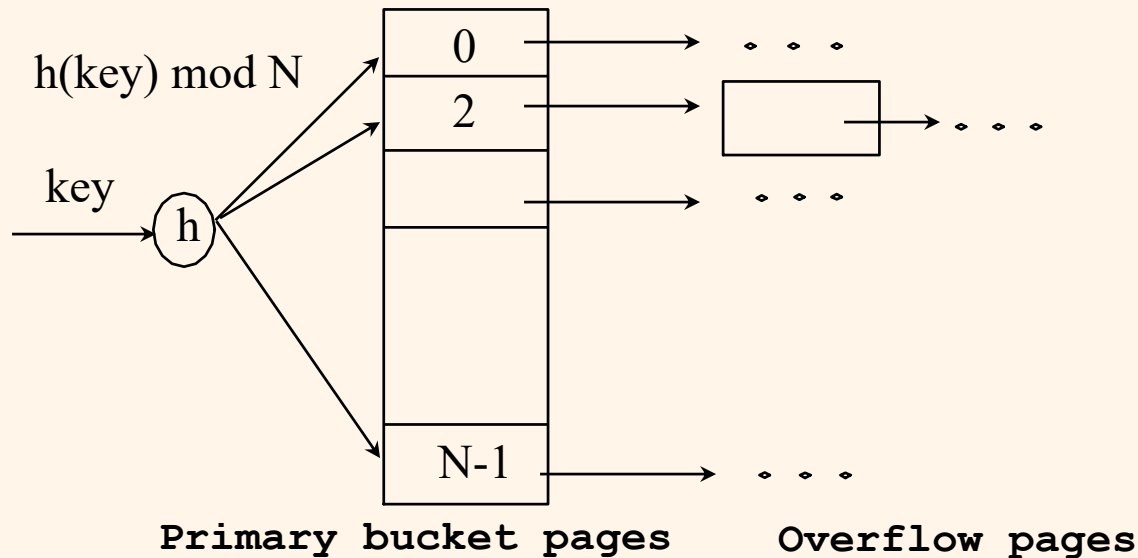
Introduction

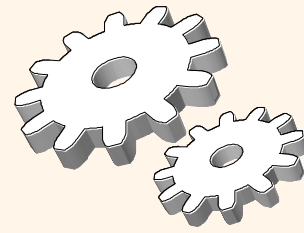
- ❖ *As for any index, 3 alternatives for data entries \mathbf{k}^* :*
 - Data record with key value \mathbf{k}
 - $\langle \mathbf{k}, \text{rid of data record with search key value } \mathbf{k} \rangle$
 - $\langle \mathbf{k}, \text{list of rids of data records with search key } \mathbf{k} \rangle$
 - Choice orthogonal to the *indexing technique*
- ❖ *Hash-based* indexes are best for *equality selections*.
Cannot support range searches.
- ❖ Static and dynamic hashing techniques exist;
trade-offs similar to ISAM vs. B+ trees.



Static Hashing

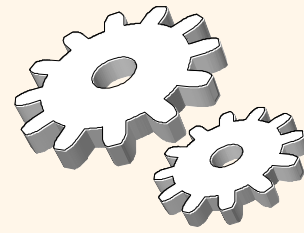
- ❖ # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- ❖ $h(k) \bmod M =$ bucket to which data entry with key k belongs. ($M = \#$ of buckets)





Static Hashing (Contd.)

- ❖ Buckets contain *data entries*.
- ❖ Hash fn works on *search key* field of record r . Must distribute values over range $0 \dots M-1$.
 - $h(key) = (a * key + b)$ usually works well.
 - a and b are constants; lots known about how to tune h .
- ❖ **Long overflow chains** can develop and degrade performance.
 - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

