Database System Concepts Overview

本学期考试覆盖内容

- **Ch1 Introduction**
- **Ch2 Introduction to the Relational Model**
- **Ch3 Introduction to SQL**
- **Ch4 Intermediate SQL**
- **Ch5 Advanced SQL**
- **Ch7 Entity-Relationship Model**
- **Ch8 Relational Database Design**
- **Ch11 Indexing and Hashing**
- **Ch14 Transactions**
- **Ch15 Concurrency Control**
- **Ch16 Recovery System**

Ch1: Key Points

- Data models and databases
- Database Design
- Database Engine
  - Storage Manager
  - Query Processing
  - Transaction Manager
- Database Architecture
- 考察基本概念

Data Models

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

```
type instructor = record
        ID : string;
        name : string;
        dept_name : string;
        salary : integer;
end;
```

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

DBMS Architecture

成绩单

学号：xxxxx　姓名：xxx　性别：x

离散数据	75
数据库系统	
……	……

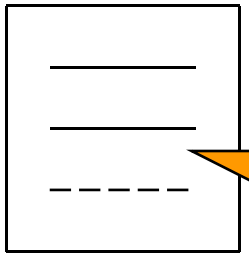
☞外模式（External Schema，也称子模式或用户模式）

——数据库用户使用的局部数据的逻辑结构和特征的描述

☞模式（Logical Schema，Schema，也称逻辑模式）

——数据库中全体数据的逻辑结构和特征（型）的描述

Cno	Cname	Cpno	Ccredit



☞内模式（Physical Schema，也称存储模式）

——数据物理结构和存储方式的描述

Data Models

- **Data Model**
  - A collection of tools for describing
    - Data
    - Data relationships
    - Data semantics
    - Data constraints
  - Relational model
  - Entity-Relationship data model (mainly for database design)
  - Object-based data models (Object-oriented and Object-relational)
  - ……

Ch2: Key Points

- Fundamental Relational-Algebra Operations
  - select(选择): σ
  - project(投影): π
  - union(并): ∪
  - set difference(集合差): –
  - Cartesian product(笛卡尔积): ×
  - rename: ρ
- Additional Relational-Algebra Operations
  - Set intersection(集合交)
  - Natural join(自然连接)
  - Division(除法)

Structure of Relational Database

- Keys(键/码)
  - Let K $\subseteq$ R
  - *K* is a superkey of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*
    - Example: {*ID*} and {ID,name} are both superkeys of *instructor.*
  - Superkey *K* is a candidate key if *K* is minimal

    Example: {*ID*} is a candidate key for *Instructor*
  - One of the candidate keys is selected to be the primary key.
    - which one?
  - Foreign key constraint: Value in one relation must appear in another
    - Referencing relation
    - Referenced relation
    - Example – *dept_name* in i*nstructor* is a foreign key from *instructor* referencing *department*

Relational-Algebra Operations

- **Select Operation**
  - Notation: $\sigma_p(r)$
  - p is called the selection predicate
  - Defined as:

    $$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

    Where p is a formula in propositional calculus consisting of terms connected by : $\wedge$ (and), $\vee$ (or), $\neg$ (not)
    Each term is one of:

    <attribute> *op* <attribute> or <constant>

    where *op* is one of: $=, \neq, >, \geq, <, \leq$
  - Example of selection:

    $$\sigma_{dept\_name=\text{“}Physics\text{”}}(instructor)$$

Relational-Algebra Operations

- **Project Operation**
  - Notation:

$$\Pi_{A_1, A_2, \ldots, A_k}(r)$$

    where A_1, A_2 are attribute names and r is a relation name.
  - The result is defined as the relation of k columns obtained by erasing the columns that are not listed
  - Duplicate rows removed from result, since relations are sets
  - Example: To eliminate the *dept_name* attribute of *instructor*

$$\Pi_{ID,\ name,\ salary}(instructor)$$

Relational-Algebra Operations

- **Union Operation**
  - Notation: $r \cup s$
  - Defined as:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

  - For $r \cup s$ to be valid.
    1. *r, s* must have the *same* arity (same number of attributes)
    2. The attribute domains must be compatible (example: 2nd column of *r* deals with the same type of values as does the 2nd column of *s*)
  - Example: to find all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or in both

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \wedge year=2009}(section)) \cup \Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \wedge year=2010}(section))$$

Relational-Algebra Operations

- **Set Difference Operation**
  - Notation r – s
  - Defined as:

$$r - s = \{t \mid t \in r \text{ and } t \notin s\}$$

  - Set differences must be taken between compatible relations.
    - r and s must have the same arity
    - attribute domains of r and s must be compatible
  - Example: to find all courses taught in the Fall 2009 semester, but not in the Spring 2010 semester

$$\prod_{\text{course\_id}} (\sigma_{\text{semester="Fall"} \wedge \text{year}=2009} (\text{section})) - \prod_{\text{course\_id}} (\sigma_{\text{semester="Spring"} \wedge \text{year}=2010} (\text{section}))$$

Relational-Algebra Operations

- **Cartesian-Product Operation**
  - Notation $r \times s$
  - Defined as:

$$r \times s = \{t\, q \mid t \in r \text{ and } q \in s\}$$

  - Assume that attributes of r(R) and s(S) are disjoint. (That is, $R \cap S = \varnothing$).
  - If attributes of *r(R)* and *s(S)* are not disjoint, then renaming must be used.

