CS3402 Database system

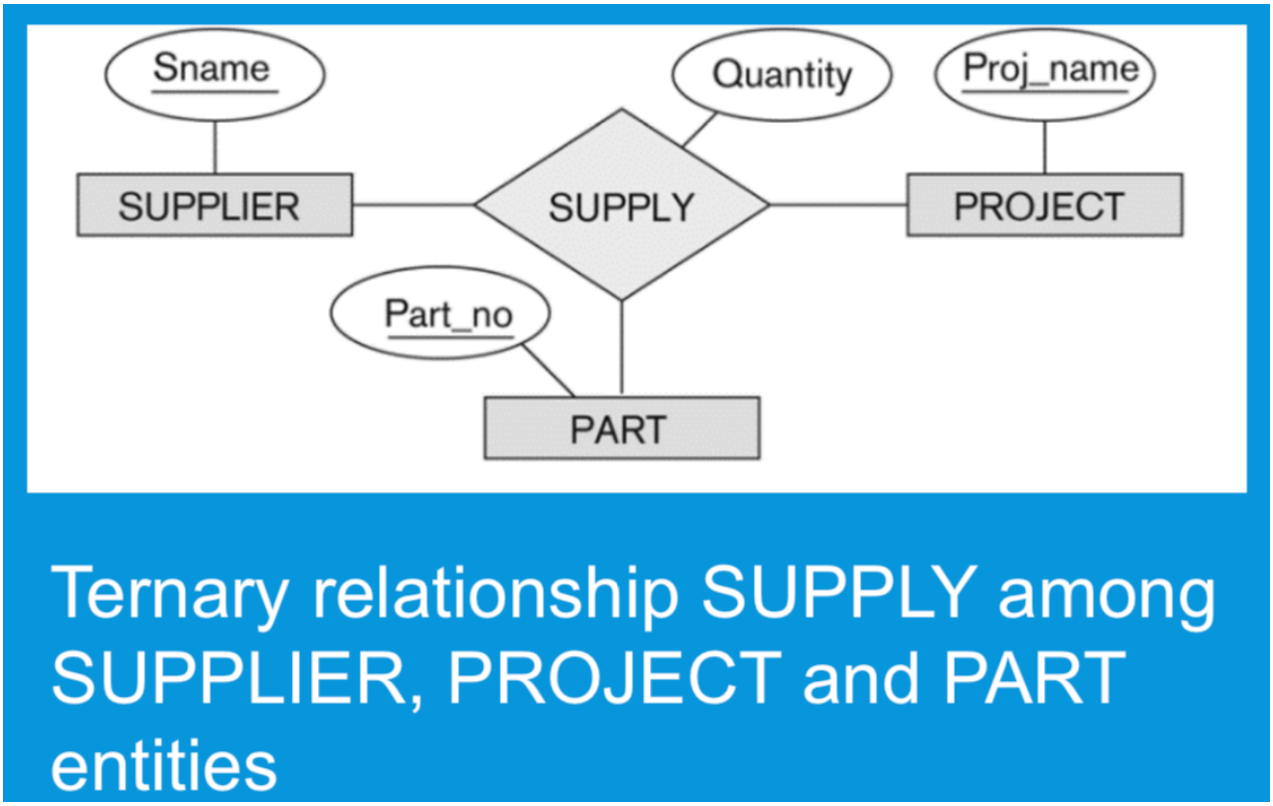
Information about the Final Exam

- ERModel and Relational Model and normalization is not important
- Time is tight
- Part A: short(40%): 2 questions
  - 1(30%).similar to midterm(sql & algebra)
  - 2(10%). True and False and reason
- Part B: 3/4 long(60%)
  - **indexing(hashing+B+ tree(deletion is not that important but can exist a simple question) => similar to tutorial&lecture)**
  - **query optimization**
  - **transaction(similar to the tutorial)**
  - concurrency control(understand the question)
- Plan:
  - Go through the lecture note
  - Do the important tutorial (Lab)
  - practice SQL on W3school

Lecture 01: Entity-Relationship (ER) Model

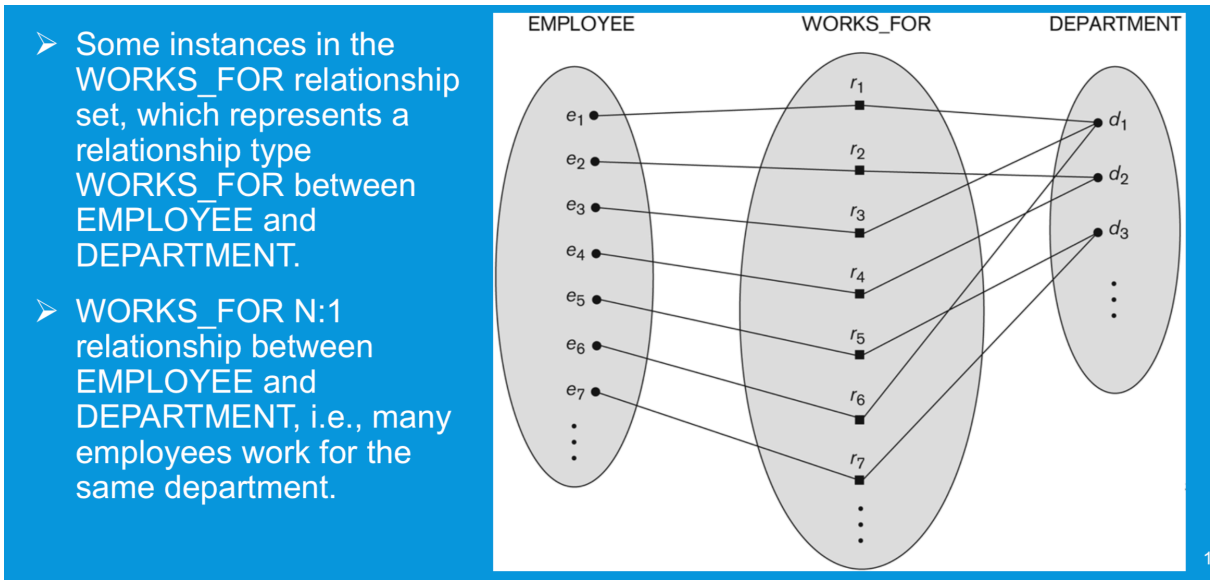
- Definition of **ER model**: describes *interrelated things of interest* in a specific domain of knowledge. Becomes an abstract data model, that defines a data or information structure which can be implemented in a database. Composed of:
  - Entity types(classify the things of interest)
  - Specifies relationships: exist between entities, instances of those entity types
- Entity, Entity type and Entity Set
  - **Entity**: a thing capable of an independent existence that **can be uniquely identified** and exists either **physically or logically**. (represented as rectangle)(object)
  - **Entity type**: collection of entities that have the **same kinds of attributes**(class)
  - **Entity set**: set of entities of the same type(a set of objects)
- Relationship, Relationship types and Relationship Set
  - **Relationship** (ties): captures how entities are related to one another(can be thought of as *verbs, linking two or more nouns(entities)*).
    - For example, a *work_for* relationship between an *employee* and a *department*.(represented as a diamond)
  - **Relationship type**(same kinds of ties) : Defines a relationship among entities of certain entity types
    - **Degree** of a relationship type: **number of participating entity types**
      - binary(ternary) relation type: involving two(three) entity types
  - **Relationship set** (a bunch of ties) : collection of relationships all belonging to one relationship type represented in the database

- Attribute
  - Both entities and relationships can have



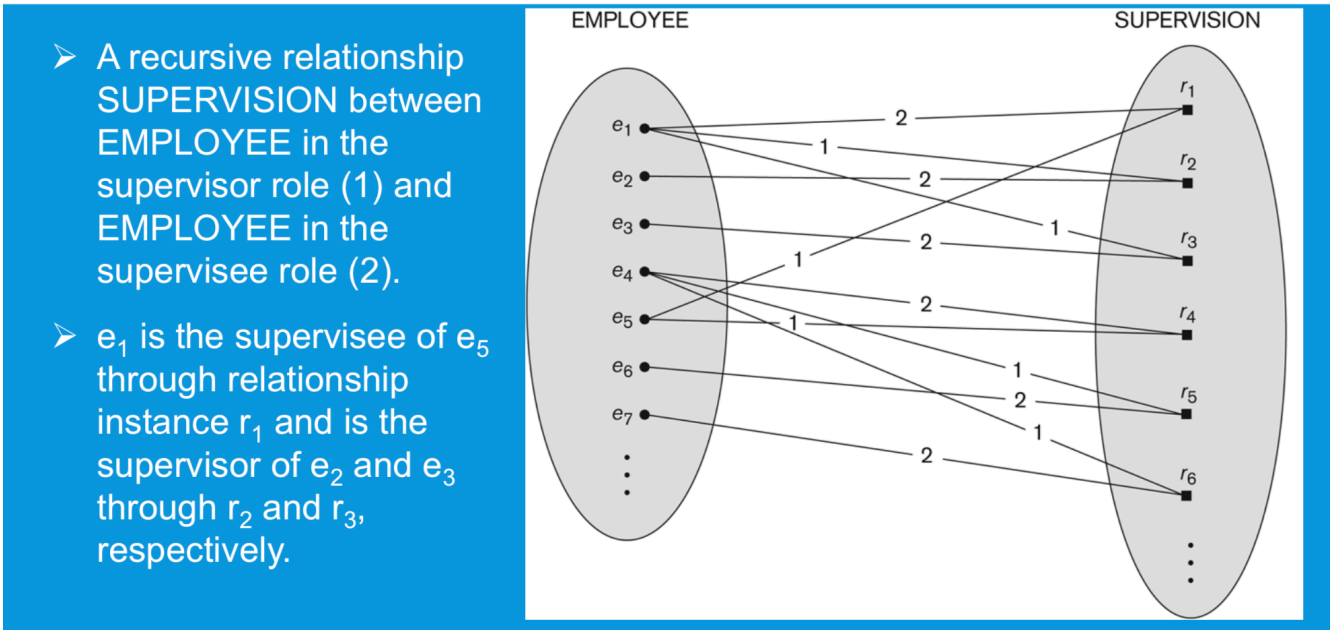
  - **Key attribute** : A set of attributes (one or more attributes) that uniquely identify an entity(super keys) => There might be multiple key attribute that can determine a tuple => different with primary key
  - Types of attribute
    - **Simple attribute** : Has a single atomic value that does not contain any smaller meaningful components
    - **Composite attributes** : composed of several components(simple attributes).
    - **Multi-valued attribute** : Has multiple values.
      - For example, color of a product (i.e., red and white) and major of a student (i.e., computer science and mathematics).(stored as a collection)
      - In general, composite and multi-valued attributes may be nested to any number of levels although this is rare. (Muti-valued consists of multi-valued)
    - **Derived attribute** : An attribute whose value is calculated from other attributes(need not be physically stored within the database)
- Value sets(domains) of attributes
  - Each **simple attribute** is associated with a **value se**t (or domain)
  - The value set specifies the set of values associated with an attribute
  - Value sets are similar to data types in most programming languages(**normally do not use float in database**, unsteadily, use double)
  - Char(20), int ....
- Constrains on relationships
  - **Participation constraint**(focus on pariticipation of each entity): Indicate the minimum number of relationship instances that an entity can participate in(must be larger than what)
    - **Total participation** requires that each entity is involved in the relationship.

        - In other words, an entity must exist related to another entity(represented by **double lines** in ER model) => **Key word: A must have at least one B**
    - **Partial participation** means that not all entities are involved in the relationship. (represented by single lines in ER model)
- **Cardinality constraint**(focus on the map between A and B) : Indicates the maximum number of relationship instances that an entity can participate in(cannot be larger than what, defines the upper bound)
    - A **1:1 or one-to-one relationship** from entity type S to entity type T is one in which an entity from S is related to **at most one** entity from T and vice versa.
    - An **N:1 or many-to-one relationship** from entity type S to entity type T is one in which an entity from T can be related to two or more entities from S.

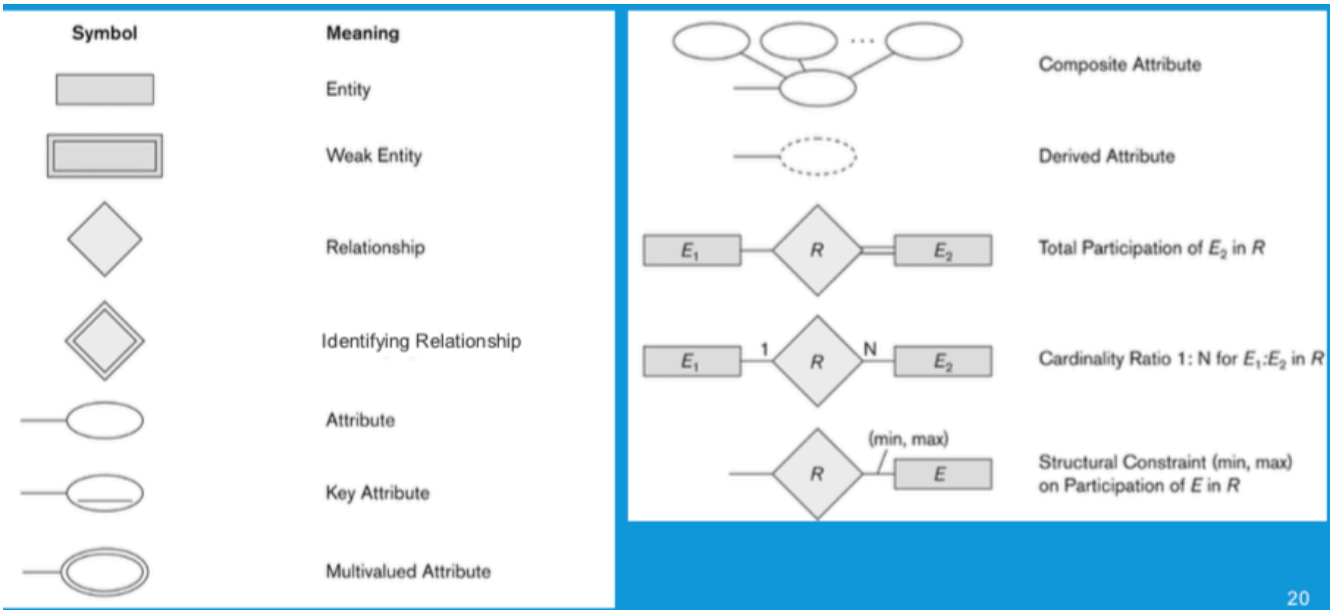


    - A **1:N or one-to-many relationship** from entity type S to entity type T is one in which an entity from S can be related to two or more entities from T.
    - An **N:M or many-to-many relationship** from entity type S to entity type T is one in which an entity from S can be related to two or more entities from T, and an entity from T can be related to two or more entities from S.
    - *(min, max)* notation for relationship structural constraints (focus on each of the entity in one entity type)
        - This notation specifies that **each entity** participates in at least min and at most max relationship instances(of relationship) in a relationship.
            - total participation: min > 0
            - 1:1, max = 1
            - 1:1 total: (1,1)
            - 1:1 partial: (0,1)
        - min must be at least 0 and at most max (0 <= min and min <= max)
        - max must be at least 1 (max >= 1)

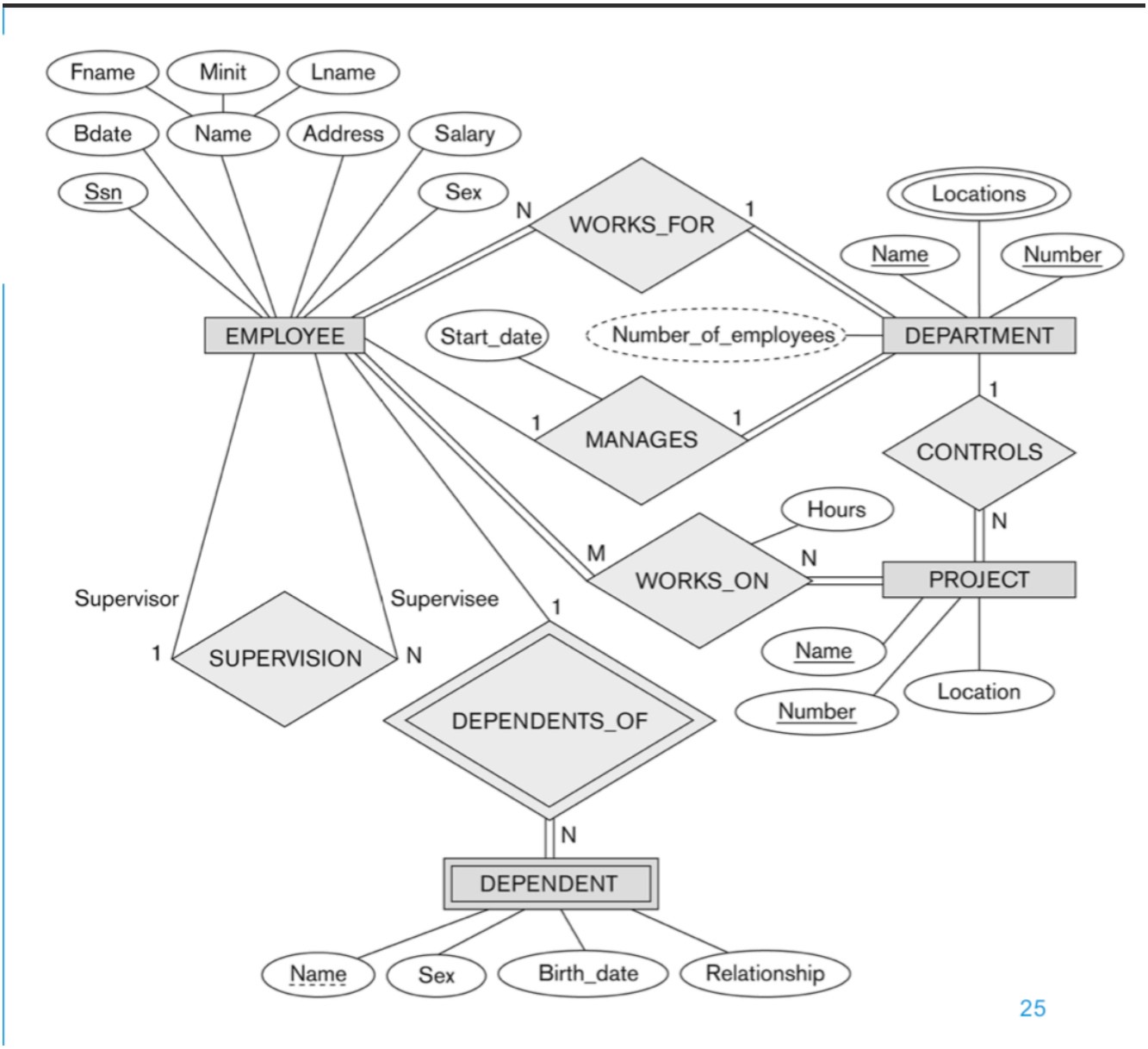
- Recursive relationship type



  - A recursive relationship is one in which **the same entity participates more than once** in the relationship. The relationship should be marked by the role that an entity takes in the participation(supervisor).
  - It is also called a **self-referencing relationship** type.
- Weak entity type: defined as entity types that **doesn't have super key itself**, therefore determined by other entity by reference(foreign key)
  - A **weak entity** that does not have a key attribute and is identification- dependent on another entity type. It **must** participate in an **identifying relationship** type with an owner or identifying entity type. In other words, weak entity type **must be owned by some owner entity type (total participation)**.
  - A weak entity is identified by the combination of: (1) its **partial key** and (2) the **identifying entity type** related to the identifying relationship type.
    - **because partial key may be the same**
  - e.g.:
    - Ada Chan is an employee. She has a dependent(受抚养者) Cindy Chan.
    - Bob Chan is an employee. He has a dependent Cindy Chan.
    - The two dependent entities are identical and determined by employee name(foreign key) and dependent name(partial key)
    - The EMPYLOEE entity type owns the DEPENDENT entity type.
- Notations for ER Diagrams

