

# **Chapter 7: Normalization**

# **Overview of Normalization**

# Features of Good Relational Designs

- Suppose we combine *instructor* and *department* into *in\_dep*, which represents the natural join on the relations *instructor* and *department*

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

- There is repetition of information – **REDUNDANCY**
- Runs the risk that budget for a specific dept. be updated for some tuples but not for the rest – **INCONSISTENCY**

## Causes of Redundancy and Inconsistency:-

- INSERTION ANOMALY** - Need to use null values (if we add a new department with no instructors)
- DELETION ANOMALY** – Delete last employee from a dept.
- UPDATION ANOMALY** – Update budget for a specific dept.

# A Combined Schema Without Repetition

Not all combined schemas result in repetition of information

- Consider combining relations
  - $\text{sec\_class}(\text{sec\_id}, \text{building}, \text{room\_number})$  and
  - $\text{section}(\text{course\_id}, \text{sec\_id}, \text{semester}, \text{year})$
- into one relation
  - $\text{section}(\text{course\_id}, \text{sec\_id}, \text{semester}, \text{year}, \text{building}, \text{room\_number})$
- No repetition in this case

# Decomposition

- The only way to avoid the repetition-of-information problem in the *in\_dep* schema is to decompose it into two schemas – *instructor* and *department* schemas.
- Not all decompositions are good. Suppose we decompose

*employee*(*ID*, *name*, *street*, *city*, *salary*)

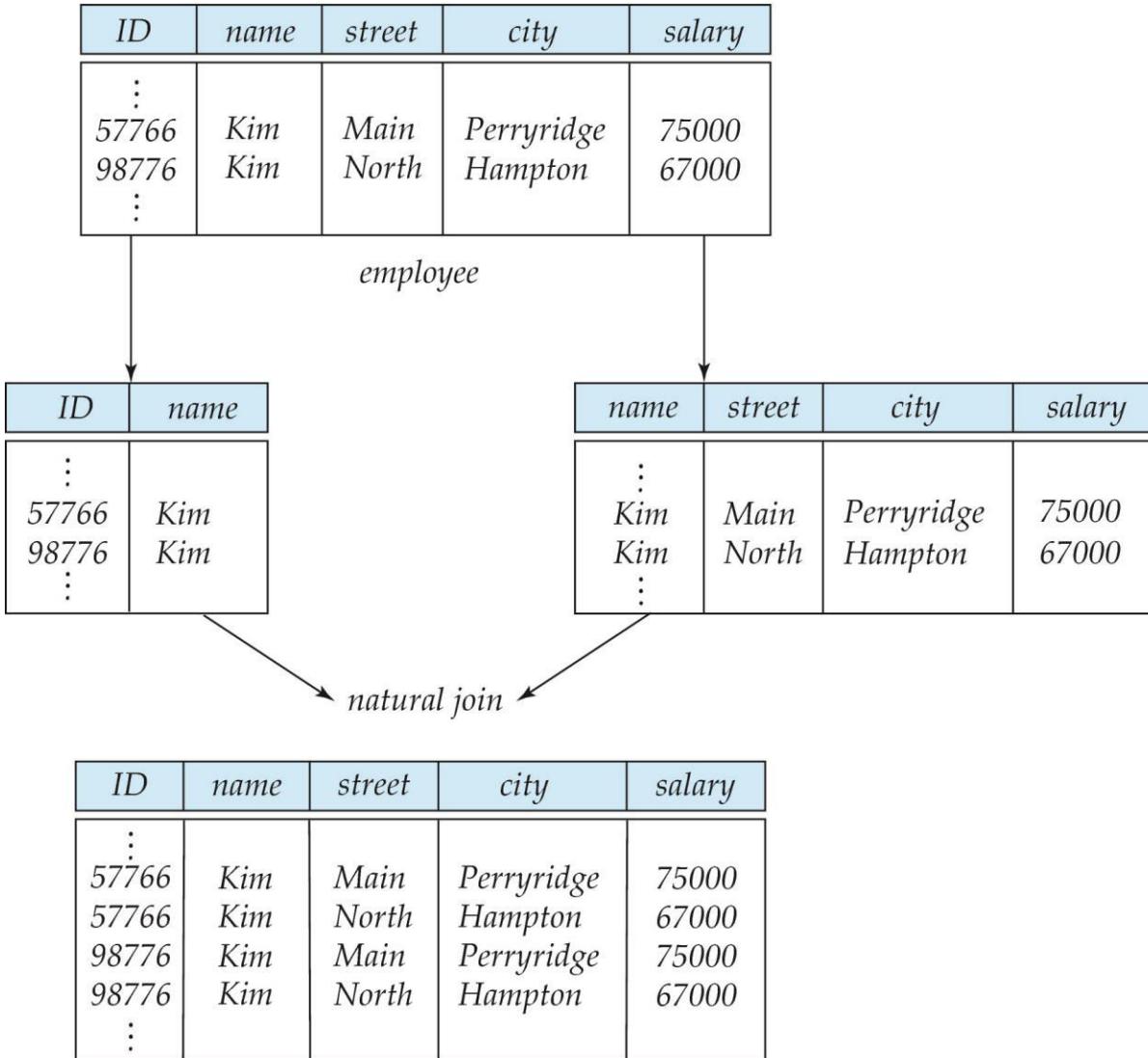
into

*employee1* (*ID*, *name*)

*employee2* (*name*, *street*, *city*, *salary*)

- The problem arises when we have two employees with the same name
- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

# A Lossy Decomposition



# Lossless Decomposition

- Let  $R$  be a relation schema and let  $R_1$  and  $R_2$  form a decomposition of  $R$ . That is  $R = R_1 \cup R_2$
- We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing  $R$  with the two relation schemas  $R_1 \cup R_2$
- Formally,

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

- And, conversely a decomposition is lossy if  
 $r \subset \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$   $\rightarrow$  Although #tuples increases, content of information decreases

# Example of Lossless Decomposition

- Decomposition of  $R = (A, B, C)$

$$R_1 = (A, B) \quad R_2 = (B, C)$$

A	B	C
$\alpha$	1	A

$r$

A	B
$\alpha$	1

$\Pi_{A,B}(r)$

B	C
1	A

$\Pi_{B,C}(r)$

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B	C
$\alpha$	1	A

# Normalization Theory

- Decide whether a particular relation  $R$  is in “good” form.
- In the case that a relation  $R$  is “not in good” form, decompose it into set of relations  $\{R_1, R_2, \dots, R_n\}$  such that
  - Each relation is in good form
  - The decomposition is a lossless decomposition
- Our theory is based on:
  - Functional dependencies
  - Multivalued dependencies

# Functional Dependencies

- There are usually a variety of constraints (rules) on the data in the real world.
- For example, some of the constraints that are expected to hold in a university database are:
  - Students and instructors are uniquely identified by their ID.
  - Each student and instructor has only one name.
  - Each instructor and student is associated with only one department.
  - Each department has only one value for its budget, and only one associated building.

# Functional Dependencies (Cont.)

- An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation;
  - (A legal instance of a database is one where all the relation instances are legal instances)
- Constraints on the set of legal relations, require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key*.

# Functional Dependencies Definition

- Let  $R$  be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

**holds on  $R$**  if and only if for any legal relations  $r(R)$ , whenever any two tuples  $t_1$  and  $t_2$  of  $r$  agree on the attributes  $\alpha$ , they also agree on the attributes  $\beta$ . That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider  $r(A,B)$  with the following instance of  $r$ .

1	4
1	5
3	7

- On this instance,  $B \rightarrow A$  hold;  $A \rightarrow B$  does **NOT** hold,

# Keys and Functional Dependencies

- $K$  is a superkey for relation schema  $R$  if and only if  $K \rightarrow R$
- $K$  is a candidate key for  $R$  if and only if
  - $K \rightarrow R$ , and
  - for no  $\alpha \subset K$ ,  $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

*in\_dep (ID, name, salary, dept\_name, building, budget ).*

We expect these functional dependencies to hold:

*dept\_name*  $\rightarrow$  *building*

*ID*  $\rightarrow$  *building*

but would not expect the following to hold:

*dept\_name*  $\rightarrow$  *salary*

# Use of Functional Dependencies

- We use functional dependencies to:
  - To test relations to see if they are legal under a given set of functional dependencies.
    - If a relation  $r$  is legal under a set  $F$  of functional dependencies, we say that  $r$  **satisfies**  $F$ .
  - To specify constraints on the set of legal relations
    - We say that  $F$  **holds on**  $R$  if all legal relations on  $R$  satisfy the set of functional dependencies  $F$ .
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
  - For example, a specific instance of *instructor* may, by chance, satisfy  $name \rightarrow ID$ .

# Trivial Functional Dependencies

- A functional dependency is **trivial** if it is bound to be satisfied by all instances of a relation
- Example:
  - $ID, name \rightarrow ID$
  - $name \rightarrow name$
- In general,  $\alpha \rightarrow \beta$  is trivial if  $\beta \subseteq \alpha$

# Closure of a Set of Functional Dependencies

- It is not sufficient to consider only the given set of functional dependencies, the given functional dependencies may imply other functional dependencies
- Given a set  $F$  of functional dependencies, there are certain other functional dependencies that are logically implied by  $F$ .
  - If  $A \rightarrow B$  and  $B \rightarrow C$ , then we can infer that  $A \rightarrow C$
  - etc.
- The set of **all** functional dependencies logically implied by  $F$  is the **closure** of  $F$ .
- We denote the *closure* of  $F$  by  $F^*$ .

# Closure of a Set of Functional Dependencies

- We can compute  $F^+$ , the closure of  $F$ , by repeatedly applying **Armstrong's Axioms**:
  - **Reflexive rule:** if  $\beta \subseteq \alpha$ , then  $\alpha \rightarrow \beta$
  - **Augmentation rule:** if  $\alpha \rightarrow \beta$ , then  $\gamma\alpha \rightarrow \gamma\beta$
  - **Transitivity rule:** if  $\alpha \rightarrow \beta$ , and  $\beta \rightarrow \gamma$ , then  $\alpha \rightarrow \gamma$
- These rules are
  - **Sound** -- generate only functional dependencies that actually hold, and
  - **Complete** -- generate all functional dependencies that hold.

# Closure of Functional Dependencies (Cont.)

- Additional rules:
  - **Union rule:** If  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds, then  $\alpha \rightarrow \beta\gamma$  holds.
  - **Decomposition rule:** If  $\alpha \rightarrow \beta\gamma$  holds, then  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds.
  - **Pseudotransitivity rule:** If  $\alpha \rightarrow \beta$  holds and  $\beta \rightarrow \delta$  holds, then  $\alpha\beta \rightarrow \delta$  holds.
- The above rules can be inferred from Armstrong's axioms.

# Example of $F^+$

- $R = (A, B, C, G, H, I)$   
 $F = \{ A \rightarrow B$   
     $A \rightarrow C$   
     $CG \rightarrow H$   
     $CG \rightarrow I$   
     $B \rightarrow H\}$
- Some members of  $F^+$ 
  - $A \rightarrow H$ 
    - by transitivity from  $A \rightarrow B$  and  $B \rightarrow H$
  - $AG \rightarrow I$ 
    - by augmenting  $A \rightarrow C$  with G, to get  $AG \rightarrow CG$  and then transitivity with  $CG \rightarrow I$
  - $CG \rightarrow HI$ 
    - by augmenting  $CG \rightarrow I$  with G to infer  $CG \rightarrow CGI$ , and augmenting of  $CG \rightarrow H$  with I to infer  $CGI \rightarrow HI$ , and then transitivity

# Procedure for Computing $F^+$

- To compute the closure of a set of functional dependencies  $F$ :

$F^+ = F$

**repeat**

**for each** functional dependency  $f$  in  $F^+$

    apply reflexivity and augmentation rules on  $f$

    add the resulting functional dependencies to  $F^+$

**for each** pair of functional dependencies  $f_1$  and  $f_2$  in  $F^+$

**if**  $f_1$  and  $f_2$  can be combined using transitivity

**then** add the resulting functional dependency to  $F^+$

**until**  $F^+$  does not change any further

# Closure of Attribute Sets

- Given a set of attributes  $\alpha$ , define the ***closure*** of  $\alpha$  **under**  $F$  (denoted by  $\alpha^+$ ) as the set of attributes that are functionally determined by  $\alpha$  under  $F$
- Algorithm to compute  $\alpha^+$ , the closure of  $\alpha$  under  $F$

```
result :=  $\alpha$ ;  
while (changes to result) do  
  for each  $\beta \rightarrow \gamma$  in  $F$  do  
    begin  
      if  $\beta \subseteq result$  then result := result  $\cup$   $\gamma$   
    end
```

# Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- $(AG)^+$ 
  1.  $result = AG$
  2.  $result = ABCG$       ( $A \rightarrow C$  and  $A \rightarrow B$ )
  3.  $result = ABCGH$       ( $CG \rightarrow H$  and  $CG \subseteq AGBC$ )
  4.  $result = ABCGHI$       ( $CG \rightarrow I$  and  $CG \subseteq AGBCH$ )
- Is  $AG$  a candidate key?
  1. Is  $AG$  a super key?
    1. Does  $AG \rightarrow R$ ? == Is  $R \subseteq (AG)^+$
    2. Is any subset of  $AG$  a superkey?
      1. Does  $A \rightarrow R$ ? == Is  $R \subseteq (A)^+$
      2. Does  $G \rightarrow R$ ? == Is  $R \subseteq (G)^+$
      3. In general: check for each subset of size upto  $n-1$

# Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if  $\alpha$  is a superkey, we compute  $\alpha^+$ , and check if  $\alpha^+$  contains all attributes of  $R$ .
- Testing functional dependencies
  - To check if a functional dependency  $\alpha \rightarrow \beta$  holds (or, in other words, is in  $F^+$ ), just check if  $\beta \subseteq \alpha^+$ .
  - That is, we compute  $\alpha^+$  by using attribute closure, and then check if it contains  $\beta$ .
  - Is a simple and cheap test, and very useful
- Computing closure of  $F$ 
  - For each  $\gamma \subseteq R$ , we find the closure  $\gamma^+$ , and for each  $S \subseteq \gamma^+$ , we output a functional dependency  $\gamma \rightarrow S$ .

# Canonical Cover

- Suppose that we have a set of functional dependencies  $F$  on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies; that is, all the functional dependencies in  $F$  are satisfied in the new database state.
- If an update violates any functional dependencies in the set  $F$ , the system must roll back the update.
- We can reduce the effort spent in checking for violations by testing a **simplified set of functional dependencies that has the same closure as the given set**.
- This simplified set is termed the **canonical cover**
- To define canonical cover we must first define **extraneous attributes**
  - An attribute of a functional dependency in  $F$  is **extraneous** if we can remove it without changing  $F^+$

# Extraneous Attributes

- Removing an attribute from the **left side** of a functional dependency could make it a **stronger constraint**.
  - For example, if we have  $AB \rightarrow C$  and remove B, we get the possibly stronger result  $A \rightarrow C$ . It may be stronger because  $A \rightarrow C$  logically implies  $AB \rightarrow C$ , but  $AB \rightarrow C$  does not, on its own, logically imply  $A \rightarrow C$
- But, depending on what our set F of functional dependencies happens to be, we may be able to remove B from  $AB \rightarrow C$  safely.
  - For example, suppose that
  - $F = \{AB \rightarrow C, A \rightarrow D, D \rightarrow C\}$
  - Then we can show that F logically implies  $A \rightarrow C$ , making **B** extraneous in  $AB \rightarrow C$ .

# Extraneous Attributes (Cont.)

- Removing an attribute from the **right side** of a functional dependency could make it a **weaker constraint**.
  - For example, if we have  $AB \rightarrow CD$  and remove C, we get the possibly weaker result  $AB \rightarrow D$ . It may be weaker because using just  $AB \rightarrow D$ , we can no longer infer  $AB \rightarrow C$ .
- But, depending on what our set F of functional dependencies happens to be, we may be able to remove C from  $AB \rightarrow CD$  safely.
  - For example, suppose that
$$F = \{ AB \rightarrow CD, A \rightarrow C \}$$
  - Then we can show that even after replacing  $AB \rightarrow CD$  by  $AB \rightarrow D$ , we can still infer  $AB \rightarrow CD$ .

# Extraneous Attributes

- An attribute of a functional dependency in  $F$  is **extraneous** if we can remove it without changing  $F^+$
- Consider a set  $F$  of functional dependencies and the functional dependency  $\alpha \rightarrow \beta$  in  $F$ .
  - **Remove from the left side:** Attribute  $A$  is **extraneous** in  $\alpha$  if
    - $A \in \alpha$  and
    - $F$  logically implies  $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ .
  - **Remove from the right side:** Attribute  $A$  is **extraneous** in  $\beta$  if
    - $A \in \beta$  and
    - The set of functional dependencies
$$(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$
 logically implies  $F$ .
- Note: implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one

# Testing if an Attribute is Extraneous

- Let  $R$  be a relation schema and let  $F$  be a set of functional dependencies that hold on  $R$ . Consider an attribute in the functional dependency  $\alpha \rightarrow \beta$ .
- To test if attribute  $A \in \alpha$  is extraneous in  $\alpha$ 
  - Let  $\gamma = \alpha - \{A\}$ . Check if  $\gamma \rightarrow \beta$  can be inferred from  $F$ .
    - Compute  $\gamma^+$  using the dependencies in  $F$
    - If  $\gamma^+$  includes all attributes in  $\beta$  then,  $A$  is extraneous in  $\alpha$
- To test if attribute  $A \in \beta$  is extraneous in  $\beta$ 
  - Consider the set:  
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\},$$
  - check whether  $\alpha^+$  in  $F'$  contains  $A$ ; if it does,  $A$  is extraneous in  $\beta$

# Examples of Extraneous Attributes

- Let  $F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$
- To check if  $C$  is extraneous in  $AB \rightarrow CD$ , we:
  - Compute the attribute closure of  $AB$  under  $F = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$
  - The closure is  $ABCDE$ , which includes  $CD$
  - This implies that  $C$  is extraneous
- *Is D extraneous in  $AB \rightarrow CD$ ? Check.*

# Canonical Cover

A **canonical cover** for  $F$  is a set of dependencies  $F_c$  such that

- $F$  logically implies all dependencies in  $F_c$ , and
- $F_c$  logically implies all dependencies in  $F$ , and
- No functional dependency in  $F_c$  contains an extraneous attribute, and
- Each left side of functional dependency in  $F_c$  is unique. That is, there are no two dependencies in  $F_c$ 
  - $\alpha_1 \rightarrow \beta_1$  and  $\alpha_2 \rightarrow \beta_2$  such that
  - $\alpha_1 = \alpha_2$

# Canonical Cover

- $F_c = F$

**repeat**

    Use the union rule to replace any dependencies in  $F$  of the form

$$\alpha_1 \rightarrow \beta_1 \text{ and } \alpha_1 \rightarrow \beta_2 \text{ with } \alpha_1 \rightarrow \beta_1 \beta_2$$

    Find a functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  with an extraneous attribute either in  $\alpha$  or in  $\beta$

    /\* Note: test for extraneous attributes done using  $F_c$ , not  $F^*$  \*/

    If an extraneous attribute is found, delete it from  $\alpha \rightarrow \beta$

**until** ( $F_c$  does not change)

- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

# Example: Computing a Canonical Cover

- $R = (A, B, C)$   
 $F = \{A \rightarrow BC$   
     $B \rightarrow C$   
     $A \rightarrow B$   
     $AB \rightarrow C\}$
- Combine  $A \rightarrow BC$  and  $A \rightarrow B$  into  $A \rightarrow BC$ 
  - Set is now  $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- $A$  is extraneous in  $AB \rightarrow C$ 
  - Check if the result of deleting  $A$  from  $AB \rightarrow C$  is implied by the other dependencies
    - Yes: in fact,  $B \rightarrow C$  is already present!
  - Set is now  $\{A \rightarrow BC, B \rightarrow C\}$
- $C$  is extraneous in  $A \rightarrow BC$ 
  - Check if  $A \rightarrow C$  is logically implied by  $A \rightarrow B$  and the other dependencies
    - Yes: using transitivity on  $A \rightarrow B$  and  $B \rightarrow C$ .
      - Can use attribute closure of  $A$  in more complex cases
- The canonical cover is:
  - $A \rightarrow B$
  - $B \rightarrow C$

# **Chapter 7: Normalization**

# Lossless Decomposition

- We can use functional dependencies to show when certain decompositions are lossless.
- For the case of  $R = (R_1, R_2)$ , we require that for all possible relations  $r$  on schema  $R$

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- A decomposition of  $R$  into  $R_1$  and  $R_2$  is lossless decomposition if at least one of the following dependencies is in  $F^+$ :
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$ 
    - In other words,  $R_1 \cap R_2$  forms a **superkey** of either  $R_1$  or  $R_2$
  - The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies (there may be multivalued dependencies too – *later*)

# Example

- $R = (A, B, C)$   
 $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B), \quad R_2 = (B, C)$

- Lossless decomposition:

$$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$

- $R_1 = (A, B), \quad R_2 = (A, C)$ 
  - Lossless decomposition:

$$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

- Note:
  - $B \rightarrow BC$   
is a shorthand notation for
  - $B \rightarrow \{B, C\}$

# Dependency Preservation

- Testing functional dependency constraints each time the database is updated can be costly
- It is useful to design the database in a way that constraints can be tested efficiently
- **When decomposing a relation it is possible that it is no longer possible to do the testing without having to perform a Cartesian Product – See next slide**
- A decomposition that makes it infeasible to enforce one or more functional dependencies in original F, is said to be NOT **dependency preserving**.

# Dependency Preservation Counter-Example

- Consider a schema:  
 $\text{dept\_advisor}(s\_ID, i\_ID, \text{dept\_name})$
- With function dependencies:  
 $i\_ID \rightarrow \text{dept\_name}$   
 $s\_ID, \text{dept\_name} \rightarrow i\_ID$
- In the above design we are forced to repeat the department name once for each time an instructor participates in a *dept\_advisor* relationship.
- To fix this, we need to decompose *dept\_advisor*
- Any decomposition will not include all the attributes in  
 $s\_ID, \text{dept\_name} \rightarrow i\_ID$
- Thus, the decomposition can NOT be dependency preserving

# Dependency Preservation

- Let  $F_i$  be the set of dependencies in  $F^+$  that includes only attributes in  $R_i$ .
  - A decomposition is **dependency preserving**, if
$$(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$$
- **(Note that the definition of restriction uses all dependencies in  $F^+$ , not just those in  $F$ )**
- Using the above definition, testing for dependency preservation take **exponential time**.
- Note that if a decomposition is NOT dependency preserving **then checking updates for violation of functional dependencies may require computing joins, which is expensive.**

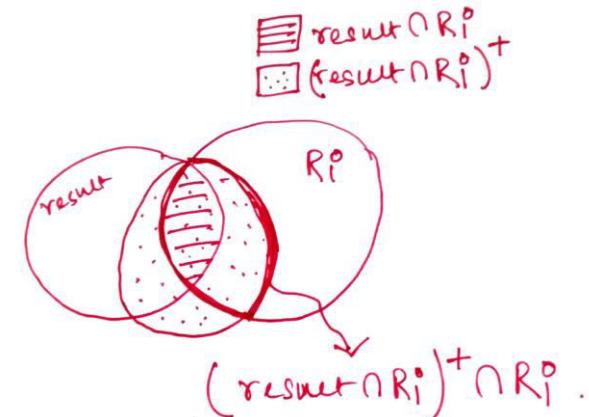
# Testing for Dependency Preservation

- To check if a dependency  $\alpha \rightarrow \beta$  is preserved in a decomposition of  $R$  into  $R_1, R_2, \dots, R_n$ , we apply the following test (with attribute closure done with respect to  $F$ )

- $result = \alpha$   
**repeat**  
    **for each**  $R_i$  in the decomposition  
         $t = (result \cap R_i)^+ \cap R_i$   
         $result = result \cup t$   
**end for**

- If  $result$  contains all attributes in  $\beta$ , then the functional dependency  $\alpha \rightarrow \beta$  is preserved.

- We apply the test on all dependencies in  $F$  to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute  $F^+$  and  $(F_1 \cup F_2 \cup \dots \cup F_n)^+$



# Example

- $R = (A, B, C)$   
 $F = \{A \rightarrow B$   
     $B \rightarrow C\}$   
Key = {A}
- $R$  is not in BCNF
- Decomposition  $R_1 = (A, B)$ ,  $R_2 = (B, C)$ 
  - $R_1$  and  $R_2$  in BCNF
  - Lossless-join decomposition
  - Dependency preserving

# An Example of Dependency Preserving, but Lossy Decomposition

- R (A,B,C) → decomposed into R1 (A,B) and R2 (B,C)

The diagram illustrates the decomposition of a relation R into two smaller relations, R1 and R2. On the left, there is a large blue-bordered table labeled R with columns A, B, and C, and rows numbered 1, 2, and 3. The values are: Row 1: A=1, B=x, C=p; Row 2: A=2, B=x, C=q; Row 3: A=3, B=y, C=r. An arrow points from this table to two smaller tables on the right. The first table, labeled R1, has columns A and B, and rows 1, 2, and 3. Its values are: Row 1: A=1, B=x; Row 2: A=2, B=x; Row 3: A=3, B=y. The second table, labeled R2, has columns B and C, and rows x, y, and z. Its values are: Row 1: B=x, C=p; Row 2: B=x, C=q; Row 3: B=y, C=r.

A	B	C
1	x	p
2	x	q
3	y	r

→

A	B
1	x
2	x
3	y

B	C
x	p
x	q
y	r

- After applying natural join on R1 and R2 above, we get

The diagram shows the result of performing a natural join on the decomposed relations R1 and R2. The resulting table has columns A, B, and C, and rows 1, 2, and 3. The values are: Row 1: A=1, B=x, C=p; Row 2: A=2, B=x, C=q; Row 3: A=3, B=y, C=r. The second and third rows (2 and 3) are highlighted with a red rectangle, indicating they are lost during the join process.

A	B	C
1	x	p
1	x	q
2	x	p
2	x	q
3	y	r

# **Chapter 7: Normal Forms**

# Normalization Theory

- Decide whether a particular relation  $R$  is in “good” form.
- In the case that a relation  $R$  is “not in good” form, decompose it into set of relations  $\{R_1, R_2, \dots, R_n\}$  such that
  - Each relation is in good form
  - The decomposition is a lossless decomposition
- Our theory is based on:
  - Functional dependencies
  - Multivalued dependencies

# **Normal Forms**

# Goals of Normalization

- Let  $R$  be a relational schema with a set  $F$  of functional dependencies.
- Decide whether a relation scheme  $R$  is in “good” form.
- In the case that a relation scheme  $R$  is not in “good” form, need to decompose it into a set of relation schema  $\{R_1, R_2, \dots, R_n\}$  such that:
  - Each relational schema is in good form
  - The decomposition is a lossless decomposition
  - Preferably, the decomposition should be dependency preserving.