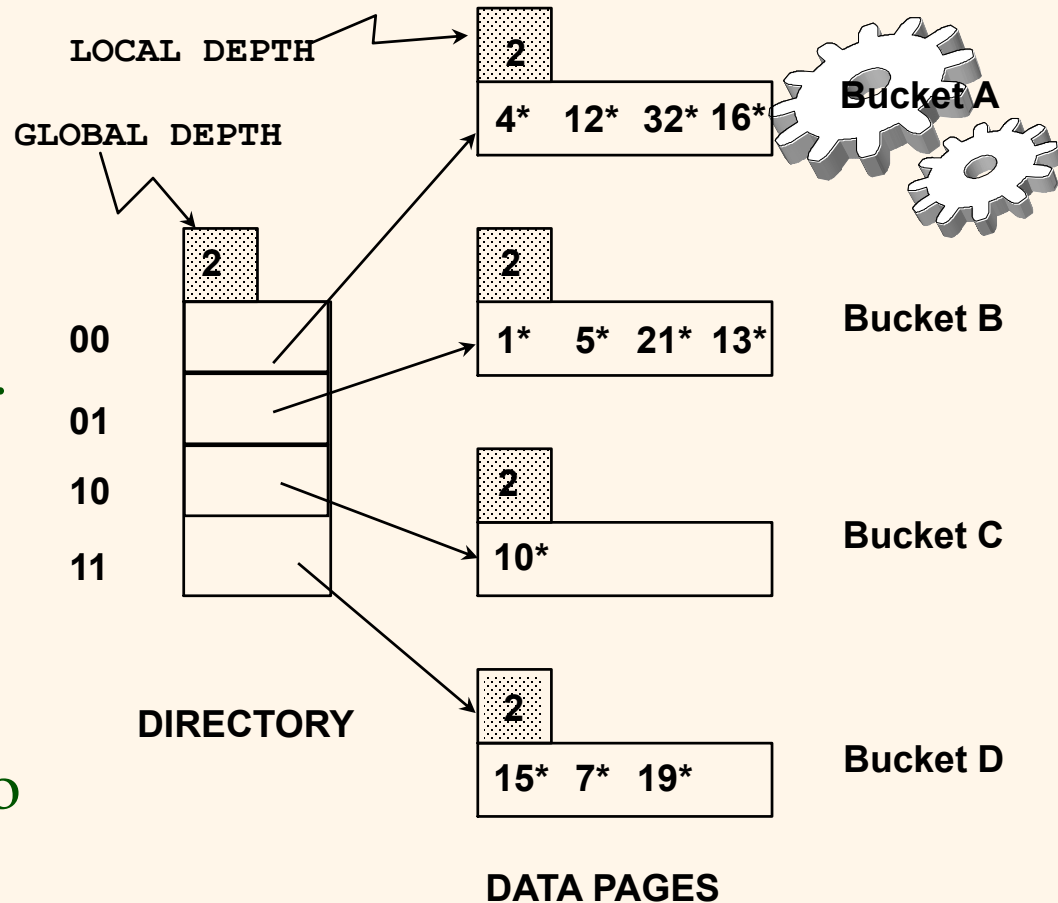


Extendible Hashing

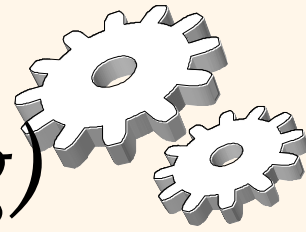
- ❖ Situation: Bucket (primary page) becomes full.
Why not re-organize file by *doubling* # of buckets?
 - Reading and writing all pages is expensive!
 - Idea: Use directory of pointers to buckets, double # of buckets by *doubling the directory*, splitting just the bucket that overflowed!
 - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split.
No overflow page!
 - Trick lies in how hash function is adjusted!

Example

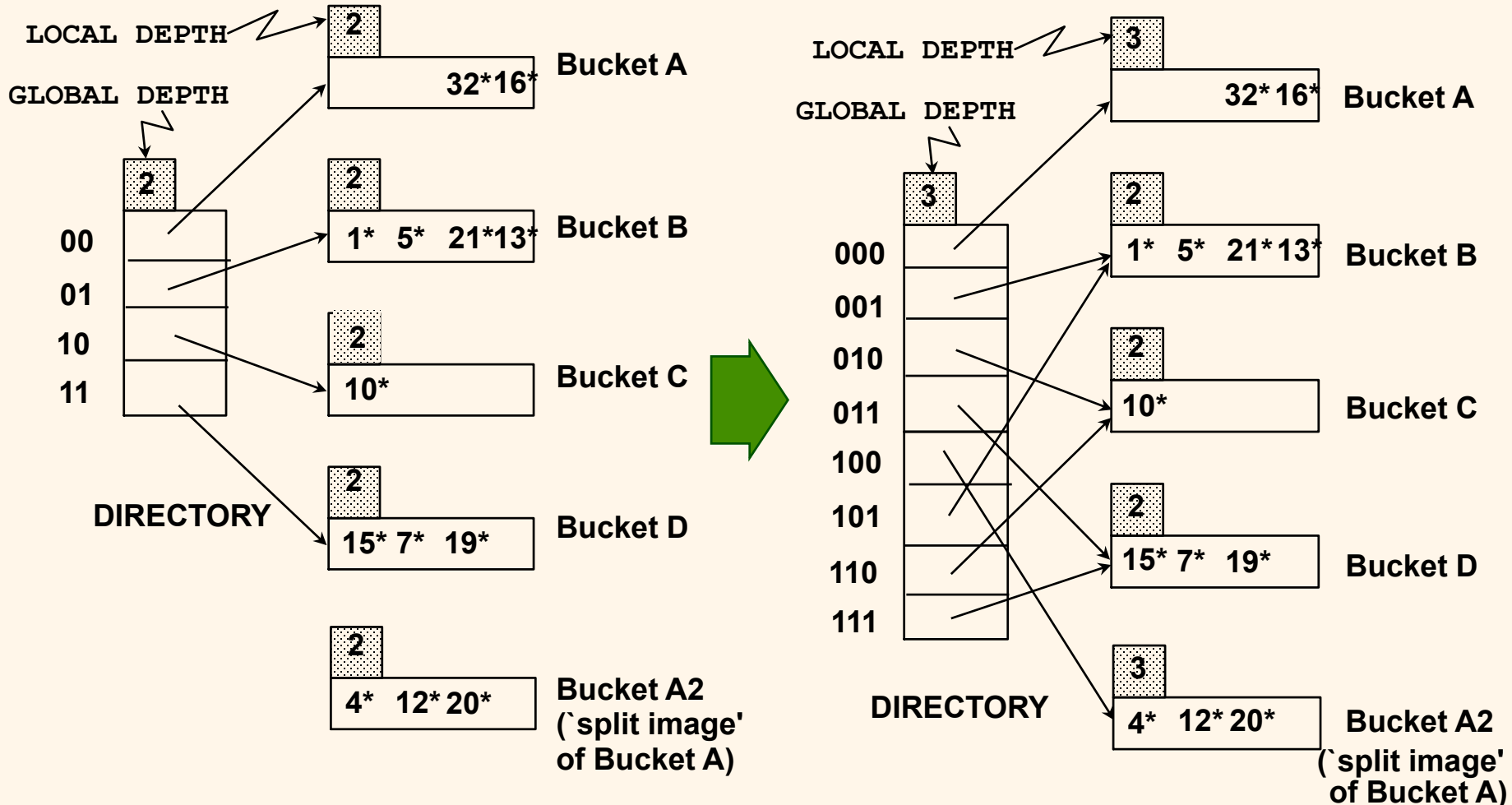
- ❖ Directory is array of size 4.
- ❖ To find bucket for r , take last '*global depth*' # bits of $h(r)$; we denote r by $h(r)$.
 - If $h(r) = 5 = \text{binary } 101$, it is in bucket pointed to by 01.