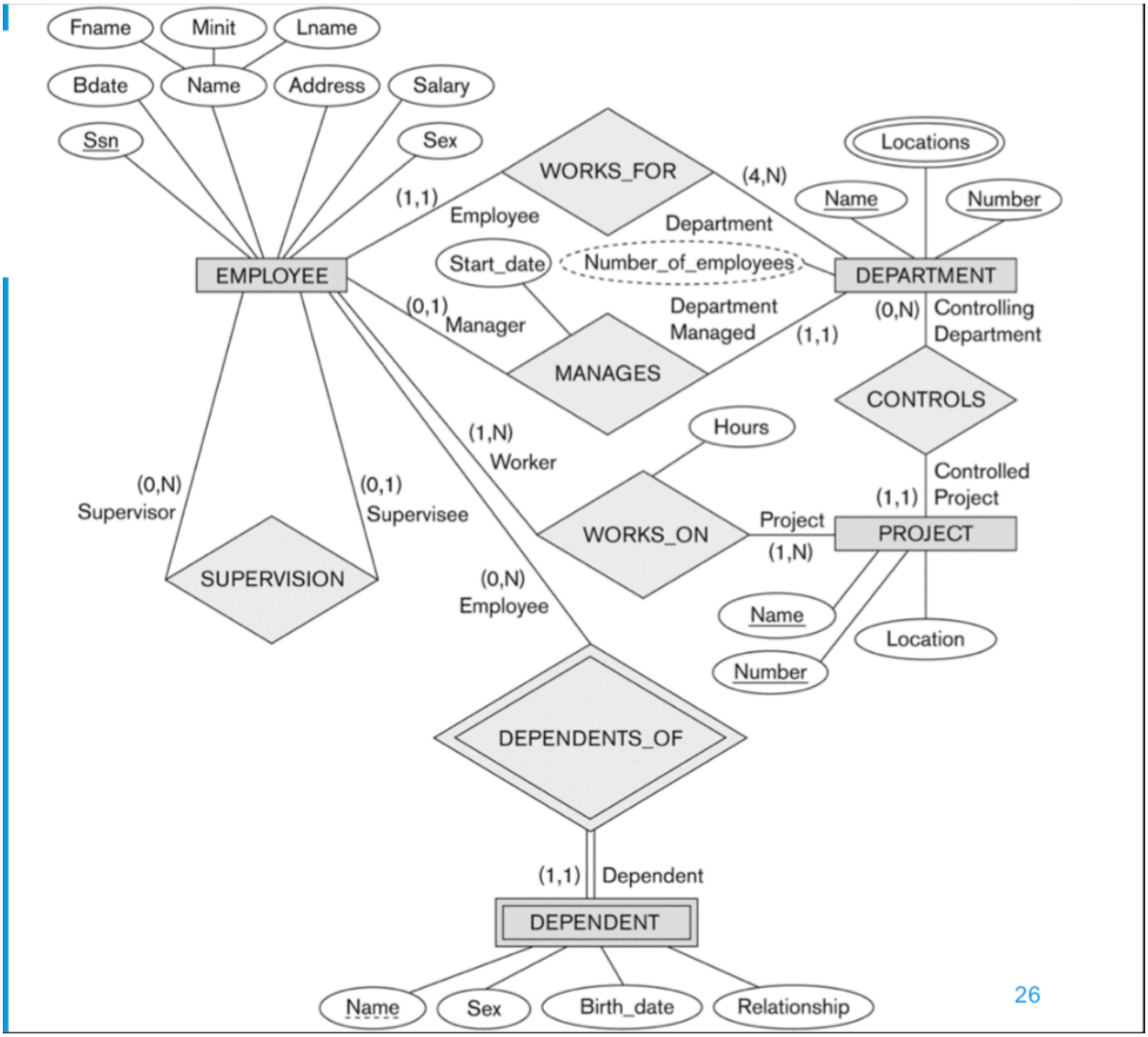


- Case Study:



  - An ER diagram for the company database.
  - 3 entities: EMPLOYEE, DEPARTMENT, and PROJECT

- 1 weak entity: DEPENDENT
- 4 relationships: WORKS_FOR, MANAGES, WORKS_ON, and CONTROLS
- 1 identifying relationship: DEPENDENTS_OF
- 1 recursive relationship: SUPERVISION
- The company is organized into DEPARTMENTs
- Each DEPARTMENT has a unique name, unique number, many EMPLOYEEs and an EMPLOYEE who manages the DEPARTMENT.
- A DEPARTMENT may have several locations.
- We keep track of the start date of the department manager and the number of employees for each DEPARTMENT.
- A DEPARTMENT controls a number of PROJECTs.
- Each PROJECT has a unique name, unique number and is located at a single location and is controlled by a DEPARTMENT.
- Each EMPLOYEE has social security number (Ssn), address, salary, sex, and birthdate. Ssn is a key attribute and address is composite attribute.
- Each EMPLOYEE works for one DEPARTMENT. Many EMPLOYEEs work for the same DEPARTMENT.
- Each EMPLOYEE may work on several PROJECTs.
- Many EMPLOYEEs work on the same PROJECT.
- An EMPLOYEE manages at most one DEPARTMENT.
- It is required to keep track the number of hours per week that each EMPLOYEE currently works on each PROJECT and the direct supervisor of each EMPLOYEE.
- A supervisor can supervise many EMPLOYEEs.
- An EMPLOYEE may have a number of DEPENDENTs. For each dependent, it is required to keep a record of name, sex, birthdate, and relationship to the EMPLOYEE.



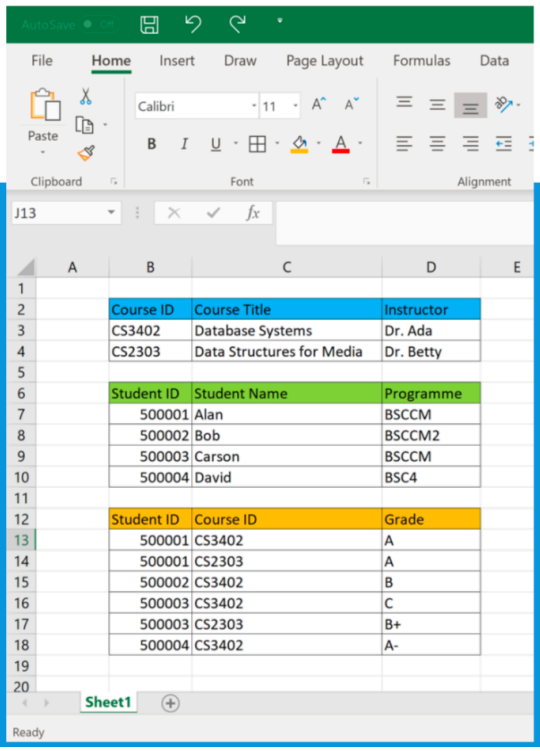
- An ER diagram for the company database with structural constraints specified using (min, max) notation and role name.
- A DEPARTMENT has **exactly one** manager and an EMPLOYEE can manage at most one DEPARTMENT.
- An EMPLOYEE can work for exactly one DEPARTMENT but a DEPARTMENT has at least 4 EMPLOYEEs.
- An EMPLOYEE works on at least one project. A PROJECT has at least one worker.
- A DEPARTMENT can control no PROJECT or any number of PROJECTs, but a PROJECT has exactly one controlling department.
- An EMPLOYEE can have no dependent or many dependents, but a dependent belongs to exactly one EMPLOYEE.
- An EMPLOYEE has at most one supervisor and may be a supervisor supervising any number of supervisees.

Tutorial 01:ER Model

Question 1

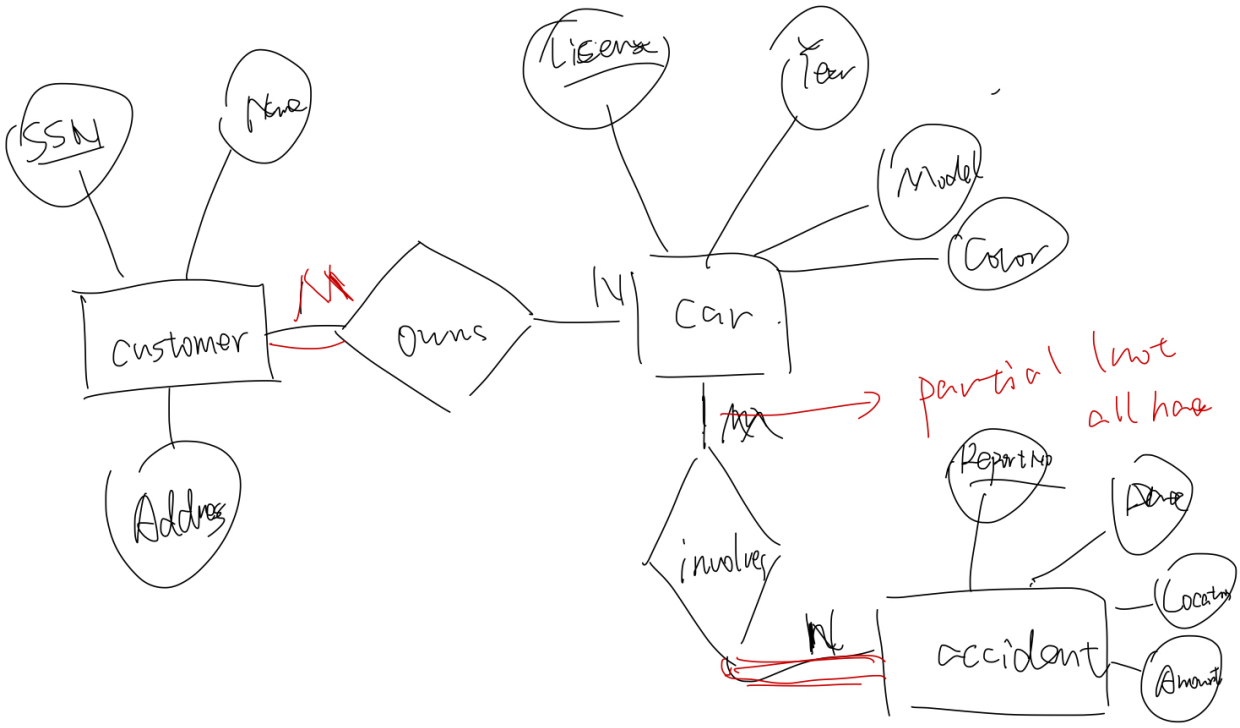
- An instructor Peter does not know relational database management systems. He uses an Excel file to store university data. Here are some sample data stored in the Excel file.
  a) Identify entity, entity set, attribute, relationship, relationship set in this application.
  b) Is there any integrity constraint in this application? If so, is it easy to make sure the constraint(s) is not violated in Excel?

entity: course, student, and instructor
entity set(unique): set of rows
attributes: course ID, student ID,Instructor ID
(simple attributes is not needed to be added in the exam,
just a entry of simple attribute)
Relationship: teaches(instructor-student), takes(stu-course)(add grade as simple attriubte)
constraint: many attribute need to be unique => need a relational database management system



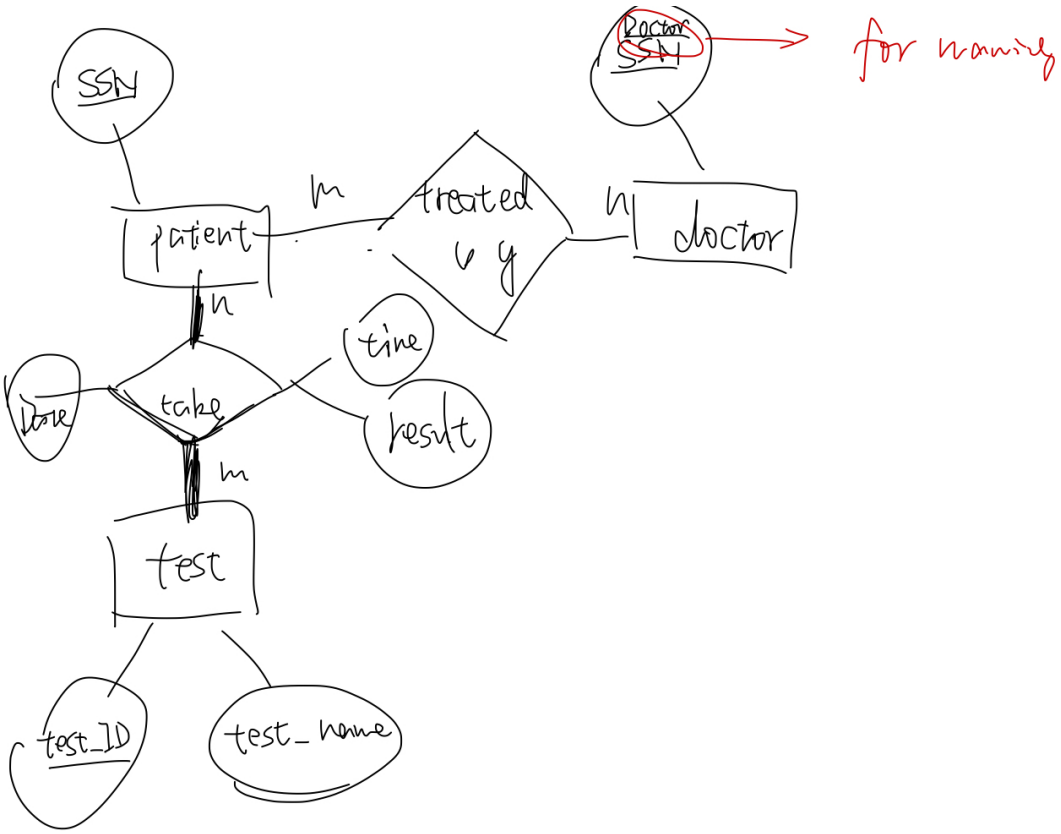
	A	B	C	D	E
1					
2		Course ID	Course Title	Instructor	
3		CS3402	Database Systems	Dr. Ada	
4		CS2303	Data Structures for Media	Dr. Betty	
5					
6		Student ID	Student Name	Programme	
7		500001	Alan	BSCCM	
8		500002	Bob	BSCCM2	
9		500003	Carson	BSCCM	
10		500004	David	BSC4	
11					
12		Student ID	Course ID	Grade	
13		500001	CS3402	A	
14		500001	CS2303	A	
15		500002	CS3402	B	
16		500003	CS3402	C	
17		500003	CS2303	B+	
18		500004	CS3402	A-	
19					
20					

- Question 2: Construct an ER diagram for a car insurance company. Identify the key entities, relationships and their attributes in the ER diagram.
  - A customer owns **at least one(total participation)** car.
  - A car may be owned by more than one customer.
  - An accident involves **at least one** car.
  - A car may have a number of recorded accidents associated with it.



- Question 3: Construct an ER diagram for a hospital. Identify the key entities, relationships and their attributes in the ER diagram.

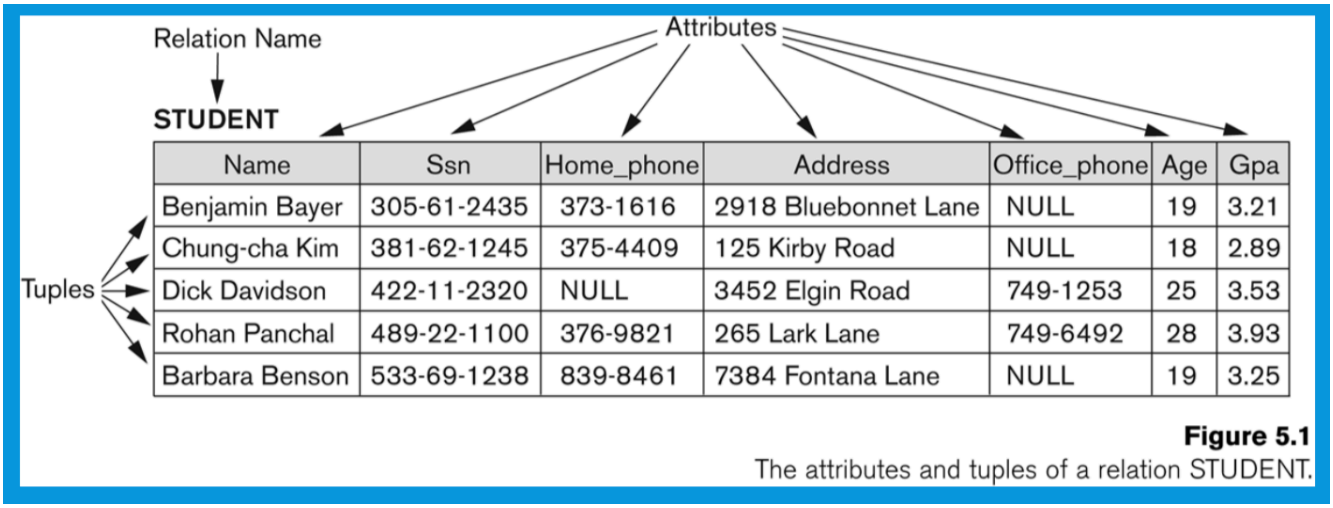
- The hospital has a set of patients and a set of medical doctors.
- A patient may be treated by more than one doctor.
- A doctor may have a number of patients.
- A log of the various conducted tests and results is associated with each patient.