# First Normal Form

- A relational schema  $R$  is in **First Normal Form** if the domains of all attributes of  $R$  are **atomic**
- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Attribute *Name* with first, middle and surname parts
    - Composite attributes like *Location* (for a particular project)
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account

# First Normal Form

- Atomicity is actually a property of how the elements of the domain are used.
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.
- We assume all such relations are in first normal form (1NF)

# First Normal Form

- **1NF Decomposition Rule:** Remove those attributes from the relational schema R which violate the 1NF criteria, and place them in a separate relational schema along with their determining attributes
- Example:

Dept ( dno, dmgr, dname, dloc )

dloc is NOT atomic since it can have multiple values

Dept decomposed into Dept\_details and Dept\_loc

Dept\_details ( dno, dmgr, dname )

Dept\_loc ( dno, dloc )

# Second Normal Form

- **Partial Dependency**

A dependency  $\alpha \rightarrow \beta$  is partial if  $(\alpha - A) \rightarrow \beta$  holds where  $A \subseteq \alpha$

- A relational schema is in 2NF if

- It is in 1NF
- Every *non-prime attribute* (attributes which are not a part of any candidate key) is fully functionally dependent on candidate keys

# Second Normal Form

- Example:

Project ( eno, pno, whour, ename, pname )

$F = \{ ( \text{eno}, \text{pno} ) \rightarrow \text{whour}, \text{eno} \rightarrow \text{ename}, \text{pno} \rightarrow \text{pname} \}$

From the above F, we can see ( **eno, pno** ) is a candidate key and **whour, ename, pname** are non-prime attributes

But, ename is dependent on eno and pname is dependent on pno

So,  $( \text{eno}, \text{pno} ) \rightarrow \text{ename}$  and  $( \text{eno}, \text{pno} ) \rightarrow \text{pname}$  are partial dependencies

Therefore, the relation Project is NOT in 2NF

- Partial dependencies are considered bad

# Second Normal Form

- **2NF Decomposition Rule:** Remove those attributes from the relational schema R which violate the 2NF criteria, and place them in a separate relational schema along with their determining attributes
- Example:

Project ( eno, pno, whour, ename, pname )

$F = \{ ( eno, pno ) \rightarrow whour, eno \rightarrow ename, pno \rightarrow pname \}$

decomposed into:-

R1 ( eno, ename ) with  $F1 = \{ eno \rightarrow ename \}$

R2 ( pno, pname ) with  $F2 = \{ pno \rightarrow pname \}$

R3 ( eno, pno, whour ) with  $F3 = \{ ( eno, pno ) \rightarrow whour \}$

# Boyce-Codd Normal Form

- A relation schema  $R$  is in BCNF with respect to a set  $F$  of functional dependencies if for all functional dependencies in  $F^+$  of the form

$$\alpha \rightarrow \beta$$

where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following holds:

- $\alpha \rightarrow \beta$  is trivial (i.e.,  $\beta \subseteq \alpha$ )
- $\alpha$  is a superkey for  $R$

# Boyce-Codd Normal Form (Cont.)

- Example schema that is *not* in BCNF:

*in\_dep (ID, name, salary, dept\_name, building, budget )*

because :

- $\text{dept\_name} \rightarrow \text{building}, \text{budget}$ 
  - holds on *in\_dep*
  - but
- *dept\_name* is not a superkey

- When we decompose *in\_dept* into *instructor* and *department*
  - *instructor* is in BCNF
  - *department* is in BCNF

# Boyce-Codd Normal Form

- Example:

Teach ( course, teacher, sroll )

( sroll, course ) → teacher

teacher → course

Therefore, (sroll, course) is the candidate key

teacher → course is not trivial, neither “teacher” is a superkey

Therefore, above relation Teach is **not** in BCNF

# BCNF Decomposition Rule

- **BCNF Decomposition Rule:** Remove those attributes from the relational schema  $R$  which violate the BCNF criteria, and place them in a separate relational schema along with their determining attributes
- Let  $R$  be a schema  $R$  that is not in BCNF. Let  $\alpha \rightarrow \beta$  be the FD that causes a violation of BCNF.
- We decompose  $R$  into:
  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$
- In our example of  $in\_dep$ ,
  - $\alpha = dept\_name$
  - $\beta = building, budget$and  $in\_dep$  is replaced by
  - $(\alpha \cup \beta) = (dept\_name, building, budget)$
  - $(R - (\beta - \alpha)) = (ID, name, dept\_name, salary)$
- Teach ( course, teacher, sroll ) decomposed into:
  - $R1$  ( course, teacher ) with  $F1 = \{ teacher \rightarrow course \}$
  - $R2$  ( sroll, teacher )
    - If  $R2$  ( sroll, course ) → what is the problem?

# BCNF Decomposition Algorithm (Same as the rule in previous slide)

```
result := {R};  
done := false;  
compute  $F^+$ ;  
while (not done) do  
  if (there is a schema  $R_i$  in result that is not in BCNF)  
    then begin  
      let  $\alpha \rightarrow \beta$  be a nontrivial functional dependency that  
      holds on  $R_i$  such that  $\alpha$  is not a superkey of  $R_i$ ;  
      result := (result -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );  
    end  
  else done := true;
```

Note: each  $R_i$  is in BCNF, and decomposition is lossless-join.

# Example of BCNF Decomposition

- *class (course\_id, title, dept\_name, credits, sec\_id, semester, year, building, room\_number, capacity, time\_slot\_id)*
- Functional dependencies:
  - $course\_id \rightarrow title, dept\_name, credits$
  - $building, room\_number \rightarrow capacity$
  - $course\_id, sec\_id, semester, year \rightarrow building, room\_number, time\_slot\_id$
- A candidate key  $\{course\_id, sec\_id, semester, year\}$ .
- BCNF Decomposition:
  - $course\_id \rightarrow title, dept\_name, credits$  holds
    - but  $course\_id$  is not a superkey.
  - We replace *class* by:
    - *course(course\_id, title, dept\_name, credits)*
    - *class-1 (course\_id, sec\_id, semester, year, building, room\_number, capacity, time\_slot\_id)*

# BCNF Decomposition (Cont.)

- *course* is in BCNF
  - How do we know this?
- *building, room\_number*→*capacity* holds on *class-1*
  - but {*building, room\_number*} is not a superkey for *class-1*.
  - We replace *class-1* by:
    - *classroom* (*building, room\_number, capacity*)
    - *section* (*course\_id, sec\_id, semester, year, building, room\_number, time\_slot\_id*)
- *classroom* and *section* are in BCNF.

# Miscellaneous

- **Claim:** Every relational schema with exactly TWO attributes is always in BCNF
- Say  $R(A,B)$
- All possible functional dependencies:
  - $A \rightarrow B \rightarrow A$  is superkey
  - $B \rightarrow A \rightarrow B$  is superkey
  - $AB \rightarrow A \rightarrow$  trivial
  - $AB \rightarrow B \rightarrow$  trivial

# Example

- $R = (A, B, C)$   
 $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B), R_2 = (B, C)$ 
  - Lossless-join decomposition:

$$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$

- Dependency preserving
- $R_1 = (A, B), R_2 = (A, C)$ 
  - Lossless-join decomposition:

$$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

- Not dependency preserving  
(cannot check  $B \rightarrow C$  without computing  $R_1 \bowtie R_2$ )

# Testing for BCNF

- To check if a non-trivial dependency  $\alpha \rightarrow \beta$  causes a violation of BCNF
  1. compute  $\alpha^+$  (the attribute closure of  $\alpha$ ), and
  2. verify that it includes all attributes of  $R$ , that is, it is a superkey of  $R$ .
- **Simplified test:** To check if a relation schema  $R$  is in BCNF, it suffices to check only the dependencies in the given set  $F$  for violation of BCNF, rather than checking all dependencies in  $F^+$ .
  - If none of the dependencies in  $F$  causes a violation of BCNF, then none of the dependencies in  $F^+$  will cause a violation of BCNF either.
- However, **simplified test** using **only**  $F$  is incorrect when testing a relation in a decomposition of  $R$ 
  - Consider  $R = (A, B, C, D, E)$ , with  $F = \{ A \rightarrow B, BC \rightarrow D \}$ 
    - Decompose  $R$  into  $R_1 = (A, B)$  and  $R_2 = (A, C, D, E)$
    - Neither of the dependencies in  $F$  contain only attributes from  $(A, C, D, E)$  so we might be mislead into thinking  $R_2$  satisfies BCNF.
    - In fact, dependency  $AC \rightarrow D$  in  $F^+$  shows  $R_2$  is not in BCNF.

# Testing Decomposition for BCNF

To check if a relation  $R_i$  in a decomposition of  $R$  is in BCNF

- Either test  $R_i$  for BCNF with respect to the **restriction** of  $F^+$  to  $R_i$  (that is, all FDs in  $F^+$  that contain only attributes from  $R_i$ )
- Or use the original set of dependencies  $F$  that hold on  $R$ , but with the following test:
  - for every set of attributes  $\alpha \subseteq R_i$ , check that  $\alpha^+$  (the attribute closure of  $\alpha$ ) either includes no attribute of  $R_i - \alpha$ , or includes all attributes of  $R_i$ .
  - If the condition is violated by some set of attributes  $\alpha$  in  $R_i$ , the following functional dependency can be shown to hold on  $R_i$ :
$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$
This above dependency shows that  $R_i$  violates BCNF
  - We use above dependency to decompose  $R_i$

# BCNF and Dependency Preservation

- It is not always possible to achieve both BCNF and dependency preservation
- Consider a schema:  
 $\text{dept\_advisor}(s\_ID, i\_ID, \text{dept\_name})$
- With function dependencies:  
 $i\_ID \rightarrow \text{dept\_name}$   
 $s\_ID, \text{dept\_name} \rightarrow i\_ID$
- $\text{dept\_advisor}$  is not in BCNF
  - $i\_ID$  is not a superkey.
- Any decomposition of  $\text{dept\_advisor}$  will not include all the attributes in  
 $s\_ID, \text{dept\_name} \rightarrow i\_ID$
- Thus, the decomposition can NOT be dependency preserving

# Third Normal Form

- A relation schema  $R$  is in **third normal form (3NF)** if for all:

$$\alpha \rightarrow \beta \text{ in } F^+$$

at least one of the following holds:

- $\alpha \rightarrow \beta$  is trivial (i.e.,  $\beta \in \alpha$ )
- $\alpha$  is a superkey for  $R$
- Each attribute  $A$  in  $\beta - \alpha$  is contained in a candidate key for  $R$ .

(**NOTE**: each attribute may be in a different candidate key)

- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

# 3NF Example

- Consider a schema:  
 $dept\_advisor(s\_ID, i\_ID, dept\_name)$
- With function dependencies:  
 $i\_ID \rightarrow dept\_name$   
 $s\_ID, dept\_name \rightarrow i\_ID$
- Two candidate keys =  $\{s\_ID, dept\_name\}$ ,  $\{s\_ID, i\_ID\}$
- We have seen before that  $dept\_advisor$  is **not** in BCNF
- $R$ , however, is in 3NF
  - $s\_ID, dept\_name$  is a superkey
  - $i\_ID \rightarrow dept\_name$  and  $i\_ID$  is NOT a superkey, but:
    - $\{ dept\_name \} - \{ i\_ID \} = \{ dept\_name \}$  is contained in a candidate key

# Testing for 3NF

- Need to check only FDs in  $F$ , need not check all FDs in  $F^+$ .
- Use attribute closure to check for each dependency  $\alpha \rightarrow \beta$ , if it is trivial,
- Else if  $\alpha$  is a superkey.
- Else we have to verify if each attribute in  $\beta - \alpha$  is contained in a candidate key of  $R$ 
  - This test is rather more expensive, since it involve finding candidate keys
  - **Testing for 3NF has been shown to be NP-hard**
  - **Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time**

# 3NF Decomposition Algorithm

**3NF Decomposition Rule:** Remove those attributes from the relational schema R which violate the 3NF criteria, and place them in a separate relational schema along with their determining attributes

Let  $F_c$  be a canonical cover for  $F$ ;

$i := 0$ ;

**for each** functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  **do**

**if** none of the schemas  $R_j$ ,  $1 \leq j \leq i$  contains  $\alpha \beta$

**then begin**

$i := i + 1$ ;

$R_i := \alpha \beta$

**end**

**if** none of the schemas  $R_j$ ,  $1 \leq j \leq i$  contains a candidate key for  $R$

**then begin**

$i := i + 1$ ;

$R_i :=$  any candidate key for  $R$ ;

**end**

**/\* Optionally, remove redundant relations \*/**

**repeat**

**if** any schema  $R_j$  is contained in another schema  $R_k$

**then /\* delete  $R_j$  \*/**

$R_j = R_i$ ;

$i = i - 1$ ;

**return**  $(R_1, R_2, \dots, R_i)$

# 3NF Decomposition Algorithm (Cont.)

Above algorithm ensures

- Each relation schema  $R_i$  is in 3NF
- Decomposition is dependency preserving and lossless-join
- Proof of correctness is at end of this presentation

# 3NF Decomposition: An Example

- Relation schema:  
 $cust\_banker\_branch = (customer\_id, employee\_id, branch\_name, type)$
- The functional dependencies for this relation schema are:
  - $customer\_id, employee\_id \rightarrow branch\_name, type$
  - $employee\_id \rightarrow branch\_name$
  - $customer\_id, branch\_name \rightarrow employee\_id$
- We first compute a canonical cover
  - $branch\_name$  is extraneous in the RHS of the 1<sup>st</sup> dependency
  - No other attribute is extraneous, so we get  $F_C =$   
 $customer\_id, employee\_id \rightarrow type$   
 $employee\_id \rightarrow branch\_name$   
 $customer\_id, branch\_name \rightarrow employee\_id$

# 3NF Decomposition Example (Cont.)

- The **for** loop generates following 3NF schema:
  - (*customer\_id, employee\_id, type*)
  - (*employee\_id, branch\_name*)
  - (*customer\_id, branch\_name, employee\_id*)
- Observe that (*customer\_id, employee\_id, type*) contains a candidate key of the original schema, so no further relation schema needs be added
- At end of for loop, detect and delete schemas, such as (*employee\_id, branch\_name*), which are subsets of other schemas
  - result will not depend on the order in which FDs are considered
- The resultant simplified 3NF schema is:
  - (*customer\_id, employee\_id, type*)
  - (*customer\_id, branch\_name, employee\_id*)

# Redundancy in 3NF

- Consider the schema R below, which is in 3NF

- $R = (J, K, L)$
- $F = \{JK \rightarrow L, L \rightarrow K\}$
- And an instance table:

$J$	$L$	$K$
$j_1$	$l_1$	$k_1$
$j_2$	$l_1$	$k_1$
$j_3$	$l_1$	$k_1$
<i>null</i>	$l_2$	$k_2$

- What is wrong with the table?
  - Repetition of information
  - Need to use null values (e.g., to represent the relationship  $l_2, k_2$  where there is no corresponding value for  $J$ )

# Comparison of BCNF and 3NF

- There are some situations where
  - BCNF is not dependency preserving, and
  - efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form (3NF)
  - Allows some redundancy
  - But functional dependencies can be checked on individual relations without computing a join.
  - There is always a lossless-join, dependency-preserving decomposition into 3NF.
- Advantages to 3NF over BCNF. It is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation.
- Disadvantages to 3NF.
  - We may have to use null values to represent some of the possible meaningful relationships among data items.
  - There is the problem of repetition of information.

# **Summary: Comparison of BCNF and 3NF**

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - The decomposition is lossless
  - The dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - The decomposition is lossless
  - It may not be possible to preserve dependencies.

# Design Goals

- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - 3NF with associated Redundancy

# How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation

*inst\_info (ID, child\_name, phone)*

- where an instructor may have more than one phone and can have multiple children
- Instance of *inst\_info*

<i>ID</i>	<i>child_name</i>	<i>phone</i>
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	William	512-555-4321

## How good is BCNF? (Cont.)

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

(99999, David, 981-992-3443)  
(99999, William, 981-992-3443)

# Higher Normal Forms

- It is better to decompose *inst\_info* into:
  - *inst\_child*:

<i>ID</i>	<i>child_name</i>
99999	David
99999	William

- *inst\_phone*:

<i>ID</i>	<i>phone</i>
99999	512-555-1234
99999	512-555-4321

- This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later → deals with MULTIVALUED DEPENDENCY

# **Multivalued Dependencies**

# Multivalued Dependencies (MVDs)

- Suppose we record names of children, and phone numbers for instructors:
  - $inst\_child(ID, child\_name)$
  - $inst\_phone(ID, phone\_number)$
- If we were to combine these schemas to get
  - $inst\_info(ID, child\_name, phone\_number)$
  - Example data:
    - (99999, David, 512-555-1234)
    - (99999, David, 512-555-4321)
    - (99999, William, 512-555-1234)
    - (99999, William, 512-555-4321)

# Multivalued Dependencies

- Let  $R$  be a relation schema and let  $\alpha \subseteq R$  and  $\beta \subseteq R$ . The **multivalued dependency**

$$\alpha \rightarrow\!\!\!\rightarrow \beta$$

holds on  $R$  if in any legal relation  $r(R)$ , for all pairs for tuples  $t_1$  and  $t_2$  in  $r$  such that  $t_1[\alpha] = t_2[\alpha]$ , there exist tuples  $t_3$  and  $t_4$  in  $r$  such that:

$$\begin{aligned}t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\t_3[\beta] &= t_1[\beta] \\t_3[R - \beta] &= t_2[R - \beta] \\t_4[\beta] &= t_2[\beta] \\t_4[R - \beta] &= t_1[R - \beta]\end{aligned}$$

- Tabular representation of  $\alpha \rightarrow\!\!\!\rightarrow \beta$

	$\alpha$	$\beta$	$R - \alpha - \beta$
$t_1$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
$t_2$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
$t_3$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
$t_4$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Note: If  $Y \rightarrow\!\!\!\rightarrow Z$  then  
 $Y \rightarrow\!\!\!\rightarrow [R - Y - Z]$

# Example

- In our example:

$ID \rightarrow\!\!\! \rightarrow child\_name$

$ID \rightarrow\!\!\! \rightarrow phone\_number$

- The above formal definition is supposed to formalize the notion that given a particular value of  $Y$  ( $ID$ ) it has associated with it a set of values of  $Z$  ( $child\_name$ ) and a set of values of  $W$  ( $phone\_number$ ), and these two sets are in some sense independent of each other.

- From the definition of multivalued dependency, we can derive the following rule:

- **If  $\alpha \rightarrow \beta$ , then  $\alpha \rightarrow\!\!\! \rightarrow \beta$**

That is, every functional dependency is also a multivalued dependency

# Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:
  1. To test relations to **determine** whether they are legal under a given set of functional and multivalued dependencies
  2. To specify **constraints** on the set of legal relations. We shall concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.
- If a relation  $r$  fails to satisfy a given multivalued dependency, we can construct a relations  $r'$  that does satisfy the multivalued dependency by adding tuples to  $r$ .

# Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:
  - If  $\alpha \rightarrow \beta$ , then  $\alpha \twoheadrightarrow \beta$
- That is, every functional dependency is also a multivalued dependency
- The **closure**  $D^+$  of  $D$  is the set of all functional and multivalued dependencies logically implied by  $D$ .
  - We can compute  $D^+$  from  $D$ , using the formal definitions of functional dependencies and multivalued dependencies.
  - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice
  - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules (Appendix C).

# Fourth Normal Form

- A relation schema  $R$  is in **4NF** with respect to a set  $D$  of functional and multivalued dependencies if for all multivalued dependencies in  $D^+$  of the form  $\alpha \twoheadrightarrow \beta$ , where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following hold:
  - $\alpha \twoheadrightarrow \beta$  is trivial (i.e.,  $\beta \subseteq \alpha$  or  $\alpha \cup \beta = R$ )
  - $\alpha$  is a superkey for schema  $R$
- If a relation is in 4NF it is in BCNF

# Restriction of Multivalued Dependencies

- The restriction of  $D$  to  $R_i$  is the set  $D_i$  consisting of
  - All functional dependencies in  $D^+$  that include only attributes of  $R_i$
  - All multivalued dependencies of the form
$$\alpha \rightarrow\rightarrow (\beta \cap R_i)$$
where  $\alpha \subseteq R_i$  and  $\alpha \rightarrow\rightarrow \beta$  is in  $D^+$

# 4NF Decomposition Algorithm

*result* := { $R$ };

*done* := false;

*compute*  $D^+$ ;

Let  $D_i$  denote the restriction of  $D^+$  to  $R_i$

**while** (**not** *done*)

**if** (there is a schema  $R_i$  in *result* that is not in 4NF) **then**

**begin**

            let  $\alpha \rightarrow\!\!\!\rightarrow \beta$  be a nontrivial multivalued dependency that holds  
            on  $R_i$  such that  $\alpha \rightarrow R_i$  is not in  $D_i$ , and  $\alpha \cap \beta = \emptyset$ ;

*result* := (*result* -  $R_i$ )  $\cup$  ( $R_i$  -  $\beta$ )  $\cup$  ( $\alpha, \beta$ );

**end**

**else** *done* := true;

Note: each  $R_i$  is in 4NF, and decomposition is lossless-join



# Example

- $R = (A, B, C, G, H, I)$   
 $F = \{ A \rightarrow\!\!\!\rightarrow B$   
     $B \rightarrow\!\!\!\rightarrow HI$   
     $CG \rightarrow\!\!\!\rightarrow H \}$
- $R$  is not in 4NF since  $A \rightarrow\!\!\!\rightarrow B$  and  $A$  is not a superkey for  $R$
- Decomposition
  - a)  $R_1 = (A, B)$   $(R_1$  is in 4NF)
  - b)  $R_2 = (A, C, G, H, I)$   $(R_2$  is not in 4NF, decompose into  $R_3$  and  $R_4$ )
  - c)  $R_3 = (C, G, H)$   $(R_3$  is in 4NF)
  - d)  $R_4 = (A, C, G, I)$   $(R_4$  is not in 4NF, decompose into  $R_5$  and  $R_6$ )
    - $A \rightarrow\!\!\!\rightarrow B$  and  $B \rightarrow\!\!\!\rightarrow HI \Rightarrow A \rightarrow\!\!\!\rightarrow HI$ , (MVD transitivity), and
    - and hence  $A \rightarrow\!\!\!\rightarrow I$  (*MVD restriction to  $R_4$* )
  - e)  $R_5 = (A, I)$   $(R_5$  is in 4NF)
  - f)  $R_6 = (A, C, G)$   $(R_6$  is in 4NF)

**End of Chapter 7**

# **Additional issues**

# Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
  - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used

# Overall Database Design Process

We have assumed schema  $R$  is given

- $R$  could have been generated when converting E-R diagram to a set of tables.
- $R$  could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
- Normalization breaks  $R$  into smaller relations.
- $R$  could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

# ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
  - Example: an *employee* entity with
    - attributes  
*department\_name* and *building*,
    - functional dependency  
 $\textit{department\_name} \rightarrow \textit{building}$
    - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary

# Denormalization for Performance

- Designers may want to use non-normalized schema for tuning performance to support time-critical operations
- For example, displaying *prereqs* along with *course\_id* and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
  - faster lookup
  - extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined a *course*  $\bowtie$  *prereq*
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

# Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:
  - Instead of *earnings* (*company\_id*, *year*, *amount* ), which is in BCNF, use
    - *earnings\_2004*, *earnings\_2005*, *earnings\_2006*, etc., all on the schema (*company\_id*, *earnings*).
      - Above are also in BCNF, but make querying across years difficult and needs new table each year
    - *company\_year* (*company\_id*, *earnings\_2004*, *earnings\_2005*, *earnings\_2006*)
      - Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
      - Is an example of a **crosstab**, where values for one attribute become column names
      - Used in spreadsheets, and in data analysis tools

# Modeling Temporal Data

- **Temporal data** have an association time interval during which the data are *valid*.
- A **snapshot** is the value of the data at a particular point in time
- Several proposals to extend ER model by adding valid time to
  - attributes, e.g., address of an instructor at different points in time
  - entities, e.g., time duration when a student entity exists
  - relationships, e.g., time during which an instructor was associated with a student as an advisor.
- But no accepted standard
- Adding a temporal component results in functional dependencies like
$$ID \rightarrow street, city$$
not holding, because the address varies over time
- A **temporal functional dependency**  $X \rightarrow Y$  holds on schema  $R$  if the functional dependency  $X \rightarrow Y$  holds on all snapshots for all legal instances  $r(R)$ .

# Modeling Temporal Data (Cont.)

- In practice, database designers may add start and end time attributes to relations
  - E.g.,  $course(course\_id, course\_title)$  is replaced by  
 $course(course\_id, course\_title, start, end)$
  - Constraint: no two tuples can have overlapping valid times
    - Hard to enforce efficiently
- Foreign key references may be to current version of data, or to data at a point in time
  - E.g., student transcript should refer to course information at the time the course was taken

# **Proof of Correctness of 3NF Decomposition Algorithm**

# Correctness of 3NF Decomposition Algorithm

- 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in  $F_c$ )
- Decomposition is lossless
  - A candidate key ( $C$ ) is in one of the relations  $R_i$  in decomposition
  - Closure of candidate key under  $F_c$  must contain all attributes in  $R$ .
  - Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in  $R_i$

## Correctness of 3NF Decomposition Algorithm (Cont.)

- Claim: if a relation  $R_i$  is in the decomposition generated by the above algorithm, then  $R_i$  satisfies 3NF.
- Proof:
  - Let  $R_i$  be generated from the dependency  $\alpha \rightarrow \beta$
  - Let  $\gamma \rightarrow B$  be any non-trivial functional dependency on  $R_i$ . (We need only consider FDs whose right-hand side is a single attribute.)
  - Now,  $B$  can be in either  $\beta$  or  $\alpha$  but not in both. Consider each case separately.

# Correctness of 3NF Decomposition (Cont.)

- Case 1: If  $B$  in  $\beta$ :
  - If  $\gamma$  is a superkey, the 2nd condition of 3NF is satisfied
  - Otherwise  $\alpha$  must contain some attribute not in  $\gamma$
  - Since  $\gamma \rightarrow B$  is in  $F^+$  it must be derivable from  $F_c$ , by using attribute closure on  $\gamma$ .
  - Attribute closure not have used  $\alpha \rightarrow \beta$ . If it had been used,  $\alpha$  must be contained in the attribute closure of  $\gamma$ , which is not possible, since we assumed  $\gamma$  is not a superkey.
  - Now, using  $\alpha \rightarrow (\beta - \{B\})$  and  $\gamma \rightarrow B$ , we can derive  $\alpha \rightarrow B$  (since  $\gamma \subseteq \alpha \beta$ , and  $B \notin \gamma$  since  $\gamma \rightarrow B$  is non-trivial)
  - Then,  $B$  is extraneous in the right-hand side of  $\alpha \rightarrow \beta$ ; which is not possible since  $\alpha \rightarrow \beta$  is in  $F_c$ .
  - Thus, if  $B$  is in  $\beta$  then  $\gamma$  must be a superkey, and the second condition of 3NF must be satisfied.

# Correctness of 3NF Decomposition (Cont.)

- Case 2:  $B$  is in  $\alpha$ .
  - Since  $\alpha$  is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.
  - In fact, we cannot show that  $\gamma$  is a superkey.
  - This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.

# **Chapter 7: Normal Forms**

# Normalization Theory

- Decide whether a particular relation  $R$  is in “good” form.
- In the case that a relation  $R$  is “not in good” form, decompose it into set of relations  $\{R_1, R_2, \dots, R_n\}$  such that
  - Each relation is in good form
  - The decomposition is a lossless decomposition
- Our theory is based on:
  - Functional dependencies
  - Multivalued dependencies

# **Normal Forms**

# Goals of Normalization

- Let  $R$  be a relational schema with a set  $F$  of functional dependencies.
- Decide whether a relation scheme  $R$  is in “good” form.
- In the case that a relation scheme  $R$  is not in “good” form, need to decompose it into a set of relation schema  $\{R_1, R_2, \dots, R_n\}$  such that:
  - Each relational schema is in good form
  - The decomposition is a lossless decomposition
  - Preferably, the decomposition should be dependency preserving.