Relational-Algebra Operations

- **Rename Operation**
  - Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
  - Allows us to refer to a relation by more than one name.
  - Example:

$$\rho_x(E)$$

returns the expression E under the name X

  - If a relational-algebra expression E has arity n, then

$$\rho_{x(A_1, A_2, \ldots, A_n)}(E)$$

returns the result of expression E under the name X, and with the attributes renamed to $A_1, A_2, \ldots, A_n$.

Additional Operations

- Intersection Operation(交操作)
  - Notation: $r \cap s$
  - Assume:
    - r, s have the *same degree*
    - attributes of r and s are compatible
  - Defined as:

    $r \cap s = \{ t \mid t \in r \wedge t \in s \}$
  - Note: $r \cap s = r - (r - s)$

☞ Find the names of all customers who have a loan and an account at bank.

$$\Pi_{customer\_name}(borrower) \cap \Pi_{customer\_name}(depositor)$$

Additional Operations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations **match** (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the from clause

Additional Operations

- **Outer Join**
  - An extension of the join operation that avoids loss of information.
  - Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
  - Uses *null* values.

Additional Operations

- **Left Outer Join**
  - ***course* natural left outer join *prereq***

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

  - ***course* natural right outer join *prereq***

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Additional Operations

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

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

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

Additional Operations

- **Full Outer Join**
  - ***course* natural full outer join *prereq***

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Additional Operations

- **Division Operation(除)**
  - Suited to queries that include the phrase "for all".
  - Let *r* and *s* be relations on schemas R and S respectively where
    - $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
    - $S = (B_1, \ldots, B_n)$

    The result of r ÷ s is a relation on schema

    $$R \div S = (A_1, \ldots, A_m)$$
  - Defined as:

    $$r \div s = \{ t \mid t \in \pi_{R-S}(r) \wedge \forall u \in s \rightarrow tu \in r \}$$

除运算是同时从关系的水平方向和垂直方向进行运算。给定关系 R(X，Y)和 S(Y，Z)，X、Y、Z 为属性组。R÷S 应当满足元组在 X 上的分量值 x 的象集 Y_x 包含关系 S 在属性组 Y 上投影的集合。其形式定义为：

$$R \div S = \{t_n[X] \mid t_n \in R \wedge \pi_y(S) \subseteq Y_x\}$$

其中 Y_x 为 x 在 R 中的象集，$x = t_n[X]$，且 R÷S 的结果集的属性组为 X。

Additional Operations

Division Operation(除)

1：找出关系R和关系S中相同的属性，即Y属性。在关系S中对Y做投影（即将Y列取出）

2：被除关系R中与S中不相同的属性列是X，关系R在属性（X）上做取消重复值的投影为{X1，X2}；

3：求关系R中X属性对应的像集Y

4：判断包含关系 。

R：

X	Y
X1	Y1
X2	Y2
X2	Y3
X2	Y1

S：

Y	F
Y1	F1
Y2	F3

Y
Y1
Y2

R÷S其实就是判断关系R中X各个值的像集Y是否包含关系S中属性Y的所有值。对比即可发现：

X1的像集只有Y1，不能包含关系S中属性Y的所有值，所以排除掉X1；

而X2的像集包含了关系S中属性Y的所有值，所以R÷S的最终结果就是X2

R

X	Y
X1	Y1

图 3 X 的象集 Y

R

X	Y
X2	Y1
	Y2
	Y3

图 4 X2 值的象集 Y

R÷S

X
X2

Ch3: Key Points

- Basic Query Structure
- Null Values
- Aggregate Functions
- Nested Subqueries

Basic Query Structure

- Executing Orders of Query Clauses

④ SELECT [ALL | DISTINCT] [*] | [<*object_exp1*>[, <*object_exp1*>]...]

① FROM <*table_name1*>[, <*table_name2* >] ...

② [WHERE <*condition_exp*>]

③ [GROUP BY <*column_names*>]
[HAVING <*condition_exp*>]

⑤ [ORDER BY <*column_name1*> [ASC | DESC]
[<*column_name2*> [ASC | DESC] ...]] ;

NULL Values

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

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value
  - **avg**: average value
  - **min**: minimum value
  - **max**: maximum value
  - **sum**: sum of values
  - **count**: number of values

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A subquery is a select-from-where expression that is nested within another query.
- The nesting can be done in the following SQL query

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

  as follows:

  - A_i can be replaced be a subquery that generates a single value.
  - r_i can be replaced by any valid subquery
  - P can be replaced with an expression of the form:

    B <operation> (subquery)

    Where B is an attribute and <operation> to be defined later.

Nested Subqueries

- Subqueries in the Where Clause
  - A common use of subqueries is to perform tests:
    - For set membership
    - For set comparisons
    - For set cardinality.