- Steps to draw a ER diagram
  - Entities: customer,car, accident
  - Relationships: owns involve
    - Partial/Total Participation
    - Constraints on Relationship
  - Attributes/Key Attributes

Lecture 02: Relational Model

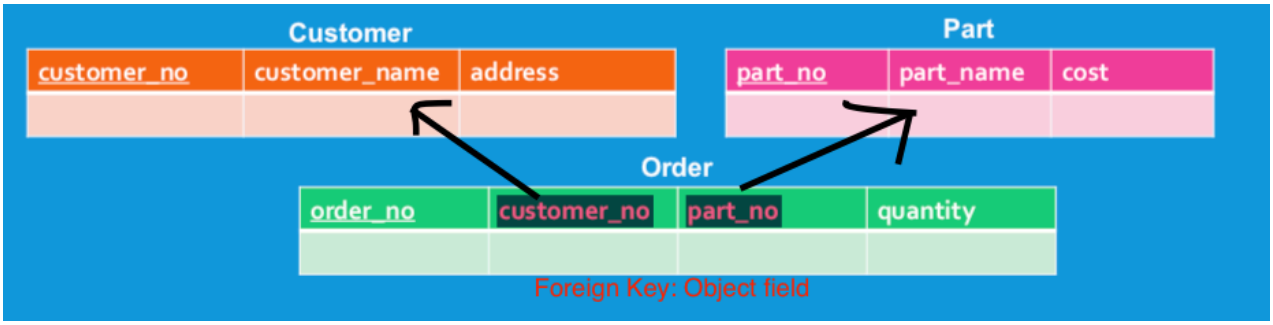
- Many database implementations are always based on **relational approach**
  - ER diagram => relation model
- Relation : Looks like a table of values
  - A relation contains a set of **rows (tuples(元组))** and each **column (attribute)** has a column header that gives an indication of the meaning of the data items in that column
    - Associated with each attribute of a relation is a set of values (domain)
    - Students(SSN:string, Name:string, GPA:double)
  - The data elements in **each row** (tuple) represent certain facts that **correspond to a real-world entity or relationship(also multivalued attribute)**



Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	749-1253	25	3.53
Rohan Panchal	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	NULL	19	3.25

Figure 5.1
The attributes and tuples of a relation STUDENT.

- Primary Key vs Foreign Key
  - **Primary Key**: Uniquely indentify a record in the table. (We can have **only one** primary key in a table)
  - **Foreign Key**: Foreign key is a field in the table that is primary key in another table(reference to other table, can be treated as a object field or pointer). (We can have **more than one** foreign key in a table)
- Relational Data Model: Basic Structure
  - Each row/turple in a relation is a record/turple (an entity)
  - Each attribute in a relation corresponds to a particular field of a record



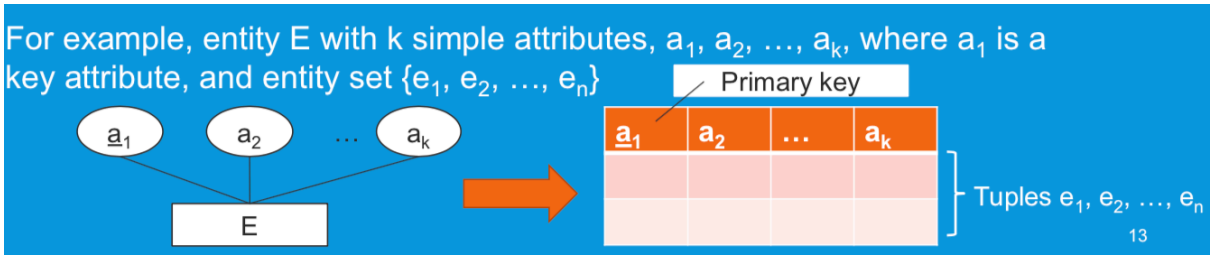
  - Definition Summary

Informal Terms	Formal Terms
Table	Relation
Column header	Attribute
All possible values for a column	Domain
Row	Tuple
Table definition	Schema of a relation
Populated table	State of the relation

- Relation State
  - Each populated relation has many records or tuples in its relation state
  - Whenever the database is changed a new state arises (change the data => change the state)
  - Basic operations for changing the database:
    - Insert – add a new tuple in a relation
    - Delete – remove an existing tuple from a relation
    - Update – modify an attribute of an existing tuple
  - Query is not a part of changing state operations
- Characteristics of Relations

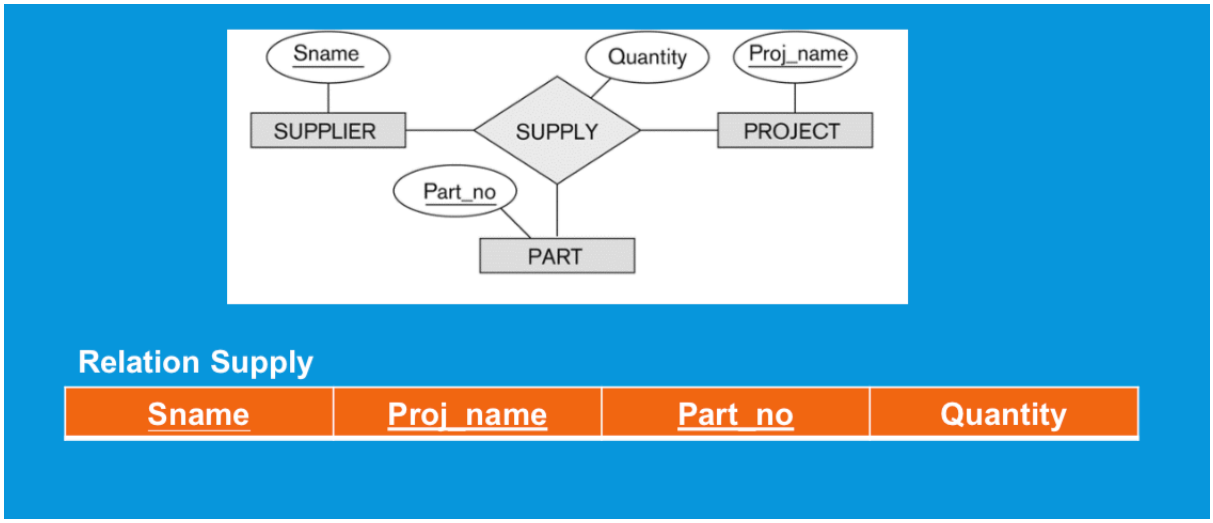
- The tuple **are not considered to be ordered**, even though they appear to be in a tabular(列成表的) form (may have dfferent presentation orders)
  - **same relation state can be with different order of tuples** => unordered set
- Values in a tuple
  - **All values are considered atomic(indivisible) => the reason of construct a new relation for multivalued attributes and separate composite attribute**
  - Basic unit for manipulation(add or change)
- Each value in a turple must be from the domain(set of values) of the attribute for that column => data type and size should fulfill the requirements initialized in table schema
- A special null value is used to represent values that are unknown or not available or inapplicable in certain tuples

- From ER Diagram to Relations:
  - **notice: all composite attribute should be divided into multiple simple attributes in the relation model**
  - Step 1: Mapping of strong Entity types
    - Create a relation R that includes all the simple attributes of E(The strong enity)
    - Choose one of the key attributes E as the primary key for R
    - R is called an entity relation(each tuple represent an entity instance)

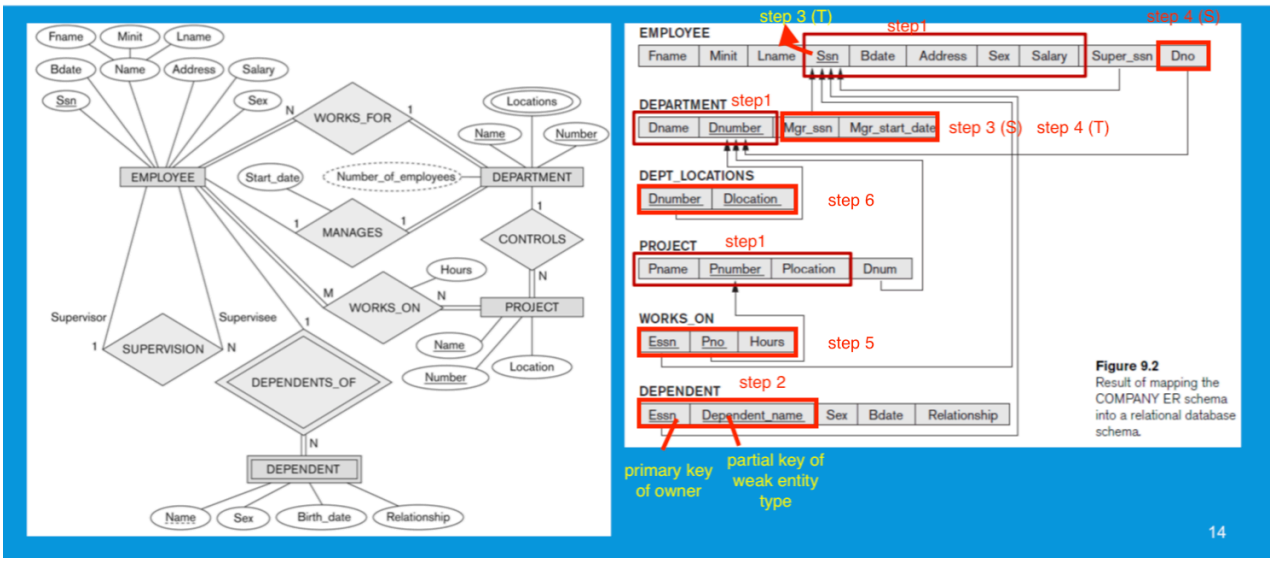


  - Step 2: Mapping of weak Entity types
    - Create a relation R and includes all the simple attributes of the entity type as the attributes of R
    - Include the **primary key attribute of the owner** as the **foreign key attributes** of R
    - The primary key of R is the **combination** of(1) the **primary key of the owner** and(2) the **partial key of the weak entity type**(may be the name of the dependence)
  - Step 3: Mapping of 1:1 Relation**ship** types
    - Identify relations that correspond to the entity types participating R (Says S and T)
    - Approaches:
      - **Foreign key approah**(let one of the entity remember the relationship) used in the example
        - Choose one of relations(says S, **normally the total participation side**) and include the primary key of T as the foreign key in S
        - Include all the **simple attributes of the relationship** as the attributes of S
      - Merged relationship approach: merge the 2 entity types and the relationship(simple attributes) into a single relation (**not efficient**)
      - Cross reference or **relationship relation approach**
        - Set up a third relation **R** for the purpose of cross-referencing(including) the **primary keys of the two relations S and T** representing the entity types
        - Also including the primary key attributes of S and T as foreign keys to S and T respectively

        - primary key of R will be **one of the two foreign keys** (because it is 1: 1 relation, 1 key is enough to identify a relation)
        - include the simple attributes of the relationship
- Step 4: Mapping of 1:N Relationship Types(let N-side remember the relation=>more efficient)
    - Identify relation S that represents participating entity type at **N-side** of relationship type
    - Include **the primary key of relation T as the foreign key in S**
    - Include the **simple attributes** of the 1:N relationship type as the attributes of S
    - Alternative approach(create a new relationship)
        - Use the relationship relation option as in the third approach for binary 1:1 relationships, but the **primary key of R will be two foreign keys of both involvoing entities**
- Step 5: Mapping of Binary M:N Relationship Types(cross reference)
    - Create a new relation R
    - Include all primary key of the participating entity types as the foreign key attributes in R
    - The **combination of all foreign key attributes** forms the primary keys of R
    - Include **all the simple attributes** of M:N relationship type as attribute of R
- Step 6: Mapping of Multivalued Attributes
    - Create a new relation R
    - Primary key of R. is the **combination of A and the primary key attribute of "owner" (relationship or entity) that has A as an attribute**
    - If the multivalued attribute is composite, include its simple components.
- Step 7: Mapping of N-ary Relationship Types
    - Create a new relation to represent R
    - Include **primary keys of participating entity types** as foreign keys
    - Include all the **simple attributes of R** as the attributes of S
    - The primary key of S is a combination of **all the foreign keys that reference the relations representing the participating entity types**



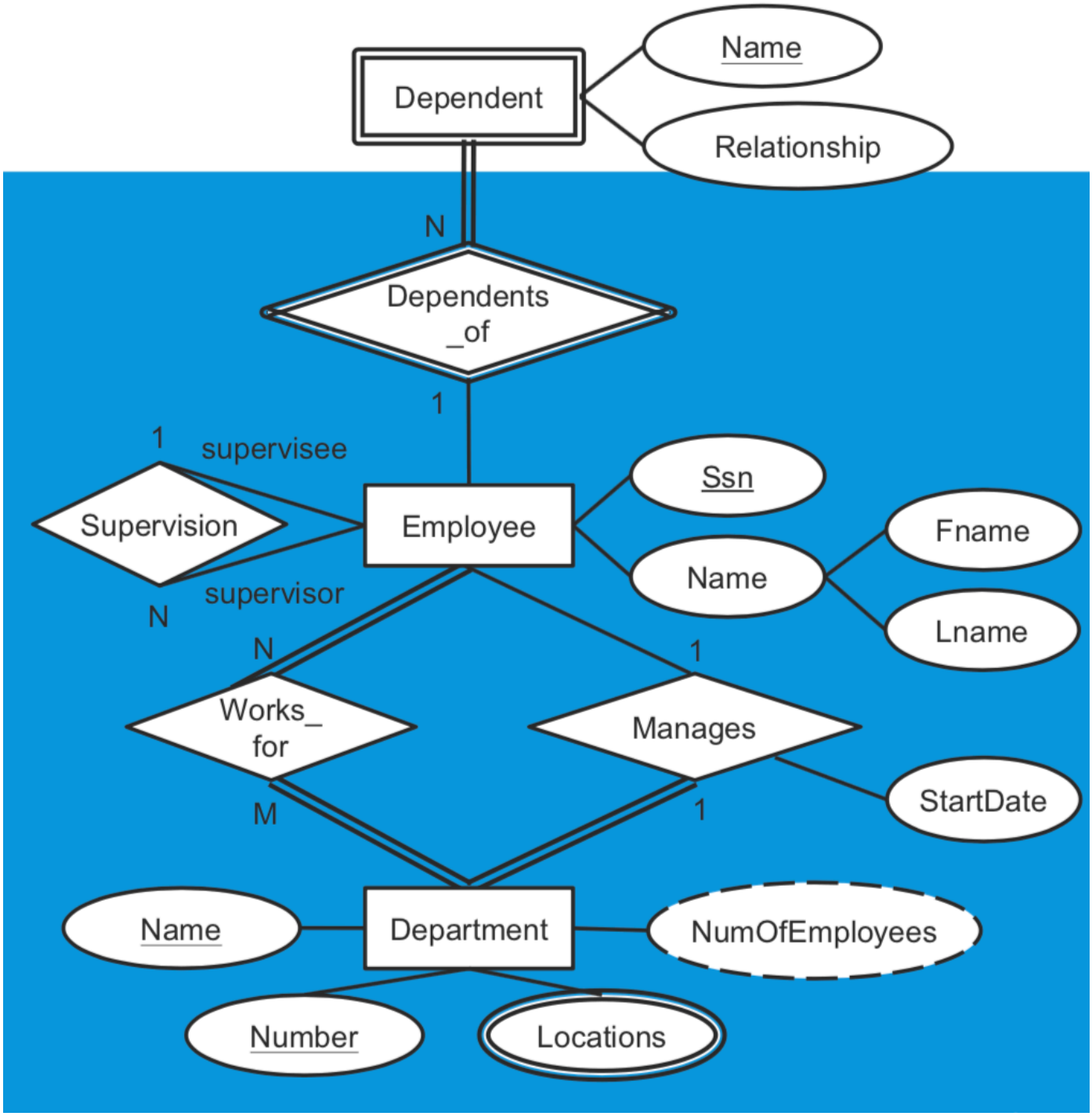
- The example



- Terms

ER Model	Relational Model
Entity type	Entity relation
1:1 or 1:N relationship type	Foreign key (or relationship relation)
M:N relationship type	Relationship relation and two foreign keys
n-ary relationship type	Relationship relation and n foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Doman
Key attribute	Primary key

Tutorial 02 : Relational Model



- (a)For each strong entity type
  - Include simple (or atomic) attributes of the entity
  - include components of composite attributes
  - Identify the primary key from the key attributes
  - Do not include : non-simple component of composite attributes, derived attributes, multivalued attributes (not yet)
  - Employee(SSN, Fname, Lname)
  - Department(Number, Name)
- (b) Weak Entity(partial key, foreign key, simple attributes)
  - **Dependent**(Name,EmployeeSSN,Relationship(simple_attr))
- (c) Binary 1:1 relation ship type(total side remember the information)

    - Options: store the Manager information into Dept or store Dept information in manager
        - choose the total participation side to store the other side => avoid empty entry to save memory
    - **Department**(Number, Name, **ManagerSSN**, **StartDate**)
- (d) Binary 1:N Relationship type(N side remember the information)
    - **Employee** (SSN, Fname, Lname, **SupervisorSSN**)
- (e) Binary M:N Relationship type(create new relation type)
    - **Work_for**(**EmployeeSSN**,**DeptNum**,simple_attrs)
- (f) Multi-valued attribute(primary key of owner, simple_attrs values)(they are all prime keys) (Why??)
    - each value create an entry
    - **Dept_location**(**DeptNum**,**single Location_value**)

Lecture 03: Structured Query Languages

- Relational Query Language :
    - Data Definition Language (DDL) : standard commands for defining the different structures in a database. DDL statements **create, modify, and remove** database objects such as tables, indexes, and users, Common DDL statements ate CREATE, ALTER, and DROP x
    - Data Manipulation Language (DML): standard commands for dealing with th**e manipulation of data present in database**. Common DDL statements SELECT, INSERT, UPDATE, and DELETE.
    - Each statement in SQL ends with a semicolon(;)
- `CREATE SCHEMA` Statement
    - A schema is a way to **logically group objects in a single collection and provide a unique namespace for objects**. => like a package (different with the schema in relational model)
    - The `CREATE SCHEMA` statement is used to create a schema. A schema name **cannot exceed 128 characters**. Schema names must be **unique within the database**.
    - Syntax

```
CREATE SCHEMA schemaName AUTHORIZATION user-name;
```