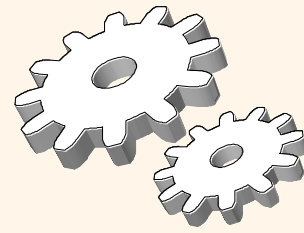


- ❖ Insert: If bucket is full, *split* it (allocate new page, re-distribute).
- ❖ If necessary, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)



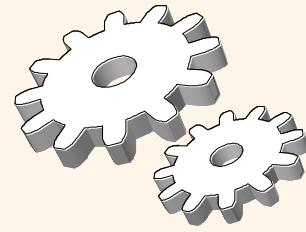
Insert $h(r)=20$ (Causes Doubling)





Points to Note

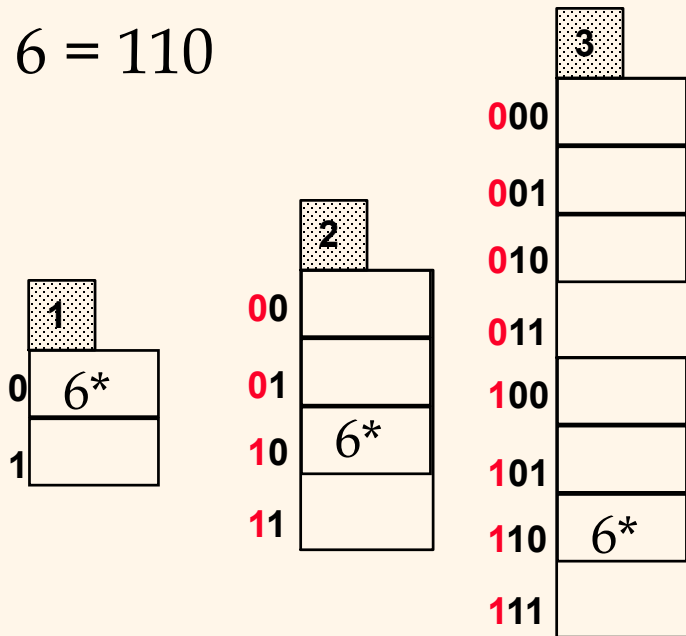
- ❖ 20 = binary 10100. Last 2 bits (00) tell us r belongs in A or A2. Last 3 bits needed to tell which.
 - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
 - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.
- ❖ When does bucket split cause directory doubling?
 - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become $>$ *global depth*; directory is doubled by *copying it over* and 'fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)