Ch4: Key Points

- Views
- Integrity Constraints

Views

- A view is defined using the create view statement which has the form

  create view *v* as < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v*.
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

Integrity Constraints

- Integrity Constraints on a Single Relation
  - not null
  - primary key
  - unique
  - check (P), where P is a predicate

Ch5: Key Points

- Accessing SQL From a Programming Language
- Functions and Procedural Constructs
- Triggers

Functions and Procedural Constructs

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
        returns integer
        begin
        declare d_count  integer;
              select count (* ) into d_count
              from instructor
              where instructor.dept_name = dept_name
        return d_count;
end
```

- The function *dept*_count can be used to find the department names and budget of all departments with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

Triggers

- Triggering **event** can be insert, delete or update
- Triggers on update can be restricted to specific attributes
  - For example, after update of *takes* on *grade*
- Values of attributes **before and after** an update can be referenced
  - referencing **old** row as : for deletes and updates
  - referencing **new** row as : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ‘ ‘)
begin atomic
        set nrow.grade = null;
end;
```

Triggers

- When Not To Use Triggers
  - Triggers were used earlier for tasks such as
    - Maintaining summary data (e.g., total salary of each department)
    - Replicating databases by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica
  - There are better ways of doing these now:
    - Databases today provide built in materialized view facilities to maintain summary data
    - Databases provide built-in support for replication
  - Encapsulation facilities can be used instead of triggers in many cases
    - Define methods to update fields
    - Carry out actions as part of the update methods instead of through a trigger

Triggers

- When Not To Use Triggers
  - Risk of unintended execution of triggers, for example, when
    - Loading data from a backup copy
    - Replicating updates at a remote site
    - Trigger execution can be disabled before such actions.
  - Other risks with triggers:
    - Error leading to failure of critical transactions that set off the trigger
    - Cascading execution

Ch7: Key Points

- Design Process
- Modeling - E-R Diagram

Design Process

- Design Approaches
- Entity Relationship Model (covered in this chapter)
  - Models an enterprise as a collection of *entities* and *relationships*
    - Entity: a "thing" or "object" in the enterprise that is distinguishable from other objects
      - Described by a set of *attributes*
    - Relationship: an association among several entities
  - Represented diagrammatically by an *entity-relationship diagram:*
- Normalization Theory (Chapter 8)
  - Formalize what designs are bad, and test for them

Modeling

- The ER data mode was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database.
- The ER model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the ER model.
- The ER data model employs three basic concepts:
  - entity sets,
  - relationship sets,
  - attributes.
- The ER model also has an associated diagrammatic representation, the ER diagram, which can express the overall logical structure of a database graphically.

Modeling

- Entity Sets
  - An entity is an object that exists and is distinguishable from other objects.
    - Example: specific person, company, event, plant
  - An entity set is a set of entities of the same type that share the same properties.
    - Example: set of all persons, companies, trees, holidays
  - An entity is represented by a set of attributes; i.e., descriptive properties possessed by all members of an entity set.
    - Example:

      instructor = (*ID, name, street, city, salary*)
      course= (*course_id, title, credits*)
  - A subset of the attributes form a primary key of the entity set; i.e., uniquely identifiying each member of the set.

Modeling

- Relationship Sets
  - A relationship is an association among several entities

    Example:

    44553 (Peltier) — *advisor* — 22222 (Einstein)

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

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

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

      $(44553,22222) \in advisor$

Modeling

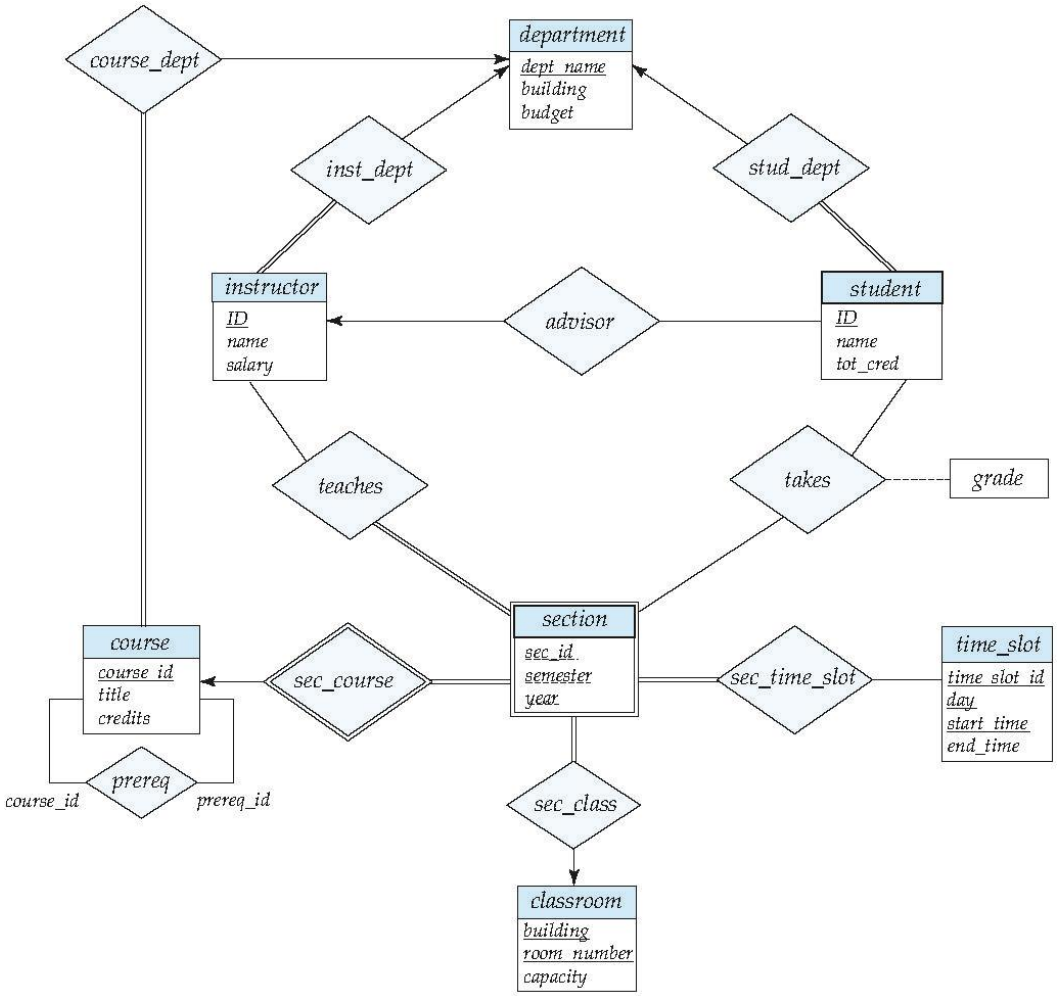
- Degree of a Relationship Set
  - binary relationship
    - involve two entity sets (or degree two).
    - most relationship sets in a database system are binary.
  - Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
    - Example: *students* work on research *projects* under the guidance of an *instructor*.
    - relationship *proj_guide* is a ternary relationship between *instructor, student,* and *project*