    - Example

```
CREATE SCHEMA COMPANY AUTHORIZATION ‘Jsmith’;
```

- `CREATE TABLE` Statement
    - A `CREATE TABLE` statement creates a table. Tables contain **columns and constraints(primary key, foreign key, data type, simple attributes)**, rules to which data must conform. Table-level constraints specify a column or columns. Columns have a data type and can specify column constraints (column-level constraints => **determined the domain of attributes**).

```
CREATE TABLE EMPLOYEE
        ( FNAME          VARCHAR(15)      NOT NULL ,
          MINIT          CHAR ,
          LNAME          VARCHAR(15)      NOT NULL ,
          SSN            CHAR(9)          NOT NULL ,
          BDATE          DATE
          ADDRESS        VARCHAR(30) ,
          SEX            CHAR ,
          SALARY         DECIMAL(10,2) ,
          SUPERSSN       CHAR(9) ,
          DNO            INT              NOT NULL ,
    PRIMARY KEY (SSN) ,
    FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN) ,
    FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER) ) ;
```

```
CREATE TABLE DEPARTMENT
        ( DNAME          VARCHAR(15)      NOT NULL ,
          DNUMBER        INT              NOT NULL ,
          MGRSSN         CHAR(9)          NOT NULL ,
          MGRSTARTDATE   DATE ,
        PRIMARY KEY (DNUMBER) ,
        UNIQUE (DNAME) ,
        FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN) ) ;
CREATE TABLE DEPT_LOCATIONS
        ( DNUMBER        INT              NOT NULL ,
          DLOCATION      VARCHAR(15)      NOT NULL ,
        PRIMARY KEY (DNUMBER, DLOCATION) ,
        FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER) ) ;
```

CREATE TABLE Statement (3/3)

- Create 6 tables
  - EMPLOYEE
  - DEPARTMENT
  - DEPT_LOCATIONS
  - PROJECT
  - WORKS_ON
  - DEPENDENT

```
CREATE TABLE PROJECT
        ( PNAME          VARCHAR(15)      NOT NULL ,
          PNUMBER        INT              NOT NULL ,
          PLOCATION      VARCHAR(15) ,
          DNUM           INT              NOT NULL ,
        PRIMARY KEY (PNUMBER) ,
        UNIQUE (PNAME) ,
        FOREIGN KEY (DNUM) REFERENCES DEPARTMENT(DNUMBER) ) ;
CREATE TABLE WORKS_ON
        ( ESSN           CHAR(9)          NOT NULL ,
          PNO            INT              NOT NULL ,
          HOURS          DECIMAL(3,1)     NOT NULL ,
        PRIMARY KEY (ESSN, PNO) ,
        FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN) ,
        FOREIGN KEY (PNO) REFERENCES PROJECT(PNUMBER) ) ;
CREATE TABLE DEPENDENT
        ( ESSN           CHAR(9)          NOT NULL ,
          DEPENDENT_NAME VARCHAR(15)      NOT NULL ,
          SEX            CHAR ,
          BDATE          DATE ,
          RELATIONSHIP   VARCHAR(8) ,
              PRIMARY KEY (ESSN, DEPENDENT_NAME) ,
              FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN) ) ;
```

- Table Manipulation
  - The `ALTER TABLE` statement allows you to => **manipulate columns of table instead of insert an instance into the table**:

    - Add/Drop a column/constraint to a table
    - Increase the width of a VARCHAR or VARCHAR FOR BIT DATA column
    - change the default value for a column
  - `DROP TABLE` statement **removes the specified table**.
  - The `TRUNCATE TABLE` statement allows you to quickly remove all content from the specified table and return it to its initial empty state.
- `SELECT` Statement
  - The basic statement for retrieving(getter) information from a database

```
SELECT <attribute list>
// A list of attrbibute names whose values are to be retrieved by the query
// projection in algebra
FROM <table list>
// a list of relation names required to process the query
WHERE <condition> ;
// condition is a boolean expression
// select in algebra
```

  - try one of single/double quotation for a string

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q0:  SELECT  Bdate, Address
     FROM    EMPLOYEE
     WHERE   Fname='John' AND Minit='B' AND Lname='Smith';
```

Projection attributes

Selection conditions

EMPLOYEE

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

Result:

Bdate	Address
1965-01-09	731Fondren, Houston, TX

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1:  SELECT  Fname, Lname, Address
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   Dname='Research' AND Dnumber=Dno;
```

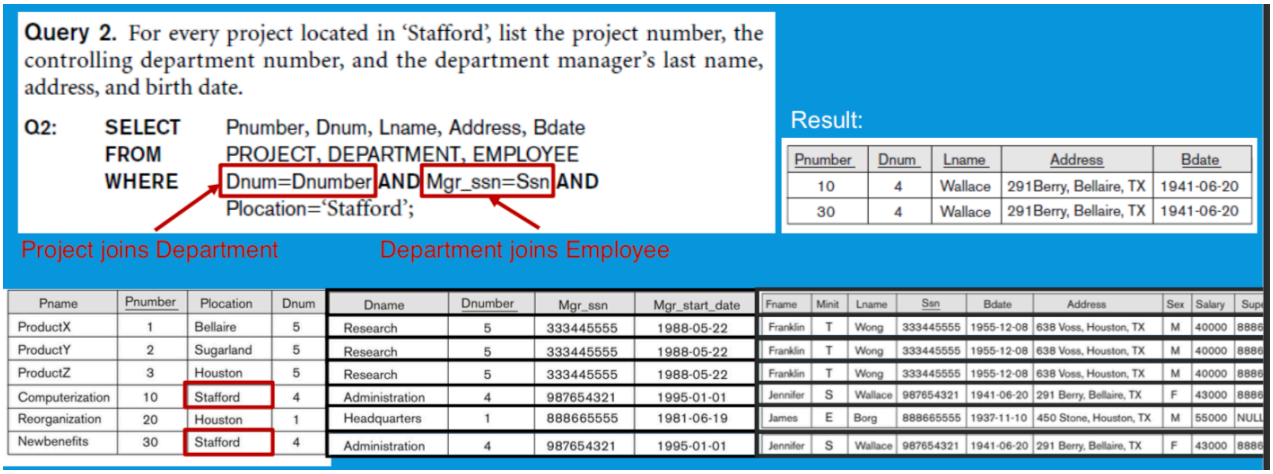
Join conditions

Result:

Fname	Lname	Address
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

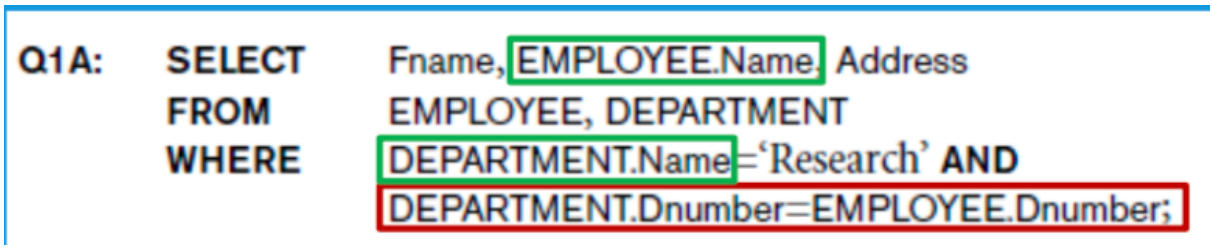
Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer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

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Administration	4	987654321	1995-01-01
Research	5	333445555	1988-05-22
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19



- Each connection between 2 tables will be as one constraint in WHERE closure
- `WHERE` will find the information according to the attributes' name

- Ambiguous Attribute Names
  - Same name can be used for two (or more) attributes in different relations
    - As long as the attributes are in different relations
    - Must qualify the attribute name with the relation name to prevent ambiguity (Name space)



- Aliasing, Renaming and Tuple Variable
  - Aliases or tuple variables: Declare alternative relation names E and S to refer to the EMPLOYEE relation twice in a query (**important when there is a recursive relation and some query based on comparation between entries within the same relation**)
  - e.g. For each employee, retrieve the employee's first and last name and the first and last name of his of her immediate supervisor

```
SELECT E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE AS E, EMPLOYEE AS S // E and S are alias
WHERE E.Super_ssn=S.Ssn
```

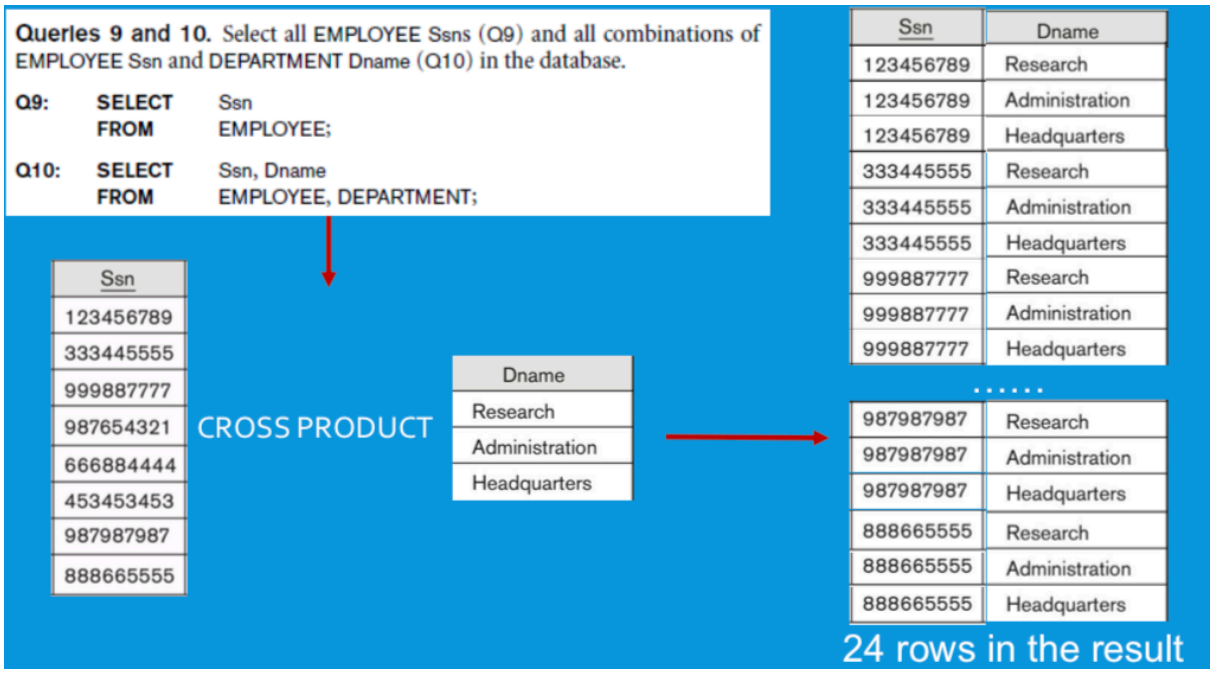
  - Recommended practice to **abbreviate names and to prefix same or similar attribute** from multiple tables
  - Attributes can also be aliased

```
EMPLOYEE AS E (Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno)
```

  - Note that the relation `EMPLOYEE` now has a variable name `E` which corresponds to a tuple variable
  - The "AS" may be **dropped in most SQL implementations**

```
EMPLOYEE E (Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno)
```

- Unspecified `WHERE` Clause(分句)
  - Missing `WHERE` clause: Indicates no condition on tuple selection(select ALL)
  - **notice that**: to select all tuples => do not write where; to select all columns => use `SELECT *`
  - The resultant effect is a CROSS PRODUCT (`JOIN` n x m) => multiple relations without `WHERE`
    - Result is **all possible tuple combinations** between the participating relations



  - Specify an asterisk*: Retrive all the **attributes** values of the **selected tuples**
- Table as Sets in SQL
  - The `ALL` and `DISTINCT` keywords determine **whether duplicates are eliminated from the result of the operation**
    - `DISTINCT` : Result have no duplicate row(**default**) => **Thus, the returned row of selecting different attribute might be different**
    - `ALL` : May have duplicate row(depends on the original data set)

```
SELECT DISTINCT Salary
```

  - `SELECT` statement using the set operators `UNION`, `INTERSECT` and `MINUS`, all set operators have equal precedences(**set operations work on two returned queries got by 2 distinct select**)
  - Each `SELECT` statement within the operator must have the **same number of fields** in the result sets with similar data types.
  - `UNION` operator
    - The UNION operator is used to combine the result sets of 2 or more SELECT statements. It **removes duplicate rows** between the various SELECT statements.
    - e.g.: Select a list of all project numbers for projects that involve an employee whose last name is 'Smith', **either as a worker or as a manager of the department that controls the project**(2 tables are involved)

```
(
  SELECT PNUMBER
    FROM PROJECT, DEPARTMENT, EMPLOYEE
    WHERE DNUM = DNUMBER  (project <=> department)
    AND MGR_SSN = SSN   (department <=> employee)
    AND LNAME = 'Smith'   (the condition)
) UNION (
  SELECT DISTINCT PNUMBER
    FROM PROJECT, WORKS_ON, EMPLOYEE
    WHERE PNUMBER = PNO   (project <=> works_on relationship)
    AND ESSN = SSN      (employee <=> works_on relationship)
    AND LNAME = 'Smith'   ()
);
```

- `UNION` ALL operator
  - It returns all rows from query and it **does not remove duplicate rows** between the various SELECT statements.

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

- `INTERSECT` operator
  - The INTERSECT operator is used to return the results of 2 or more SELECT statements.It only returns the rows selected by all queries or data sets. In other words, if a record exists in one query and not in the other, it will be omitted from the INTERSECT results.
- `MINUS` operator
  - The MINUS operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement. Each SELECT statement will define a dataset. The MINUS operator will retrieve all records from the first dataset and then **remove from the results all records from the second dataset.**

- `LIKE` Conditions
  - The `LIKE` condition allows wildcards(通配符) to be used in the `WHERE` clause of a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. This allows you to perform pattern matching.

Wildcard	Explanation
%	Allows you to match any string of any length (including zero length)
_	Allows you to match on a single character

  - select first_name begins with 'P'

```
SELECT first_name
FROM customers
WHERE customers.last_name LIKE 'P%'
```

- `ORDER BY` Clause
  - The `ORDER BY` clause is used to sort the records in your result set. The `ORDER BY` clause can only be used in `SELECT` statements.
  - Syntax: `ORDER BY expression [ ASC | DESC ]`
    - expressions: The columns or calculations that you wish to retrieve
    - ASC: Optional. It sorts the result set in ascending order by expression (**default**, if no modifier is provider).
    - DESC: Optional. It sorts the result set in descending order by expression.
    - There may be many of the condition, when first condition is same
  - e.g.: List the entire borrow table in descending order of amount, and if several loans have the same amount, order them in ascending order by loan#:

```
SELECT *
FROM Borrow
ORDER BY amount DESC, loan# ASC;
```

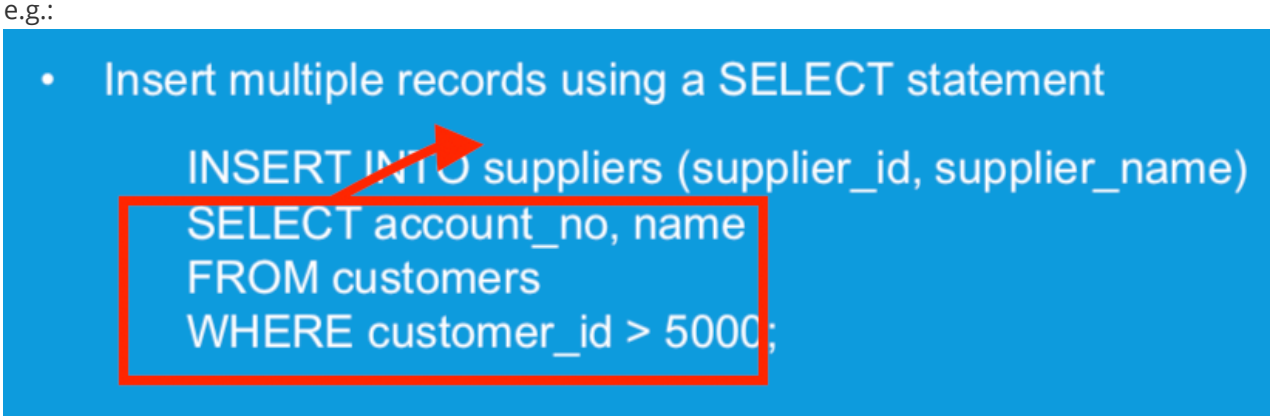
- `INSERT` statement: insert single/ multiple records
  - insert single record using the `VALUES` keyword

```
INSERT INTO target_table (column1, column2, ... column_n)
VALUES (expression1, expression2, ... expression_n);
```

  - insert multiple records using a `SELECT` statement(list attrs after target_table because may insert only part of the value, others attrs use default)

```
INSERT INTO target_table (column1, column2, ... column_n)
SELECT expression1, expression2, ... expression_n
FROM source_table
[WHERE conditions];
```

  - e.g.:



- `DELETE` Statement: delete a single/multiple records from a table(different with drop)

```
DELETE FROM target_table
[WHERE conditions;]
```

  - target_table: The table that you wish to delete records from.