# First Normal Form

- A relational schema  $R$  is in **First Normal Form** if the domains of all attributes of  $R$  are **atomic**
- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Attribute *Name* with first, middle and surname parts
    - Composite attributes like *Location* (for a particular project)
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account

# First Normal Form

- Atomicity is actually a property of how the elements of the domain are used.
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.
- We assume all such relations are in first normal form (1NF)

# First Normal Form

- **1NF Decomposition Rule:** Remove those attributes from the relational schema R which violate the 1NF criteria, and place them in a separate relational schema along with their determining attributes
- Example:

Dept ( dno, dmgr, dname, dloc )

dloc is NOT atomic since it can have multiple values

Dept decomposed into Dept\_details and Dept\_loc

Dept\_details ( dno, dmgr, dname )

Dept\_loc ( dno, dloc )

# Second Normal Form

- **Partial Dependency**

A dependency  $\alpha \rightarrow \beta$  is partial if  $(\alpha - A) \rightarrow \beta$  holds where  $A \subseteq \alpha$

- A relational schema is in 2NF if

- It is in 1NF
- Every *non-prime attribute* (attributes which are not a part of any candidate key) is fully functionally dependent on candidate keys

# Second Normal Form

- Example:

Project ( eno, pno, whour, ename, pname )

$F = \{ ( \text{eno}, \text{pno} ) \rightarrow \text{whour}, \text{eno} \rightarrow \text{ename}, \text{pno} \rightarrow \text{pname} \}$

From the above F, we can see ( **eno, pno** ) is a candidate key and **whour, ename, pname** are non-prime attributes

But, ename is dependent on eno and pname is dependent on pno

So,  $( \text{eno}, \text{pno} ) \rightarrow \text{ename}$  and  $( \text{eno}, \text{pno} ) \rightarrow \text{pname}$  are partial dependencies

Therefore, the relation Project is NOT in 2NF

- Partial dependencies are considered bad

# Second Normal Form

- **2NF Decomposition Rule:** Remove those attributes from the relational schema R which violate the 2NF criteria, and place them in a separate relational schema along with their determining attributes
- Example:

Project ( eno, pno, whour, ename, pname )

$F = \{ ( eno, pno ) \rightarrow whour, eno \rightarrow ename, pno \rightarrow pname \}$

decomposed into:-

R1 ( eno, ename ) with  $F1 = \{ eno \rightarrow ename \}$

R2 ( pno, pname ) with  $F2 = \{ pno \rightarrow pname \}$

R3 ( eno, pno, whour ) with  $F3 = \{ ( eno, pno ) \rightarrow whour \}$



# Chapter 12: Physical Storage Systems

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use

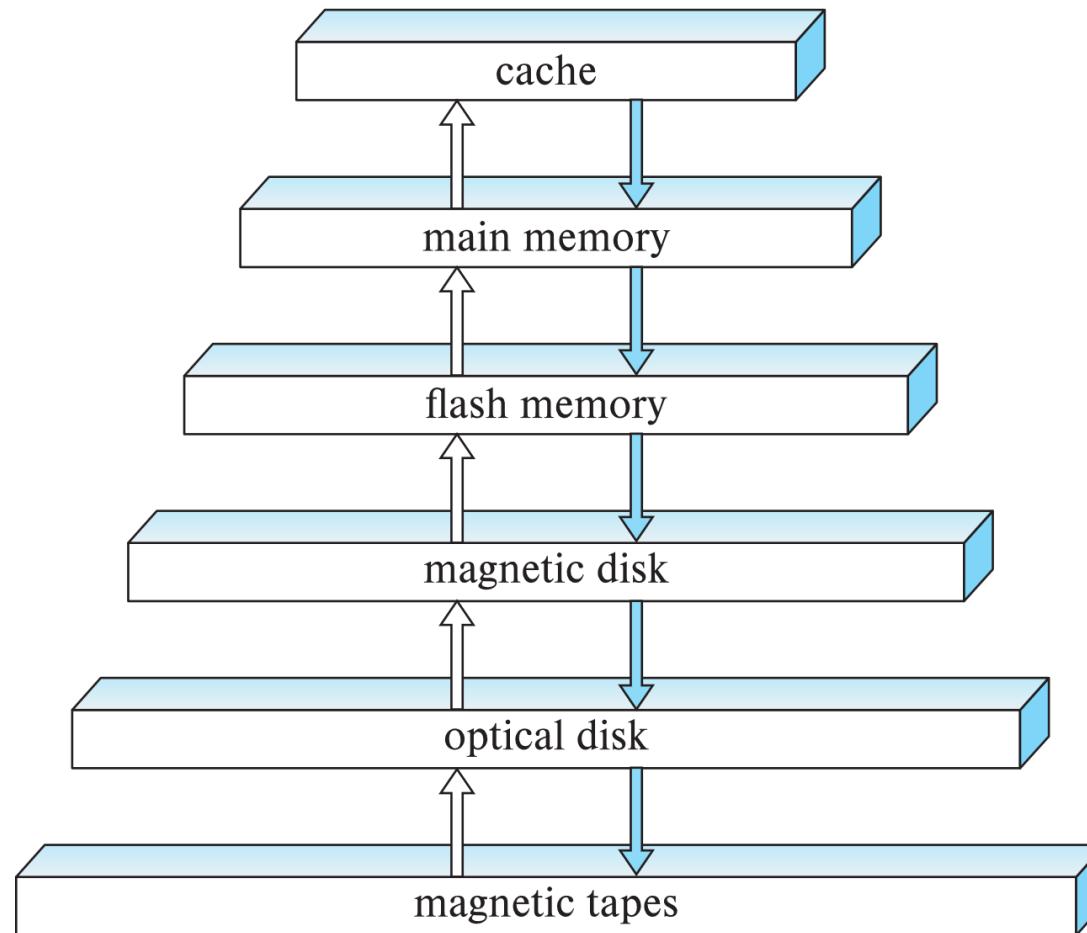


# Classification of Physical Storage Media

- Can differentiate storage into:
  - **volatile storage:** loses contents when power is switched off
  - **non-volatile storage:**
    - Contents persist even when power is switched off.
    - Includes secondary and tertiary storage, as well as battery-backed up main-memory.
- Factors affecting choice of storage media include
  - Speed with which data can be accessed
  - Cost per unit of data
  - Reliability



# Storage Hierarchy





# Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - Also called **on-line storage**
  - E.g., flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage** and used for **archival storage**
  - e.g., magnetic tape, optical storage
  - Magnetic tape
    - Sequential access, 1 to 12 TB capacity
    - A few drives with many tapes
    - Juke boxes with petabytes (1000's of TB) of storage

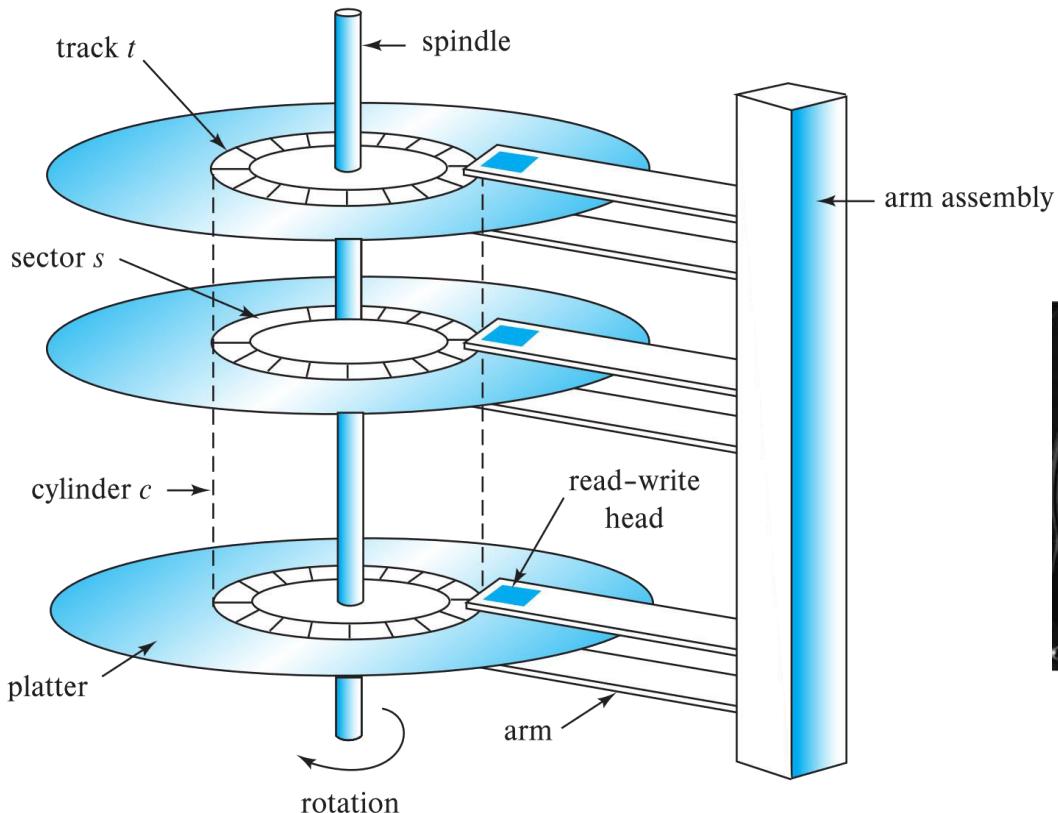


# Storage Interfaces

- Disk interface standards families
  - **SATA** (Serial ATA)
    - SATA 3 supports data transfer speeds of up to 6 gigabits/sec
  - **SAS** (Serial Attached SCSI)
    - SAS Version 3 supports 12 gigabits/sec
  - **NVMe** (Non-Volatile Memory Express) interface
    - Works with PCIe connectors to support lower latency and higher transfer rates
    - Supports data transfer rates of up to 24 gigabits/sec
- Disks usually connected directly to computer system
- In **Storage Area Networks (SAN)**, a large number of disks are connected by a high-speed network to a number of servers
- In **Network Attached Storage (NAS)** networked storage provides a file system interface using networked file system protocol, instead of providing a disk system interface



# Magnetic Hard Disk Mechanism



Schematic diagram of magnetic disk drive

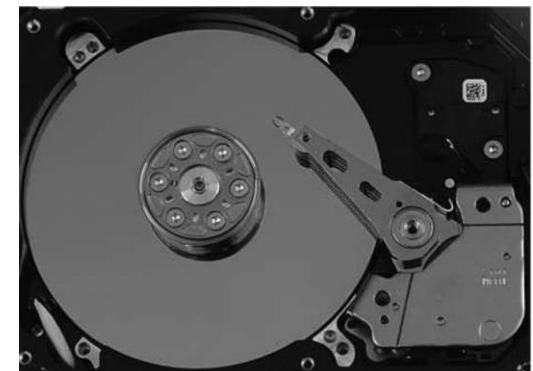


Photo of magnetic disk drive



# Magnetic Disks

- **Read-write head**
- Surface of platter divided into circular **tracks**
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**.
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
  - multiple disk platters on a single spindle (1 to 5 usually)
  - one head per platter, mounted on a common arm.
- **Cylinder  $i$**  consists of  $i^{\text{th}}$  track of all the platters



# Magnetic Disks (Cont.)

- **Disk controller** – interfaces between the computer system and the disk drive hardware.
  - accepts high-level commands to read or write a sector
  - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
  - Computes and attaches **checksums** to each sector to verify that data is read back correctly
    - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
  - Ensures successful writing by reading back sector after writing it
  - Performs **remapping of bad sectors**



# Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
    - Average latency is 1/2 of the above latency.
  - Overall latency is 5 to 20 msec depending on disk model
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 200 MB per second max rate, lower for inner tracks



# Performance Measures (Cont.)

- **Disk block** is a logical unit for storage allocation and retrieval
  - 4 to 16 kilobytes typically
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks
- **Sequential access pattern**
  - Successive requests are for successive disk blocks
  - Disk seek required only for first block
- **Random access pattern**
  - Successive requests are for blocks that can be anywhere on disk
  - Each access requires a seek
  - Transfer rates are low since a lot of time is wasted in seeks
- **I/O operations per second (IOPS)**
  - Number of random block reads that a disk can support per second
  - 50 to 200 IOPS on current generation magnetic disks



# Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
    - E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
  - MTTF decreases as disk ages



# Magnetic Tapes

- Hold large volumes of data and provide high transfer rates
  - Few GB for DAT (Digital Audio Tape) format, 10-40 GB with DLT (Digital Linear Tape) format, 100 GB+ with Ultrium format, and 330 GB with Ampex helical scan format
  - Transfer rates from few to 10s of MB/s
- Tapes are cheap, but cost of drives is very high
- Very slow access time in comparison to magnetic and optical disks
  - limited to sequential access.
  - Some formats (Accelis) provide faster seek (10s of seconds) at cost of lower capacity
- Used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another.
- Tape jukeboxes used for very large capacity storage
  - Multiple petabytes ( $10^{15}$  bytes)



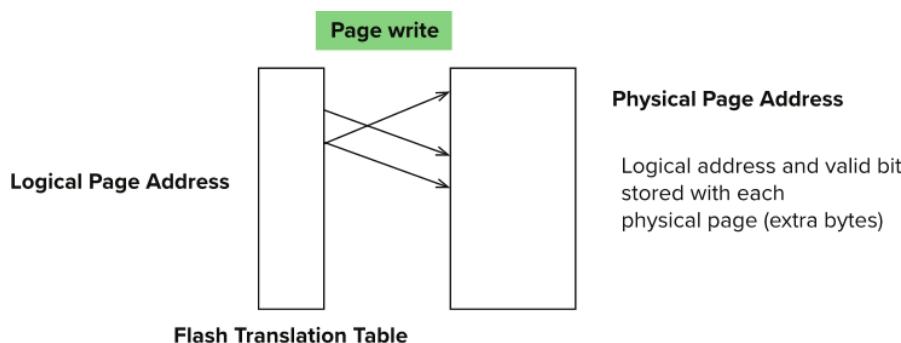
# Flash Storage

- NOR flash vs NAND flash
- NAND flash
  - used widely for storage, cheaper than NOR flash
  - requires page-at-a-time read (page: 512 bytes to 4 KB)
    - 20 to 100 microseconds for a page read
    - Not much difference between sequential and random read
  - Page can only be written once
    - Must be erased to allow rewrite
- **Solid state disks**
  - Use standard block-oriented disk interfaces, but store data on multiple flash storage devices internally
  - Transfer rate of up to 500 MB/sec using SATA, and up to 3 GB/sec using NVMe PCIe



# Flash Storage (Cont.)

- Erase happens in units of **erase block**
  - Takes 2 to 5 millisecs
  - Erase block typically 256 KB to 1 MB (128 to 256 pages)
- **Remapping** of logical page addresses to physical page addresses avoids waiting for erase
- **Flash translation table** tracks mapping
  - also stored in a label field of flash page
  - remapping carried out by **flash translation layer**



- After 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used
  - **wear leveling**



# SSD Performance Metrics

- Random reads/writes per second
  - Typical 4 KB reads: 10,000 reads per second (10,000 IOPS)
  - Typical 4KB writes: 40,000 IOPS
  - SSDs support parallel reads
    - Typical 4KB reads:
      - 100,000 IOPS with 32 requests in parallel (QD-32) on SATA
      - 350,000 IOPS with QD-32 on NVMe PCIe
    - Typical 4KB writes:
      - 100,000 IOPS with QD-32, even higher on some models
- Data transfer rate for sequential reads/writes
  - 400 MB/sec for SATA3, 2 to 3 GB/sec using NVMe PCIe
- **Hybrid disks:** combine small amount of flash cache with larger magnetic disk



# Storage Class Memory

- 3D-XPoint memory technology pioneered by Intel
- Available as Intel Optane
  - SSD interface shipped from 2017
    - Allows lower latency than flash SSDs
  - Non-volatile memory interface announced in 2018
    - Supports direct access to words, at speeds comparable to main-memory speeds



# RAID

- **RAID: Redundant Arrays of Independent Disks**
  - disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
    - **high capacity** and **high speed** by using multiple disks in parallel,
    - **high reliability** by storing data redundantly, so that data can be recovered even if a disk fails
- The chance that some disk out of a set of  $N$  disks will fail is much higher than the chance that a specific single disk will fail.
  - E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
  - Techniques for using redundancy to avoid data loss are critical with large numbers of disks



# Improvement of Reliability via Redundancy

- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring (or shadowing)**
  - Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
    - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
    - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
      - Probability of combined event is very small
        - Except for dependent failure modes such as fire or building collapse or electrical power surges
- **Mean time to data loss** depends on mean time to failure, and **mean time to repair**
  - E.g., MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of  $500 \times 10^6$  hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)



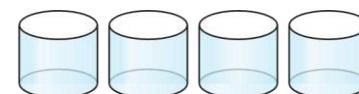
# Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
  1. Load balance multiple small accesses to increase throughput
  2. Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
- **Bit-level striping** – split the bits of each byte across multiple disks
  - In an array of eight disks, write bit  $i$  of each byte to disk  $i$ .
  - Each access can read data at eight times the rate of a single disk.
  - But seek/access time worse than for a single disk
    - Bit level striping is not used much any more
- **Block-level striping** – with  $n$  disks, block  $i$  of a file goes to disk  $(i \bmod n) + 1$ 
  - Requests for different blocks can run in parallel if the blocks reside on different disks
  - A request for a long sequence of blocks can utilize all disks in parallel

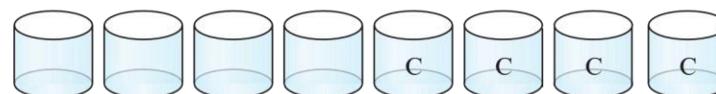


# RAID Levels

- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
  - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- **RAID Level 0:** Block striping; non-redundant.
  - Used in high-performance applications where data loss is not critical.
- **RAID Level 1:** Mirrored disks with block striping
  - Offers best write performance.
  - Popular for applications such as storing log files in a database system.



(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks



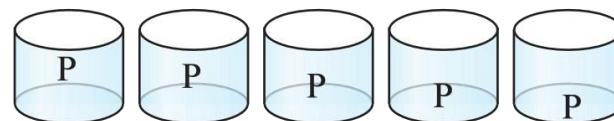
# RAID Levels (Cont.)

- **Parity blocks:** Parity block  $j$  stores XOR of bits from block  $j$  of each disk
  - When writing data to a block  $j$ , parity block  $j$  must also be computed and written to disk
    - Can be done by using old parity block, old value of current block and new value of current block (2 block reads + 2 block writes)
    - Or by recomputing the parity value using the new values of blocks corresponding to the parity block
      - More efficient for writing large amounts of data sequentially
  - To recover data for a block, compute XOR of bits from all other blocks in the set including the parity block



# RAID Levels (Cont.)

- **RAID Level 5:** Block-Interleaved Distributed Parity; partitions data and parity among all  $N + 1$  disks, rather than storing data in  $N$  disks and parity in 1 disk.
  - E.g., with 5 disks, parity block for  $n$ th set of blocks is stored on disk  $(n \bmod 5) + 1$ , with the data blocks stored on the other 4 disks.



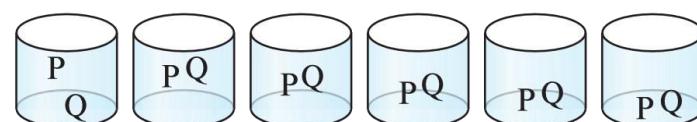
(c) RAID 5: block-interleaved distributed parity

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4



# RAID Levels (Cont.)

- **RAID Level 5 (Cont.)**
  - Block writes occur in parallel if the blocks and their parity blocks are on different disks.
- **RAID Level 6: P+Q Redundancy** scheme; similar to Level 5, but stores two error correction blocks (P, Q) instead of single parity block to guard against multiple disk failures.
  - Better reliability than Level 5 at a higher cost
    - Becoming more important as storage sizes increase



(d) RAID 6: P + Q redundancy



# RAID Levels (Cont.)

- Other levels (not used in practice):
  - **RAID Level 2:** Memory-Style Error-Correcting-Codes (ECC) with bit striping.
  - **RAID Level 3:** Bit-Interleaved Parity
  - **RAID Level 4:** Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate **parity disk** for corresponding blocks from  $N$  other disks.
    - RAID 5 is better than RAID 4, since with RAID 4 with random writes, parity disk gets much higher write load than other disks and becomes a bottleneck



# Choice of RAID Level

- Factors in choosing RAID level
  - Monetary cost
  - Performance: Number of I/O operations per second, and bandwidth during normal operation
  - Performance during failure
  - Performance during rebuild of failed disk
    - Including time taken to rebuild failed disk
- RAID 0 is used only when data safety is not important
  - E.g., data can be recovered quickly from other sources



# Choice of RAID Level (Cont.)

- Level 1 provides much better write performance than level 5
  - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
- Level 1 had higher storage cost than level 5
- Level 5 is preferred for applications where writes are sequential and large (many blocks), and need large amounts of data storage
- RAID 1 is preferred for applications with many random/small updates
- Level 6 gives better data protection than RAID 5 since it can tolerate two disk (or disk block) failures
  - Increasing in importance since latent block failures on one disk, coupled with a failure of another disk can result in data loss with RAID 1 and RAID 5.



# Hardware Issues

- **Software RAID:** RAID implementations done entirely in software, with no special hardware support
- **Hardware RAID:** RAID implementations with special hardware
  - Use non-volatile RAM to record writes that are being executed
  - Beware: power failure during write can result in corrupted disk
    - E.g., failure after writing one block but before writing the second in a mirrored system
    - Such corrupted data must be detected when power is restored
      - Recovery from corruption is similar to recovery from failed disk
      - NV-RAM helps to efficiently detect potentially corrupted blocks
        - Otherwise all blocks of disk must be read and compared with mirror/parity block



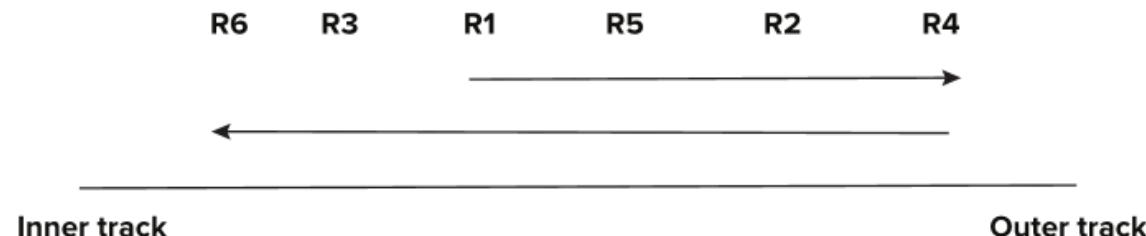
# Hardware Issues (Cont.)

- **Latent failures:** data successfully written earlier gets damaged
  - can result in data loss even if only one disk fails
- **Data scrubbing:**
  - continually scan for latent failures, and recover from copy/parity
- **Hot swapping:** replacement of disk while system is running, without power down
  - Supported by some hardware RAID systems,
  - reduces time to recovery, and improves availability greatly
- Many systems maintain **spare disks** which are kept online, and used as replacements for failed disks immediately on detection of failure
  - Reduces time to recovery greatly
- Many hardware RAID systems ensure that a single point of failure will not stop the functioning of the system by using
  - Redundant power supplies with battery backup
  - Multiple controllers and multiple interconnections to guard against controller/interconnection failures



# Optimization of Disk-Block Access

- **Buffering:** in-memory buffer to cache disk blocks
- **Read-ahead:** Read extra blocks from a track in anticipation that they will be requested soon
- **Disk-arm-scheduling** algorithms re-order block requests so that disk arm movement is minimized
  - **elevator algorithm**





# Optimization of Disk Block Access (Cont.)

- **File organization**

- Allocate blocks of a file in as contiguous a manner as possible
- Allocation in units of **extents**
- Files may get **fragmented**
  - E.g., if free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
  - Sequential access to a fragmented file results in increased disk arm movement
  - Some systems have utilities to **defragment** the file system, in order to speed up file access

- **Non-volatile write buffers**



## End of Chapter 12



# Chapter 13: Data Storage Structures

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.
- One approach
  - Assume record size is fixed
  - Each file has records of one particular type only
  - Different files are used for different relations
- This case is easiest to implement; will consider variable length records later
- We assume that records are smaller than a disk block



# Fixed-Length Records

- Simple approach:
  - Store record  $i$  starting from byte  $n * (i - 1)$ , where  $n$  is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



# Fixed-Length Records

- Deletion of record  $i$ : alternatives:
  - **move records  $i + 1, \dots, n$  to  $i, \dots, n - 1$**
  - move record  $n$  to  $i$
  - do not move records, but link all free records on a *free list*

**Record 3 deleted**

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



# Fixed-Length Records

- Deletion of record  $i$ : alternatives:
  - move records  $i + 1, \dots, n$  to  $i, \dots, n - 1$
  - **move record  $n$  to  $i$**
  - do not move records, but link all free records on a *free list*

**Record 3 deleted and replaced by record 11**

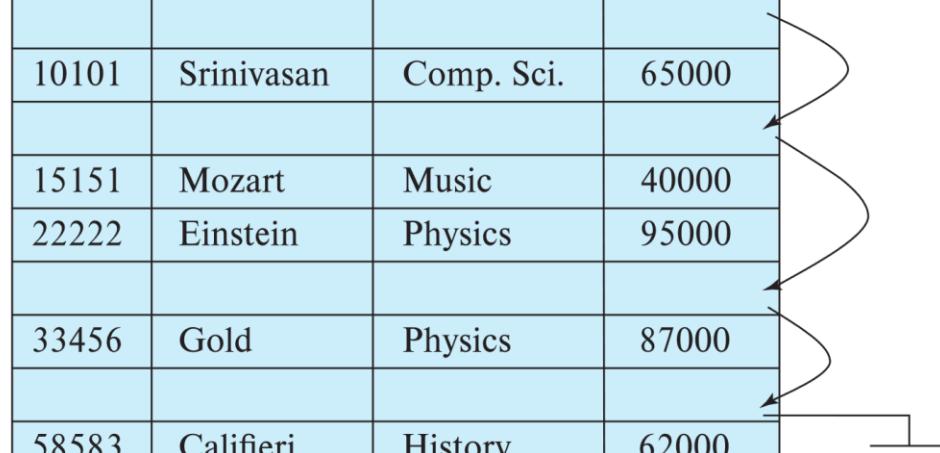
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000



# Fixed-Length Records

- Deletion of record  $i$ : alternatives:
  - move records  $i + 1, \dots, n$  to  $i, \dots, n - 1$
  - move record  $n$  to  $i$
  - **do not move records, but link all free records on a *free list***

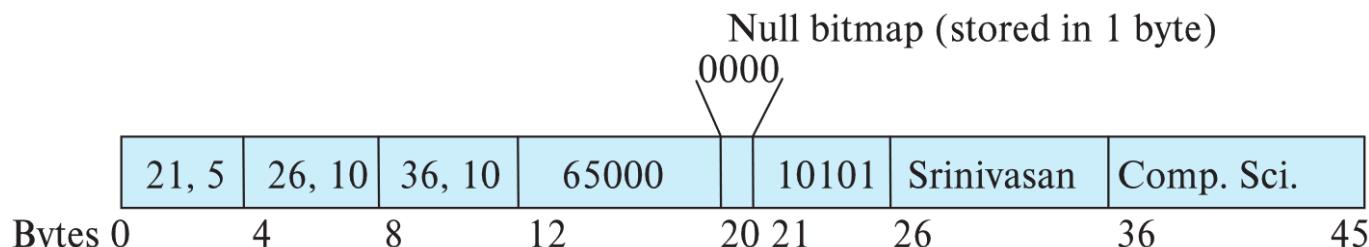
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000





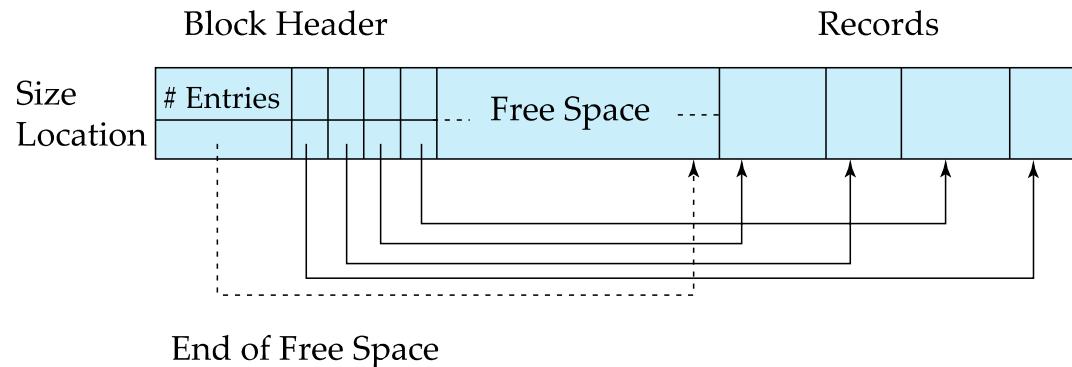
# Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap





# Variable-Length Records: Slotted Page Structure



- Used for organizing records within a block
- Slotted page header contains:
  - number of record entries
  - end of free space in the block
  - An array whose entries contain location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.
  - This level of indirection allows records to move – prevents fragmentation



# Organization of Records in Files

- **Heap** – record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O
- **B+-tree file organization**
  - Ordered storage even with inserts/deletes
  - More on this in next chapter
- **Hashing** – a hash function computed on search key; the result specifies in which block of the file the record should be placed
  - More on this in next chapter



# Heap File Organization

- Records can be placed anywhere in the file where there is free space
- Records usually do not move once allocated
- Important to be able to efficiently find free space within file
- **Free-space map**
  - Array with 1 entry per block. Each entry is a few bits to a byte, and records fraction of block that is free
  - In example below, 3 bits per block, value divided by 8 indicates fraction of block that is free

4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Can have second-level free-space map
- In example below, each entry stores maximum from 4 entries of first-level free-space map

4	7	2	6
---	---	---	---

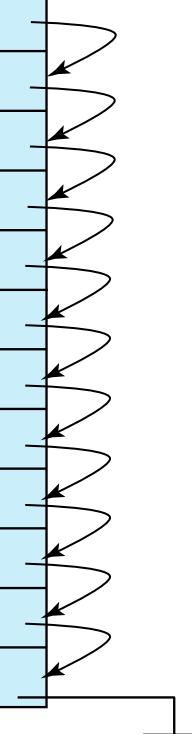
- Free space map written to disk periodically, OK to have wrong (old) values for some entries (will be detected and fixed)



# Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

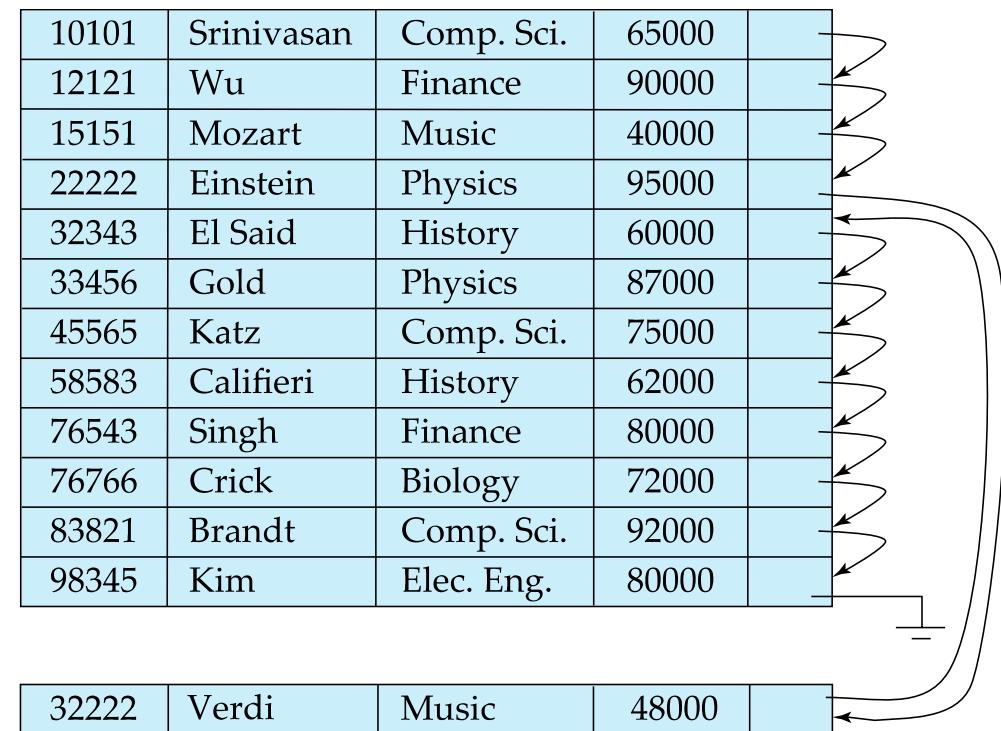
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	





# Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion – locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an **overflow block**
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order





# Multitable Clustering File Organization

Store several relations in one file using a **multitable clustering** file organization

*department*

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

*instructor*

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

multitable clustering  
of *department* and  
*instructor*

Comp. Sci.	Taylor	100000	
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
Physics	Watson	70000	
33456	Gold	Physics	87000



# Multitable Clustering File Organization (cont.)

- good for queries involving *department*  $\bowtie$  *instructor*, and for queries involving one single department and its instructors
- bad for queries involving only *department*
- results in variable size records
- Can add pointer chains to link records of a particular relation



# Partitioning

- **Table partitioning:** Records in a relation can be partitioned into smaller relations that are stored separately
- E.g., *transaction* relation may be partitioned into *transaction\_2018*, *transaction\_2019*, etc.
- Queries written on *transaction* must access records in all partitions
  - Unless query has a selection such as *year=2019*, in which case only one partition is needed
- Partitioning
  - Reduces costs of some operations such as free space management
  - Allows different partitions to be stored on different storage devices
    - E.g., *transaction* partition for current year on SSD, for older years on magnetic disk



# Data Dictionary Storage

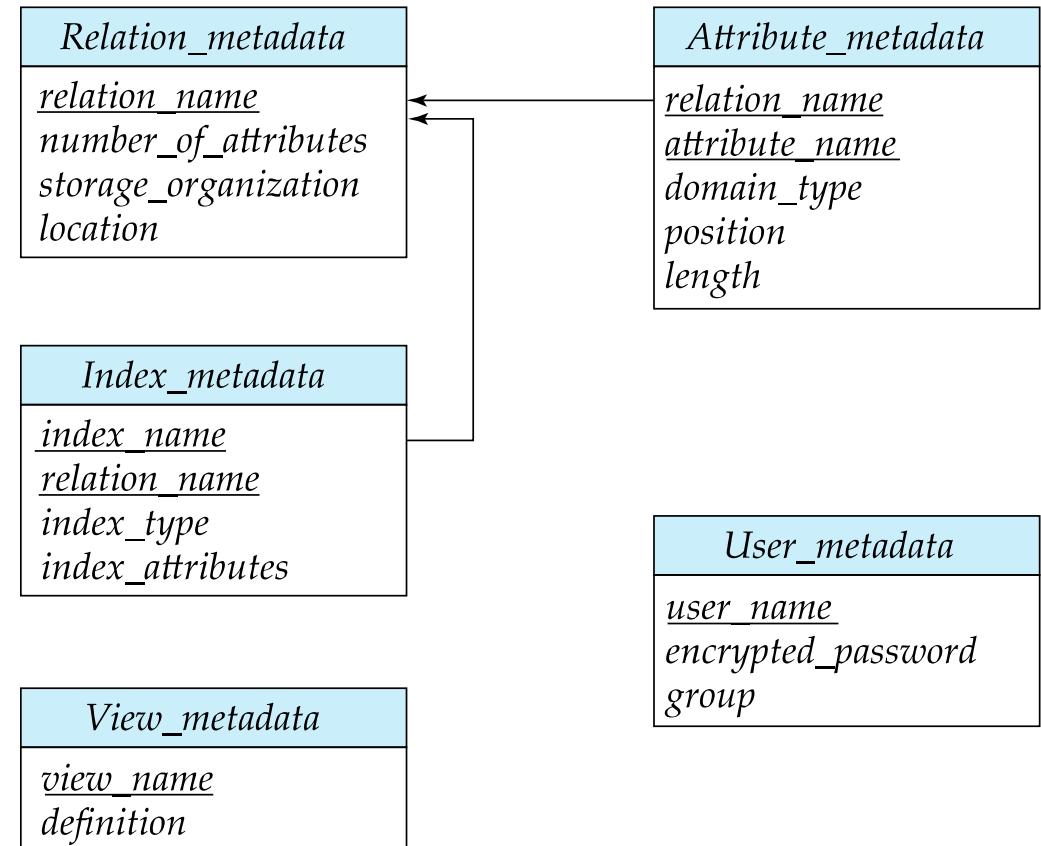
The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

- Information about relations
  - names of relations
  - names, types and lengths of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/...)
  - Physical location of relation
- Information about indices



# Relational Representation of System Metadata

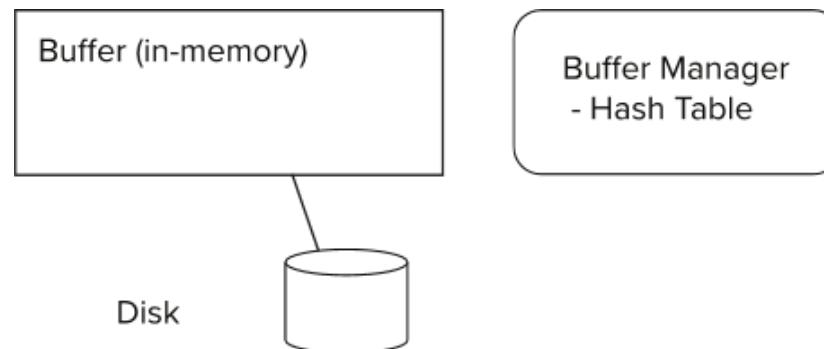
- A miniature database consisting of all this metadata
- Specialized data structures designed for efficient access, in memory
- Whenever to retrieve records from a relation, first consult *Relation\_metadata* to find storage org. and location





# Storage Access

- Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.





# Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  - If the block is already in the buffer, buffer manager returns the address of the block in main memory
  - If the block is not in the buffer, the buffer manager
    - Allocates space in the buffer for the block
      - Replacing (throwing out) some other block, if required, to make space for the new block.
      - Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
    - Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.



# Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
  - Idea behind LRU – use past pattern of block references as a predictor of future references
- LRU may NOT be best idea for database queries
- Database queries have well-defined access patterns
  - Possible to predict pattern of future queries more accurately than OS
  - Database system can use the information in a user's query to predict future references
- Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.



# Buffer-Replacement Policies (Cont.)

- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Operating system or buffer manager may reorder writes
  - Can lead to corruption of data structures on disk
    - E.g., linked list of blocks with missing block on disk
    - File systems perform consistency check to detect such situations
  - Careful ordering of writes can avoid many such problems



# Column-Oriented Storage

- Also known as **columnar representation**
- Store each attribute of a relation separately
- Example

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000



# Columnar Representation

- Benefits:
  - Reduced IO if only some attributes are accessed
  - Improved CPU cache performance
  - Improved compression
  - **Vector processing** on modern CPU architectures
- Drawbacks
  - Cost of tuple reconstruction from columnar representation
  - Cost of tuple deletion and update
  - Cost of decompression
- Columnar representation found to be more efficient for decision support than row-oriented representation
- Traditional row-oriented representation preferable for transaction processing
- Some databases support both representations
  - Called **hybrid row/column stores**



## End of Chapter 13



# Chapter 14: Indexing

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

- Basic Concepts
- Ordered Indices
- B<sup>+</sup>-Tree Index Files
- B-Tree Index Files
- Hashing
- Write-optimized indices
- Spatio-Temporal Indexing



# Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form



- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** index entries are stored sorted on the search key value
  - **Hash indices:** index entries are distributed uniformly across “buckets”, determined using a “hash function” on search key values.



# Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - Finding records with a specified value in the attribute
  - Finding records with an attribute value falling in a specified range
- Access time
- Insertion time
- Deletion time
- Space overhead



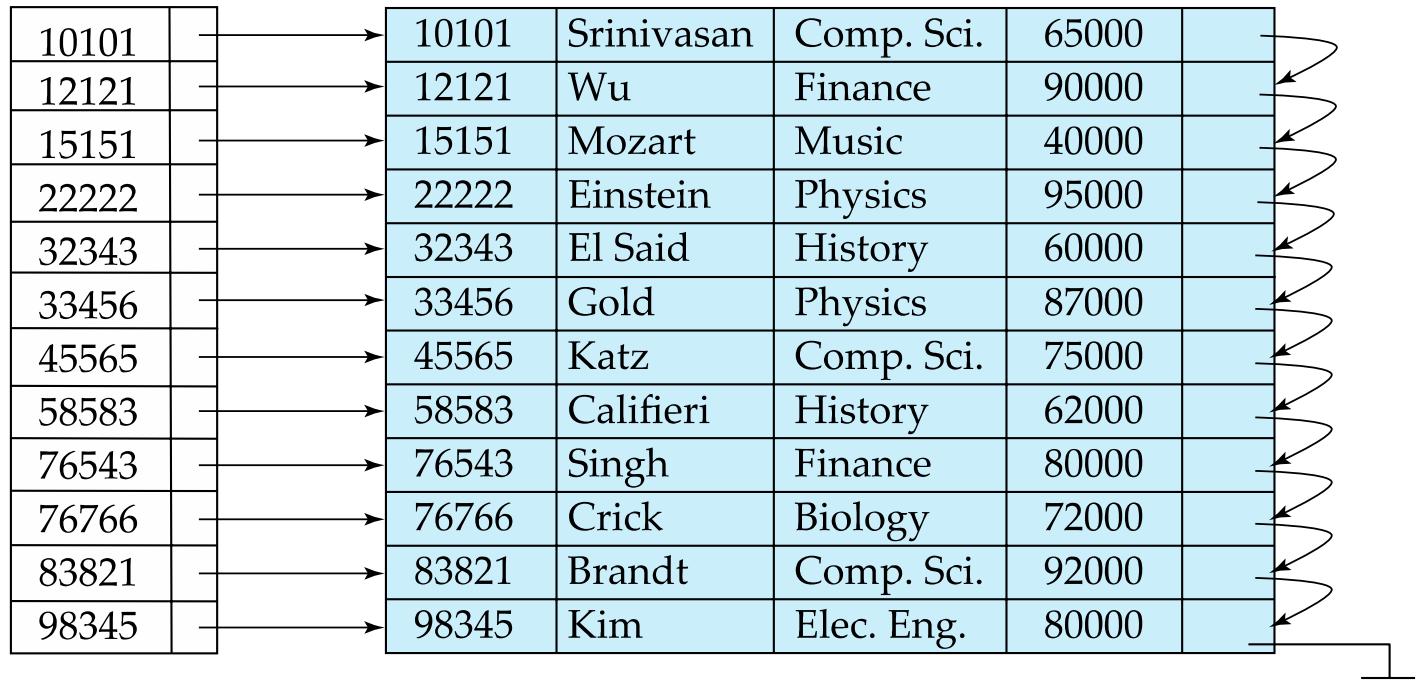
# Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value.
- **Primary index** : in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.
- Files which are sequentially ordered w.r.t. a search key, and has a clustering index on the search key, are called **index-sequential files**.



# Primary Index: Dense Index Files

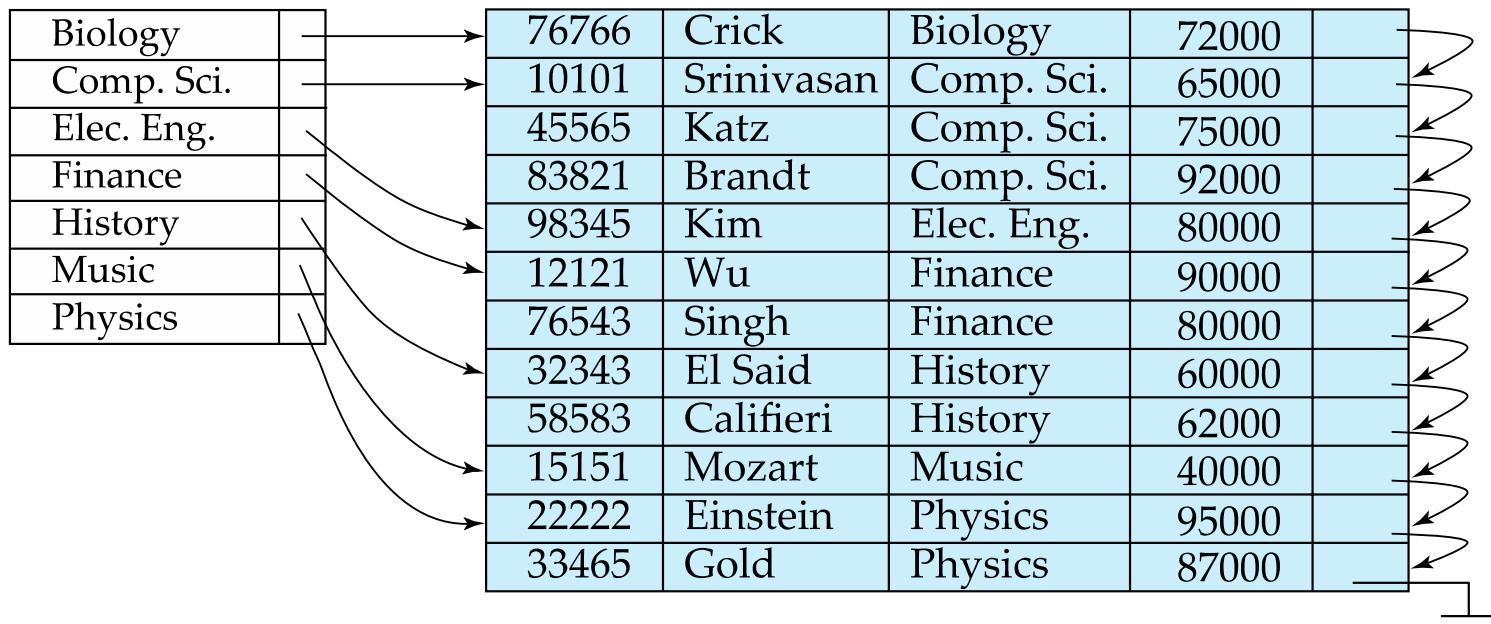
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation





# Primary Index: Dense Index Files (Cont.)

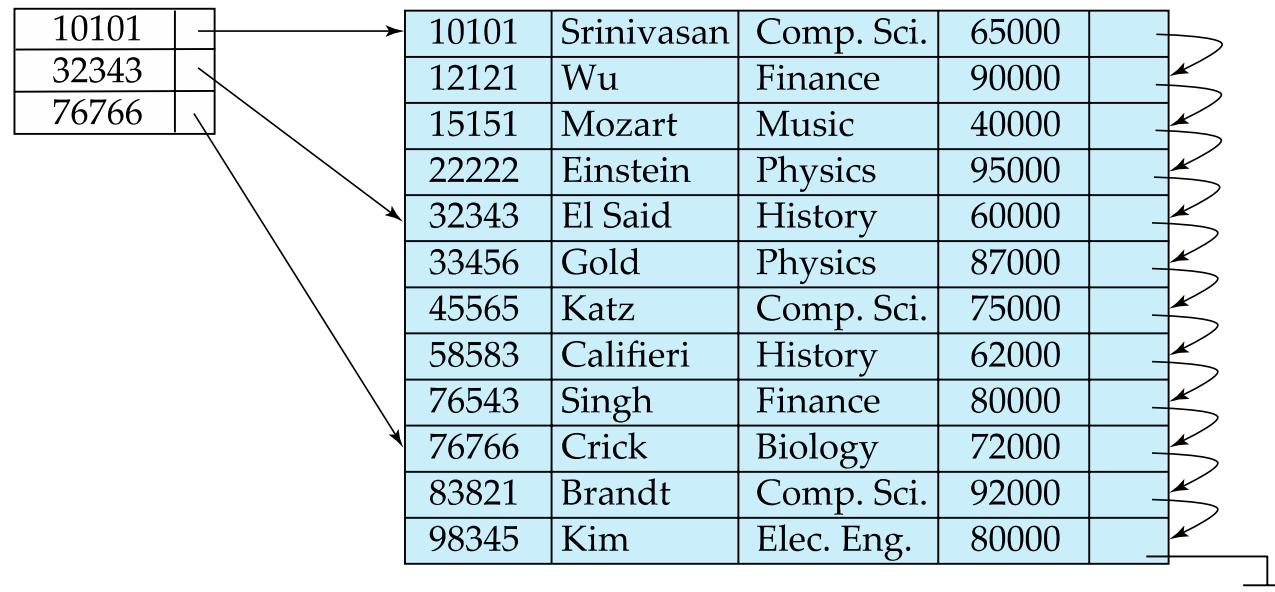
- Dense index on *dept\_name*, with *instructor* file sorted on *dept\_name*





# Primary Index: Sparse Index Files

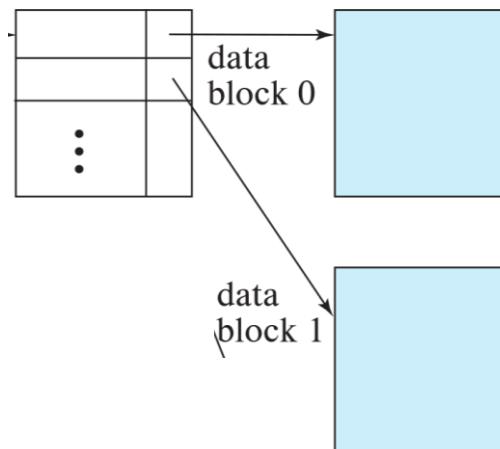
- **Sparse Index:** contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value  $K$  we:
  - Find index record with largest search-key value  $< K$
  - Search file sequentially starting at the record to which the index record points





# Primary Index: Sparse Index Files (Cont.)

- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff:**
  - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

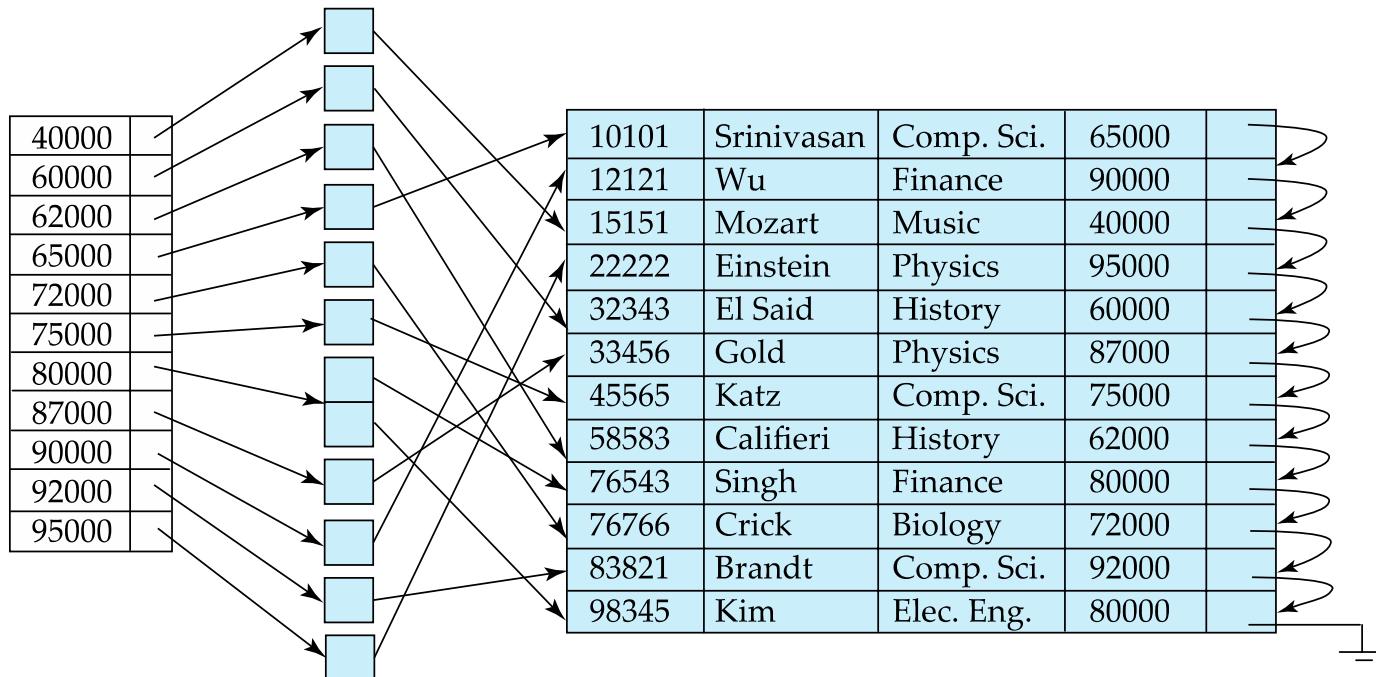


- For unclustered index (next slide): sparse index on top of dense index (multilevel index)



# Secondary Indices Example

- Secondary index on salary field of instructor



- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense



# Clustering vs Nonclustering Indices

- Indices offer substantial benefits when searching for records.
- BUT: indices imposes overhead on database modification
  - when a record is inserted or deleted, every index on the relation must be updated
  - When a record is updated, any index on an updated attribute must be updated
- Sequential scan using clustering index is efficient, but a sequential scan using a secondary (nonclustering) index is expensive on magnetic disk

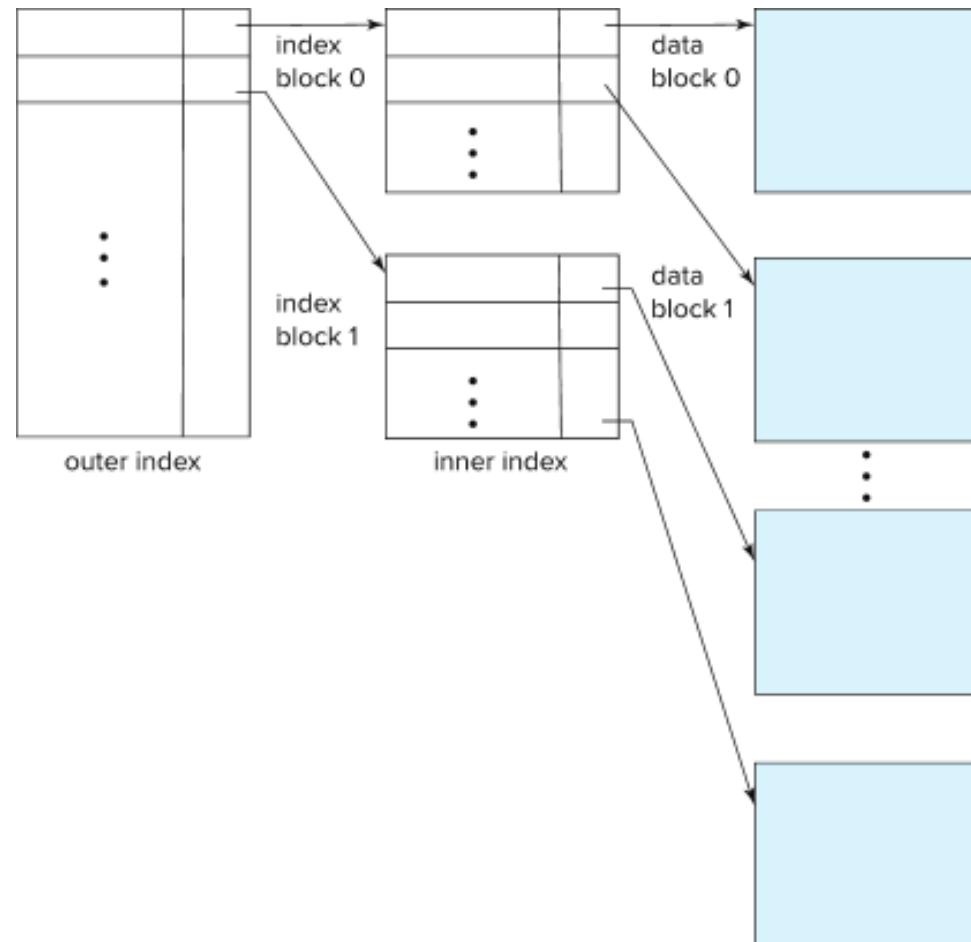


# Multilevel Index

- If index does not fit in memory, access becomes expensive.
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of the basic index
  - inner index – the basic index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



# Multilevel Index (Cont.)





# Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value of the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it
    - Indices are maintained as sequential files
    - Need to create space for new entry, overflow blocks may be required
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms



# Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- **Single-level index entry deletion:**
  - **Dense indices** – deletion of search-key is similar to file record deletion.
  - **Sparse indices** –
    - If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

10101	-			
32343				
76766				
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	



# Indices on Multiple Keys

- **Composite search key**

- E.g., index on *instructor* relation on attributes (*name*, *ID*)
- Values are sorted lexicographically
  - E.g. (John, 12121) < (John, 13514) and  
(John, 13514) < (Peter, 11223)
- Can query on just *name*, or on (*name*, *ID*)

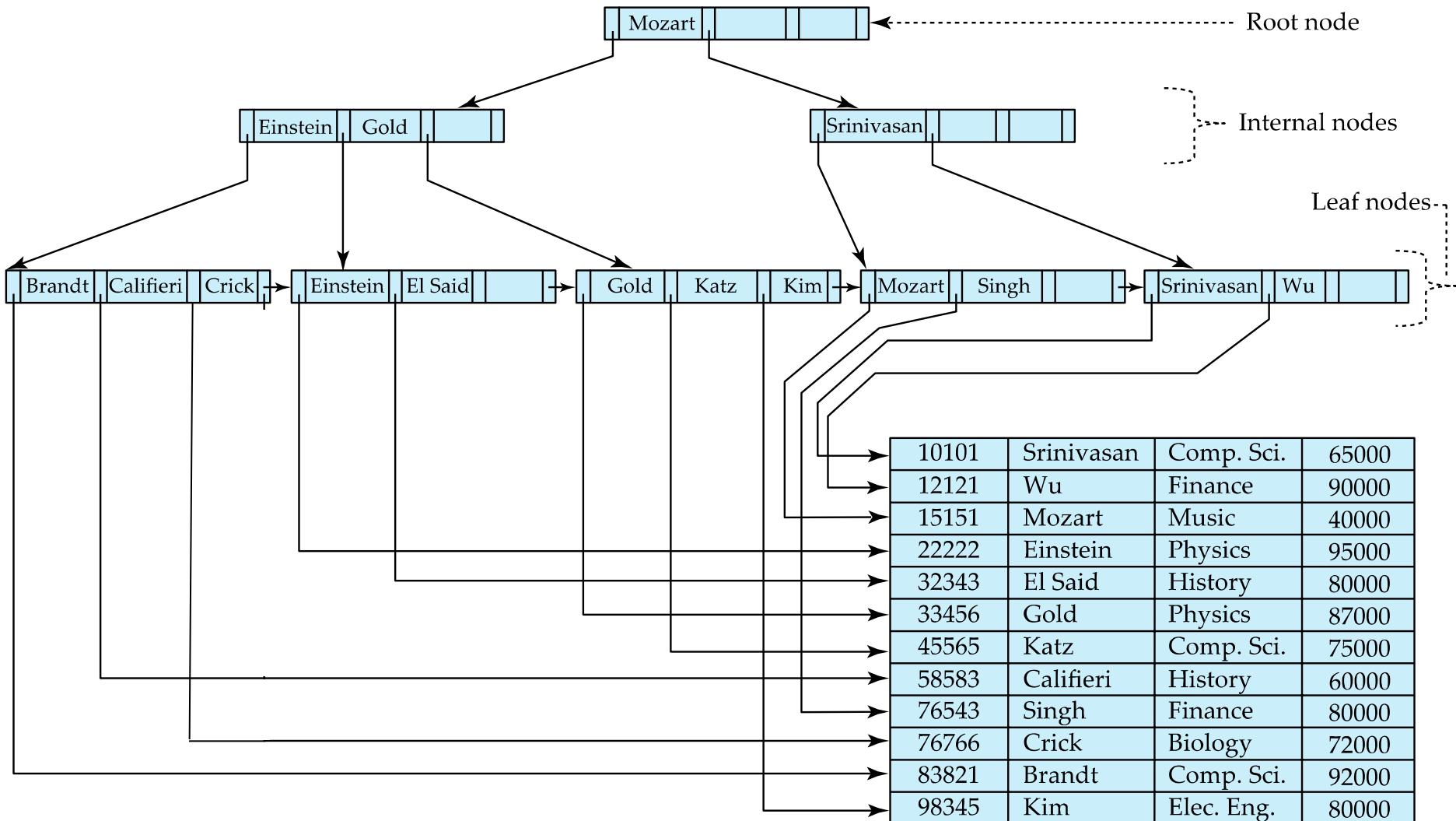


# B<sup>+</sup>-Tree Index Files

- Disadvantage of indexed-sequential files
  - Performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B<sup>+</sup>-tree index files:
  - Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B<sup>+</sup>-trees:
  - Extra insertion and deletion overhead, space overhead.
- Advantages of B<sup>+</sup>-trees outweigh disadvantages
  - B<sup>+</sup>-trees are used extensively



# Example of B+-Tree





## B<sup>+</sup>-Tree Index Files (Cont.)

A B<sup>+</sup>-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between  $\lceil n/2 \rceil$  and  $n$  children.
- A leaf node has between  $\lceil (n-1)/2 \rceil$  and  $n-1$  values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and  $(n-1)$  values.