Modeling

- Mapping Cardinality Constraints
  - Express the number of entities to which another entity can be associated via a relationship set.
  - Most useful in describing binary relationship sets.
  - For a binary relationship set the mapping cardinality must be one of the following types:
    - One to one
    - One to many
    - Many to one
    - Many to many

E-R Diagram

- E-R Diagram for a University Enterprise



Ch8: Key Points

- Normal Form：1NF、2NF、3NF、BCNF
- Functional Dependency Theory

Features of Good Relational Design

- What About Smaller Schemas?
  - Suppose we had started with *inst_dept*. How would we know to split up (**decompose**) it into *instructor* and *department*?
  - Write a rule "if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key"
  - Denote as a **functional dependency**:

    $dept\_name \rightarrow building, budget$
  - In *inst_dept*, because *dept_name* is not a candidate key, the building and budget of a department may have to be repeated.
    - This indicates the need to decompose *inst_dept*
  - Not all decompositions are good. Suppose we decompose *employee(ID, name, street, city, salary)* into

    employee1 (*ID*, *name*)

    employee2 (*name*, *street, city, salary*)
  - The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

Atomic Domains and First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts.
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic
- 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
  - We assume all relations are in first normal form.

Atomic Domains and First Normal Form

- Goal — Devise a Theory for the Following.
  - 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 a set of relations $\{R_1, R_2, ..., R_n\}$ such that
    - each relation is in good form
    - the decomposition is a lossless-join decomposition
  - Our theory is based on:
    - functional dependencies
    - multivalued dependencies

Functional Dependency Theory

- Let R be a relation schema

$$\alpha \subseteq R \quad and \quad \beta \subseteq R$$

- 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 α, they also agree on the attributes β. That is,

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

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

1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does NOT hold, but $B \rightarrow A$ does hold.

Functional Dependency Theory

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

Functional Dependency Theory

- **Closure of a Set of Functional Dependencies**
- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F.
  - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of **all** functional dependencies logically implied by F is the **closure** of F.
- We denote the *closure* of F by $\mathbf{F^+}$.
- F^+ is a superset（超集） of F.

Functional Dependency Theory

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., $\beta \subseteq \alpha$)
  - α is a superkey for R

- Example schema not in BCNF:

  instr_dept (ID, name, salary, dept_name, building, budget)

- because dept_name→ building, budget
- holds on instr_dept, but dept_name is not a superkey

Functional Dependency Theory

Decomposing a Schema into BCNF

- Suppose we have a schema *R* and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

  We decompose *R* into:

  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$

- In our example,

  - α = *dept_name*
  - β = *building, budget*

  and *inst_dept* is replaced by

  - $(\alpha \cup \beta)$ = (*dept_name, building, budget*)
  - $(R - (\beta - \alpha))$ = (*ID, name, salary, dept_name*)

Functional Dependency Theory

- 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., $\beta \in \alpha$)
  - α 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).

Functional Dependency Theory

Closure of Attribute Sets

- Given a set of attributes α, define the *closure* of α under F (denoted by α^+) as the set of attributes that are functionally determined by α under F

- Algorithm to compute α^+, the closure of α under F

```
result := α;
while (changes to result) do
        for each β → γ in F do
            begin
                if β ⊆ result then  result := result ∪ γ
            end
```

Functional Dependency Theory

- **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 $(AG)^+ \supseteq R$
    2. Is any subset of AG a superkey?
       1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
       2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$

Functional Dependency Theory

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if α is a superkey, we compute α^+, and check if α^+ 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 α^+ by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful
- Computing closure of F
  - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Functional Dependency Theory

Canonical Cover（正则覆盖）

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B,\ B \rightarrow C, A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g.: on RHS: $\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow CD\}$ can be simplified to

      $$\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow D\}$$

    - E.g.: on LHS: $\{\mathrm{A} \rightarrow B,\ B \rightarrow C,\ AC \rightarrow D\}$ can be simplified to

      $$\{\mathrm{A} \rightarrow B,\ B \rightarrow C,\ A \rightarrow D\}$$

- Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

Functional Dependency Theory

- $R = (A, B, C)$
  $F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C\}$
- 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: $A \rightarrow B$
  $B \rightarrow C$

Functional Dependency Theory

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

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

- A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- 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

Functional Dependency Theory

- **Lossless-join Decomposition – Example**
  - $R = (A, B, C)$
    $F = \{A \rightarrow B, B \rightarrow C\}$
    - Can be decomposed in two different ways
  - $R_1 = (A, B)$, $R_2 = (B, C)$
    - Lossless-join decomposition:
      $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
    - Dependency preserving
  - $R_1 = (A, B)$, $R_2 = (A, C)$
    - Lossless-join decomposition:
      $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
    - Not dependency preserving
      (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

Functional Dependency Theory

- **Third Normal Form: Motivation**
  - 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 (with resultant problems; we will see examples later)
    - But functional dependencies can be checked on individual relations without computing a join.
    - There is always a lossless-join, dependency-preserving decomposition into 3NF.