- `WHERE` conditions: Optional. The conditions that must be met for the records to be deleted. If no c**onditions are provided, then all records from the table will be deleted.**
- Example: Delete all records from the employee table where the first_name is Bob

```
DELETE FROM EMPLOYEE
WHERE FIRST_NAME = 'Bob';
```

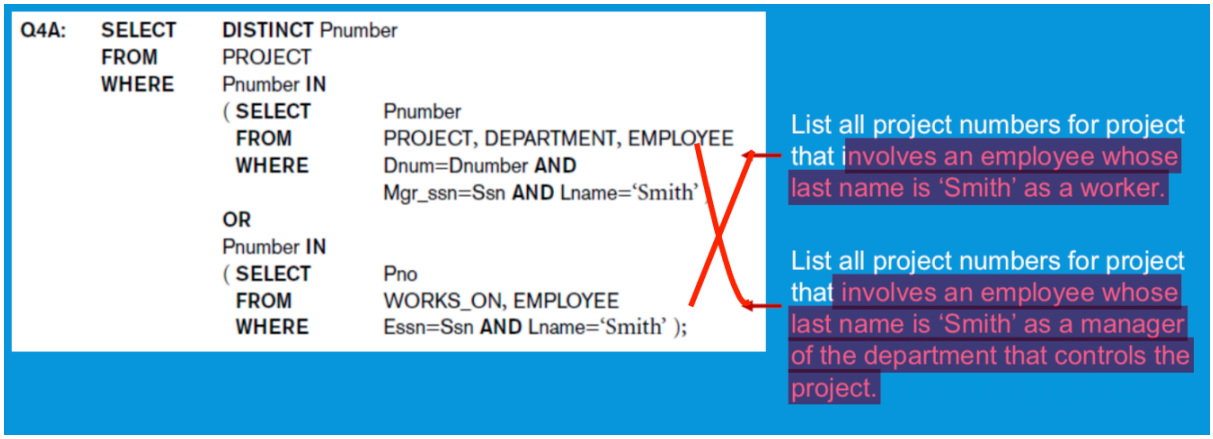
- `UPDATE` Statement: Used to update exsiting records in a table

```
UPDATE table
SET column1 = expression1,
  column2 = expression2, ...
  column_n = expression_n
[WHERE conditions];
```

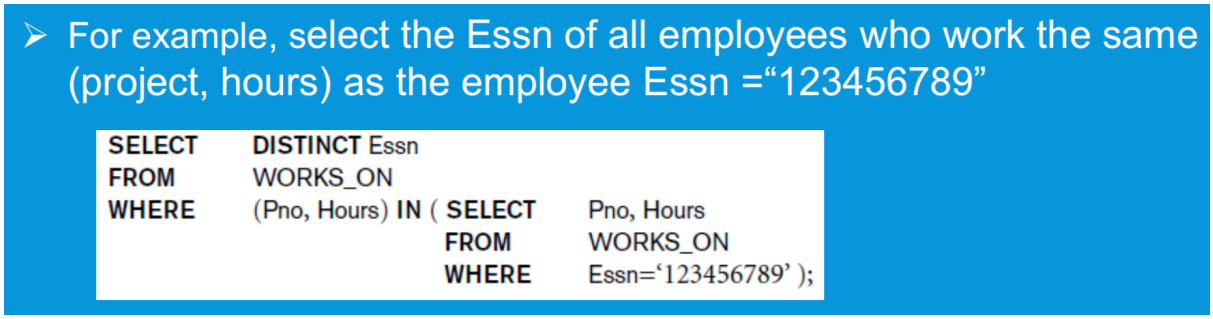
- e.g.: Update the last_name to 'Bob' in the employee table where the employee_id is 123, and increase the balance by 5%

```
UPDATE employee
SET last_name = 'Bob', balance = balance*1.05
WHERE employee_id = 123;
```

- Nested queries and set comparisons
  - Nested queries : make some trick on `WHERE` closure
    - `SELECT-FROM-WHERE` blocks within `WHERE` clause of another query
    - For example, some queries require that existing values in the database be fetched and then used in a comparison condition
  - Comparison operator IN (focus on an **entry** in the **selected set** or not)
    - Compares value v with a set (or multiset) of values v
    - Evaluate to `TRUE` if v is one of the elements in V



    - multiple attributes can be collected by a parenthesis for comparison



- `=ANY` (or `=SOME`) operator returns `TRUE` if the value b is equal to **some value** in the set V and is hence equivalent to `IN`
- `=ALL` returns `TRUE` if the value b is equal to all the values in the set V (meaningless but make sence when used with '>' or '<')
- Other operators that can be combined: >, >=, < , <= and <>(!=)
- e.g.:
  - Find the last name and first name of the employees with salary **higher than all the employees** in the department with Dno=5

```
SELECT Lname, Fname
FROM Employee
WHERE Salary > ALL( SELECT salary
                    From  Employee
                    WHERE Dno = 5);
```

  - Find the name of braches that not the least assets in "central city"

```
SELECT DISTINCT T.cname
FROM Deposit T
WHERE assets > SOME (SELECT assets
            FROM Branch
            WHERE b-city = "Central");
//or use alias
SELECT (DISTINCT) X.bname
FROM Branch X, Branch Y
WHERE X. assets > Y.assets AND Y.b-city= "Central";

```

  - Find all customers who have an account at some branch in which Jones has an account

```
// bname is branch name, cname is customer name
// first select all bname of customer "Jones"
// second select all customer except Jones whose branch name is in the selected branches
SELECT DISTINCT T.cname
FROM Deposit T
WHERE T.cname != "Jones"
    AND T.bname IN(SELECT S.bname
                        FROM Deposit S
                        WHERE S.cname = "Jones");
```

```
//A set of branch name with “Jones” as cname.
SELECT DISTINCT T.cname
FROM Deposit S, Deposit T
WHERE S.cname = “Jones” AND S.bname = T.bname
    AND T.cname != S.cname;
```

- `EXISTS` Condition:
  - `EXISTS` : The EXISTS operator is **used to test for the existence of any record in a subquery.** The EXISTS operator returns true if the subquery returns one or more records.(**not existence of entry but existence of selected type**)
  - `NOT EXISTS` condition: return true when there is no answer
  - Thus, **the sub selection always use the attributes as a `WHERE` constraint from super selection in order to return the existence information**
  - e.g.:
    - Find all customers of Central branch who have an account there but no loan there

```
SELECT C.cname
FROM Customer C
WHERE EXISTS
      (SELECT *
       FROM Deposit D
       WHERE D.cname = C.cname  (account <=> customer)
       AND D.bname = “Central”)
    AND NOT EXISTS
      (SELECT *
       FROM Borrow B
       WHERE B.cname = C.cname  (loan <=> customer)
       AND B.bname = “Central”);
```

    - Find branches having greater assets than all branches in N.T

```
SELECT X.bname
FROM Branch X
WHERE NOT EXISTS(SELECT *
                 FROM Brach Y
                 WHERE Y.b-city="N.T."
                 AND Y.assets>=X.assets);
//or
SELECT bname
FROM Branch
WHERE assets>ALL(SELECT assets
                 FROM Brach
                 WHERE b-city="N.T.")
```

    - Find all customers who have a deposit account at ALL braches located in Kowloon(no branch do not contain its deposit)

```
SELECT DISTINCT S.cname
```

```
FROM Deposit S
WHERE NOT EXIST(
    (
        SELECT bname
      FROM Branch
      WHERE b-city = "Kowloon"
    )
    MINUS(
      SELECT T.bname
        FROM Deposit T
        WHERE S.cname = T.cname
    )
);
所有的s使得 不存在(在九龙但不包含s的branch)
在九龙-包括他 个数为零
//not sure correct
SELECT DISTINCT S.cname
FROM Deposit S
WHERE (
  SELECT COUNT(*)
    FROM BRANCH B, DEPOSIT T
    WHERE B.cname = T.cname
    AND B.b-city = "Kowloon"
    AND S.cname != T.cname
)=0;
```

- Aggregate(总数) Functions
  - Built-in aggregate functions: `COUNT`, `SUM`, `MAX`, `MIN`, `AVG`
  - summarize information from multiple tuples into a single tuple
  - use `where` to limit the rows that under the consideration

```
SELECT SUM(Salary), MAX(Salary), MIN(Salary), AVG(Salary)
FROM EMPLOYEE;
```

  - e.g.: Find the sum of the salaries of all employees of the 'Reasearch' department, as well as the maximum salary, the minimum salary, and the average salary in this department

```
SELECT SUM(Salary), MAX(Salary), MIN(Salary), AVG(Salary)
FROM EMPLOYEE, DEPARTMENT
WHERE Dno = Dnumber AND Dname = 'Research';
```

  - Retrieve the total number of employees in the company
    - `COUNT(*)` is different with `SELECT *`, it is used to calculate total number of rows instead of working on the columns

```
SELECT COUNT(*)
FROM EMPLOYEE;
```

- Retrieve the number of employees in the 'Reasearch' department

```
SELECT COUNT(*)
FROM EMPLOYEE, DEPARTMENT
WHERE Dno = Dnumber AND Dname = "Reasearch"
```

- Count how many different salary values in the data base

```
SELECT COUNT(DISTINCT Salary)
FROM EMPLOYEE
```

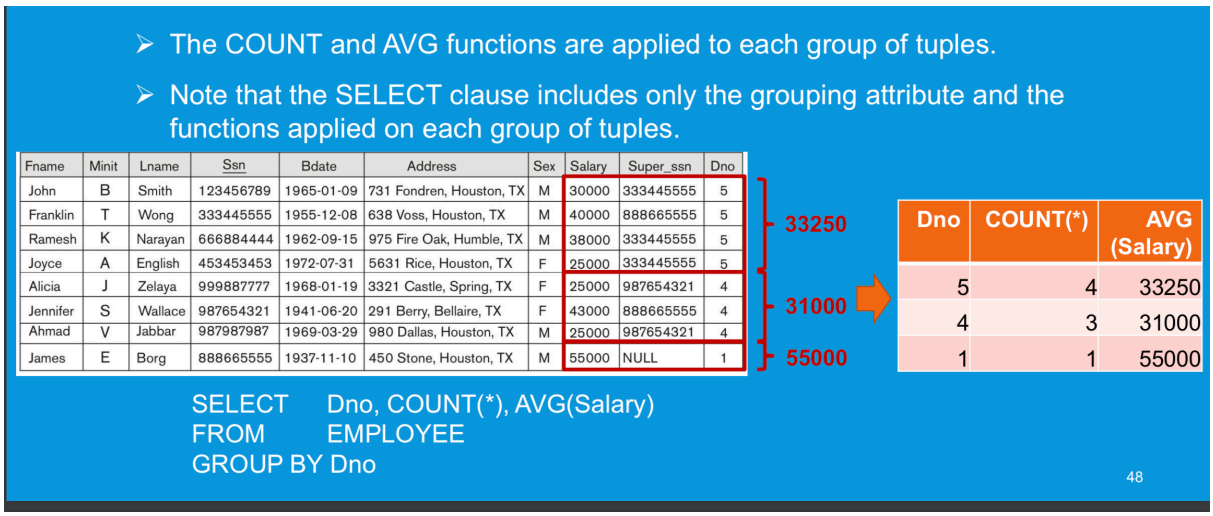
- Retrieve the names of all employees who have two or more dependents(more specific query than `EXIST`)

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE (
    SELECT COUNT(*)
    FROM DEPENDENT
    WHERE SSN=ESSN
) >= 2 ;
```

- `GROUP BY` clause(used to construct a new table, each row is identified by the attribute used to form a group) => **when you what to calculate some things according to one entity type but the type is in another relation (referenced in the current relation)**
  - group tuples by some of its attributes
    - The same value of attributes for group members called **grouping attribute(s)**, and the aggregate function is applied to each subgroup independently
    - SQL has the GROUP BY clause for this purpose
    - The **GROUP BY clause** specifies the grouping attributes, which should also appear in the SELECT clause => **value resulting from applying each function to a group of tuples** appears along with the value of the grouping attributes
    - e.g.: For each department, retrieve the department number, the number of employees in the department, and their average salary (salary is not stored in department table)

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

```sql
SELECT Dno, COUNT(*), AVG(Salary)
FROM EMPLOYEE
GROUP BY Dno;
```



- For each project, retrieve the project number, the project name, and the number of employees who work on that project
  - `GROUP BY` restricts the type of tuples that the `COUNT(*)` function works on
    - `COUNT(*)` : from all tuples to a group that shares the same Pnumber an Pname
  - If we want to select all distinct Pnumber and Pname only, we can omit `GROUP BY`

```sql
SELECT Pnumber, Pname, COUNT(*)
FROM PROJECT, WORKS_ON
WHERE Pnumber=Pno
GROUP BY Pnumber, Pname;
```

- HAVING Clause(conjunction with the GROUP BY clause)
  - Retrieve the values of the aggregate functions only for **groups that satisfying certain conditions**(select some rows in the GROUP BY table) => **constraint on the groups**, used when you need to make some constraints to the value that calculated by functions that only works correctly under `GROUP BY`（是where的后置条件，即先选择entry组成group， 再判断）.(`where` only works when you are dealing with each of the row)
  - Example 1: For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

```sql
SELECT Pnumber, Pname, COUNT(*)
FROM PROJECT, WORKS_ON
WHERE Pnumber=Pno
GROUP BY Pnumber, Pname
HAVING COUNT(*)>2
```

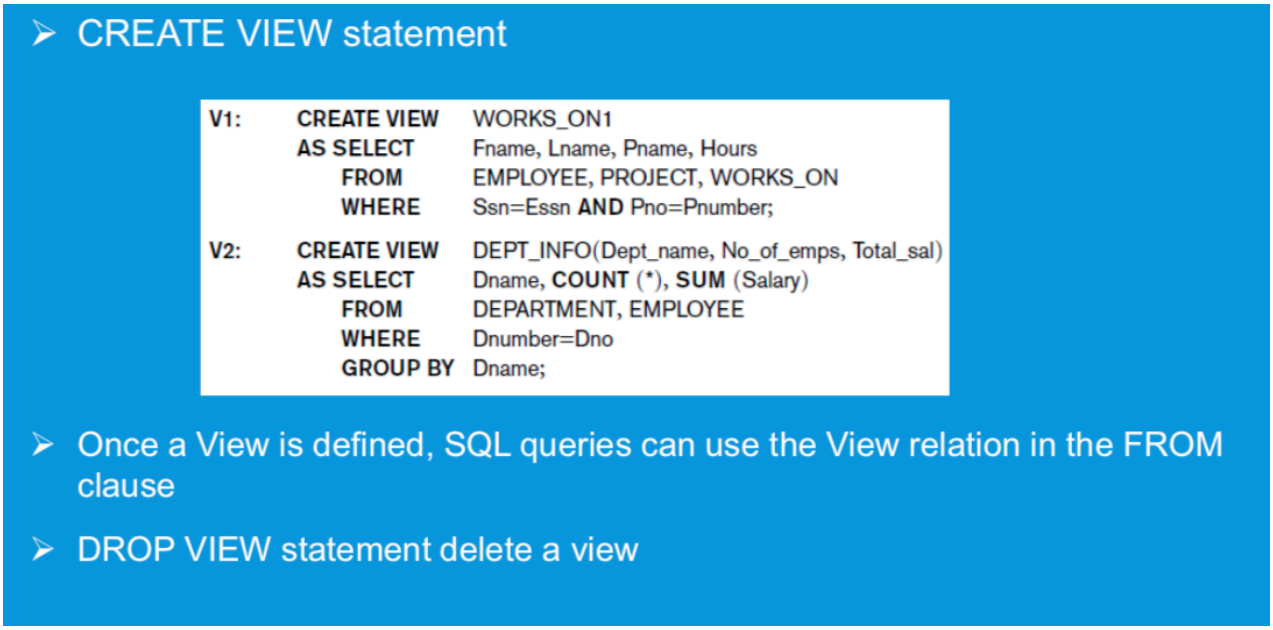
    - `WHERE` limit the tuples to which functions applies (correspond to WORKS_ON), `HAVING` serves to choose groups
  - Example 2: For **each department that has more than five employees**(前置条件), retrieve the department number and the number of its employees who are marking more than $40,000.

```sql
SELECT Dname, COUNT(*)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber = Dno AND Salary>40000
GROUP BY Dname
HAVING COUNT(*)>5
```

- This is **incorrect** because it will select only departments that have more than five employees who earn more than $40,000. This is because the WHERE clause is executed first to select individual tuples, and then the HAVING clause is applied later to select individual groups of tuples.

```sql
SELECT Dname, COUNT(*)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber = Dno AND Salary>40000 AND Dno in
    (
          SELECT Dno
            FROM EMPLOYEE
            GROUP BY Dno
            HAVING COUNT(*)>5
        )
GROUP BY Dnumber;
```

- Views(Virtual Tables)
  - A VIEW is a virtual table that does not physically exist.
  - A view contains rows and columns, just like a real table. The fields in a view are fields from **one or more real tables** in the database.
  - You can add SQL functions, WHERE, and JOIN statements to a view and present the data **as if the data were coming from one single table**.
  - A view always shows up-to-date data! The database engine recreates the data(automatically), using the view's SQL statement, every time a user queries a view.
  - When you update record(s) in a VIEW, it updates the records in the underlying tables that make up the View. However, most SQL-based DBMSs restrict that a modification is permitted through a view ONLY IF the view is defined in terms of ONE underlying table.



- NULL Values
  - Information can be very often incomplete in the real world
  - Unknown attributes are assigned a null value
  - One proposal to deal with NULL values is by using 3-valued logic

NOT	
T	F
U	U
F	T

AND	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

OR	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

  - Syntax: expression IS NULL
    - Expression: The value to test whether it is a null value
    - If *expression* is a NULL value, the condition evaluates to TRUE.
    - If *expression* is not a NULL value, the condition evaluates to FALSE.
- Summary:

Table 7.2 Summary of SQL Syntax

```
CREATE TABLE <table name> ( <column name> <column type> [ <attribute constraint> ]
                                { , <column name> <column type> [ <attribute constraint> ] }
                                [ <table constraint> { , <table constraint> } ] )

DROP TABLE <table name>
ALTER TABLE <table name> ADD <column name> <column type>

SELECT [ DISTINCT ] <attribute list>
FROM ( <table name> { <alias> } | <joined table> ) { , ( <table name> { <alias> } | <joined table> ) }
[ WHERE <condition> ]
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]

<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) )
                        { , ( <column name> | <function> ( ( [ DISTINCT] <column name> | * ) ) } ) )

<grouping attributes> ::= <column name> { , <column name> }

<order> ::= ( ASC | DESC )

INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { , <constant value> } ) }
| <select statement> )
```

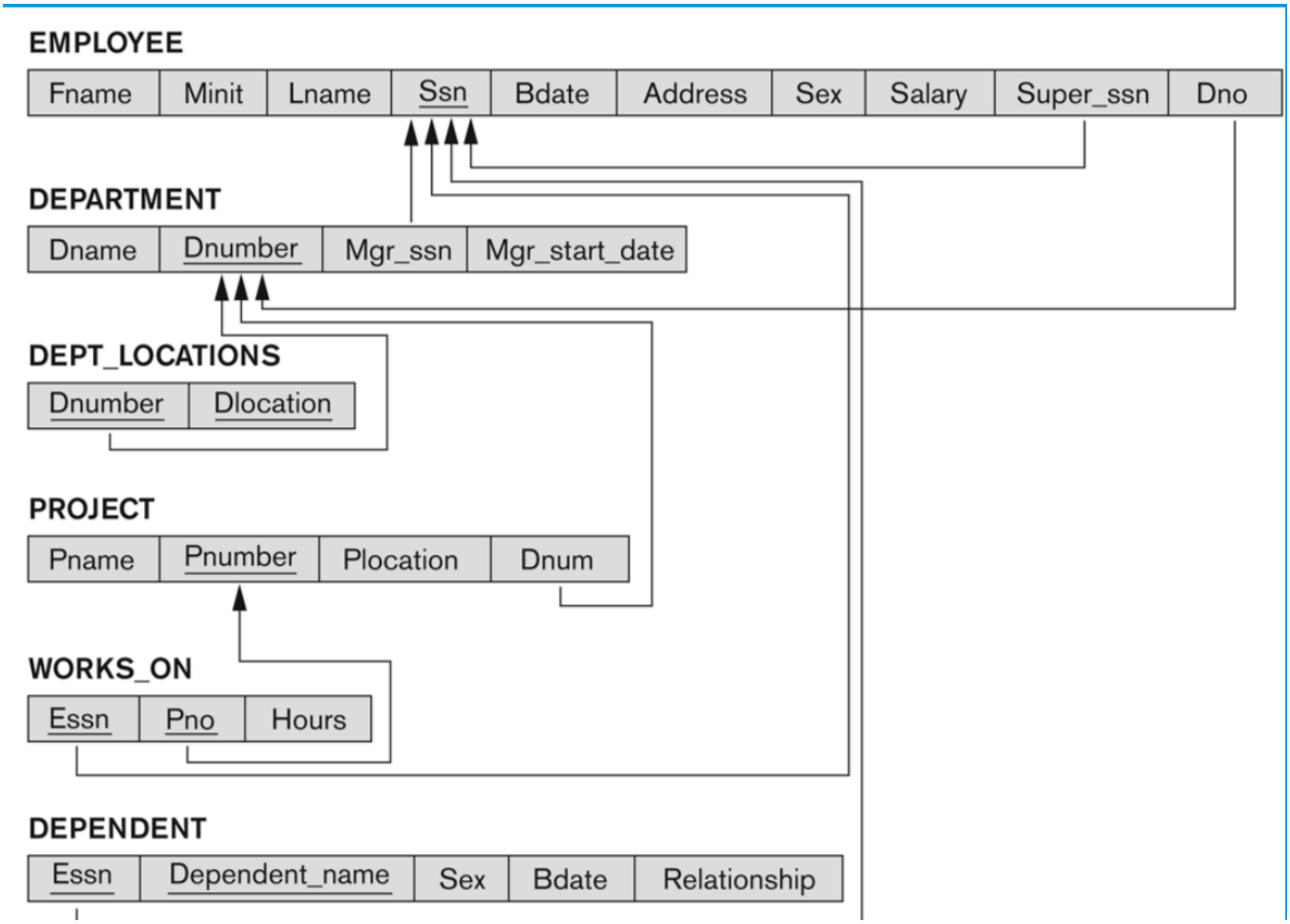
Table 7.2 Summary of SQL Syntax

```
DELETE FROM <table name>
[ WHERE <selection condition> ]

UPDATE <table name>
SET <column name> = <value expression> { , <column name> = <value expression> }
[ WHERE <selection condition> ]

CREATE [ UNIQUE] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ]

DROP INDEX <index name>

CREATE VIEW <view name> [ ( <column name> { , <column name> } ) ]
AS <select statement>

DROP VIEW <view name>
```

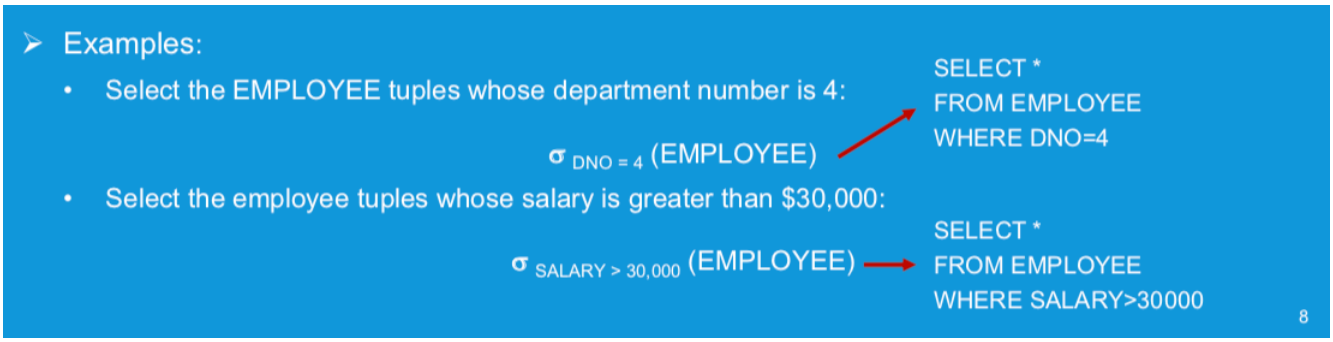
NOTE: The commands for creating and dropping indexes are not part of standard SQL.

Lecture 04: Relational Algebra

- Relational Algebra: a formal language for the relational model
  - The operations in relational algebra enable a user to **specify basic retrieval requests(or queries)**
  - Relational algebra consists of **a set of operations on relations to generate relations**
  - The result of an operation is a new relation that can be further manipulated using operations
  - A sequence of relational algebra operations forms a relational algebra expression
  - **provides a formal foundation for relational model**
- Overview
  - Unary(一元) Relational Operations
    - SELECT (symbol: σ (sigma))
    - PROJECT (symbol: π (pi))
    - RENAME (symbol: ρ (rho))
  - Relational algebra operations from set theory
    - UNION (∪), INTERSECTION (∩), DIFFERENCE (or MINUS, −)
    - CARTESIAN PRODUCT (**x**)
  - Binary Relational Operations
    - JOIN (several variations of JOIN exist) (cross product and products under some constraints)
    - DIVISION
  - Additional Relational Operations
    - OUTER JOINS, OUTER UNION
    - AGGREGATE FUNCTIONS (These compute summary of information: for example, SUM, COUNT, AVG, MIN, and MAX)
- The COMPANY relational database schema



- SELECT: Used to select **a subset of the tuples** from a relation based on a selection condition(different with the SQL select, similar to WHERE in SQL)
  - only keep tulples satisfying the condition => **horizontal partitioning**



  - Properties:
    - (1) selected relation has **the same attrbutes(columns)** with the parent relation;
    - (2) commutative(order of selected attributes doesn't matter) => A cascade (sequence) of SELECT operation may be applied in any order;
    - (3) A cascade of SELECT operations may be replaced by a **single selection with a conjunction (and) of all the conditions**

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\sigma_{<cond3>}(R)) = \sigma_{<cond1> \text{ AND } <cond2> \text{ AND } <cond3>}(R))$$

    - The **number of tuples** in the result of a SELECT operation is **less than (or equal to)** the number of tuples in the input relation R.
    - The fraction of tuples selected by a selection condition is called the **selectivity of the condition**
- Unary Relational Operations: PROJECT(π) (similar to SELECT in SQL)

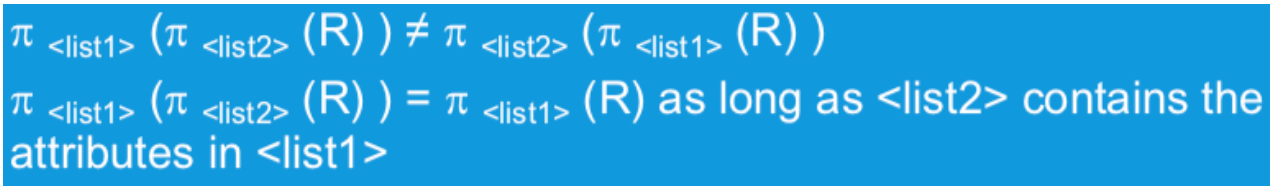
- Keeps certain attributes from a relation and discards the other attributes: create **vertical partitioning** => spercified attributes are kept in each tuple and the other attributes in each tuple are discarded

- General form of the project operation is $\pi_{<attribute\ list>}(R)$
- **Removes duplicate tuples**: Result of the project operation must be a mathmetical set(do not allow duplicate) of tuples => equivalent to `SELECT DISTINCT` in SQL
- Number of tuples in the result of projection is always **less or equal** to the number of tuples(duplicate removed => `SELECT DISTINCT`)
- If the list of attributes **includes a key of R**, then the number of tuples in the result of PROJECT is equal to the number of tuples in R, because it uniquely determine each tuple (i.e. they shares same column space => keys can be treated as bases in Linear algebra)
  - $\pi_{Lname,Fname,Salary}(R)$ (with key)and $\pi_{Sex,Salary}(R)$ (without key)

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

- Not commutative:
  - list1 must be part of list2



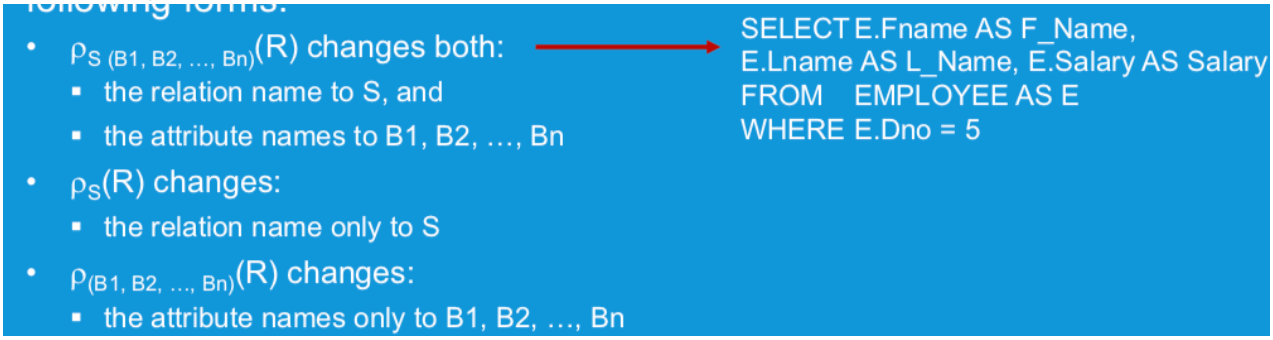
- Relational Algebra Expressions
  - Apply several relational algebra operations one after the other: Either we can write the operations as a single relational algebra expression by nesting the operation(**must select before projection** because we need to use the DNO as the condition)
  - **if you do not know the order, selecting before projecting is always correct**

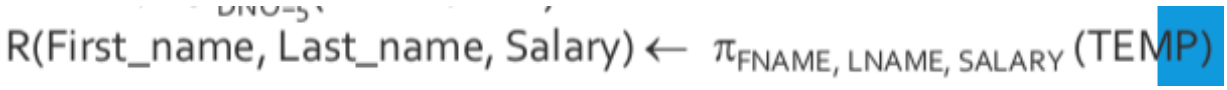
  - Or we can apply one operation at a time and create intermediate result relations

$$DEP5\_EMPS \leftarrow \sigma_{DNO=5}(EMPLOYEE)$$
$$RESULT \leftarrow \pi_{FNAME, LNAME, SALARY}(DEP5\_EMPS)$$

- In the latter case, we must give names to the relations that hold the intermediate results

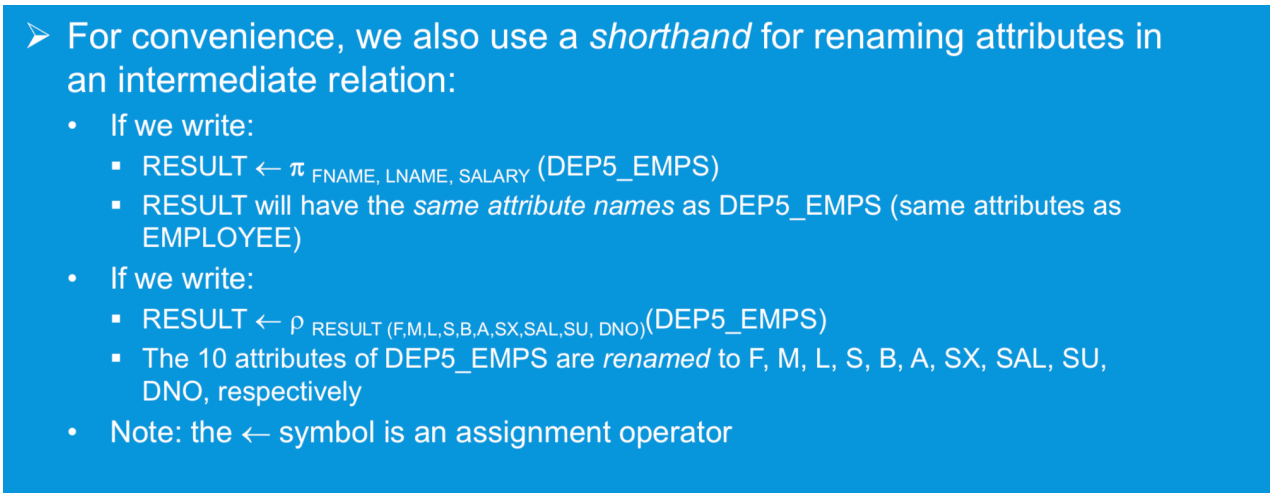
- Unary relational Operations: RENAME(ρ)
  - Forms of expression





Create R and attributes(similar to class init in cpp)

  - *Shorthand* for renaming



- Set Theory : UNION Operation ($R \cup S$)
  - Either in R or S; remove duplicate
  - R and S must be "type compatible" (double and int are different types but compatible) (same for $R \cap S$ and R-S)
    - R and S have same number of attributes
    - Domains of each corresponding pair of attr must be type compatible
    - Resulting relation has the same attr names as the **first operand relation** as default
  - e.g. Retrieve all employees that are either in Dept 5 or supervise an employee in Dept 5

$$DEP5\_EMPS \leftarrow \sigma_{DNO=5}(EMPLOYEE)$$
$$RESULT1 \leftarrow \pi_{SSN}(DEP5\_EMPS)$$
$$RESULT2(SSN) \leftarrow \pi_{SUPERSSN}(DEP5\_EMPS)$$
$$RESULT \leftarrow RESULT1 \cup RESULT2$$

    - SSN and SUPERSSN are compatible
- INTERSECTION and DIFFERENCE (∩ and -)
    - $R \cap S = (R \cup S) - (R - S) - (S - R)$
- Some properties of set operation (3 operations above)
    - commutative(except minus)
    - associative
- CARTESIAN (or CROSS) PRODUCT
    - $R(A_1, A_2, \ldots, A_n)\ X\ S(B_1, B_2, \ldots, B_m))$
    - In order take all the possible combination
    - result is a relation Q with degree $col_A + col_B$ attributes and $row_A * row_B$ tuples
    - R and S **do not have to be compatible**
    - meaningless because some tuple do not exist in real world, but meaningful followed by other operations
        - e.g. all combination of part of female employee information and dependent

$$FEMALE\_EMPS \leftarrow \sigma_{SEX='F'}(EMPLOYEE)$$
$$EMPNAMES \leftarrow \pi_{FNAME, LNAME, SSN}(FEMALE\_EMPS)$$
$$EMP\_DEPENDENTS \leftarrow EMPNAMES \times DEPENDENT$$

        - meaningful after checking the reference of SSN(the name of female employees and their dependents)

$$ACTUAL\_DEPS \leftarrow \sigma_{SSN=ESSN}(EMP\_DEPENDENTS)$$
$$RESULT \leftarrow \pi_{FNAME, LNAME, DEPENDENT\_NAME}(ACTUAL\_DEPS)$$

- Binary Relational Operations: **JOIN** (denote by $\bowtie$)
    - represents a sequence of CROSS PRODUCT followed by SELECT

A special operation, called JOIN combines this sequence into a single operation

The general form of a join operation on two relations R(A1, A2, . . ., An) and S(B1, B2, . . ., Bm) is:

$$R \bowtie_{\text{<join condition>}} S$$

    - similar property with CROSS PRODUCT but has number of tuples is less than or equal to $row_A * row_B$
    - The general case of JOIN operation is called a **Theta-join**: R S

    - **The join condition is called theta**
    - Theta can be any general boolean expression on the attributes of R and S
    - Most join conditions involve one or more equality conditions “AND”ed together
- **EQUIJOIN**: join conditions with equality comparisons only(the only operator used is =)
    - In the result of an EQUIJOIN, we always have **one or more pairs of attributes that have identical values** in every tuple(because it is the join condition)
- **NATURAL JOIN** Operation(denoted by *)
    - created to get rid of the second(superfluous(多余的)) attribute in an EQUIJOIN condition(**attributes as part of condition in table 2 are not included in the result table**)
    - The standard definition of natural join **requires that the two join attributes, or each pair of corresponding join attributes**, have the **same name in both relations**
    - The same name attributes are all included
    - If this is not the case, a **renaming operation is applied first** or specify the join conditions
    - e.g.: For 2 relations(DEPARTMENT and DEPT_LOCATIONS) has only 1 same name attributes, sufficient to write:

      DEPT_LOCS <- DEPARTMENT*DEPT_LOCATIONS
    - Only attribute with the same name is DNUMBER -> An implicit join condition is created based on this attribute:

      DEPARTMENT.DNUMBER=DEPT_LOCATIONS.DNUMBER
    - Another example: Q <- R(A,B,C,D) * S(C,D,E)
        - The implicit join condition includes each pair of attributes with the same name,

          “AND”ed together: R.C=S.C AND R.D=S.D
        - Result keeps only one attribute of each such pair: Q(A,B,C,D,E)
- **Complete Set** of Relational Operations: SELECT, PROJECT, UNION, DIFFERENCE, RENAME, and CARTESIAN PRODUCT X (any other relational algebra expression can be expressed by a combination of these five)
- DIVISION: The division operation is applied to two relations
    - R(Z) ÷ S(X), where X is a subset of Z (both of them are set of attributes)
    - Let Y = Z - X (and hence Z = X ∪ Y); that is, let Y be the set of attributes of R that are not attributes of S
    - The **result of DIVISION is a relation T(Y)** that includes a tuple t if tuples t_R appear in R with t_R [Y] = t, and with t_R [X]=t_s for every tuple t_s in S (**S X T⊆R**)
    - For a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with every tuple in S

The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

(a)

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

S

A
a1
a2
a3

T

B
b1
b4

- summary

Table 8.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{\text{<selection condition>}}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{\text{<attribute list>}}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\text{<join condition>}} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\text{<join condition>}} R_2$, OR $R_1 \bowtie_{\text{(<join attributes 1>), (<join attributes 2>)}} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{\text{<join condition>}} R_2$, OR $R_1 *_{\text{(<join attributes 1>), (<join attributes 2>)}} R_2$ OR $R_1 * R_2$

Table 8.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

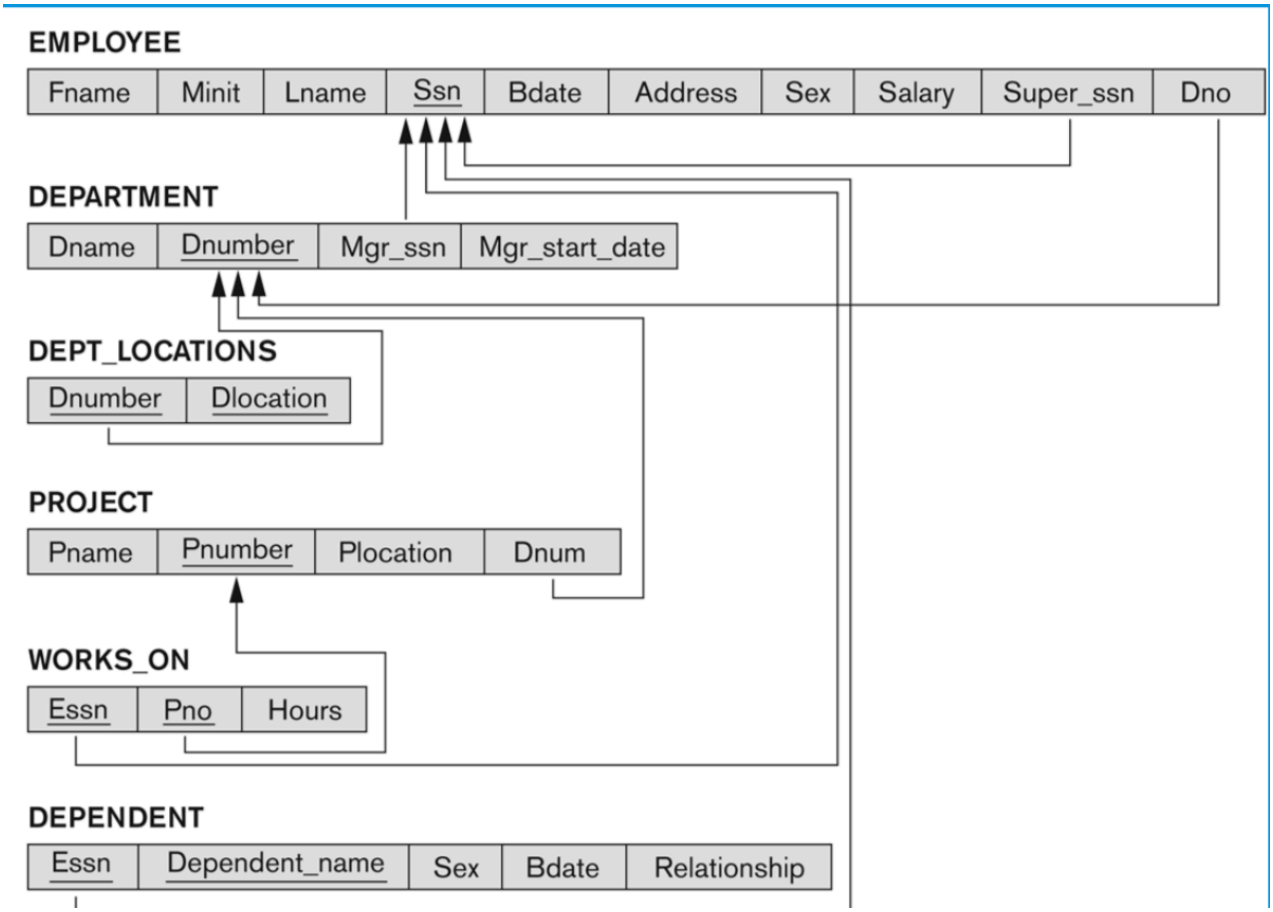
Lecture 05 Integrity Constraints

- Integrity Constraints (完整性限制)
  - Constraints determine which values are permissible(许可) and which are not in the database (table)
    - Constraints are conditions that must hold on **all valid relation states**
  - A relational database schema S is a set of relation schemes S = {R 1 , R 2 , ..., R n } and a set of integrity constraints IC
  - Valid state vs. invalid state
    - Valid state: a state that satisfies all the constraints in the defined set of integrity constraints
    - Invalid state: A database state that **does not obey all** the integrity constraints
- Three Main Types of Relational Integrity Constraints
  - Inherent or Implicit Constraints: characteristics of relations, e.g., no duplicate tuples
  - **Schema-based** or Explicit Constraints: Expressed in schemas by DDL (i.e., SQL) => added when creating table
  - Application-based or Semantic(语意的) constraints: These are **beyond the expressive power of the model** (i.e., cannot be expressed in the schemas of the data model) and must be specified and enforced by the application programs(business rules, laws, which cannot be checked in the data base)
- Schema-based Constraints
  - There are three main types of schema-based constraints that can be expressed in the relational mode
    - Key constraints (key must be minimal)
    - Entity integrity constraints (key must be not null)
    - Referential integrity constraints (between 2 relation R1 and R2 a **foreign key** must be one-to-one , must have same domain with primary key of referenced relation and must match one of the primary key or be NULL)
  - Another schema-based constraint is the **domain constraint**
    - Every value in a tuple **must be from the domain of its attribute** (or it could be null, if allowed for that attribute) (domain is determined in SQL when create the table)
- **Keys** of Relations: Example

- A **functional dependency (FD)** on a relation R is a statement of the form: if two tuples of R agree on attributes $\{A_1, A_2, \ldots, A_n\}$ (i.e., the tuples have the same values in their corresponding components for **each of** these attributes), then they **must also** agree on another attribute, "B", $\{A_1, A_2, \ldots, A_n\} \rightarrow$ B (i.e. B is dependent on the set)
- A set of one or more attributes $\{A_1, A_2, \ldots, A_n\}$ is a key for a relation if:
  - The attributes *functionally determine* **all other attributes** of the relation (can be treated as bases of the relation)
  - Relations are sets. It is **impossible** for two distinct tuples of R to agree on all $A_1, A_2, \ldots, A_n$ (if agree on the set, will agree on the whole tuple, which means duplicate and violate the implicit constraints)
  - No proper subset of $\{A_1, A_2, \ldots, A_n\}$ functionally determines all other attributes of R, i.e., a **key must be minimal**
- e.g. Movies(title, year, length, type, studioName, starName): title, year, starName → length, type, studioName
  - Attributes {title, year, starName} form a key for the relation Movie
  - Suppose two tuples agree on these three attributes: title, year, starName
  - They must agree on the other attributes, length, type and studioName
- No proper subset of {title, year, starName} functionally determines all other attributes
  - {title, year} does not determine starName since many movies have more than one star
  - {year, starName} is not a key because we could have a star in 2 movies in the same year

- Key Constraints => what is primary key (minimal+min size)
  - **super key**: A **set of attributes that contains a key** is called a superkey
  - It is a set of attributes super key SK, e.g., {A 1 , A2 } of R with the following conditions:
    - No two tuples in any valid relation state r(R) will have the same value for SK
    - For any distinct tuples t1 and t2 in r(R), t1[SK] $\neq$ t2[SK] (i.e., different SK)
  - **Key** is also superkey (minimal superkey): For a key(minimal) remove any attributes, it will not be a superkey
  - If a relation has several candidate keys(cannot be divided), one is chosen arbitrarily to be the primary key(uniquely identify each tuple in a relation)
  - **General rule**: choose the key with **smallest size** as the primary key
- Entity Integrity (完整性) => Primary Key cannot be null
  - The primary key attributes PK of each relation schema R **cannot have null values** in any tuple of R
    - t[PK] $\neq$ null for any tuple t in R because it need to be used to identify the tuple
    - if PK has several attributes, null is not allowed in any of these attributes
  - Note: Other attributes of R **may be** constrained to disallow null values, even though they are not members of the primary key (domain constraint)
- Referential Integrity
  - Key and entity integrity constraints are specified on **individual relations**
  - Referential integrity is a constraint **involving two relations**
    - To specify a relationship among tuples in two relations
    - The referencing relation and the referenced relation (R 1 → R 2)
  - Tuples in the referencing relation R 1 have attributes **FK (called foreign key attributes)** that reference the primary key attributes PK of the referenced relation R 2 if it satisfies:
    - The attributes in FK **have the same domain(s) as the primary key attributes PK of R2**

- Displaying a Relational Database Schema and its Constraints
  - Each relation schema can be displayed as **a row of attribute names**
  - The name of the relation is **written above the attribute names**
  - The primary key attribute (or attributes) will be **underlined**
  - A foreign key (referential integrity) constraints is **displayed as a directed** arc (arrow) from the foreign key attributes to the referenced table
  - Next slide shows the COMPANY relational schema diagram with referential integrity constraints