# B<sup>+</sup>-Tree Node Structure

- Typical node

$P_1$	$K_1$	$P_2$	$\dots$	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	---------	-----------	-----------	-------

- $K_i$  are the search-key values
- $P_i$  are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

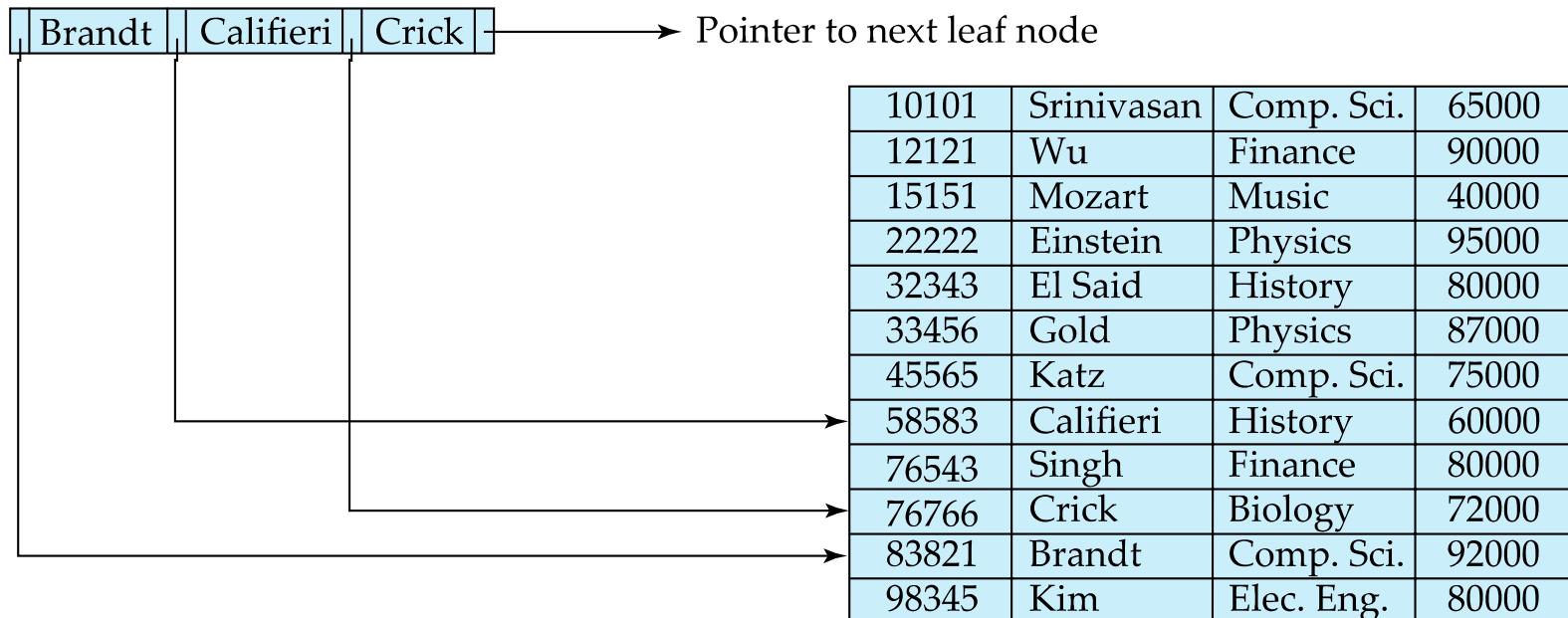
$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$



# Leaf Nodes in B<sup>+</sup>-Trees

Properties of a leaf node:

- For  $i = 1, 2, \dots, n-1$ , pointer  $P_i$  points to a file record with search-key value  $K_i$ ,
- $P_n$  points to next leaf node in search-key order
- If  $L_i, L_j$  are leaf nodes and  $i < j$ ,  $L_i$ 's search-key values are less than or equal to  $L_j$ 's search-key values  
leaf node





# Non-Leaf Nodes in B<sup>+</sup>-Trees

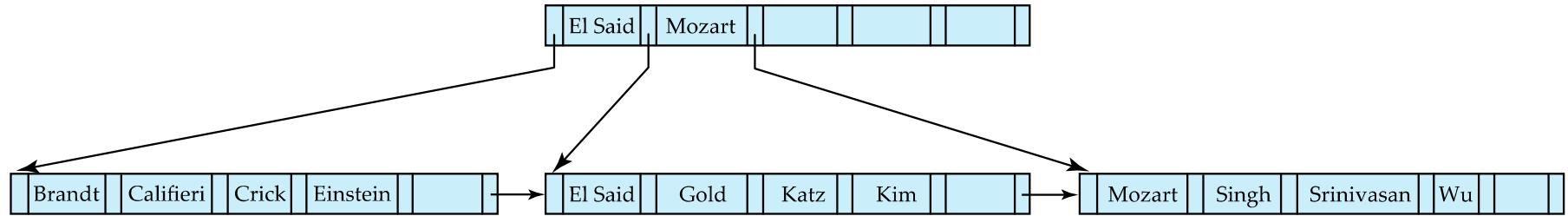
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with  $n$  pointers:
  - All the search-keys in the subtree to which  $P_1$  points are less than  $K_1$
  - For  $2 \leq i \leq n - 1$ , all the search-keys in the subtree to which  $P_i$  points have values **greater than or equal to  $K_{i-1}$**  and less than  $K_i$
  - All the search-keys in the subtree to which  $P_n$  points have values greater than or equal to  $K_{n-1}$
  - General structure

$P_1$	$K_1$	$P_2$	...	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	-----	-----------	-----------	-------



# Example of B<sup>+</sup>-tree

- B<sup>+</sup>-tree for *instructor* file ( $n = 6$ )



- Leaf nodes must have between 3 and 5 values ( $\lceil (n-1)/2 \rceil$  and  $n-1$ , with  $n = 6$ ).
- Non-leaf nodes other than root must have between 3 and 6 children ( $\lceil (n/2) \rceil$  and  $n$  with  $n = 6$ ).
- Root must have at least 2 children.



# Observations about B<sup>+</sup>-trees

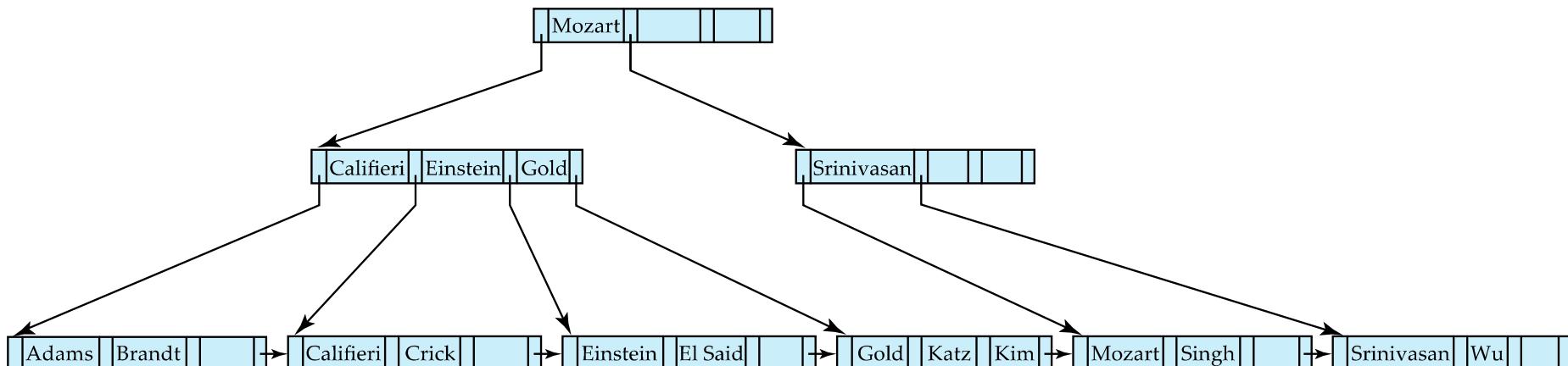
- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B<sup>+</sup>-tree form a hierarchy of sparse indices.
- The B<sup>+</sup>-tree contains a relatively small number of levels
  - Level below root has at least  $2 * \lceil n/2 \rceil$  values
  - Next level has at least  $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$  values
  - .. etc.
- If there are  $K$  search-key values in the file, the tree height is no more than  $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ 
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).



# Queries on B<sup>+</sup>-Trees

```
function find(v)
```

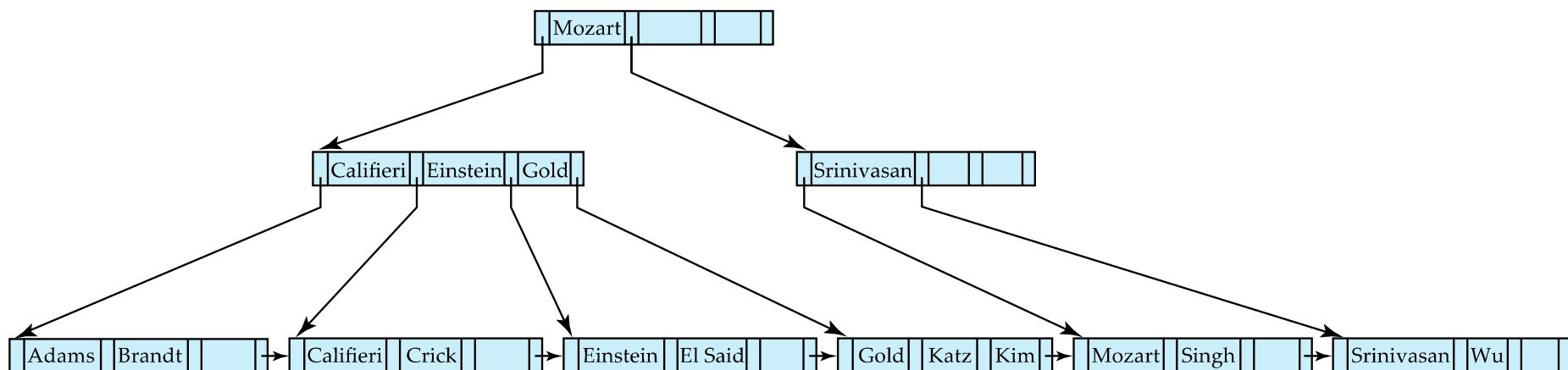
1.  $C = \text{root}$
2. **while** ( $C$  is not a leaf node)
  1. Let  $i$  be **least** number s.t.  $v \leq K_i$
  2. **if** there is no such number  $i$  **then**
    3. Set  $C = \text{last non-null pointer in } C$
  4. **else if** ( $v = C.K_i$ ) Set  $C = P_{i+1}$
  5. **else set**  $C = C.P_i$  /\*  $v < C.K_i$  \*/
3. **if** for some  $i$ ,  $C.K_i = v$  **then** return  $C.P_i$
4. **else** return null /\* no record with search-key value  $v$  exists. \*/





# Queries on B+-Trees (Cont.)

- **Range queries** find all records with search key values in a given range
  - See book for details of **function** *findRange(lb, ub)* which returns set of all such records
  - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a *next()* function





## Queries on B+-Trees (Cont.)

- If there are  $K$  search-key values in the file, the height of the tree is no more than  $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ .
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and  $n$  is typically around 100 (40 bytes per index entry).
- With 1 million search key values and  $n = 100$ 
  - at most  $\log_{50}(1,000,000) = 4$  nodes are accessed in a lookup traversal from root to leaf.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds



# Updates on B+-Trees: Insertion

Assume record already added to the file. Let

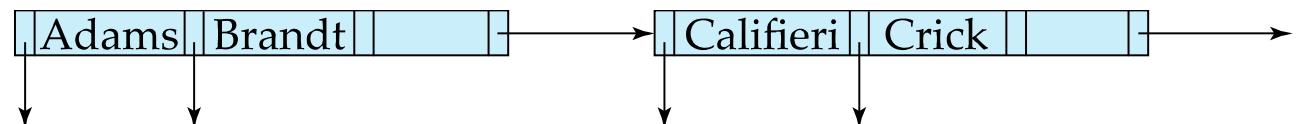
- |  $pr$  be pointer to the record, and let
- |  $v$  be the search key value of the record

1. Find the leaf node in which the search-key value would appear
  1. If there is room in the leaf node, insert  $(v, pr)$  pair in the leaf node
  2. Otherwise, split the node (along with the new  $(v, pr)$  entry) as discussed in the next slide, and propagate updates to parent nodes.



# Updates on B<sup>+</sup>-Trees: Insertion (Cont.)

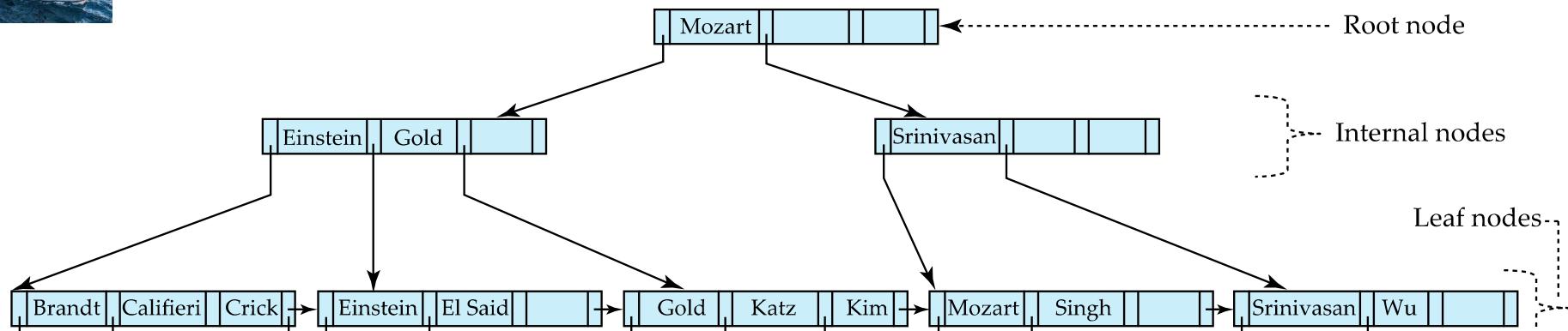
- Splitting a leaf node:
  - take the  $n$  (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first  $\lceil n/2 \rceil$  in the original node, and the rest in a new node.
  - let the new node be  $p$ , and let  $k$  be the least key value in  $p$ . Insert  $(k,p)$  in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.



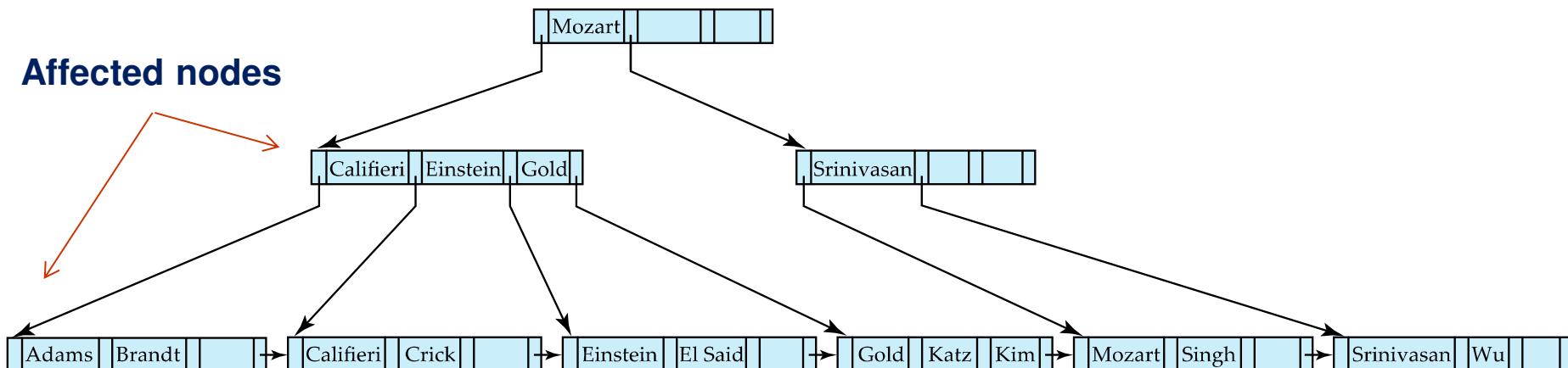
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams  
Next step: insert entry with (Califieri, pointer-to-new-node) into parent



# B+-Tree Insertion



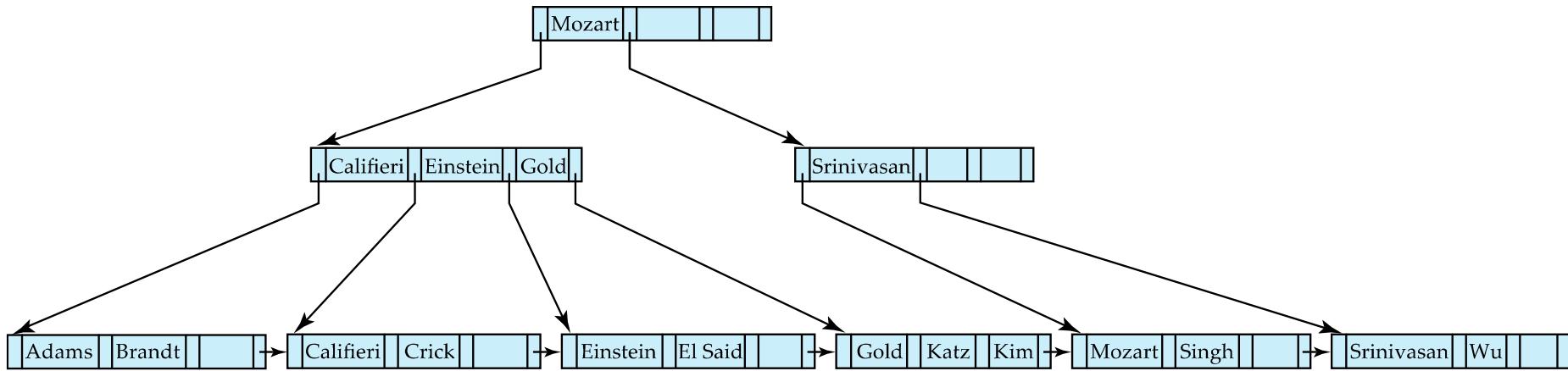
## Affected nodes



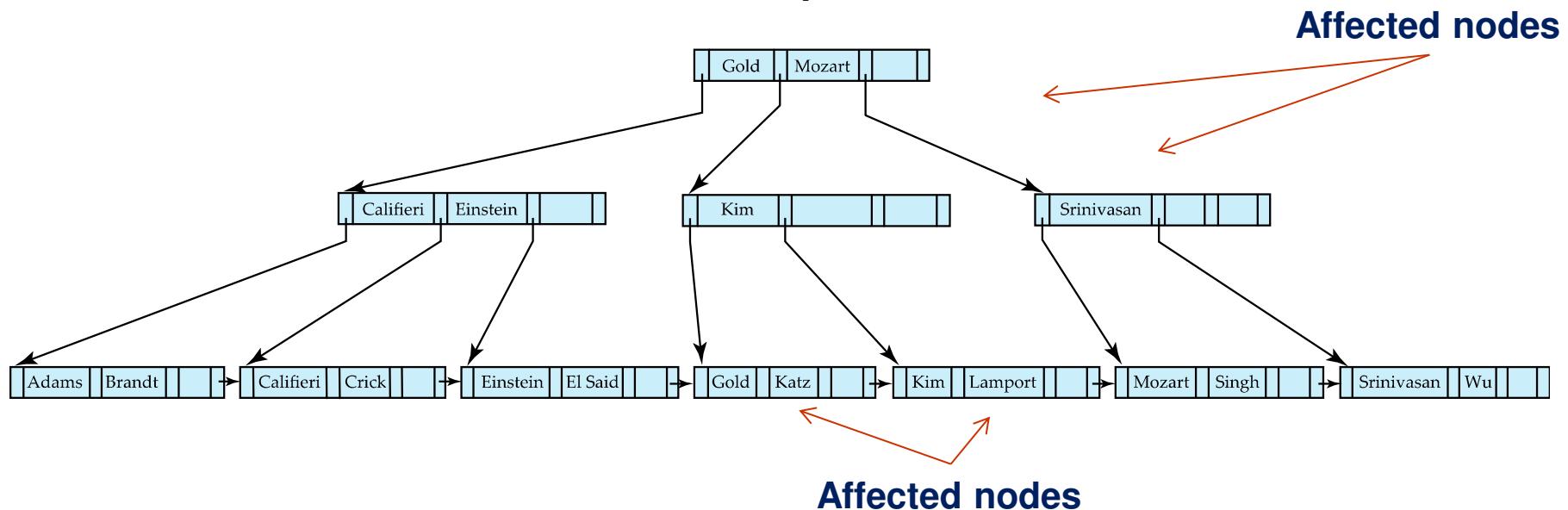
B+-Tree before and after insertion of “Adams”



# B+-Tree Insertion



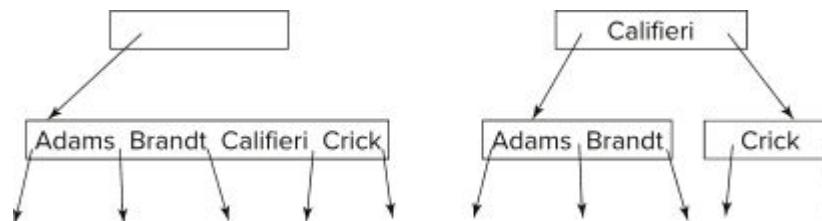
B+-Tree before and after insertion of “Lamport”





# Insertion in B+-Trees (Cont.)

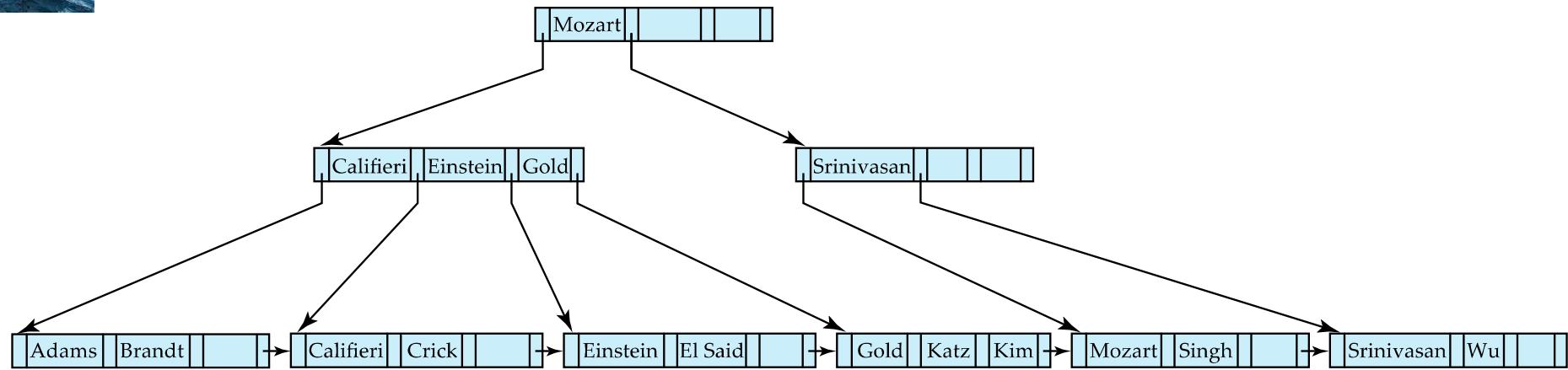
- Splitting a non-leaf node: when inserting  $(k,p)$  into an already full internal node  $N$ 
  - Copy  $N$  to an in-memory area  $M$  with space for  $n+1$  pointers and  $n$  keys
  - Insert  $(k,p)$  into  $M$
  - Copy  $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$  from  $M$  back into node  $N$
  - Copy  $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \dots, K_n, P_{n+1}$  from  $M$  into newly allocated node  $N'$
  - Insert  $(K_{\lceil n/2 \rceil}, N')$  into parent  $N$
- Example