Functional Dependency Theory

- **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.

Functional Dependency Theory

- **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
    - Redundancy due to use of 3NF
  - Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.

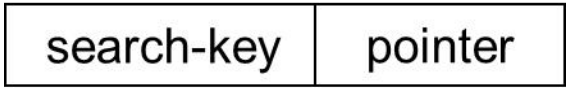
    Can specify FDs using assertions, but they are expensive to test, (and currently not supported by any of the widely used databases!)
  - Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Ch11: Key Points

- Basic Concepts
- B+-Tree Index Files

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to 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**: search keys are stored in sorted order
  - **Hash indices**: search keys are distributed uniformly across "buckets" using a "hash function".

Basic Concepts

- Index Evaluation Metrics
  - Access types supported efficiently. E.g.,
    - records with a specified value in the attribute
    - or records with an attribute value falling in a specified range of values.
  - Access time
  - Insertion time
  - Deletion time
  - Space overhead

Ordered Indices

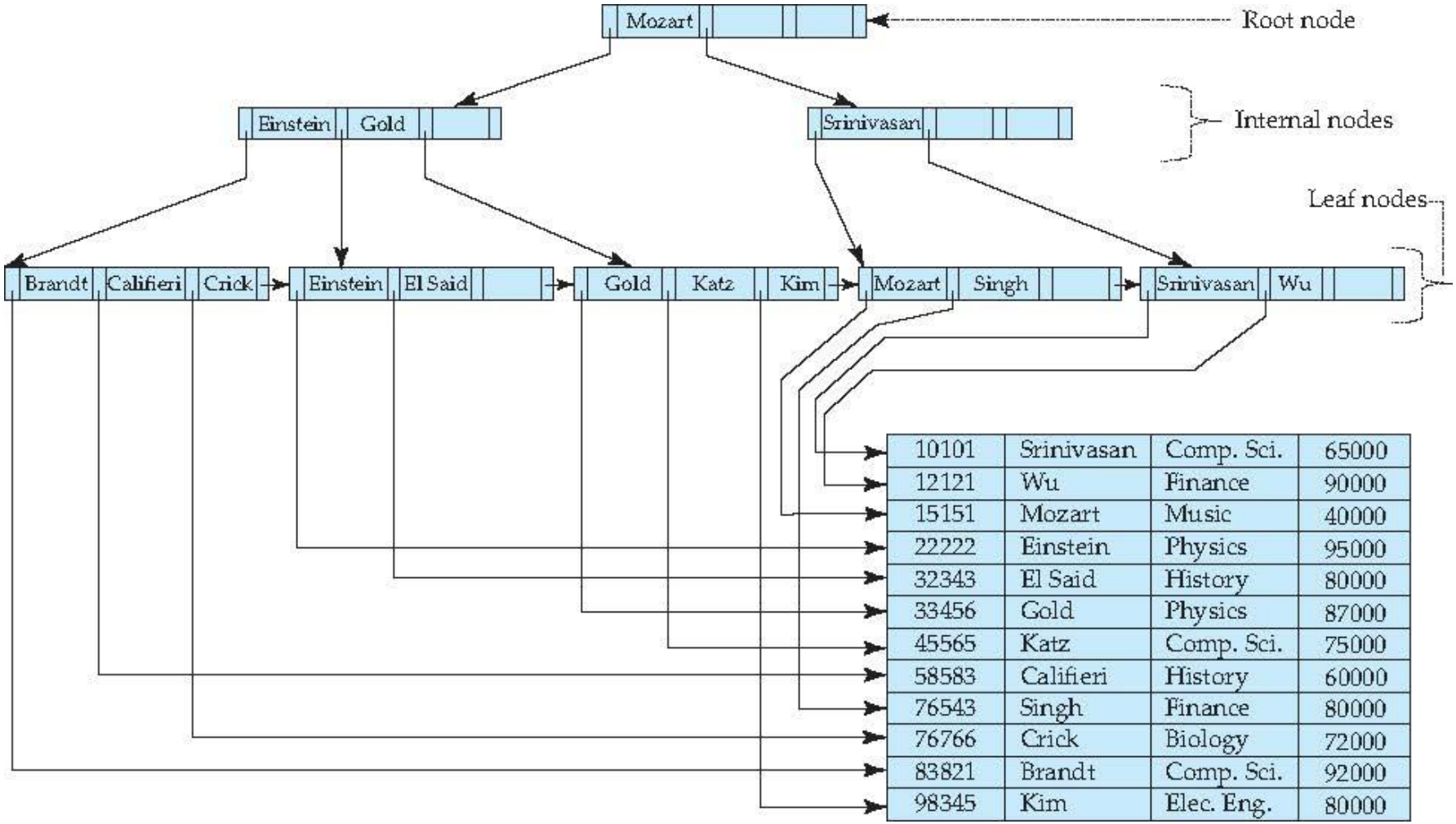
- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a 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**.
- **Index-sequential file**: ordered sequential file with a primary index.