```
CREATE TABLE EMPLOYEE
        ( FNAME             VARCHAR(15)         NOT NULL ,
          MINIT             CHAR ,
          LNAME             VARCHAR(15)         NOT NULL ,
          SSN               CHAR(9)             NOT NULL ,
          BDATE             DATE
          ADDRESS           VARCHAR(30) ,
          SEX               CHAR ,
          SALARY            DECIMAL(10,2) ,
          SUPERSSN          CHAR(9) ,
          DNO               INT                 NOT NULL ,
      PRIMARY KEY (SSN) ,
      FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN) ,
      FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER) ) ;
CREATE TABLE DEPARTMENT
        ( DNAME             VARCHAR(15)         NOT NULL ,
          DNUMBER           INT                 NOT NULL ,
          MGRSSN            CHAR(9)             NOT NULL ,
          MGRSTARTDATE      DATE ,
      PRIMARY KEY (DNUMBER) ,
      UNIQUE (DNAME) ,
      FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN) ) ;
CREATE TABLE DEPT_LOCATIONS
        ( DNUMBER           INT                 NOT NULL ,
          DLOCATION         VARCHAR(15)         NOT NULL ,
      PRIMARY KEY (DNUMBER, DLOCATION) ,
      FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER) ) ;
```

```
CREATE TABLE PROJECT
        ( PNAME             VARCHAR(15)         NOT NULL ,
          PNUMBER           INT                 NOT NULL ,
          PLOCATION         VARCHAR(15) ,
          DNUM              INT                 NOT NULL ,
      PRIMARY KEY (PNUMBER) ,
      UNIQUE (PNAME) ,
      FOREIGN KEY (DNUM) REFERENCES DEPARTMENT(DNUMBER) ) ;
CREATE TABLE WORKS_ON
        ( ESSN              CHAR(9)             NOT NULL ,
          PNO               INT                 NOT NULL ,
          HOURS             DECIMAL(3,1)        NOT NULL ,
      PRIMARY KEY (ESSN, PNO) ,
      FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN) ,
      FOREIGN KEY (PNO) REFERENCES PROJECT(PNUMBER) ) ;
CREATE TABLE DEPENDENT
        ( ESSN              CHAR(9)             NOT NULL ,
          DEPENDENT_NAME    VARCHAR(15)         NOT NULL ,
          SEX               CHAR ,
          BDATE             DATE ,
          RELATIONSHIP      VARCHAR(8) ,
              PRIMARY KEY (ESSN, DEPENDENT_NAME) ,
              FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN) ) ;
```

15

- Referential Integrity
  - Referential integrity constraints typically arise from the **relationships among the entities represented by the relation**

    - For example, in the EMPLOYEE relation, the attribute Dno refers to DEPARTMENT for which an employee works. We designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT.
    - **The constraint** (value matches a tuple in another relation or is null): A value of Dno in any tuple t1 of the EMPLOYEE relation **must match** a value of the primary key of DEPARTMENT, Dnumber, in the same tuple t2 of the DEPARTMENT relation
    - The **value of Dno can also be NULL** if the employee **does not belong to a department** or will be assigned to a department later.
- Update Operations on Relations
    - Update operations: INSERT / DELETE / MODIFY a tuple (state changing operations)
    - Integrity constraints **should not be violated by the update operations**
    - Several update operations **may have to be grouped together**
    - Updates **may propagate to cause other updates automatically**. This may be necessary to maintain integrity constraints
- Possible Violations for Update Operations
    - DELETE may violate only **referential integrity**: If the primary key value of the tuple being deleted is referenced from other tuples in the database
    - INSERT may violate any of the constraints(INSERT any entry break the law)
        - **Domain constraint**: if one of the attribute values provided for the new tuple is not of the specified attribute domain
        - **Key constraint**: if the value of a key attribute in the new tuple **already exists** in another tuple in the relation (can be duplicate)
        - **Referential integrity**: if a foreign key value in the new tuple references a primary key value that does not exist in the referenced relation
        - **Entity integrity**: if the primary key value is null in the new tuple
- Integrity Violation: In case of integrity violation, several actions can be taken:
    - **Cancel the operation** that causes the violation
    - Perform the operation but **inform the user** of the violation
    - **Trigger additional updates**(automatically) so the violation is corrected
    - Execute a **user-specified error-correction routine**
- Adding constraints in SQL

```sql
CREATE TABLE TOY
(   toy_id NUMBER(10),
  description VARCHAR(15) NOT NULL,
  purchase_date DATE,
  remaining_qnt NUMBER(6)
);
CREATE TABLE TAB1(
  col1 NUMBER(10) PRIMARY_KEY,
    col2 NUMBER(4) NOT NULL,
    col3 VARCHAR(5) REFERENCES zipcode(zip) ON DELETE CASCADE,
    --CASCADE: parent delete => child delete (called cascade delete in Oracle)
    col4 DATE,
    col5 VARCHAR(20) UNIQUE,
    --A unique constraint is a single field or combination of fields that uniquely defines a record. => cannot be duplicate
    col6 NUMBER(5) CHECK(col6<100)
    -- specify a condition(domain) on each row in the table
```

```
);
```

- Reference Constraints in SQL

```
CREATE TABLE COUNTRY
(cntry_cd       VARCHAR(3)                  NOT NULL,
 cname          VARCHAR2(32)                NOT NULL,
 ename          VARCHAR2(32)                NOT NULL,
 curr_cd        VARCHAR(3)                  NOT NULL,
 upd_dt         DATE DEFAULT SYSDATE        NOT NULL ,
 upd_uid        VARCHAR2(16)                NOT NULL);