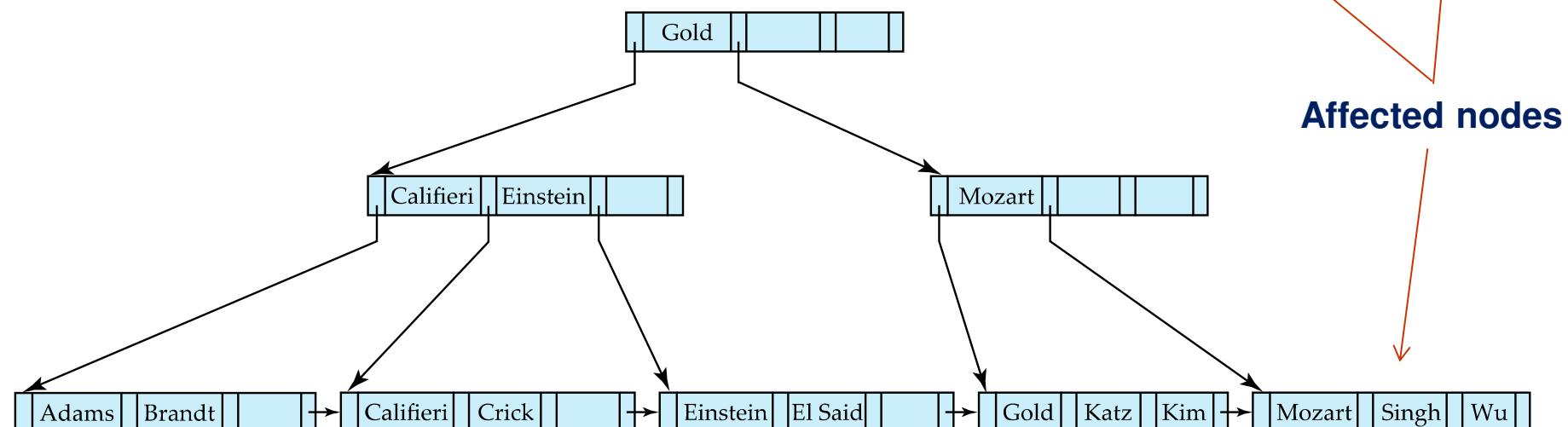
- **Read pseudocode in book!**



# Examples of B+-Tree Deletion



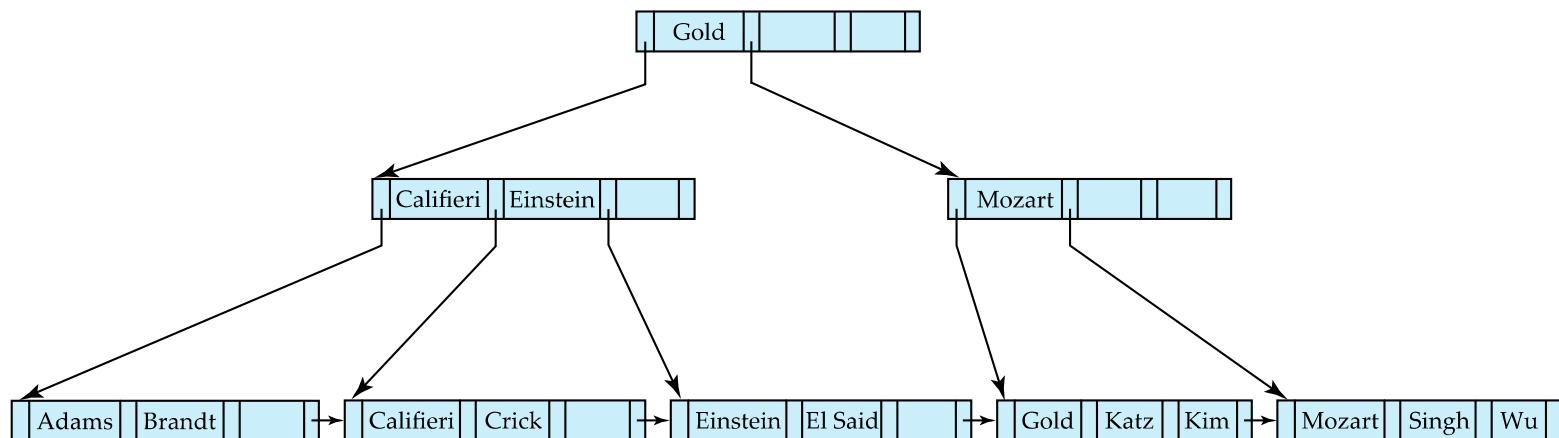
Before and after deleting “Srinivasan”



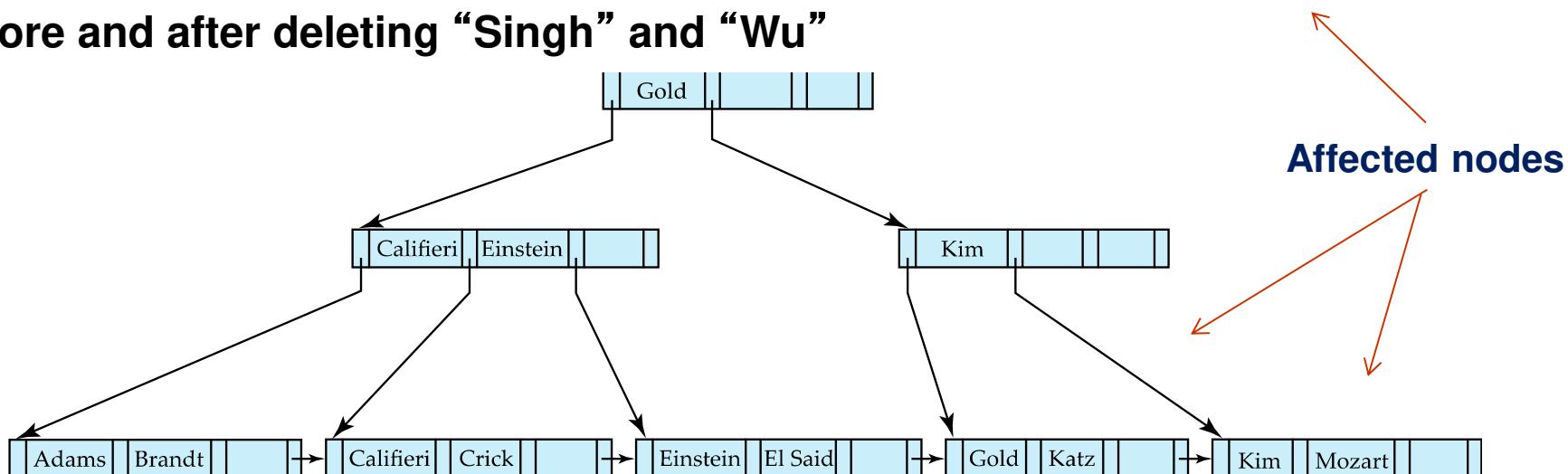
- Deleting “Srinivasan” causes **merging** of under-full leaves



# Examples of B+-Tree Deletion (Cont.)



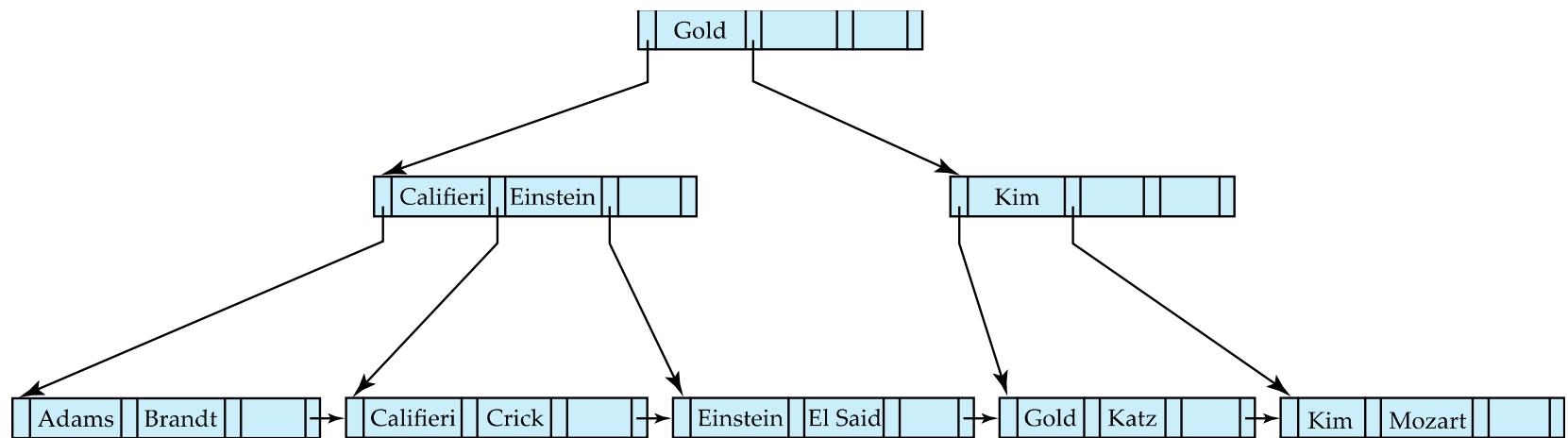
Before and after deleting “Singh” and “Wu”



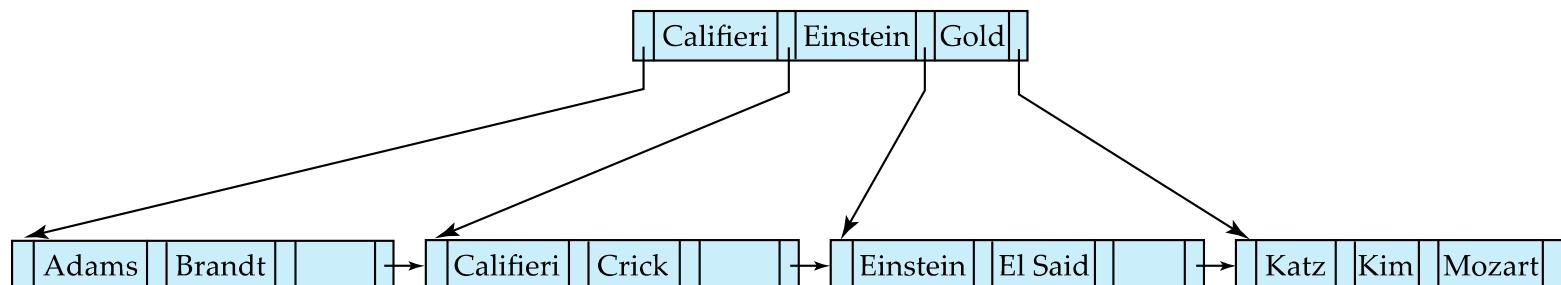
- Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result



# Example of B<sup>+</sup>-tree Deletion (Cont.)



**Before and after deletion of “Gold”**



- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted



# Updates on B<sup>+</sup>-Trees: Deletion

Assume record already deleted from file. Let  $V$  be the search key value of the record, and  $Pr$  be the pointer to the record.

- Remove  $(Pr, V)$  from the leaf node
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings***:
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair  $(K_{i-1}, P_i)$ , where  $P_i$  is the pointer to the deleted node, from its parent, recursively using the above procedure.



# Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has  $\lceil n/2 \rceil$  or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.



# Complexity of Updates

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With  $K$  entries and maximum fanout of  $n$ , worst case complexity of insert/delete of an entry is  $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, number of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, 1/2 with insertion in sorted order



# Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
      - Worst case complexity may be linear!
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used



# B<sup>+</sup>-Tree File Organization

- B<sup>+</sup>-Tree File Organization:
  - Leaf nodes in a B<sup>+</sup>-tree file organization store records, instead of pointers
  - Helps keep data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B<sup>+</sup>-tree index.



# B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

$P_1$	$K_1$	$P_2$	$\dots$	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	---------	-----------	-----------	-------

(a)

$P_1$	$B_1$	$K_1$	$P_2$	$B_2$	$K_2$	$\dots$	$P_{m-1}$	$B_{m-1}$	$K_{m-1}$	$P_m$
-------	-------	-------	-------	-------	-------	---------	-----------	-----------	-----------	-------

(b)

- Nonleaf node – pointers Bi are the bucket or file record pointers.

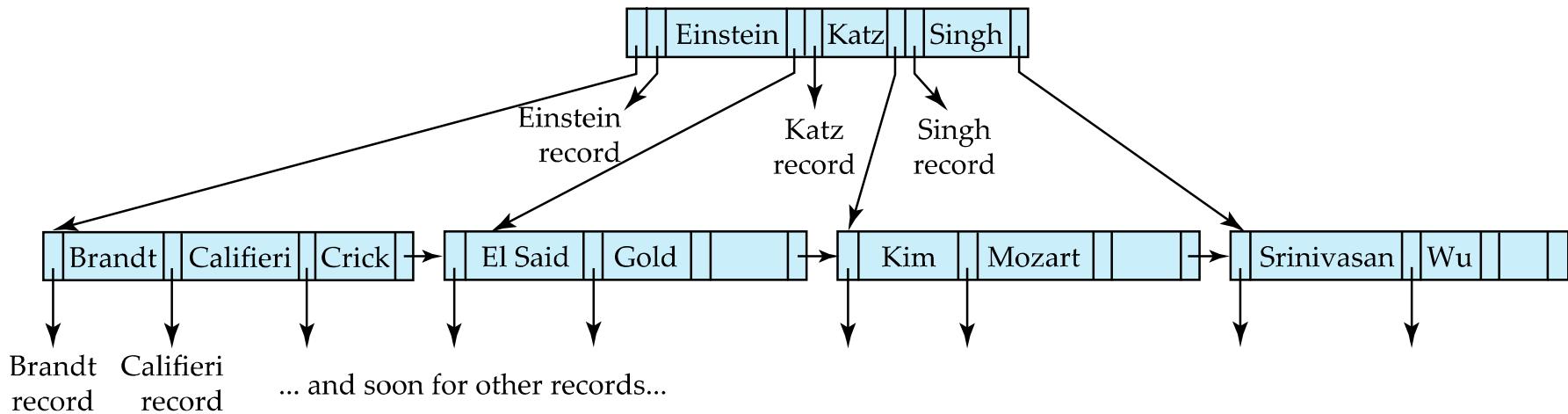


# B-Tree Index Files (Cont.)

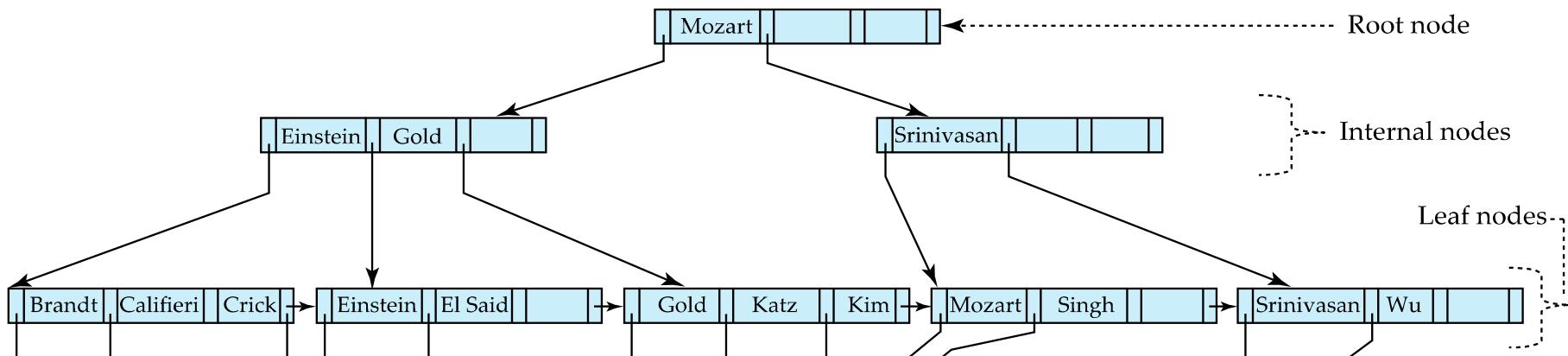
- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B<sup>+</sup>-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes store additional Bi bucket pointers, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B<sup>+</sup>-Tree
  - Insertion and deletion more complicated than in B<sup>+</sup>-Trees
  - Implementation is harder than B<sup>+</sup>-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.



# B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data





# Hashing



# Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function  $h$  is a function from the set of all search-key values  $K$  to the set of all bucket addresses  $B$ .
- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; **thus entire bucket has to be searched sequentially to locate an entry.**
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records



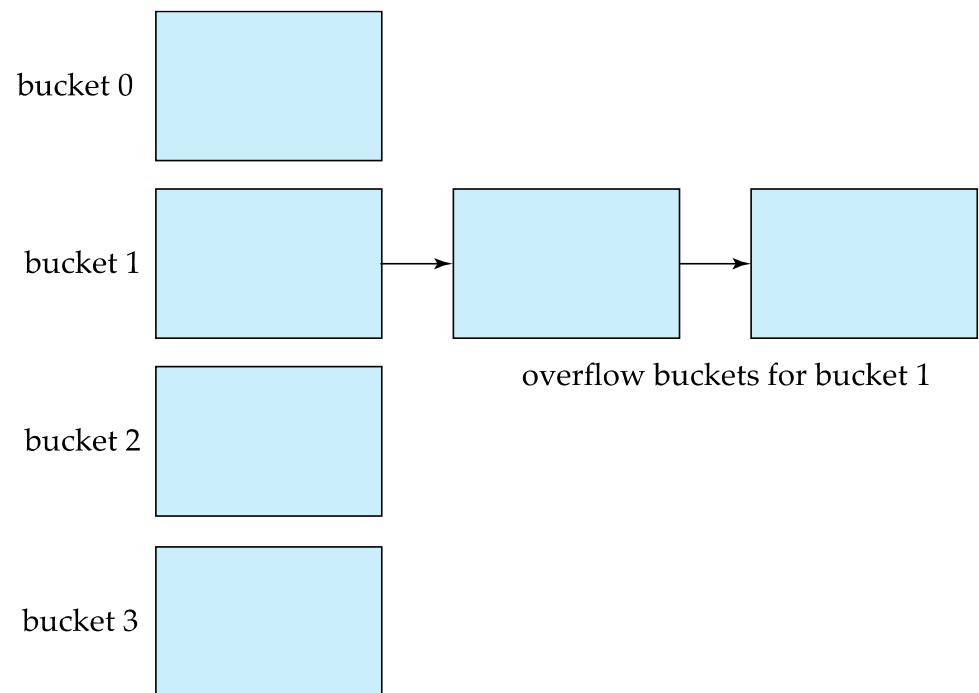
# Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using **overflow buckets**.



# Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing** or **closed hashing**
  - An alternative, called **open addressing** or **open hashing** which does not use overflow buckets, is not suitable for database applications.





# Example of Hash File Organization

Hash file organization of *instructor* file, using *dept\_name* as key  
(See figure in next slide.)

- There are 10 buckets,
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g.  $h(\text{Music}) = 1 \quad h(\text{History}) = 2$   
 $h(\text{Physics}) = 3 \quad h(\text{Elec. Eng.}) = 3$



# Example of Hash File Organization

Hash file organization of *instructor* file, using *dept\_name* as key.

bucket 0


bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7




# Deficiencies of Static Hashing

- In static hashing, function  $h$  maps search-key values to a fixed set of  $B$  of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

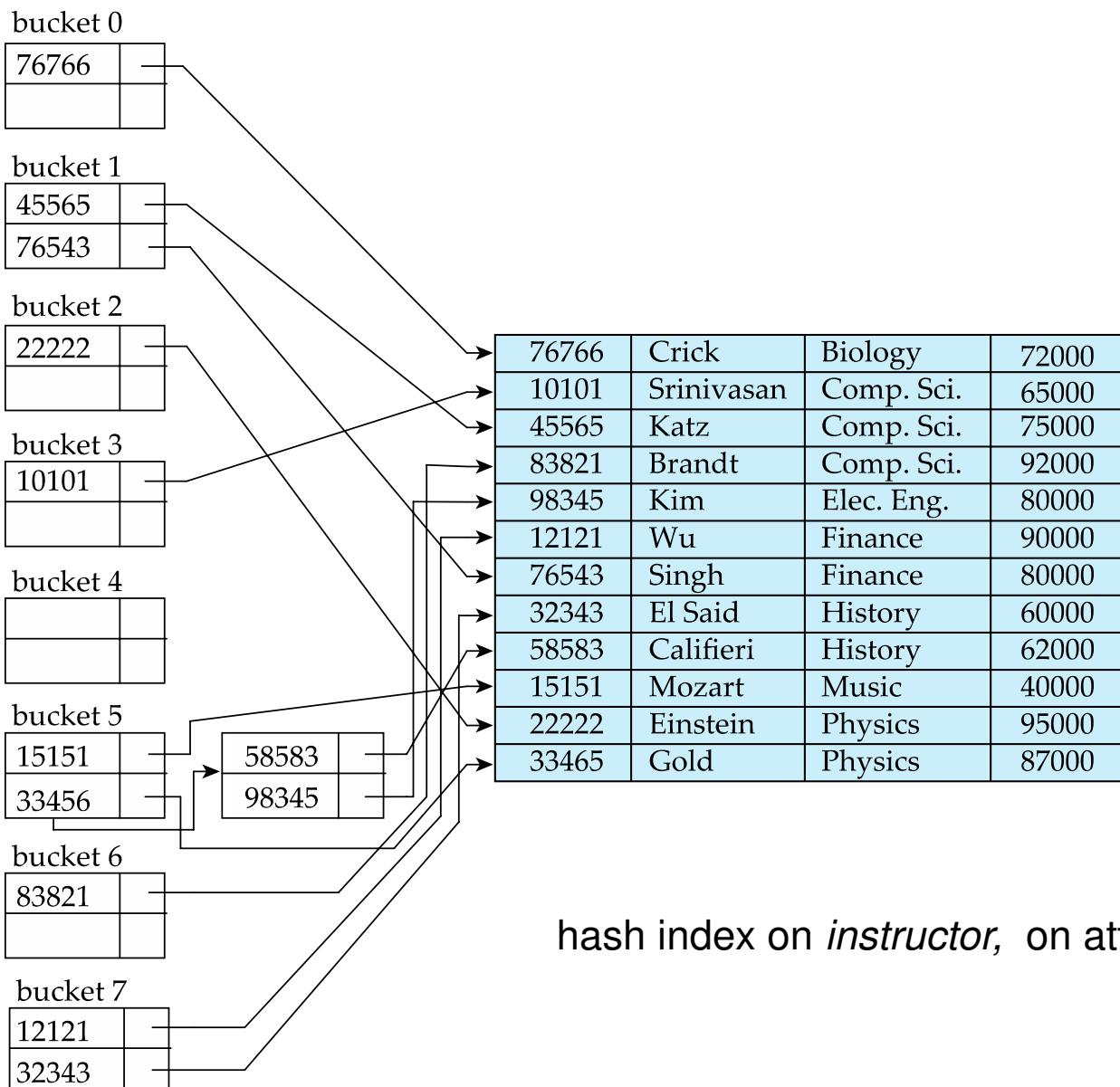


# Dynamic Hashing

- Periodic rehashing
  - If number of entries in a hash table becomes (say) 1.5 times size of hash table,
    - create new hash table of size (say) 2 times the size of the previous hash table
    - Rehash all entries to new table
- Linear Hashing
  - Do rehashing in an incremental manner
- Extendable Hashing
  - Tailored to disk based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in hash table, without doubling # of buckets



# Example of Hash Index





# Comparison of Ordered Indexing and Hashing

- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- Think about:
  - Cost of periodic re-organization
  - Relative frequency of insertions and deletions
  - Is it desirable to optimize average access time at the expense of worst-case access time?
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B<sup>+</sup>-trees



## End of Chapter 14



# Miscellaneous



# Multiple Key Access

- Until now, we have assumed that only one index on one attribute is used to process a query on a relation
- However, for certain queries,
  - It is advantageous to use multiple indices if they exist
  - Or to use an index built on a multi-attribute search key



# Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

```
select ID
```

```
from instructor
```

```
where dept_name = "Finance" and salary = 80000
```

- Possible strategies for processing query using indices on single attributes:
  1. Use index on *dept\_name* to find instructors with department name Finance; test *salary* = 80000
  2. Use index on *salary* to find instructors with a salary of \$80000; test *dept\_name* = "Finance".
  3. Use *dept\_name* index to find pointers to all records pertaining to the "Finance" department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.



# Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
  - E.g.,  $(dept\_name, salary)$
- Lexicographic ordering:  $(a_1, a_2) < (b_1, b_2)$  if either
  - $a_1 < b_1$ , or
  - $a_1 = b_1$  and  $a_2 < b_2$



# Indices on Multiple Attributes

Suppose we have an index on combined search-key  
 $(dept\_name, salary)$ .

- With the **where** clause
  - **where**  $dept\_name = \text{"Finance"} \text{ and } salary = 80000$   
the index on  $(dept\_name, salary)$  can be used to fetch only records that satisfy both conditions.
    - Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  - **where**  $dept\_name = \text{"Finance"} \text{ and } salary < 80000$
- But cannot efficiently handle
  - **where**  $dept\_name < \text{"Finance"} \text{ and } balance = 80000$ 
    - May fetch many records that satisfy the first but not the second condition



# Other Features

- **Covering indices**
  - Add extra attributes to index so (some) queries can avoid fetching the actual records
  - Store extra attributes only at leaf
    - Why?
- Particularly useful for secondary indices
  - Why?



# Creation of Indices

- Example
  - create index takes\_pk on takes (ID,course\_ID, year, semester, section)**
  - drop index takes\_pk**
- Most database systems allow specification of type of index, and clustering.
- Indices on primary key created automatically by all databases
  - Why?
- Some database also create indices on foreign key attributes
  - Why might such an index be useful for this query:
    - $takes \bowtie \sigma_{name='Shankar'} (student)$
- Indices can greatly speed up lookups, but impose cost on updates
  - Index tuning assistants/wizards supported on several databases to help choose indices, based on query and update workload



# Index Definition in SQL

- Create an index

```
create index <index-name> on <relation-name>  
          (<attribute-list>)
```

E.g.,: **create index** *b-index* **on** *branch*(*branch\_name*)

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

```
drop index <index-name>
```

- Most database systems allow specification of type of index, and clustering.



# Chapter 15: Query Processing

Database System Concepts, 7<sup>th</sup> Ed.

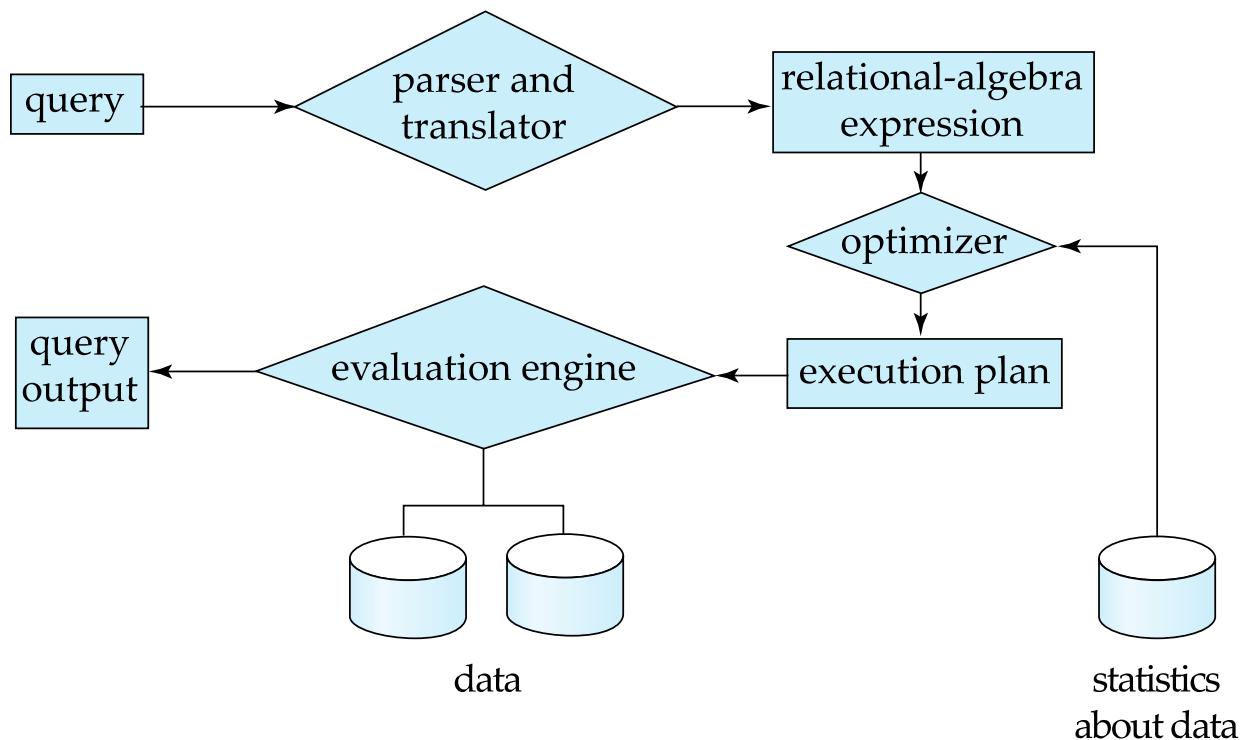
©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





# Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



# Basic Steps in Query Processing: Optimization

- A relational algebra expression may have many **equivalent expressions**
  - E.g.,  $\sigma_{\text{salary} < 75000}(\Pi_{\text{salary}}(\text{instructor}))$  is equivalent to  $\Pi_{\text{salary}}(\sigma_{\text{salary} < 75000}(\text{instructor}))$
- Each relational algebra operation can be evaluated using one of several **different algorithms**
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**. E.g.,:
  - Use an index on *salary* to find instructors with  $\text{salary} < 75000$ ,
  - Or perform complete relation scan and discard instructors with  $\text{salary} \geq 75000$



# Basic Steps: Optimization (Cont.)

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g.. number of tuples in each relation, size of tuples, etc.
- In this chapter we study
  - How to measure query costs
- In next chapter
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost



# Measures of Query Cost

- Disk cost can be estimated as:
  - Number of seeks \* average-seek-cost
  - Number of block transfers \* average-block-transfer-cost
  - Let:
  - $t_T$  – time to transfer one block
  - $t_S$  – time for one seek
  - Cost for b block transfers plus S seeks  
$$b * t_T + S * t_S$$
- $t_S$  and  $t_T$  depend on where data is stored; with 4 KB blocks:
  - High end magnetic disk:  $t_S = 4$  msec and  $t_T = 0.1$  msec
  - SSD:  $t_S = 20\text{-}90$  microsec and  $t_T = 2\text{-}10$  microsec for 4KB



# Measures of Query Cost (Cont.)

- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
- Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
  - But more optimistic estimates are used in practice



# Selection Operation

- **File scan** – Search algorithms that search a (single, dedicated) file into which the relation is stored
- Algorithm **A1 (linear search)**. Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate =  $b_r$  block transfers + 1 seek, where  $b_r$  denotes number of blocks containing records from relation  $r$
  - $t_s + b_r * t_T$
  - (*initially 1 seek required to find the first block; if blocks of the file are not stored contiguously, more seeks may be required, but we ignore this for simplicity*)
  - If selection is on a key attribute, can stop on finding the required record
    - Average cost =  $t_s + (b_r / 2) * t_T$



# Selections Using Indices

- **Index scan** – Search algorithms that use an index, selection condition must be on search-key of index.
- **A2 (primary/clustering index, equality on key)**. Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
  - Traverses *height of the index*  $h_i$  plus one I/O to fetch the record
  - Each operation requires one block transfer and one seek
- **A3 (primary index, equality on nonkey)** Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let  $b$  = number of blocks containing matching records
  - $Cost = (h_i + 1) * t_S + (h_i + b) * t_T$ 
    - *One seek for each level plus one seek for first block*
    - $(h_i + b)$  block transfers



# Selections Using Indices

- **A4 (secondary/non-clustering index, equality on key)**
  - Retrieve a single record if the search-key is a candidate key
    - $Cost = (h_i + 1) * (t_T + t_S)$
- **A4 (secondary index, equality on non-key)**
  - Retrieve multiple records if search-key is not a candidate key
    - each of  $n$  matching records may be on a different block
    - $Cost = (h_i + n) * (t_T + t_S)$ 
      - Can be very expensive!