Directory Doubling

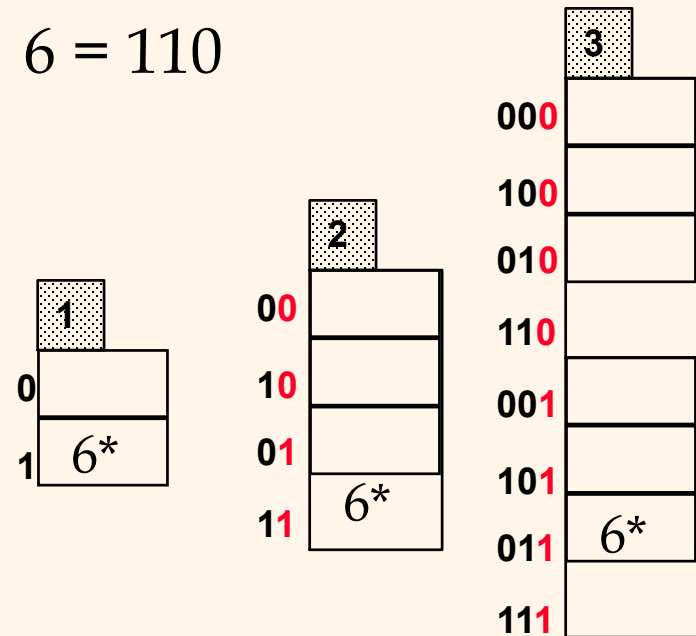
Why use least significant bits in directory?
⇔ Allows for doubling via copying!

6 = 110



Least Significant

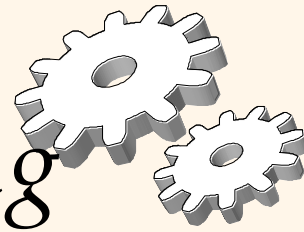
6 = 110



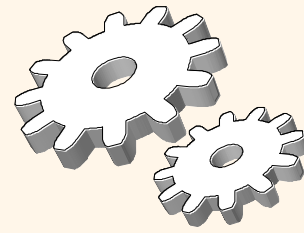
Most Significant

vs.

Comments on Extendible Hashing



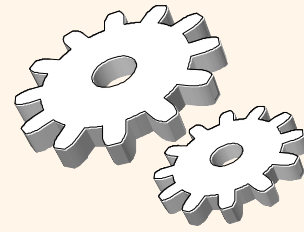
- ❖ If directory fits in memory, equality search answered with one disk access; else two.
 - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
 - Directory grows in spurts, and, if the distribution of *hash values* is skewed, directory can grow large.
 - Multiple entries with same hash value cause problems!
- ❖ **Delete**: If removal of data entry makes bucket empty, can be merged with 'split image'. If each directory element points to same bucket as its split image, can halve directory.



Linear Hashing

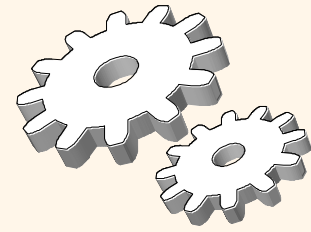
- ❖ This is another dynamic hashing scheme, an alternative to Extendible Hashing.
- ❖ LH handles the problem of long overflow chains without using a directory, and handles duplicates.
- ❖ Idea: Use a family of hash functions $\mathbf{h}_0, \mathbf{h}_1, \mathbf{h}_2, \dots$
 - $\mathbf{h}_i(\text{key}) = \mathbf{h}(\text{key}) \bmod(2^i\mathbf{N})$; \mathbf{N} = initial # buckets
 - \mathbf{h} is some hash function (range is *not* 0 to $\mathbf{N}-1$)
 - If $\mathbf{N} = 2^{d_0}$, for some d_0 , \mathbf{h}_i consists of applying \mathbf{h} and looking at the last d_i bits, where $d_i = d_0 + i$.
 - \mathbf{h}_{i+1} doubles the range of \mathbf{h}_i (similar to directory doubling)

Linear Hashing (Contd.)