B+-Tree Index Files

- B+-tree indices are an alternative to indexed-sequential 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^+-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^+-trees:
    - extra insertion and deletion overhead, space overhead.
  - Advantages of B^+-trees outweigh disadvantages
    - B^+-trees are used extensively

B+-Tree Index Files

- **Example of B$^+$-Tree**



B+-Tree Index Files

- A B+-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+-Tree Index Files

- A B+-tree is a rooted tree satisfying the following properties:
  - Typical node

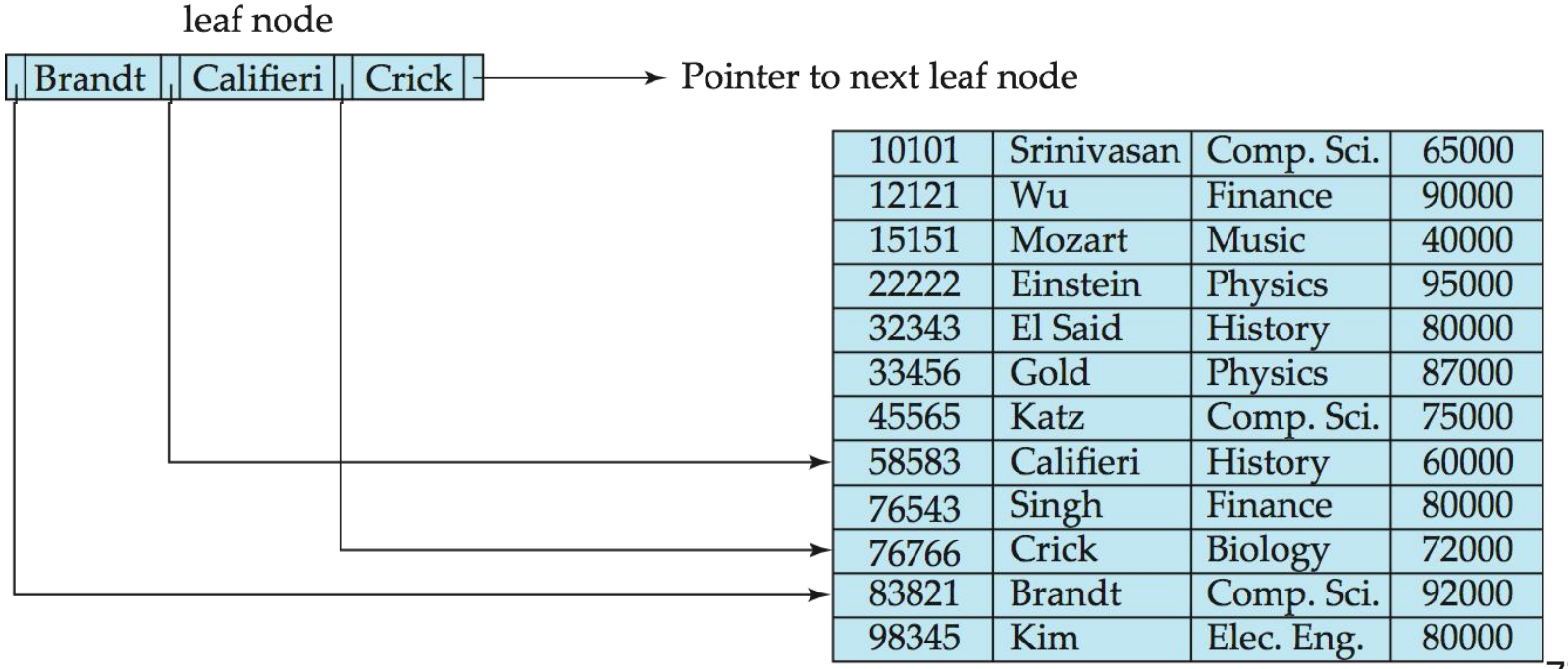
    - K_i are the search-key values
    - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
  - The search-keys in a node are ordered

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

(Initially assume no duplicate keys, address duplicates later)

B+-Tree Index Files

- Properties of a leaf node:
  - For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
  - 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
  - P_n points to next leaf node in search-key order



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

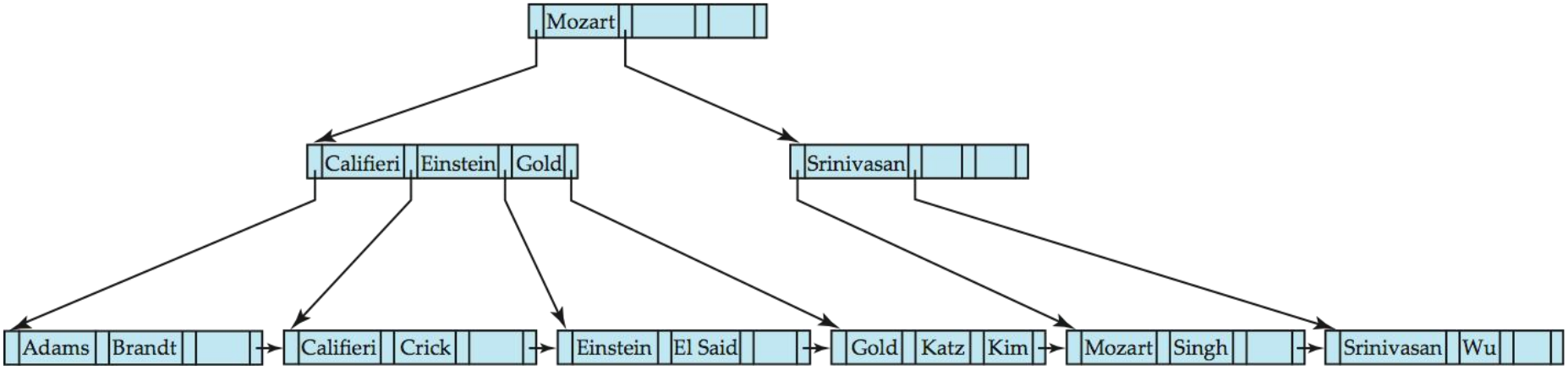
B+-Tree Index Files

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \le i \le n - 1$, 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}