# Selections Involving Comparisons

- Can implement selections of the form  $\sigma_{A \leq v}(r)$  or  $\sigma_{A \geq v}(r)$  by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5 (primary index, comparison).** (Relation is sorted on A)
  - For  $\sigma_{A \geq v}(r)$  use index to find first tuple  $\geq v$  and scan relation sequentially from there
  - For  $\sigma_{A \leq v}(r)$  just scan relation sequentially till first tuple  $> v$ ; do not use index
  - Cost =  $(h_i + 1) * t_S + (h_i + b) * t_T$
- **A6 (secondary index, comparison).**
  - For  $\sigma_{A \geq v}(r)$  use index to find first index entry  $\geq v$  and scan index sequentially from there, to find pointers to records.
  - For  $\sigma_{A \leq v}(r)$  just scan leaf pages of index finding pointers to records, till first entry  $> v$
  - In either case, retrieve records that are pointed to requires an I/O per record; Linear file scan may be cheaper!
  - Cost =  $(h_i + n) * (t_T + t_S)$



# Implementation of Complex Selections

- **Conjunction:**  $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$
- **A7 (conjunctive selection using one index)**
  - Choose a  $\theta_i$  and an algorithm from A2 to A6, that results in the least cost for  $\sigma_{\theta_i}(r)$ .
  - Test other conditions  $\theta_j$  on tuples, after fetching it into memory buffer.
- **A8 (conjunctive selection using composite index)**
  - Use appropriate composite (multiple-key) index if available.
- **A9 (conjunctive selection by intersection of identifiers)**
  - Requires indices with record pointers on the fields  $\theta_1, \theta_2 \dots$  involved
  - Use index for each  $\theta_i$ , and take intersection of all obtained sets of record pointers
  - Then fetch records from file
  - (If some conditions do not have appropriate indices, can apply test in memory)



# Algorithms for Complex Selections

- **Disjunction:**  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$ .
- **A10 (disjunctive selection by union of identifiers).**
  - Applicable if *all* conditions  $\theta_1, \theta_2, \dots$  have available indices.
    - Otherwise use linear scan.
  - Use index for each  $\theta_i$  and take union of all obtained sets of record pointers.
  - Then fetch records from file
- **Negation:**  $\sigma_{\neg\theta}(r)$ 
  - Use linear scan on file



# Join Operation

- Several different algorithms to implement joins
  - **Nested-loop join**
  - **Block nested-loop join**
  - **Indexed nested-loop join**
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following two relations: *student*, *takes*
  - Number of records:  $n_{student} = 5,000$     $n_{takes} = 10,000$
  - Number of blocks:  $b_{student} = 100$     $b_{takes} = 400$



# Nested-Loop Join

- To compute the theta join  $r \bowtie_{\theta} s$   
**for each tuple  $t_r$  in  $r$  do begin**  
    **for each tuple  $t_s$  in  $s$  do begin**  
        test pair  $(t_r, t_s)$  to see if they satisfy the join condition  $\theta$   
        if they do, add  $t_r \cdot t_s$  to the result.  
    **end**  
**end**
- $r$  is called the **outer relation** and  $s$  the **inner relation** of the join.
- Each record of  $r$  is matched with each record of  $s$
- - Requires no indices and can be used with any kind of join condition.
- - Expensive since it examines every pair of tuples in the two relations



# Nested-Loop Join (Cont.)

- In the worst case buffer can hold only one block of each relation, the estimated cost is
  - Requires seek for every block in  $r$
  - All blocks of  $s$  are scanned everytime, for each record in  $r$
  - Requires only one seek for whole  $s$  everytime, since  $s$  is read sequentially
  - $(n_r + b_r)$  seeks,  $(n_r * b_s + b_r)$  block transfers
- In best case, if buffer can hold both relations simultaneously:
  - Reduces cost to  $b_r + b_s$  block transfers and 2 seeks
- If buffer can hold any one relation ( $r$  or  $s$ ) :  $b_r + b_s$  block transfers,  $1 + (b_s$  or  $b_r)$  seeks
- Assuming worst case memory availability, cost estimate is
  - with  $student$  as outer relation:
    - $5000 * 400 + 100 = 2,000,100$  block transfers,  $5000 + 100 = 5100$  seeks
  - with  $takes$  as the outer relation
    - $10000 * 100 + 400 = 1,000,400$  block transfers and  $10,400$  seeks
  - If smaller relation ( $student$ ) fits entirely in memory, the cost estimate will be  $100 + 400 = 500$  block transfers.

$$\begin{aligned} n_{student} &= 5,000, & n_{takes} &= 10,000 \\ b_{student} &= 100, & b_{takes} &= 400 \end{aligned}$$



# Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block  $B_r$  of  $r$  do begin  
    for each block  $B_s$  of  $s$  do begin  
        for each tuple  $t_r$  in  $B_r$  do begin  
            for each tuple  $t_s$  in  $B_s$  do begin  
                Check if  $(t_r, t_s)$  satisfy the join condition  
                if they do, add  $t_r \cdot t_s$  to the result.  
            end  
        end  
    end  
end
```

- Worst case estimate where buffer can hold only one block of each relation:  $b_r * b_s + b_r$  block transfers,  $2 * b_r$  seeks
  - (Each block in) the inner relation  $s$  is read once for each *block* in the outer relation
- Best case:  $b_r + b_s$  block transfers, 2 seeks.



# Indexed Nested-Loop Join

- Index lookups can replace file scans if an index is available on the inner relation's join attribute
- For each tuple  $t_r$  in the outer relation  $r$ , use the index to look up tuples in  $s$  that satisfy the join condition with tuple  $t_r$ .
- Worst case: buffer has space for only one block of  $r$ , and, for each tuple in  $r$ , we perform an index lookup on  $s$ .
- Cost of the join:  $b_r(t_T + t_S) + n_r * c$ 
  - Where  $c$  is the cost of traversing index and fetching all matching  $s$  tuples for one tuple or  $r$
- If indices are available on join attributes of both  $r$  and  $s$ , use the relation with fewer tuples as the outer relation.



# Example of Nested-Loop Join Costs

- Compute  $student \bowtie takes$ , with  $student$  as the outer relation.

$$n_{student} = 5,000, n_{takes} = 10,000$$

$$b_{student} = 100, b_{takes} = 400$$

- Cost of block nested loops join
  - $400 * 100 + 100 = 40,100$  block transfers +  $2 * 100 = 200$  seeks
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested loops join
  - $100 + 5000 * 5 = 25,100$  block transfers and seeks.
  - CPU cost likely to be less than that for block nested loops join
- Computation for 5
  - Let  $takes$  have a primary B<sup>+</sup>-tree index on the attribute  $ID$ , which contains 20 entries in each index node.
  - Since  $takes$  has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data



# End of Chapter 15



# Chapter 16: Query Optimization

Database System Concepts, 7<sup>th</sup> Ed.

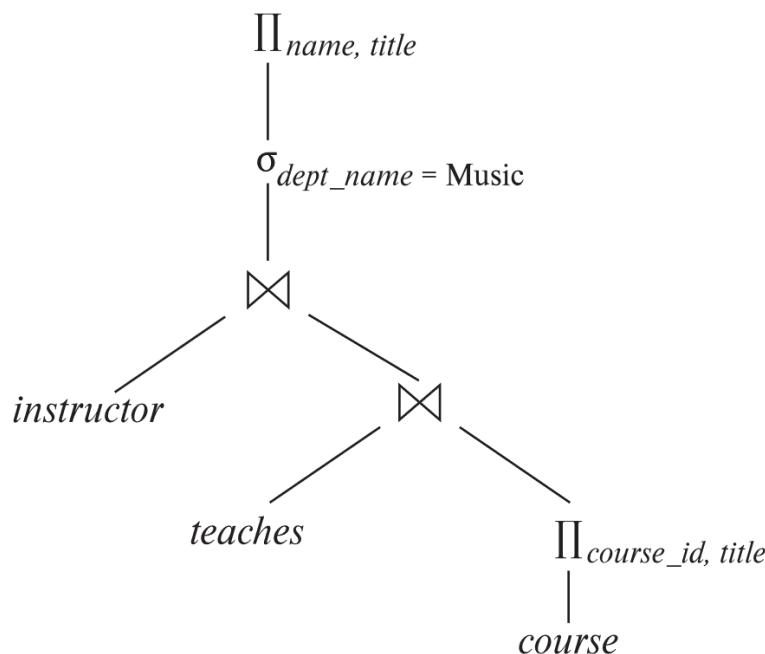
©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use

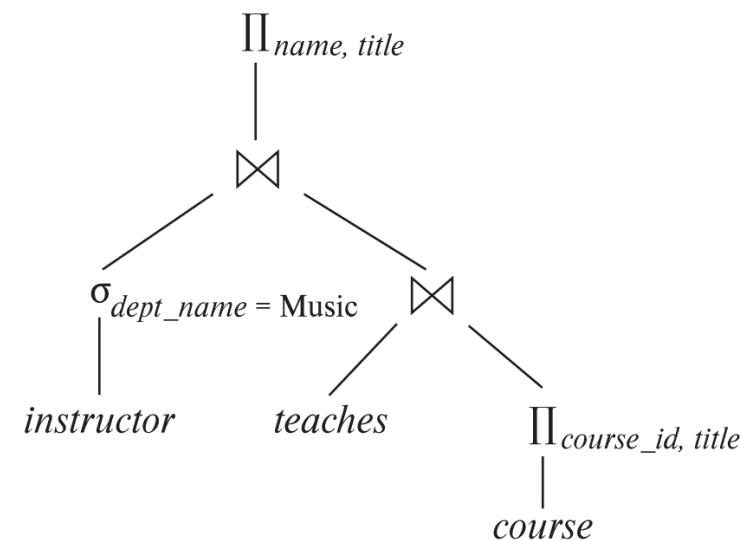


# Introduction

- Alternative ways of evaluating a given query
  - Equivalent expressions



(a) Initial expression tree

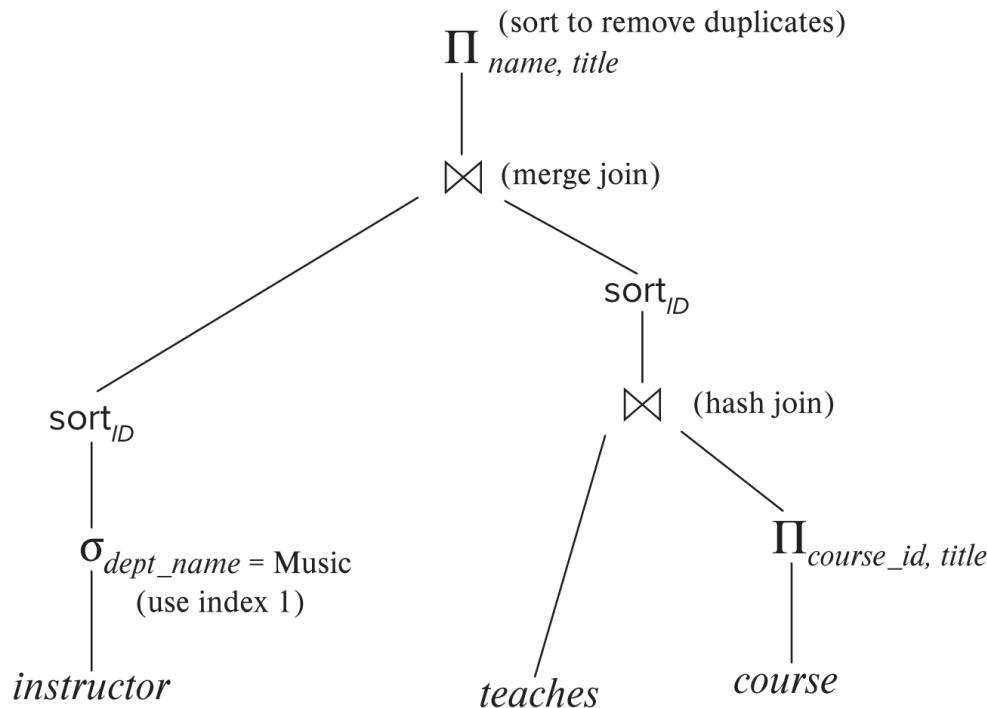


(b) Transformed expression tree



# Introduction (Cont.)

- Alternative ways of evaluating a given query
  - Different algorithms for each operation



- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



# Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g., seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions



# Viewing Query Evaluation Plans

- Most database support **explain <query>**
  - Displays plan chosen by query optimizer, along with cost estimates
  - Some syntax variations between databases
    - Oracle: **explain plan for <query>** followed by **select \* from table (dbms\_xplan.display)**
    - SQL Server: **set showplan\_text on**
- Some databases (e.g. PostgreSQL) support **explain analyse <query>**
  - Shows actual runtime statistics found by running the query, in addition to showing the plan
- Some databases (e.g. PostgreSQL) show cost as *f..l*
  - *f* is the cost of delivering first tuple and *l* is cost of delivering all results



# Generating Equivalent Expressions



# Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\Pi_{L_1}(\Pi_{L_2}(\dots(\Pi_{L_n}(E))\dots)) \equiv \Pi_{L_1}(E)$$

where  $L_1 \subseteq L_2 \dots \subseteq L_n$

4. Selections can be combined with Cartesian products and theta joins.

a.  $\sigma_{\theta}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta} E_2$

b.  $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$



# Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

6. (a) Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

- (b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where  $\theta_2$  involves attributes from only  $E_2$  and  $E_3$ .



# Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
  - (a) When all the attributes in  $\theta_0$  involve only the attributes of one of the expressions (say  $E_1$ ) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_{\theta} E_2) \equiv (\sigma_{\theta_0}(E_1)) \bowtie_{\theta} E_2$$

- (b) When  $\theta_1$  involves only the attributes of  $E_1$  and  $\theta_2$  involves only the attributes of  $E_2$ .

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$



# Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

(a) if  $\theta$  involves only attributes from  $L_1 \cup L_2$ :

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) \equiv \Pi_{L_1}(E_1) \bowtie_{\theta} \Pi_{L_2}(E_2)$$

(b) In general, consider a join  $E_1 \bowtie_{\theta} E_2$ .

- Let  $L_1$  and  $L_2$  be sets of attributes from  $E_1$  and  $E_2$ , respectively.
- Let  $L_3$  be attributes of  $E_1$  that are involved in join condition  $\theta$ , but are not in  $L_1 \cup L_2$ , and
- let  $L_4$  be attributes of  $E_2$  that are involved in join condition  $\theta$ , but are not in  $L_1 \cup L_2$ .

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) \equiv \Pi_{L_1 \cup L_2}(\Pi_{L_1 \cup L_3}(E_1) \bowtie_{\theta} \Pi_{L_2 \cup L_4}(E_2))$$

Similar equivalences hold for outerjoin operations:  $\bowtie$ ,  $\bowtie_l$ , and  $\bowtie_r$



# Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 \equiv E_2 \cup E_1$$

$$E_1 \cap E_2 \equiv E_2 \cap E_1$$

(set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over  $\cup$ ,  $\cap$  and  $-$ .

a.  $\sigma_{\theta}(E_1 \cup E_2) \equiv \sigma_{\theta}(E_1) \cup \sigma_{\theta}(E_2)$

b.  $\sigma_{\theta}(E_1 \cap E_2) \equiv \sigma_{\theta}(E_1) \cap \sigma_{\theta}(E_2)$

c.  $\sigma_{\theta}(E_1 - E_2) \equiv \sigma_{\theta}(E_1) - \sigma_{\theta}(E_2)$

d.  $\sigma_{\theta}(E_1 \cap E_2) \equiv \sigma_{\theta}(E_1) \cap E_2$

e.  $\sigma_{\theta}(E_1 - E_2) \equiv \sigma_{\theta}(E_1) - E_2$

preceding equivalence does not hold for  $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) \equiv (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$



# Equivalence Rules (Cont.)

13. Selection distributes over aggregation as below

$$\sigma_{\theta}({}_G\gamma_A(E)) \equiv {}_G\gamma_A(\sigma_{\theta}(E))$$

provided  $\theta$  only involves attributes in  $G$

14. a. Full outerjoin is commutative:

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

b. Left and right outerjoin are not commutative, but:

$$E_1 \bowtie L E_2 \equiv E_2 \bowtie R E_1$$

15. Selection distributes over left and right outerjoins as below, provided  $\theta_1$  only involves attributes of  $E_1$

a.  $\sigma_{\theta_1}(E_1 \bowtie_{\theta} E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} E_2$

b.  $\sigma_{\theta_1}(E_1 \bowtie_{\theta} E_2) \equiv E_2 \bowtie_{\theta} (\sigma_{\theta_1}(E_1))$

16. Outerjoins can be replaced by inner joins under some conditions

a.  $\sigma_{\theta_1}(E_1 \bowtie_{\theta} E_2) \equiv \sigma_{\theta_1}(E_1 \bowtie_{\theta} E_2)$

b.  $\sigma_{\theta_1}(E_1 \bowtie_{\theta} E_1) \equiv \sigma_{\theta_1}(E_1 \bowtie_{\theta} E_2)$

provided  $\theta_1$  is null rejecting on  $E_2$



# Equivalence Rules (Cont.)

Note that several equivalences that hold for joins do not hold for outerjoins

- $\sigma_{\text{year}=2017}(\text{instructor} \bowtie \text{teaches}) \not\equiv \sigma_{\text{year}=2017}(\text{instructor} \bowtie \text{teaches})$
- Outerjoins are not associative
  - $(r \bowtie s) \bowtie t \not\equiv r \bowtie (s \bowtie t)$
  - e.g. with  $r(A,B) = \{(1,1), (2,2)\}$ ,  $s(B,C) = \{(1,1), (2,2)\}$ ,  $t(A,C) = \{\}$

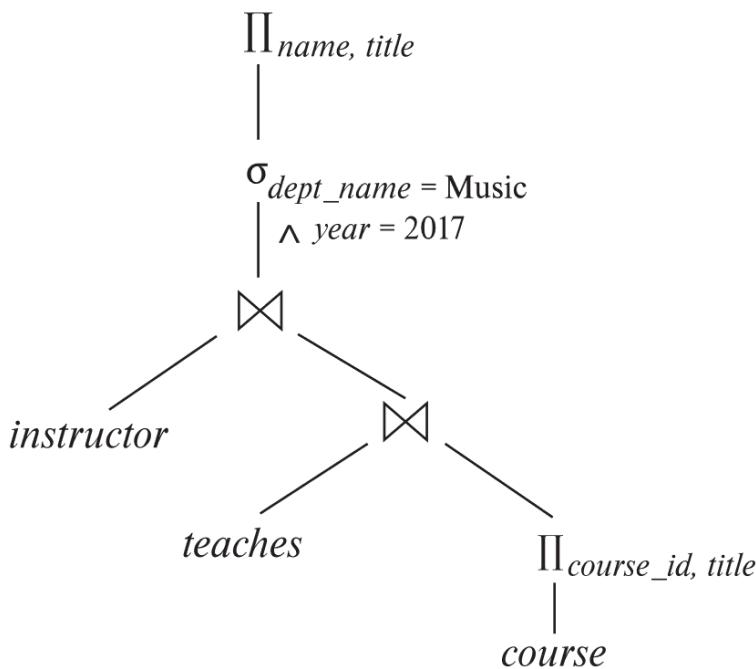


# Transformation Example: Pushing Selections

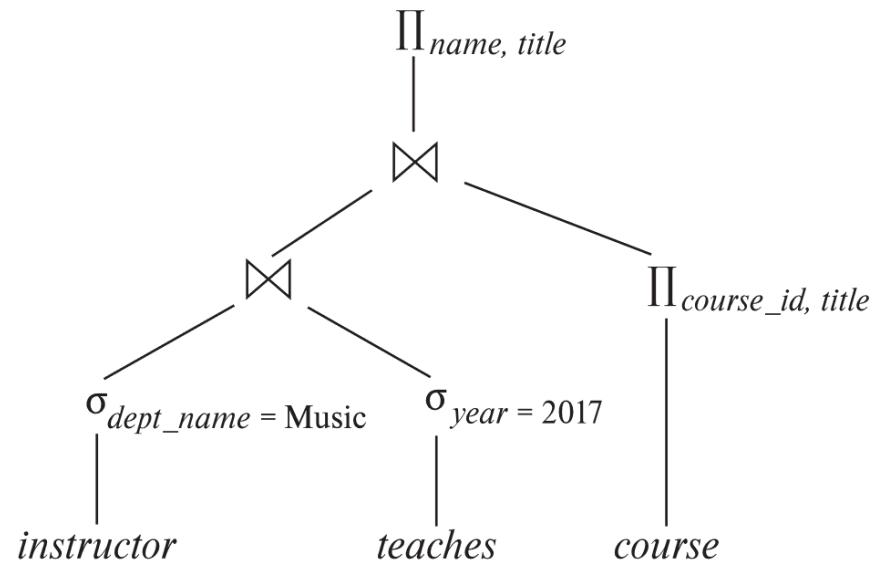
- Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
  - $\Pi_{name, title}(\sigma_{dept\_name = \text{Music}}(instructor \bowtie (teaches \bowtie \Pi_{course\_id, title}(course))))$
- Transformation using rule 7a.
  - $\Pi_{name, title}((\sigma_{dept\_name = \text{Music}}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id, title}(course)))$
- Performing the selection as early as possible reduces the size of the relation to be joined.



# Multiple Transformations (Cont.)



(a) Initial expression tree



(b) Tree after multiple transformations



# Join Ordering Example

- For all relations  $r_1$ ,  $r_2$ , and  $r_3$ ,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

(Join Associativity)  $\bowtie$

- If  $r_2 \bowtie r_3$  is quite large and  $r_1 \bowtie r_2$  is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

so that we compute and store a smaller temporary relation.



# Join Ordering Example (Cont.)

- Consider the expression

$$\begin{aligned}\Pi_{name, title}(\sigma_{dept\_name = \text{'Music'}}(instructor) \bowtie teaches \\ \bowtie \Pi_{course\_id, title}(course)))\end{aligned}$$

- Could compute  $teaches \bowtie \Pi_{course\_id, title}(course)$  first, and join result with  $\sigma_{dept\_name = \text{'Music'}}(instructor)$   
but the result of the first join is likely to be a large relation.
- Only a small fraction of the university's instructors are likely to be from the Music department
  - it is better to compute

$$\sigma_{dept\_name = \text{'Music'}}(instructor) \bowtie teaches$$

first.



# Choice of Evaluation Plans

1. Query optimizers use equivalence rules to generate expressions equivalent to the given expression
2. Cost of each operator computed as described in query evaluation chapter
  - Need statistics of input relations
    - E.g., number of tuples, sizes of tuples, number of distinct values for an attribute
3. Search all the plans and choose the best plan in a cost-based fashion
  - Or use heuristics to choose a plan



# Heuristic Optimization

- Cost-based optimization is expensive
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.



# Statistics for Cost Estimation



# Statistical Information for Cost Estimation

- $n_r$ : number of tuples in a relation  $r$ .
- $b_r$ : number of blocks containing tuples of  $r$ .
- $l_r$ : size of a tuple of  $r$ .
- $f_r$ : blocking factor of  $r$  — i.e., the number of tuples of  $r$  that fit into one block.
- $V(A, r)$ : number of distinct values that appear in  $r$  for attribute  $A$ ; same as the size of  $\Pi_A(r)$ .
- If tuples of  $r$  are stored together physically in a file, then:

$$b_r = \frac{n_r}{f_r}$$



# Selection Size Estimation

- $\sigma_{A=v}(r)$ 
  - $n_r / V(A,r)$  : number of records that will satisfy the selection, assuming uniform distribution of values
  - Equality condition on a key attribute: *size estimate* = 1
- $\sigma_{A \leq v}(r)$  (case of  $\sigma_{A \geq v}(r)$  is symmetric)
  - Let  $c$  denote the estimated number of tuples satisfying the condition.
  - If  $\min(A,r)$  and  $\max(A,r)$  are available in catalog
    - $c = 0$  if  $v < \min(A,r)$
    - $$c = n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$$
  - In absence of statistical information  $c$  is assumed to be  $n_r/2$ .



# Size Estimation of Complex Selections

- The **selectivity** of a condition  $\theta_i$  is the probability that a tuple in the relation  $r$  satisfies  $\theta_i$ .
  - If  $s_i$  is the number of satisfying tuples in  $r$ , the selectivity of  $\theta_i$  is given by  $s_i/n_r$ .
- Conjunction:**  $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$ . Assuming independence, estimate of tuples in the result is:

$$n_r \cdot \frac{s_1 \ s_2 \ \dots \ s_n}{n_r^n}$$

- Disjunction:**  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$ . Estimated number of tuples:

$$n_r \cdot 1 - (1 - \frac{s_1}{n_r}) \cdot (1 - \frac{s_2}{n_r}) \cdot \dots \cdot (1 - \frac{s_n}{n_r})$$

- Negation:**  $\sigma_{\neg \theta}(r)$ . Estimated number of tuples:

$$n_r - \text{size}(\sigma_\theta(r))$$



# Join Operation: Running Example

Running example:

$student \bowtie takes$

Catalog information for join examples:

$$n_{student} = 5,000, n_{takes} = 10,000$$
$$b_{student} = 100, b_{takes} = 400$$

- $n_{student} = 5,000$ .
- $f_{student} = 50$ , which implies that  
 $b_{student} = 5000/50 = 100$ .
- $n_{takes} = 10000$ .
- $f_{takes} = 25$ , which implies that  
 $b_{takes} = 10000/25 = 400$ .
- $V(ID, takes) = 2500$ , which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute  $ID$  in  $takes$  is a foreign key referencing  $student$ .
  - $V(ID, student) = 5000$  (*primary key!*)



# Estimation of the Size of Joins

- The Cartesian product  $r \times s$  contains  $n_r \cdot n_s$  tuples;
- If  $R \cap S = \emptyset$ , then  $r \bowtie s$  is the same as  $r \times s$ .
- If  $R \cap S$  is a key for  $R$ , then a tuple of  $s$  will join with at most one tuple from  $r$ 
  - therefore, the number of tuples in  $r \bowtie s$  is same as the number of tuples in  $s$ .
- If  $R \cap S$  in  $S$  is a foreign key in  $S$  referencing  $R$ , then the number of tuples in  $r \bowtie s$  is exactly the same as the number of tuples in  $s$ .
  - The case for  $R \cap S$  being a foreign key referencing  $S$  is symmetric.
- In the example query  $student \bowtie takes$ ,  $ID$  in  $takes$  is a foreign key referencing  $student$ 
  - hence, the result has exactly  $n_{takes}$  tuples, which is 10000

$$\begin{aligned}n_{student} &= 5,000, n_{takes} = 10,000 \\b_{student} &= 100, b_{takes} = 400\end{aligned}$$



# Estimation of the Size of Joins (Cont.)

- If  $R \cap S = \{A\}$  is not a key for  $R$  or  $S$ .  
The number of tuples in  $R \bowtie S$  is estimated to be the lower among:
$$\frac{n_r}{V(A,s)} \quad \text{and} \quad \frac{n_s}{V(A,r)}$$
- Compute the size estimates for *depositor*  $\bowtie$  *customer* without using information about foreign keys:
  - $V(ID, takes) = 2500$ , and  
 $V(ID, student) = 5000$
  - The two estimates are  $5000 * 10000/2500 = 20,000$  and  $5000 * 10000/5000 = 10000$
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.



# End of Chapter



# Miscellaneous



# Size Estimation for Other Operations

- Projection: estimated size of  $\Pi_A(r) = V(A,r)$
- Aggregation : estimated size of  $G\gamma_A(r) = V(G,r)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g.,  $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$  can be rewritten as  $\sigma_{\theta_1 \text{ or } \theta_2}(r)$
  - For operations on different relations:
    - estimated size of  $r \cup s = \text{size of } r + \text{size of } s.$
    - estimated size of  $r \cap s = \text{minimum size of } r \text{ and size of } s.$
    - estimated size of  $r - s = r.$
    - All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.



# Size Estimation (Cont.)

- Outer join:
  - Estimated size of  $r \bowtie s = \text{size of } r \bowtie s + \text{size of } r$ 
    - Case of right outer join is symmetric
  - Estimated size of  $r \bowtie s = \text{size of } r \bowtie s + \text{size of } r + \text{size of } s$



# Module 17: Transactions

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.



# Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer \$50 from account A to account B:
  1. **read(A)**
  2.  $A := A - 50$
  3. **write(A)**
  4. **read(B)**
  5.  $B := B + 50$
  6. **write(B)**
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions



# Example of Fund Transfer

- Transaction to transfer \$50 from account A to account B:
  1. **read(A)**
  2.  $A := A - 50$
  3. **write(A)**
  4. **read(B)**
  5.  $B := B + 50$
  6. **write(B)**
- **Atomicity requirement** - *whole or none*
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — *once completed, forever completed* - once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.



# Example of Fund Transfer (Cont.)

- **Consistency requirement**

- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency
- In last slide's example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints, e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand



# Example of Fund Transfer (Cont.)

- **Isolation requirement** — Although multiple transactions may execute concurrently, each transaction should occur in isolation *apparently*.
  - *Each transaction should be unaware of others executing concurrently*
- T1
  - 1. **read(A)**
  - 2.  $A := A - 50$
  - 3. **write(A)**
- T2
  - read(A), read(B), print(A+B)
- If between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum  $A + B$  will be less than it should be)
- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later



# ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that either  $T_j$  finished execution before  $T_i$  started, or  $T_j$  started execution after  $T_i$  finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

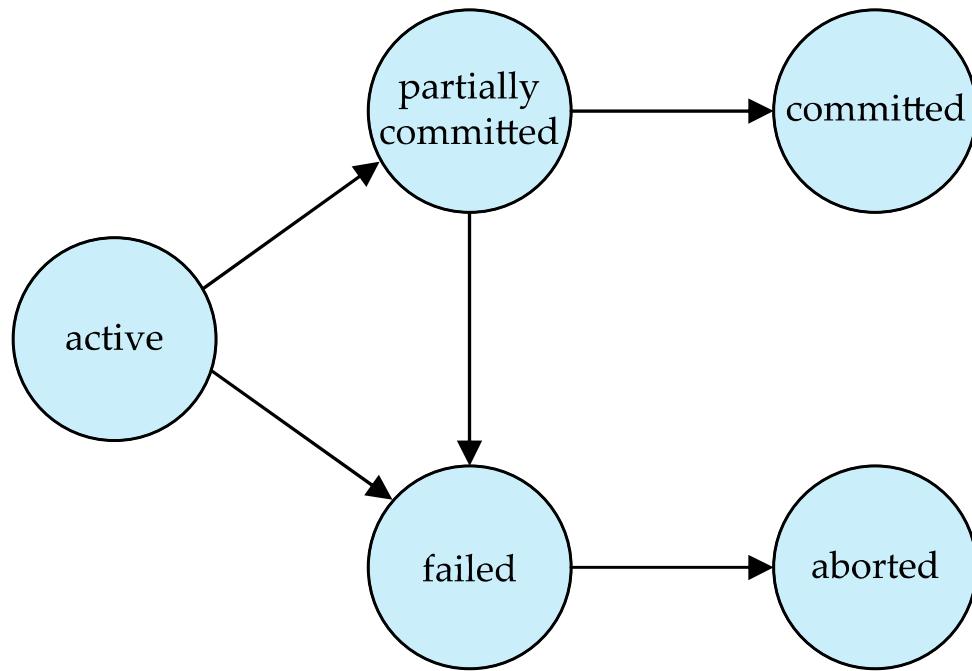


# Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.



# Transaction State (Cont.)





# Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.  
Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database



# Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
- Let  $T_1$  transfer \$50 from  $A$  to  $B$ , and  $T_2$  transfer 10% of the balance from  $A$  to  $B$ .
- A **serial** schedule in which  $T_1$  is followed by  $T_2$  :
  - A transaction that successfully completes its execution will have a commit instructions as the last statement
    - By default transaction assumed to execute commit instruction as its last step
  - A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

$T_1$	$T_2$
read ( $A$ ) $A := A - 50$ write ( $A$ ) read ( $B$ ) $B := B + 50$ write ( $B$ ) commit	read ( $A$ ) $temp := A * 0.1$ $A := A - temp$ write ( $A$ ) read ( $B$ ) $B := B + temp$ write ( $B$ ) commit
A schedule for a set of transactions must consist of all instructions of those transactions	Must preserve the order in which the instructions appear in each individual transaction.



# Schedule 1 and Schedule 2

$T_1$	$T_2$
read ( $A$ ) $A := A - 50$ write ( $A$ ) read ( $B$ ) $B := B + 50$ write ( $B$ ) commit	read ( $A$ ) $temp := A * 0.1$ $A := A - temp$ write ( $A$ ) read ( $B$ ) $B := B + temp$ write ( $B$ ) commit

$T_1$	$T_2$
read ( $A$ ) $A := A - 50$ write ( $A$ ) read ( $B$ ) $B := B + 50$ write ( $B$ ) commit	read ( $A$ ) $temp := A * 0.1$ $A := A - temp$ write ( $A$ ) read ( $B$ ) $B := B + temp$ write ( $B$ ) commit



# Schedule 3

- Let  $T_1$  and  $T_2$  be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

$T_1$	$T_2$
read ( $A$ ) $A := A - 50$ write ( $A$ )	read ( $A$ ) $temp := A * 0.1$ $A := A - temp$ write ( $A$ )
read ( $B$ ) $B := B + 50$ write ( $B$ ) commit	read ( $B$ ) $B := B + temp$ write ( $B$ ) commit

Check with initial values  
 $A = 1000, B = 2000$

- In Schedules 1, 2 and 3, the sum  $A + B$  is preserved.



# Schedule 4

- The following concurrent schedule does not preserve the value of  $(A + B)$ .

$T_1$	$T_2$
read ( $A$ ) $A := A - 50$	read ( $A$ ) $temp := A * 0.1$ $A := A - temp$ write ( $A$ ) read ( $B$ )
write ( $A$ ) read ( $B$ ) $B := B + 50$ write ( $B$ ) commit	$B := B + temp$ write ( $B$ ) commit



# Serializability

- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- A schedule may be serializable in two different ways:
  1. **Conflict serializability**
  2. **View serializability**



# Conflict Serializability

- Instructions  $I_i$  and  $I_j$  (of transactions  $T_i$  and  $T_j$  respectively) are in **conflict** iff:
  - there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ ,
  - and at least one of these instructions writes  $Q$ .
  

  1.  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  don't conflict.
  2.  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$ .  $I_i$  and  $I_j$  conflict.
  3.  $I_i = \text{write}(Q)$ ,  $I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  conflict
  4.  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$ .  $I_i$  and  $I_j$  conflict

  
- If a schedule  $S$  can be transformed into a serial schedule  $S'$  by a series of swaps of non-conflicting instructions, we say that  $S$  is **conflict serializable**
- $S$  and  $S'$  are **conflict equivalent** in the above case



# Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where  $T_2$  follows  $T_1$ , by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

$T_1$	$T_2$
read ( $A$ ) write ( $A$ )	read ( $A$ ) write ( $A$ )
read ( $B$ ) write ( $B$ )	read ( $B$ ) write ( $B$ )

Schedule 3

$T_1$	$T_2$
read ( $A$ ) write ( $A$ ) read ( $B$ ) write ( $B$ )	
	read ( $A$ ) write ( $A$ ) read ( $B$ ) write ( $B$ )

Schedule 6



# Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

$T_3$	$T_4$
read ( $Q$ )	
write ( $Q$ )	write ( $Q$ )

- We are unable to swap instructions in the above schedule to obtain either the serial schedule  $\langle T_3, T_4 \rangle$ , or the serial schedule  $\langle T_4, T_3 \rangle$ .



# View Serializability

- $S$  and  $S'$  (with the same set of transactions) are **view equivalent** if the following three conditions are met, for each data item  $Q$ ,
  1. If in schedule  $S$ , transaction  $T_i$  reads the initial value of  $Q$ , then in schedule  $S'$  also transaction  $T_i$  must read the initial value of  $Q$ .
  2. If in schedule  $S$  transaction  $T_i$  executes **read**( $Q$ ), and that value was produced by transaction  $T_j$  (if any), then in schedule  $S'$  also transaction  $T_i$  must read the value of  $Q$  that was produced by the same **write**( $Q$ ) operation of transaction  $T_j$ .
  3. The transaction (if any) that performs the final **write**( $Q$ ) operation in schedule  $S$  must also perform the final **write**( $Q$ ) operation in schedule  $S'$ .
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.



# View Serializability (Cont.)

- A schedule  $S$  is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable, but the reverse is not true.
  - Every view serializable schedule that is not conflict serializable has **blind writes**.
- Below is a schedule which is view-serializable but *not* conflict serializable.

$T_{27}$	$T_{28}$	$T_{29}$
read ( $Q$ )		
write ( $Q$ )	write ( $Q$ )	write ( $Q$ )

- What serial schedule is above equivalent to?



# Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule  $< T_1, T_5 >$ , yet is not conflict equivalent or view equivalent to it.

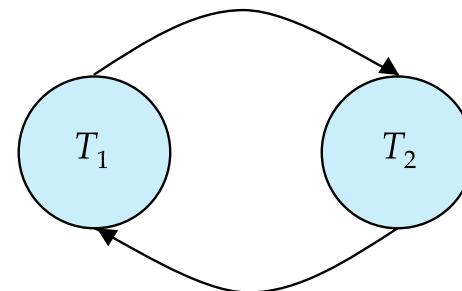
$T_1$	$T_5$
read ( $A$ ) $A := A - 50$ write ( $A$ )	read ( $B$ ) $B := B - 10$ write ( $B$ )
read ( $B$ ) $B := B + 50$ write ( $B$ )	read ( $A$ ) $A := A + 10$ write ( $A$ )

- Determining such equivalence requires analysis of operations **other than read and write**.



# Testing for Serializability

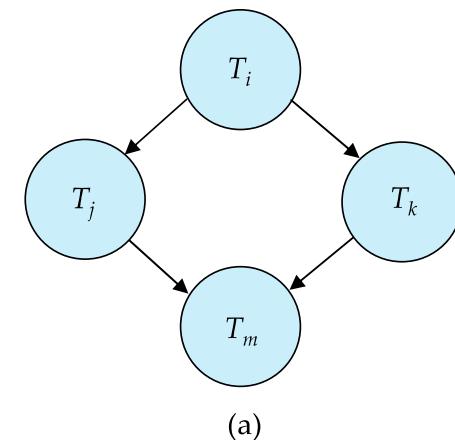
- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from  $T_i$  to  $T_j$  if the two transaction conflict, and  $T_i$  accessed the data item, on which the conflict arose, earlier.
- We may label the arc by the item that was accessed.
- Example of a precedence graph



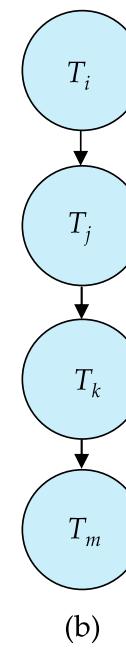


# Test for Conflict Serializability

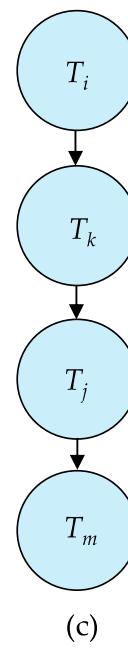
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order  $n^2$  time, where  $n$  is the number of vertices in the graph.
  - (Better algorithms take order  $n + e$  where  $e$  is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be  
 $T_i \rightarrow T_j \rightarrow T_k \rightarrow T_m$ 
    - Are there others?



(a)



(b)



(c)



# Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus, existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.



# Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction  $T_j$  reads a data item previously written by a transaction  $T_i$ , then the commit operation of  $T_i$  appears before the commit operation of  $T_j$ .
- The following schedule (Schedule 11) is not recoverable

$T_8$	$T_9$
read ( $A$ ) write ( $A$ )	
read ( $B$ )	read ( $A$ ) commit

- If  $T_8$  should abort,  $T_9$  would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.



# Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

$T_{10}$	$T_{11}$	$T_{12}$
read ( $A$ ) read ( $B$ ) write ( $A$ )		
	read ( $A$ ) write ( $A$ )	
abort		read ( $A$ )

If  $T_{10}$  fails,  $T_{11}$  and  $T_{12}$  must also be rolled back.

- Can lead to the undoing of a significant amount of work



# Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur;
  - For each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ .
- Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless



# Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- **Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless**
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.



# Transaction Definition in SQL

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- `connection.setAutoCommit(false);`
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
  - E.g. In SQL **set transaction isolation level serializable**
  - E.g. in JDBC -- `connection.setTransactionIsolation(`  
`Connection.TRANSACTION_SERIALIZABLE)`



# Implementation of Isolation Levels

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a “snapshot” of the database



# Transactions as SQL Statements

- E.g., Transaction 1:  
`select ID, name from instructor where salary > 90000`
- E.g., Transaction 2:  
`insert into instructor values ('11111', 'James', 'Marketing', 100000)`
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?
  - Instance of the **phantom phenomenon**
- Also consider T3 below, with Wu's salary = 90000  
`update instructor  
set salary = salary * 1.1  
where name = 'Wu'`
- Key idea: Detect “**predicate**” conflicts, and use some form of “**predicate locking**”



# End of Chapter 17



# Chapter 18 : Concurrency Control

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

- Lock-Based Protocols
- Multiple Granularity
- Handling Deadlocks
- Phantom Handling
- Timestamp-Based Protocols



# Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- **Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless**
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.



# Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. **exclusive (X) mode**. Data item can be both read as well as written.  
X-lock is requested using **lock-X** instruction.
  2. **shared (S) mode**. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.



# Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
- But if any transaction holds an exclusive lock on the item no other transaction may hold any lock on the item.



# Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

$T_2$ : **lock-S(A);**

**read (A);**

**unlock(A);**

**lock-S(B);**

**read (B);**

**unlock(B);**

**display(A+B)**

- Locking as above is not sufficient to guarantee serializability



# Schedule With Lock Grants

- Grants omitted in rest of chapter
  - Assume grant happens just before the next instruction following lock request
- This schedule is not serializable (why?)
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks, to enforce serializability

$T_1$	$T_2$	concurrency-control manager
lock-X( $B$ )		grant-X( $B, T_1$ )
read( $B$ ) $B := B - 50$	lock-S( $A$ )	grant-S( $A, T_2$ )
write( $B$ ) unlock( $B$ )	read( $A$ ) unlock( $A$ ) lock-S( $B$ )	grant-S( $B, T_2$ )
lock-X( $A$ )	read( $B$ ) unlock( $B$ ) display( $A + B$ )	grant-X( $A, T_1$ )
read( $A$ ) $A := A + 50$		
write( $A$ ) unlock( $A$ )		



# Deadlock

- Consider the partial schedule

$T_3$	$T_4$
lock-X( $B$ )	
read( $B$ )	
$B := B - 50$	
write( $B$ )	
	lock-S( $A$ )
	read( $A$ )
	lock-S( $B$ )
lock-X( $A$ )	

- Neither  $T_3$  nor  $T_4$  can make progress — executing **lock-S( $B$ )** causes  $T_4$  to wait for  $T_3$  to release its lock on  $B$ , while executing **lock-X( $A$ )** causes  $T_3$  to wait for  $T_4$  to release its lock on  $A$ .
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of  $T_3$  or  $T_4$  must be rolled back and its locks released.



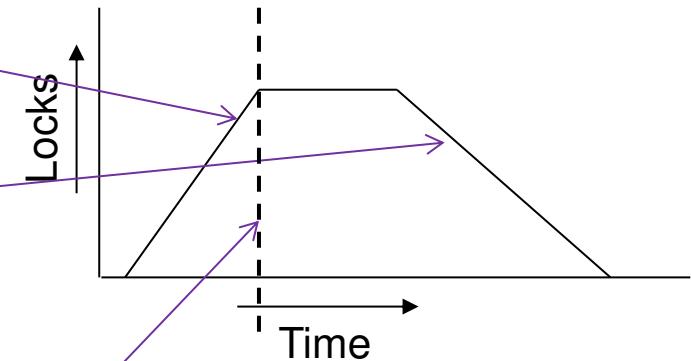
# Deadlock (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.



# The Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).





# Lock Conversions

- Two-phase locking protocol with lock conversions:
  - Growing Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can **convert** a lock-S to a lock-X (**upgrade**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability



# The Two-Phase Locking Protocol (Cont.)

- **Two-phase locking *does not* ensure freedom from deadlocks**
- Extensions to basic two-phase locking needed to ensure freedom from cascading roll-back
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking:** a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*
- Two-phase locking is not a necessary condition for serializability



# Multiple Granularity of Locking

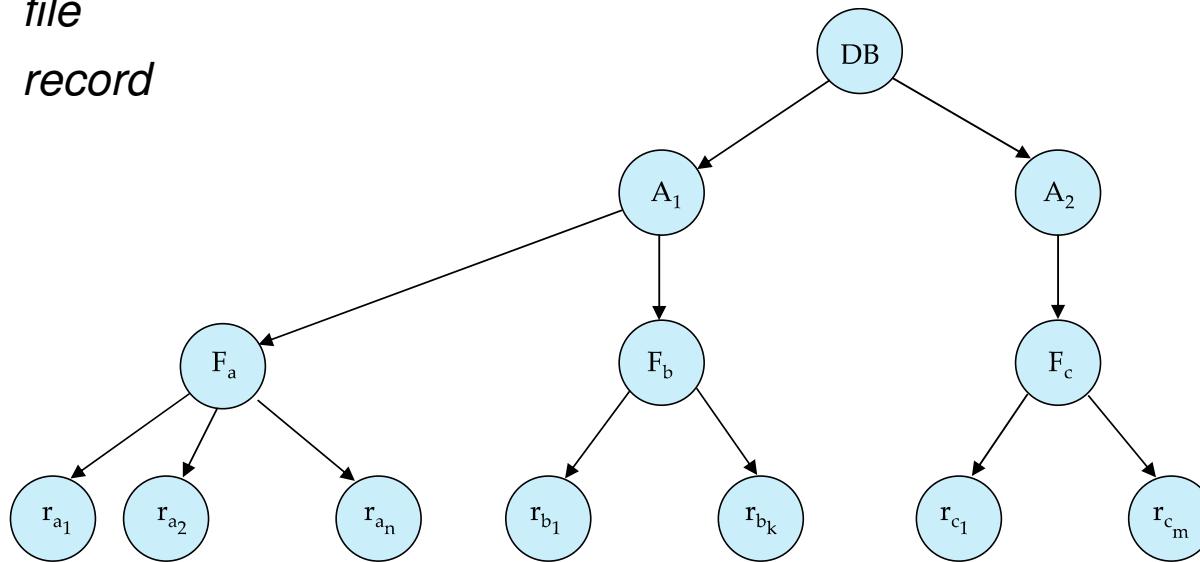
- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (see next slide)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode.
- Granularity of locking
  - **Fine granularity** (lower in tree): high concurrency, high locking overhead
  - **Coarse granularity** (higher in tree): low locking overhead, low concurrency



# Example of Granularity Hierarchy

The levels, starting from the coarsest (top) level are

- *database*
- *area (each area consists of files, no file in more than one area)*
- *file*
- *record*



**Lock granularity escalation:** in case there are too many locks at a particular level, switch to higher granularity lock



# Deadlock Handling

- System is **deadlocked** if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

$T_3$	$T_4$
$\text{lock-X}(B)$ $\text{read}(B)$ $B := B - 50$ $\text{write}(B)$  $\text{lock-X}(A)$	$\text{lock-S}(A)$ $\text{read}(A)$ $\text{lock-S}(B)$



# Deadlock Handling

- **Deadlock prevention** protocols ensure that the system will *never* enter into a deadlock state.
- Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (pre-declaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order



# Deadlock Prevention (Cont.)

- **Timeout-Based Schemes:**

- A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
- Ensures that deadlocks get resolved by timeout if they occur
- Simple to implement
- But may roll back transaction unnecessarily in absence of deadlock
  - Difficult to determine good value of the timeout interval.
- Starvation is also possible



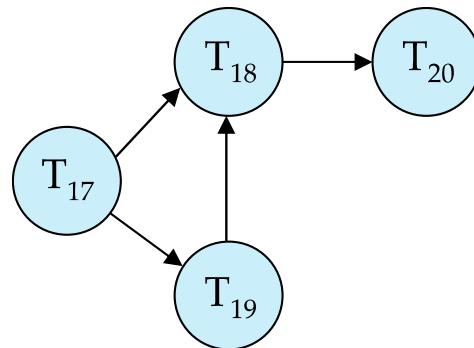
# More Deadlock Prevention Strategies

- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.
- In both schemes, a rolled back transaction is restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

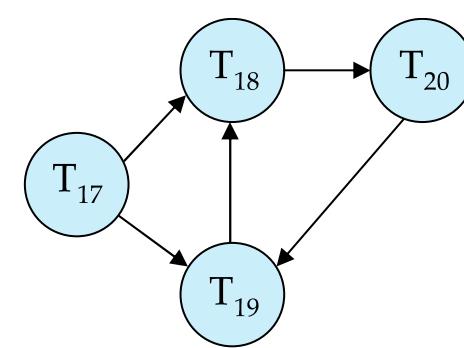


# Deadlock Detection

- **Wait-for graph**
  - *Vertices*: transactions
  - *Edge from  $T_i \rightarrow T_j$*  : if  $T_i$  is waiting for a lock held in conflicting mode by  $T_j$
- The system is in a deadlock state if and only if the wait-for graph has a **cycle**.
- Invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle



# Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to be rolled back (made a **victim**) to break deadlock cycle.
    - Select that transaction as victim that will incur minimum cost
  - Rollback -- determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim



# Phantom Phenomenon

- Example of **phantom phenomenon**.
  - A transaction T1 that performs **predicate read** (or scan) of a relation
    - **select count(\*)  
from instructor  
where dept\_name = 'Physics'**
  - and a transaction T2 that inserts a tuple while T1 is active but after predicate read
    - **insert into instructor values ('11111', 'Feynman', 'Physics', 94000)**  
(conceptually) conflict in spite of not accessing any tuple in common.
- Can also occur with updates
  - E.g. update Wu's department from Finance to Physics
- **Another Example:** T1 and T2 both find maximum instructor ID in parallel, and create new instructors with ID = maximum ID + 1
  - Both instructors get same ID, not possible in serializable schedule



# Handling Phantoms

- Locking rules for insert/delete operations
  - An X-lock must be obtained on an item before it is deleted
  - A transaction that inserts a new tuple into the database is automatically given an X-lock on the tuple
- Ensures that
  - reads/writes conflict with deletes
  - Inserted tuple is not accessible by other transactions until the transaction that inserts the tuple commits



# Index Locking To Prevent Phantoms

- **Index locking protocol** to prevent phantoms
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction  $T_i$  that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
  - A transaction  $T_i$  that inserts, updates or deletes a tuple  $t_i$  in a relation  $r$  must obtain X-locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur



# Timestamp Based Concurrency Control



# Timestamp-Based Protocols

- Each transaction  $T_i$  is issued a timestamp  $\text{TS}(T_i)$  when it enters the system.
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps strictly greater than earlier ones
  - Timestamp could be based on a logical counter
    - Real time may not be unique
    -
- Timestamp-based protocols manage concurrent execution such that  
**time-stamp order = serializability order**
- Several alternative protocols based on timestamps



# Timestamp-Ordering Protocol

The **timestamp ordering (TSO) protocol**

- For each data  $Q$  maintain two timestamp values:
  - **W-timestamp( $Q$ )** is the largest time-stamp of any transaction that executed **write( $Q$ )** successfully.
  - **R-timestamp( $Q$ )** is the largest time-stamp of any transaction that executed **read( $Q$ )** successfully.
- Imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in timestamp order
  - Out of order operations cause transaction rollback



# Timestamp-Based Protocols (Cont.)

- Suppose a transaction  $T_i$  issues a **read**( $Q$ )
  1. If  $\text{TS}(T_i) < \text{W-timestamp}(Q)$ , then  $T_i$  needs to read a value of  $Q$  that was already overwritten.
    - Hence, the **read** operation is rejected, and  $T_i$  is rolled back.
  2. If  $\text{TS}(T_i) \geq \text{W-timestamp}(Q)$ , then the **read** operation is executed, and  $\text{R-timestamp}(Q)$  is set to  
$$\max(\text{R-timestamp}(Q), \text{TS}(T_i)).$$



# Timestamp-Based Protocols (Cont.)

- Suppose that transaction  $T_i$  issues **write**( $Q$ ).
  1. If  $\text{TS}(T_i) < \text{R-timestamp}(Q)$ , then the value of  $Q$  that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the **write** operation is rejected, and  $T_i$  is rolled back.
  2. If  $\text{TS}(T_i) < \text{W-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $Q$ .
    - Hence, this **write** operation is rejected, and  $T_i$  is rolled back.
  3. Otherwise, the **write** operation is executed, and  $\text{W-timestamp}(Q)$  is set to  $\text{TS}(T_i)$ .



# Example of Schedule Under TSO

- Is this schedule valid under TSO?

Assume that initially:

$$R\text{-TS}(A) = W\text{-TS}(A) = 0$$

$$R\text{-TS}(B) = W\text{-TS}(B) = 0$$

Assume  $TS(T_{25}) = 25$  and

$$TS(T_{26}) = 26$$

$T_{25}$	$T_{26}$
read( $B$ )	read( $B$ ) $B := B - 50$ write( $B$ )
read( $A$ )	read( $A$ )
display( $A + B$ )	$A := A + 50$ write( $A$ ) display( $A + B$ )

- How about this one,  
where initially  
 $R\text{-TS}(Q)=W\text{-TS}(Q)=0$

$T_{27}$	$T_{28}$
read( $Q$ )	write( $Q$ )
write( $Q$ )	



# Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.



# Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When  $T_i$  attempts to write data item  $Q$ , if  $\text{TS}(T_i) < \text{W-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $\{Q\}$ .
  - Rather than rolling back  $T_i$  as the timestamp ordering protocol would have done, this **{write}** operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.



# End of Chapter 18



# Recoverability and Cascade Freedom

- Solution 1:
  - A transaction is structured such that its **writes are all performed at the end** of its processing
  - All **writes** of a transaction form an **atomic action**; no transaction may execute while a transaction is writing
  - A transaction that aborts is restarted with a **new timestamp**
- Solution 2:
  - Limited form of locking: wait for data to be committed before reading it
- Solution 3:
  - Use commit dependencies to ensure recoverability