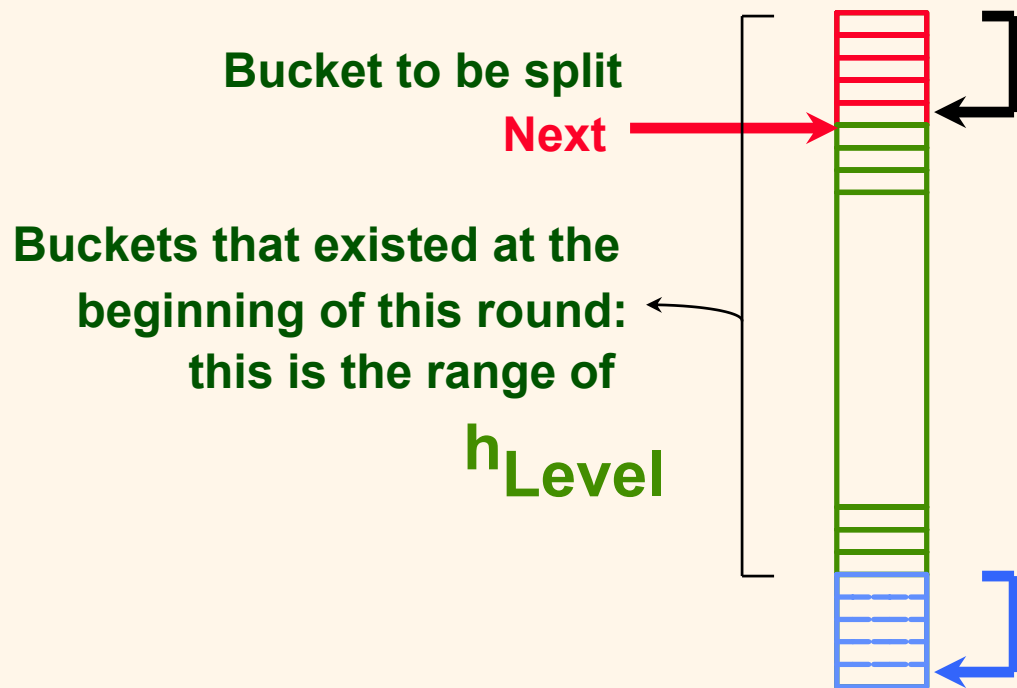


- ❖ Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
 - **Splitting proceeds in `rounds`**. Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
 - **Current round number is *Level***.
 - **Search**: To find bucket for data entry r , find $\mathbf{h}_{Level}(r)$:
 - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R ', r belongs here.
 - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.

Overview of LH File

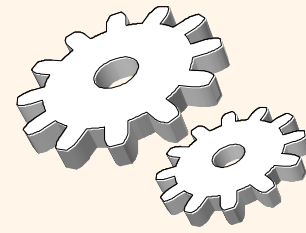


❖ In the middle of a round.



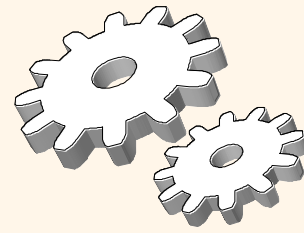
Buckets split in this round:
If h_{Level} (search key value) is in this range, must use $h_{Level+1}$ (search key value) to decide if entry is in 'split image' bucket.

'split image' buckets:
created (through splitting of other buckets) in this round



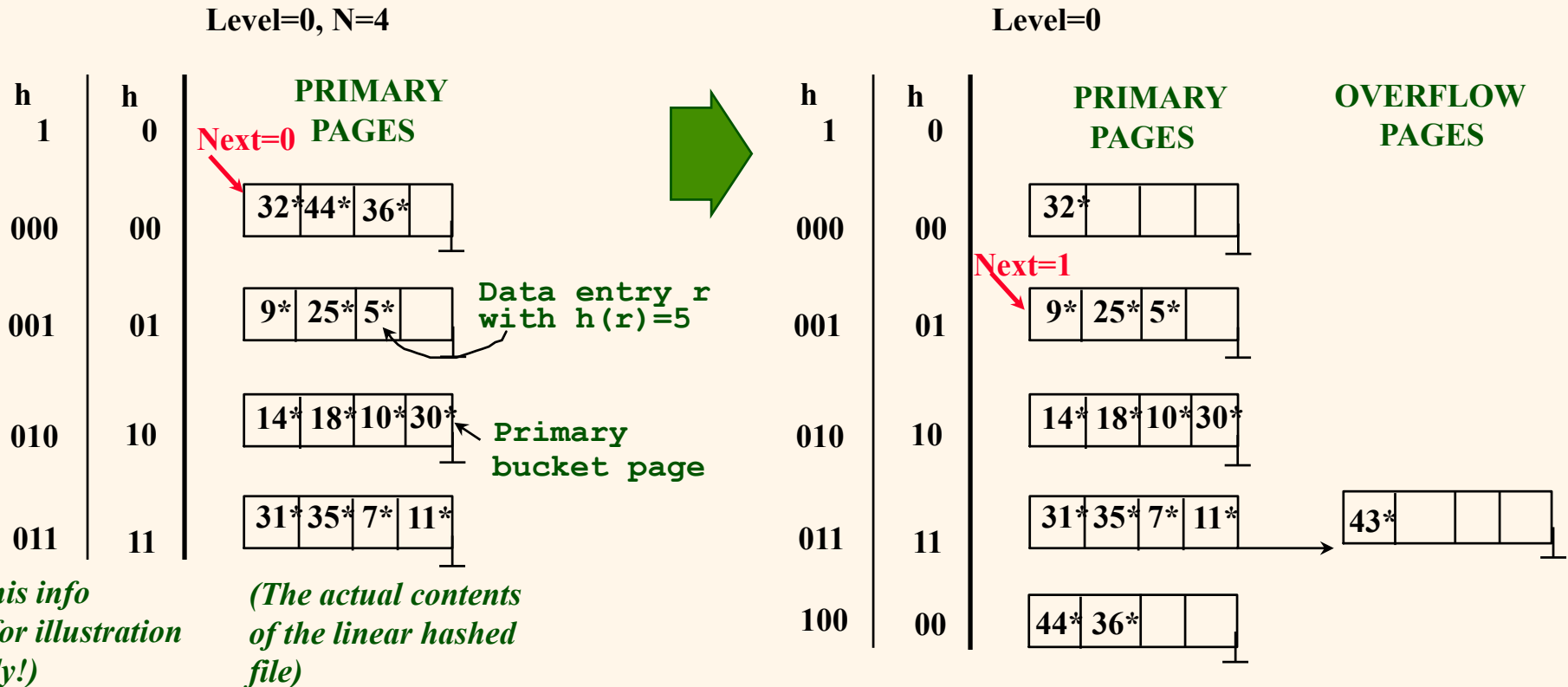
Linear Hashing (Contd.)