B+-Tree Index Files

- Queries on B+-Trees
  - Find record with search-key value V.
    1. *C=root*
    2. While C is not a leaf node {
       1. Let i be least value s.t. $V \leq K_i$.
       2. If no such exists, set C = *last non-null pointer in C*
       3. Else { if (V= K_i) Set C = P_{i+1} else set $C = P_i$}

       }
    3. Let i be least value s.t. $K_i = V$
    4. If there is such a value i, follow pointer P_i to the desired record.
    5. Else no record with search-key value k exists.



B+-Tree Index Files

- Updates on B+-Trees: Insertion
  1. Find the leaf node in which the search-key value would appear
  2. If the search-key value is already present in the leaf node
     1. Add record to the file
     2. If necessary add a pointer to the bucket.
  3. If the search-key value is not present, then
     1. add the record to the main file (and create a bucket if necessary)
     2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
     3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

Ch4: Key Points

- Transaction Concept
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation

Transaction Concept

- A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items.

- Can be defined as
  - A set of SQL statements
  - Stored procedures
  - Initiated by high level programming languages (Java, C++ etc.)
- Delimited by *begin transaction* & *end transaction*

- Example:
  - Begin transaction
  - x = select salary from person where name = “Gao H…”
  - update person set salary = x * 10 where name = “Gao H…”
  - End transaction

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.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - Increased processor and disk utilization, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - 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
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - 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.
- 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

Serializable schedule – example

- Suppose x has $100, y has $200
- Consider two operations
  - x transfer $50 to y
  - Dividend
- For serial schedules
  - If transfer comes before dividend
    - X : 100 -> 50 -> 50.5
    - Y : 200 -> 250 -> 252.5
  - If dividend comes before transfer
    - X : 100 -> 101 -> 51
    - Y : 200 -> 202 -> 252
  - In both case, X + Y = 303

Serializable schedule – example

- What's going on here?
  - Interleaving can be very bad.
  - However, some interleaving does not cause problems.
  - How can we determine what kind of interleaving is "nice"?

Conflicting Instructions

- Let l_i and l_j be two Instructions of transactions T_i and T_j respectively. Instructions l_i and l_j conflict if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
  1. l_i = read(Q), l_j = read(Q). l_i and l_j don't conflict.
  2. l_i = read(Q), l_j = write(Q). They conflict.
  3. l_i = write(Q), l_j = read(Q). They conflict
  4. l_i = write(Q), l_j = write(Q). They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
  - If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflicting Serializability

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Recoverable Schedules

- **Recoverable schedule** — if a transaction Tj reads a data item previously written by a transaction Ti , then the commit operation of Ti **must** appear before the commit operation of Tj.
- The following schedule is not recoverable if T9 commits immediately after the read(A) operation.

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

- If T8 should abort, T9 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)	
		read (A)
abort		

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 — 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
- Example of a schedule that is NOT cascadeless

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

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are both:
  - Conflict serializable.
  - Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability *after* it has executed is a little too late!
  - Tests for serializability help us understand why a concurrency control protocol is correct
- **Goal** – to develop concurrency control protocols that will assure serializability.

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.
- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
  - E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Ch15: Key Points

- Lock-Based Protocols
- Multiversion Schemes

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 the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.

Lock-Based Protocols

- 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 on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

The Two-Phase Locking Protocol

- This protocol 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).

Deadlocks

- 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.

Deadlocks

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- When a deadlock occurs there is a possibility of cascading roll-backs.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking** -- a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter. Here, *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful write results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a read(Q) operation is issued, select an appropriate version of Q based on the timestamp of the transaction, and return the value of the selected version.
- reads never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Each version Q_k contains three data fields:
  - Content -- the value of version Q_k.
  - W-timestamp(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - R-timestamp(Q_k) -- largest timestamp of a transaction that successfully read version Q_k
- When a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

Multiversion Timestamp Ordering

- Suppose that transaction T_i issues a read(Q) or write(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).
  1. If transaction T_i issues a read(Q), then the value returned is the content of version Q_k.
  2. If transaction T_i issues a write(Q)
     1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
     2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. else a new version of Q is created.
- Observe that
  - Reads always succeed
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability

Multiversion Two-Phase Locking

- When an update transaction wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.
- When update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to ts-counter + 1
  - T_i increments ts-counter by 1
- Read-only transactions that start after T_i increments ts-counter will see the values updated by T_i.
- Read-only transactions that start before T_i increments the ts-counter will see the value before the updates by T_i.
- Only serializable schedules are produced.