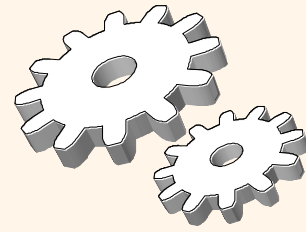
- ❖ **Insert**: Find bucket by applying $h_{Level} / h_{Level+1}$:
 - If bucket to insert into is full:
 - Add overflow page and insert data entry.
 - (*Maybe*) Split *Next* bucket and increment *Next*.
- ❖ Can choose any criterion to `trigger' split.
- ❖ Since buckets are split round-robin, long overflow chains don't develop!
- ❖ Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.



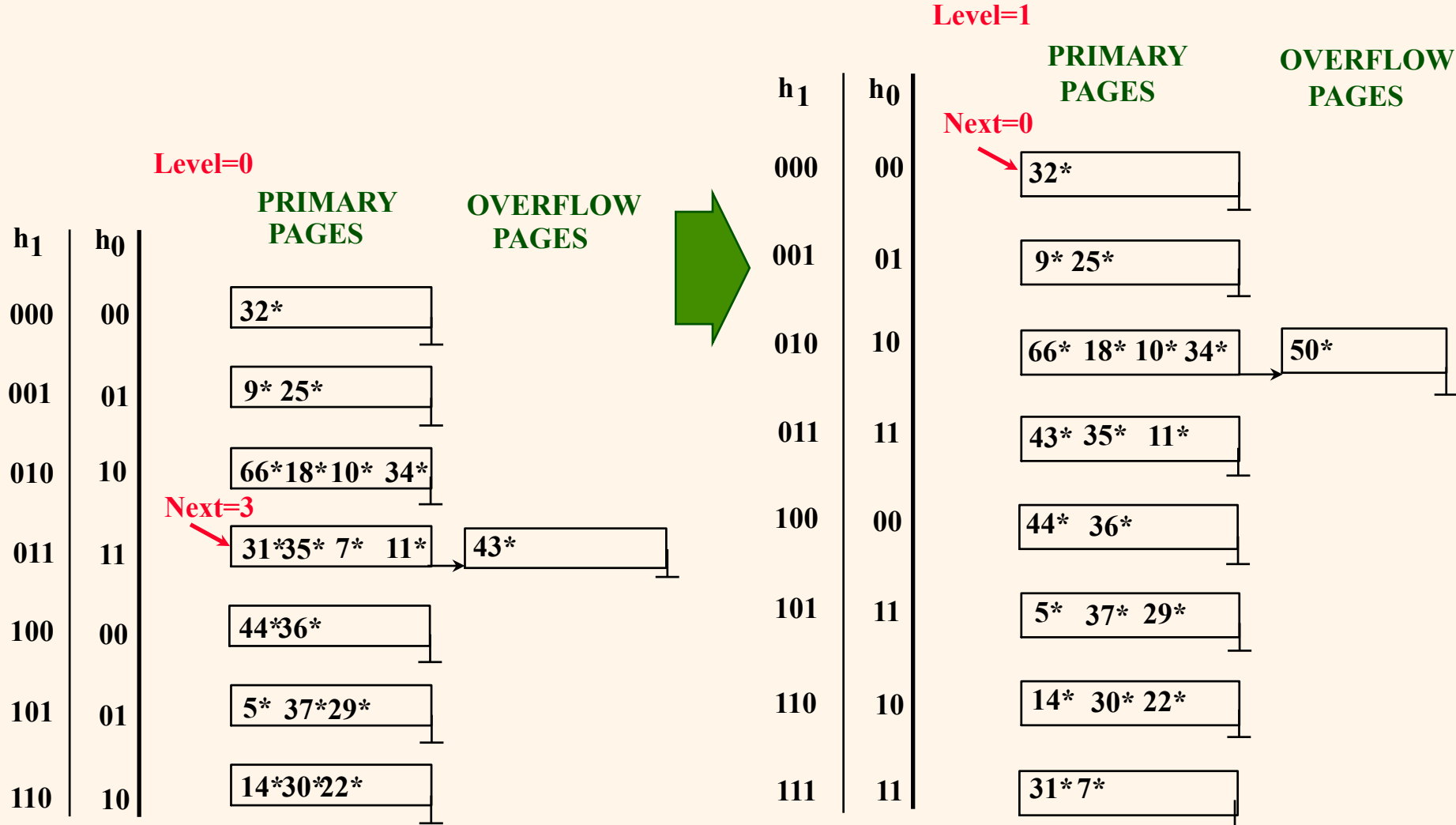
Example of Linear Hashing

- ❖ On split, $h_{\text{Level}+1}$ is used to re-distribute entries.

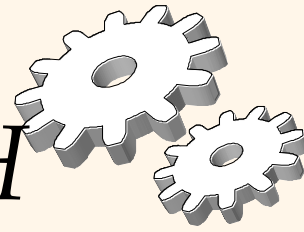




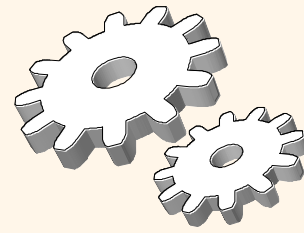
Example: End of a Round



LH Described as a Variant of EH

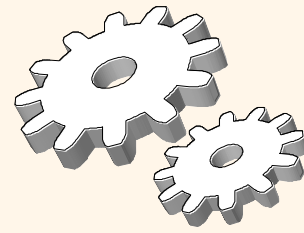


- ❖ The two schemes are actually quite similar:
 - Begin with an EH index where directory has N elements.
 - Use overflow pages, split buckets round-robin.
 - First split is at bucket 0. (Imagine directory being doubled at this point.) But elements $\langle 1, N+1 \rangle$, $\langle 2, N+2 \rangle$, ... are the same. So, need only create directory element N , which differs from 0, now.
 - When bucket 1 splits, create directory element $N+1$, etc.
- ❖ So, directory can double gradually. Also, primary bucket pages are created in order. If they are *allocated* in sequence too (so that finding i 'th is easy), we actually don't need a directory! Voila, LH.



Summary

- ❖ Hash-based indexes: best for equality searches, cannot support range searches.
- ❖ Static Hashing can lead to long overflow chains.
- ❖ Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. (*Duplicates may require overflow pages.*)
 - Directory to keep track of buckets, doubles periodically.
 - Can get large with skewed data; additional I/O if this does not fit in main memory.

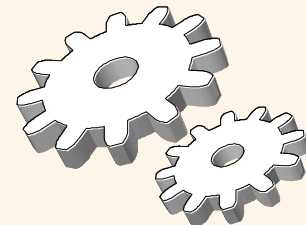


Summary (Contd.)

- ❖ Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
 - Overflow pages not likely to be long.
 - Duplicates handled easily.
 - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
 - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- ❖ For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed!

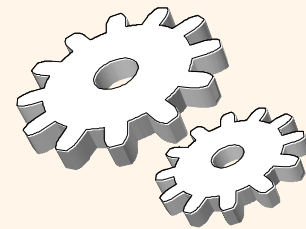
13

- Overview of Transaction Management (**Ch.16**)



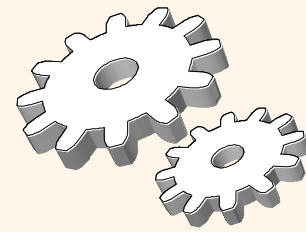
Transaction Management Overview

Chapter 16



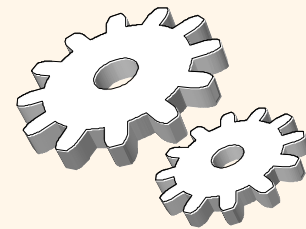
Transactions

- ❖ Concurrent execution of user programs is essential for good DBMS performance.
 - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- ❖ A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- ❖ A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes.



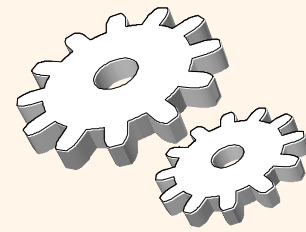
Concurrency in a DBMS

- ❖ Users submit transactions, and can think of each transaction as executing by itself.
 - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
 - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
 - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
 - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- ❖ Issues: Effect of *interleaving* transactions, and *crashes*.



Atomicity of Transactions

- ❖ A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- ❖ A very important property guaranteed by the DBMS for all transactions is that they are *atomic*. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
 - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.

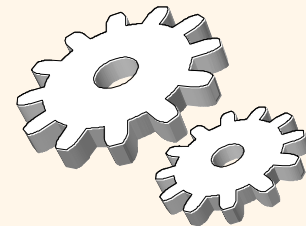


Example

- ❖ Consider two transactions (*Xacts*):

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- ❖ Intuitively, the first transaction is transferring \$100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- ❖ There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.