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again

Ch16: Key Points

- Log-Based Recovery

Recovery Algorithm

- Logging (during normal operation):
  - $<T_i$ start> at transaction start
  - $<T_i, X_j, V_1, V_2>$ for each update, and
  - $<T_i$ commit> at transaction end
- Transaction rollback (during normal operation)
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form $<T_i, X_j, V_1, V_2>$
    - perform the undo by writing V_1 to X_j,
    - write a log record $<T_i, X_j, V_1>$
      - such log records are called compensation log records
  - Once the record $<T_i$ start> is found stop the scan and write the log record $<T_i$ abort>

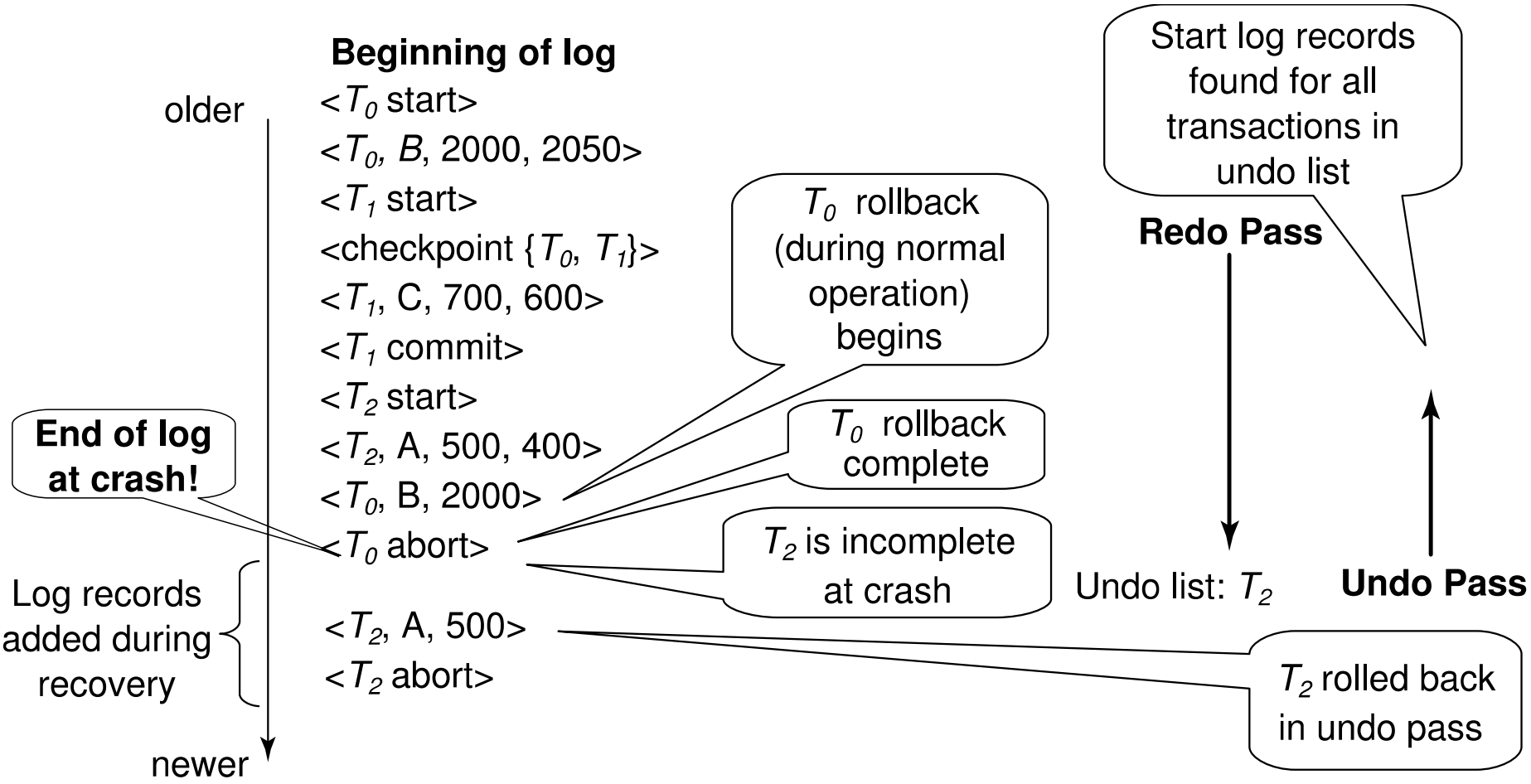
Recovery Algorithm

- Recovery from failure: Two phases
  - Redo phase: replay updates of all transactions, whether they committed, aborted, or are incomplete
  - Undo phase: undo all incomplete transactions
- Redo phase:
  1. Find last <checkpoint L> record, and set undo-list to L.
  2. Scan forward from above <checkpoint L> record
     1. Whenever a record $<T_i, X_j, V_1, V_2>$ or $<T_i, X_j, V_2>$ is found, redo it by writing V_2 to X_j
     2. Whenever a log record $<T_i$ start> is found, add T_i to undo-list
     3. Whenever a log record $<T_i$ commit> *or* $<T_i$ abort> is found, remove T_i from undo-list

Recovery Algorithm

- Undo phase:
  1. Scan log backwards from end
     1. Whenever a log record $<T_i, X_j, V_1, V_2>$ is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record $<T_i, X_j, V_1>$
     2. Whenever a log record $<T_i$ start> is found where T_i is in undo-list,
        1. Write a log record $<T_i$ abort>
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e. $<T_i$ start> has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery



试题说明

试题说明

- 全英文试卷
- 题型：
  - 单选题10个（20分）
  - 关系代数计算题（关系代数表达式和SQL）（30分）
  - 事务计算题（事务、调度）（15分）
  - 关系数据库理论题（20分）
  - 数据库设计（15分）

Question?