Example (Contd.)

- ❖ Consider a possible interleaving (*schedule*):

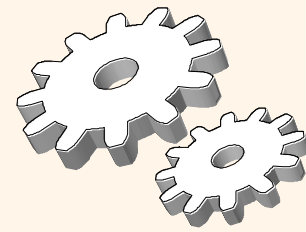
T1:	$A=A+100,$	$B=B-100$
T2:	$A=1.06*A,$	$B=1.06*B$

- ❖ This is OK. But what about:

T1:	$A=A+100,$	$B=B-100$
T2:	$A=1.06*A, B=1.06*B$	

- ❖ The DBMS's view of the second schedule:

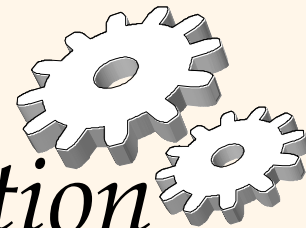
T1:	$R(A), W(A),$	$R(B), W(B)$
T2:	$R(A), W(A), R(B), W(B)$	



Scheduling Transactions

- ❖ Serial schedule: Schedule that does not interleave the actions of different transactions.
- ❖ Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- ❖ Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)



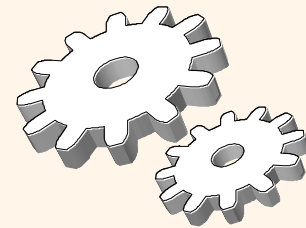
Anomalies with Interleaved Execution

- ❖ Reading Uncommitted Data (WR Conflicts, “dirty reads”):

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), C	

- ❖ Unrepeatable Reads (RW Conflicts):

T1:	R(A),	R(A), W(A), C
T2:	R(A), W(A), C	

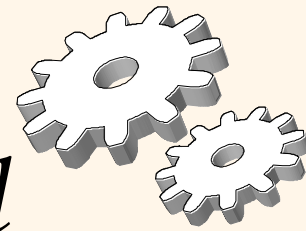


Anomalies (Continued)

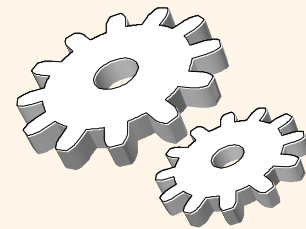
❖ Overwriting Uncommitted Data (WW Conflicts):

T1:	W(A),	W(B), C
T2:	W(A), W(B), C	

Lock-Based Concurrency Control

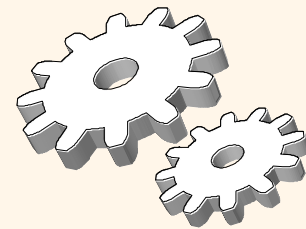


- ❖ Strict Two-phase Locking (Strict 2PL) Protocol:
 - Each Xact must obtain a *S (shared)* lock on object before reading, and an *X (exclusive)* lock on object before writing.
 - All locks held by a transaction are released when the transaction completes
 - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- ❖ Strict 2PL allows only serializable schedules.



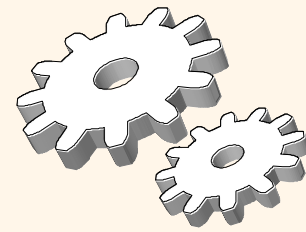
Aborting a Transaction

- ❖ If a transaction T_i is aborted, all its actions have to be undone. Not only that, if T_j reads an object last written by T_i , T_j must be aborted as well!
- ❖ Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
 - If T_i writes an object, T_j can read this only after T_i commits.
- ❖ In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.



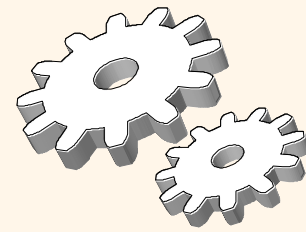
The Log

- ❖ The following actions are recorded in the log:
 - *Ti writes an object*: the old value and the new value.
 - Log record must go to disk before the changed page!
 - *Ti commits/aborts*: a log record indicating this action.
- ❖ Log records are chained together by Xact id, so it's easy to undo a specific Xact.
- ❖ Log is often *duplexed* and *archived* on stable storage.
- ❖ All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.



Recovering From a Crash

- ❖ There are 3 phases in the *Aries* recovery algorithm:
 - Analysis: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
 - Redo: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
 - Undo: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)



Summary

- ❖ Concurrency control and recovery are among the most important functions provided by a DBMS.
- ❖ Users need not worry about concurrency.
 - System automatically inserts lock/unlock requests and schedules actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- ❖ Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
 - *Consistent state*: Only the effects of committed Xacts seen.

Computers and Data Organization

CS281

Department of Computer Engineering, Bilkent University

Dr. Mustafa Değerli



Bilkent University