CREATE TABLE EXCHANGE
(exchg_cd       VARCHAR2(8)                 NOT NULL,
 cname          VARCHAR2(32)                NOT NULL,
 ename          VARCHAR2(32)                NOT NULL,
 cntry_cd       CHAR(3)                     NOT NULL);
```

  - Add constraints to tables COUNTRY and EXCHANGE
    - ALTER TABLE COUNTRY ADD CONSTRAINT PK_country PRIMARY KEY(cntry_cd);
    - ALTER TABLE EXCHANGE ADD CONSTRAINT PK_exchange PRIMARY KEY(exchg_cd);
    - ALTER TABLE EXCHANGE ADD CONSTRAINT FK_exchg_cntry FOREIGN KEY(cntry_cd) REFERENCES COUNTRY(cntry_cd);.

Lecture 6 : Functional Dependency & Normalization

Functional Dependency

- Functional Dependency
  - Functional dependency is a constraint between two sets of attributes from the database：For example, deptno and dname in DEPARTMENT, if you know the department number, you know the department name
  - A functional dependency denoted by X → Y specifies a constraint on the possible tuples between two sets of attributes X and Y that are subsets of a relation R that can form a relation state r of R (R is a relation(table))
    - The constraint is that, for any two tuples t1 and t2 in r that have t1[X] = t2[X], they must also have t1[Y] = t2[Y]
    - The values of the Y component of a tuple in r depend on, or are determined by the values of the X component
    - If you know his student ID, then I know his name (student ID → student Name)

- Formal definition :

  > ➢ Let R be a relation schema, and $\alpha \subseteq R$, $\beta \subseteq R$ (i.e., α and β are sets of R's attributes).
  >
  > ➢ We say $\alpha \rightarrow \beta$, if in any relation instance r(R), for all pairs of tuples t_1 and t_2 in r, we have $(t_1[\alpha] = t_2[\alpha]) \rightarrow (t_1[\beta] = t_2[\beta])$

  - e.g.Movies(title, year, length, type, studioName, starName): {title, year, starName} → {length, type, studioName}
  - Attributes {title, year, starName} form a key for the relation Movie
  - If two tuples agree on these three attributes, title, year, and starName, they must agree on the other attributes, length, type and studioName.
    - No **proper subset** of {title, year, starName} functionally determines all other attributes(**in the first set**)
    - {title, year} does not determine starName since many movies have more than one star
    - {year, starName} is not a key because we could have a star in two movies in the same year
  - Can it be {title, year, starName, length} → type? Yes
- Candidate Key (minimal super key)
  - If a constraint on R states X is a candidate key of R, then X → Y for any subset of attributes Y of R
  - A candidate key **uniquely identifies a tuple**
  - The values of all remaining attributes are determined
  - If X → Y in R, this does not say whether or not Y → X in R
    - {length, type, studioName} → {title, year, starName}? No
  - A functional dependency is property of the semantics (語義) or meaning of the attributes
- Trivial Functional Dependency (right hand side is a subset)
  - Some functional dependencies are "trivial", since they are always satisfied by all relations:
    - E.g.,A→A,AB→A
    - E.g., {Ename, Salary} → Ename
  - A functional dependency is trivial if and only if the right-hand side (the dependent) is a subset of the left-hand side (the determinant)
- Inference Rules for FDs

  > ➢ Given a set of FDs F, we can infer additional FDs that hold whenever the FDs in F hold
  >
  > ➢ Armstrong's inference rules:
  > - **IR1. (Reflexivity) If $Y \subseteq X$ (i.e., Y is a subset of X), then $X \rightarrow Y$**
  > - **IR2. (Augmentation) If $X \rightarrow Y$, then $XZ \rightarrow YZ$ (Note: XZ stands for $X \cup Z$)**
  > - **IR3. (Transitivity) If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$**

  - IR1(trivial FD), IR2(add to both side), IR3(transitivity) form a sound and complete set of inference rules
    - Sound: These are true
    - Complete: other true rules can be deducted from these rules
      - X -> XY; XY->YZ;X->YZ

- Some additional inference rules that are useful:
  - **Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$**
    - Since $X \rightarrow YZ$ (given) and $YZ \rightarrow Y$ (reflexivity), $X \rightarrow Y$ (transitivity)
    - Since $X \rightarrow YZ$ (given) and $YZ \rightarrow Z$ (reflexivity), $X \rightarrow Z$ (transitivity)
  - **Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$**
  - **Pseudo transitivity: If $X \rightarrow Y$ and $YW \rightarrow Z$, then $XW \rightarrow Z$**
    - Since $YW \rightarrow Z$ (given) and $XW \rightarrow YW$ (augmentation), $XW \rightarrow Z$ (transitivity)

- Suppose we are given a schema R with attributes A, B, C, D, E, F and the FDs are:
  - $A \rightarrow BC$
  - $B \rightarrow E$
  - $CD \rightarrow EF$
  - Show that FD: $AD \rightarrow F$ holds

Solution

1. $A \rightarrow BC$ (given)
2. $A \rightarrow C$ (decomposition from 1)
3. $AD \rightarrow CD$ (augmentation from 2)
4. $CD \rightarrow EF$ (given)
5. $AD \rightarrow EF$ (transitivity from 3 and 4)
6. $AD \rightarrow F$ (decomposition from 5) (proved)

- Closure of a set of FDs: Given a set of FD says F, F+ is closure of the set which contains all FDs implies by F

- $R = \{A, B, C, D, E, F\}$
- FDs in F:
  - $A \rightarrow B$
  - $A \rightarrow C$
  - $CD \rightarrow E$
  - $CD \rightarrow F$
  - $B \rightarrow D$
- Some members of F^+:
  - $A \rightarrow D$
  - $A \rightarrow BC$
  - $AD \rightarrow E$
  - $CD \rightarrow EF$
  - …

- Closure of Attribute Sets
  - The set of all FDs logically implied by F using inference rules is the closure of F, denoted by F+, where the F itself is a set of FDs

- The closure of X under F (denoted by X^+) is the set of attributes that are functionally determined by X under F (X and X^+ are a set of attributes):

$$X \rightarrow Y \text{ in } F^+ \leftrightarrow Y \subseteq X^+$$

- If X^+ consists of all attributes of R, X is a superkey for R. From the value of X, we can determine the values of the whole tuple.
- For example, given Ssn, if Ssn $\rightarrow$ Ename, then Ename is part of Ssn^+, i.e., Ssn^+ = {Ssn, Ename, …}
- If Ssn $\rightarrow$ Dmgr_ssn, then Dmgr_ssn is part of Ssn^+, i.e., Ssn^+ = {Ssn, Ename, Dmgr_ssn, …}

- Recall : super key is the one contains candidate key, but not supposed to be minimal
- Attribute sets is the set contains all attributes that one attribute can determine

- The algorithm (iteratively add the entry in **second set (right hand side) of FD where first set (left hand side) in FD is already in X** into the closure until nothing can be added)

- Input
  - R: a relation schema
  - F: a set of FDs
  - X $\subset$ R: the set of attributes for computing the closure
- Output
  - X^+ is the closure of X with respect to F

$X_0 = X$
Repeat
$X_{i+1} = X_i \cup Z$, where Z is the set of attributes such that $Y \rightarrow Z$ in F and $Y \subset X_i$
Until $X_{i+1} = X_i$
Return X_{i+1}

- e.g.

- Given a schema R={A, B, C, D, E, F}, F= {A $\rightarrow$ BC, B $\rightarrow$ E, E $\rightarrow$ CF, CD $\rightarrow$ EF}, and X={A}.
- X_0={A}
  A $\rightarrow$ BC
- X_1={A, B, C}
  B $\rightarrow$ E
- X_2={A, B, C, E}
  E $\rightarrow$ CF
- X_3={A, B, C, E, F}
- Output: X^+={A, B, C, E, F}

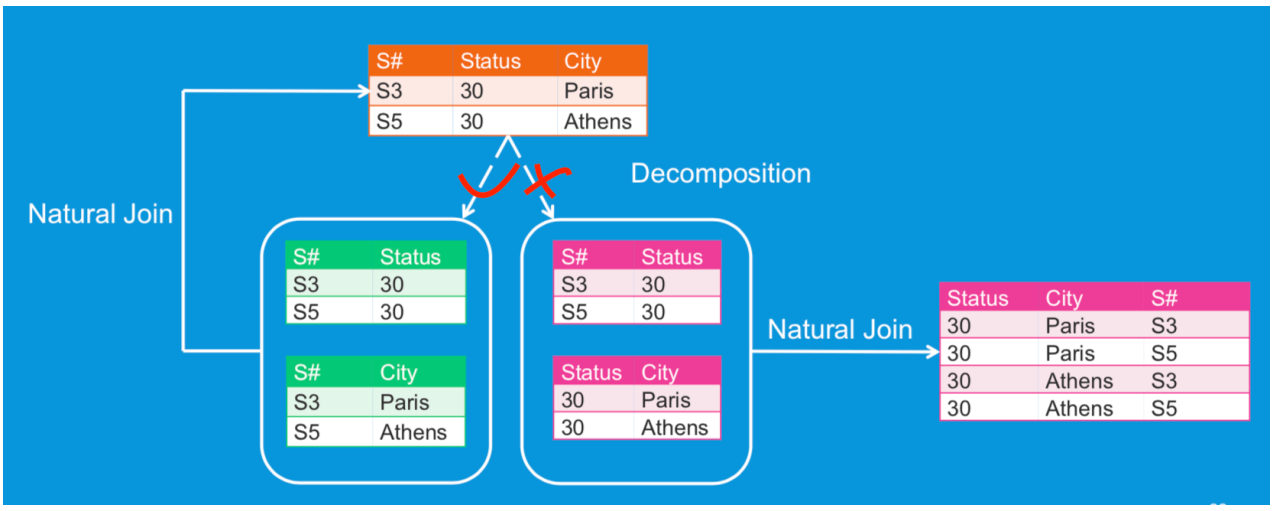
17

  - Equivalence of Sets of FDs

- A set of functional dependencies F is said to **cover** another set of functional dependency E i**f every FD in E is also in** F+(E is a subset of F+)
- Two sets of FDs F and G are equivalent if: **every FDs in F and G can be inferred from one of each others' FD**, hence F and G are equivalent if F+=G+

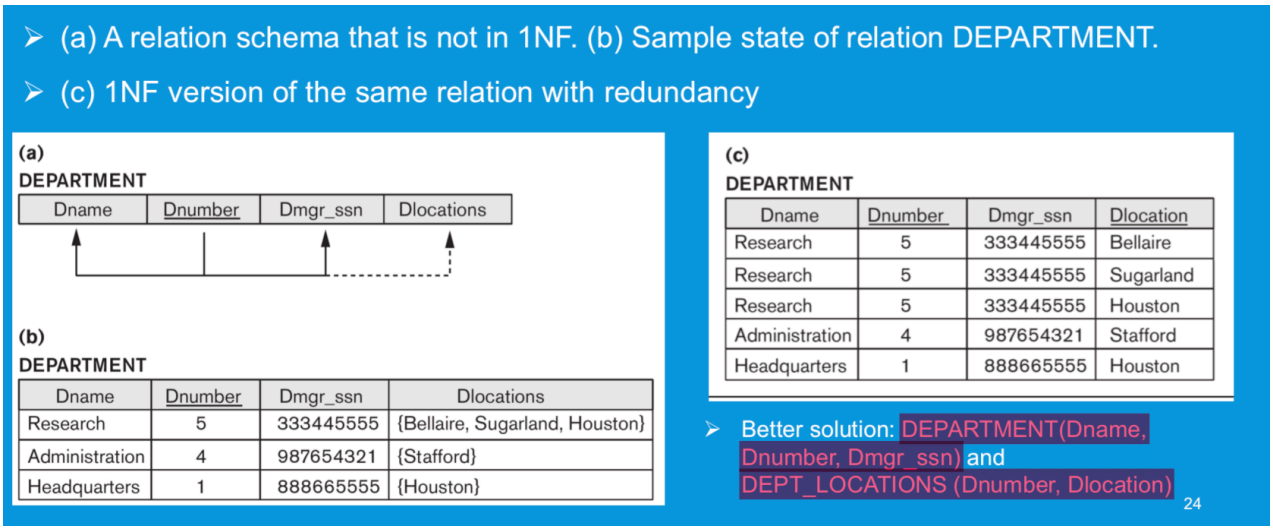
Normalization forms

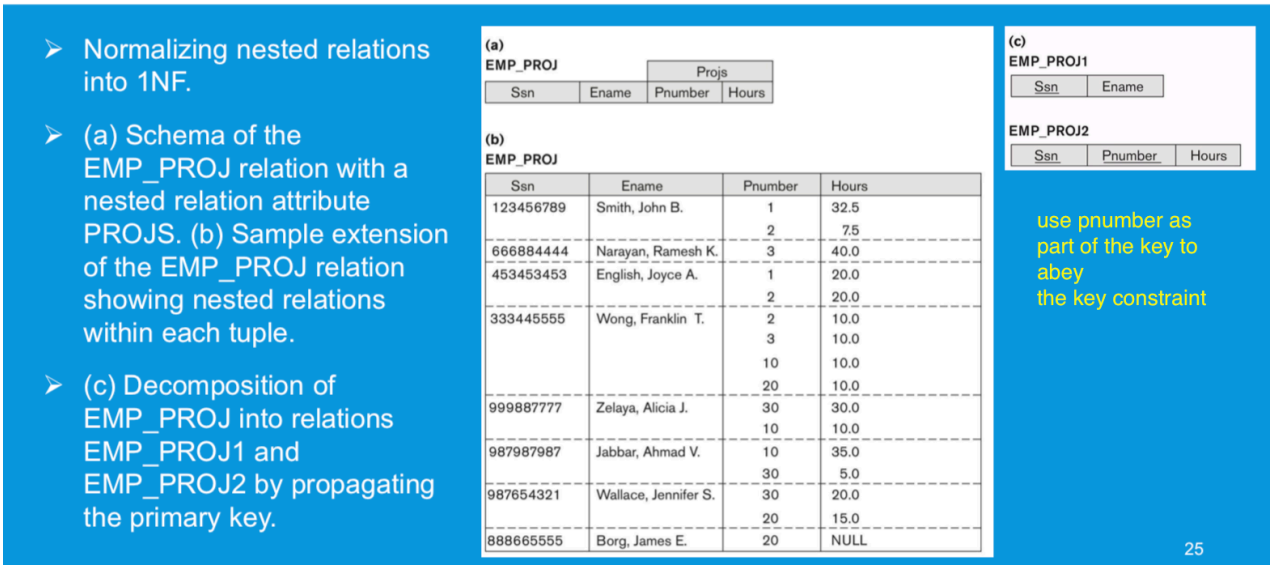
- Principle of Relational Database design
  - Logical/conceptual DB design : Schema
    - What relations (tables) are needed?
    - What their attributes should be?
  - What is a “bad” DB design?
    - **Repetition** of data/information
    - Potential inconsistency(潜在矛盾)
    - **Inability** to represent certain information
    - **Loss** of data/information => after cross product cannot construct the original table
- Normalization: Take relation schema through a series of tests to certify whether it satisfies a certain normal form
  - Analyzing the relation schema based on FD and primary keys to achieve
    - Minimizing redundancy
    - Minimizing the insertion, deletion and update anomalies (異常)
  - Properties:
    - **Non-additive** or lossless join
      - Decomposition is **reversible** and no information is loss
      - No spurious tuples (tuples that should not exist) should be generated by doing a **natural-join** (join according to the same name attributes) of any relations (extremely important)
    - Preservation of the functional dependencies
      - Ensure each functional dependency is represented in **some**(thus primary key can be broken) individual relations
      - Each of the relation must have a primary key



- First normal form (attributes must be simple):

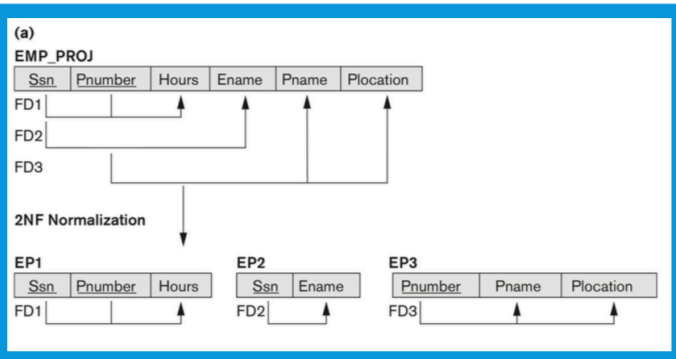
- Disallowed: **Multivalued attributes, composite attributes and their combination; multivalued attributes that themselves composite; domain of attribute must be atomic values; No repeating groups(nested relations)**





- Second Normal Form with Primary Key (attribute that is not part of candidate key cannot be determined by part of primary key)
  - **Full functional dependency**: remove one attr in the left set, the FD does not hold anymore
  - **Partial functional dependency**: If some attr removed => still hold FD
  - **Prime attribute**: An attr R is called prime attributes if it is a member of *some* candidate key of R
  - **A relation R is in 2NF iff every non-prime attr A in R is fully functional dependent on the primary key of R**
  - Change a relation to 2NF: split the relation for the non-prime **attrs that can be determined by part of the primary key** => no reduandent dependent on the primary key
  - e.g.

- The non-prime attribute Ename violates 2NF because of FD2 (i.e., is not fully functional dependent on the primary key)
- Similarly, Pname and Plocation violate 2NF because of FD3
- Solution: EP1(Ssn, Pnumber, Hours); EP2(Ssn, Ename), and EP3(Pnumber, Pname, Plocation)

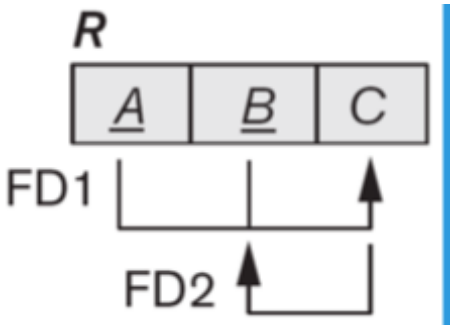


- Third Normal Form with Primary key (only super key can determine non-prime attributes, original: super key can determine all attribute => now: super key is the only type of attr sets that can determine non-prime attributes)
  - A relation schema R is in 3NF iff whenever a non-trivial (X does not have A) FD **X -> A** holds in R, either
    - X is a superkey of R **or**
    - A is a prime attribute of R
  - Why non-trivial? => X is not superkey, and A is an non-prime attribute, then (X+A)=>A always holds
  - A 3NF must follow 2NF => proof:
    - Suppose there is a relation violate 2NF, there is part of primary key determine non-prime attribute
    - The part of the primary key is not a super key since primary key is minimal
    - It violate 3NF
  - **Transitive dependent**: As we know, super key can determine all attributes, such that the existence of a super key **X** -> **Z** where **Z->Y** and Y is a set of non-prime attrs and Z is not a super key
  - 3NF = 2NF + no non-prime attrs of R is **transitively depdent on the primary key**
  - 2NF 和 3NF交集不为0
- summary of Normalization
  - both 2NF and 3NF test on the right hand side as a non-prime attribute

Table 14.1 Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation.
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

- Boyce-Codd Normal Form (part of primary key cannot be determined by set that is not a super key)
  - BCNF was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF => every BCNF relation is a 3NF, 3NF is not necessary to be BCNF
  - A relation R is BCNF iff a non-trivial function dependentcy **X->A** holds in R, then **X is a superkey of R**(only first situation in 3NF is ok)
  - Following relation is 3NF but not BCNF



  - Algorithm for BCNF Decomposition

> ➢ Let R be the initial table with FDs F and S={R}
> Until all relation schemes in S are in BCNF
> for each R in S
> for each FD X → Y that violates BCNF for R
> S = (S – {R}) ∪ (R-Y) ∪ (X,Y)
> End until
>
> ➢ When we find a table R with BCNF violation X → Y we:
> • Remove R from S
> • Add a table that has the same attributes as R except for Y
> • Add a second table that contains the attributes in X and Y

    - Primary key is possible to be broken
  - e.g. (need to consider whether it can be recovered)

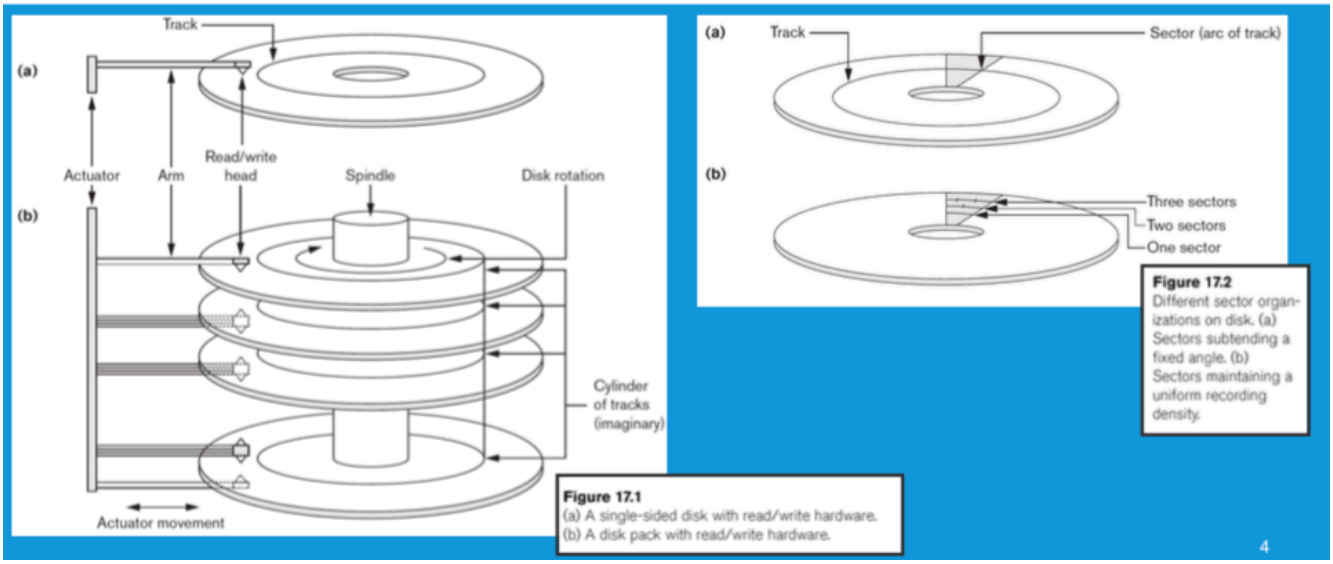
> ➢ Let us consider the relation scheme R=(A,B,C,D,E) and the FDs: {A} → {B,E}, {C} → {D}
>
> ➢ Candidate key: AC
>
> ➢ Both functional dependencies violate BCNF because the LHS is not a candidate key
>
> ➢ Pick {A} → {B,E}
> • We can also choose {C} → {D} – different choices
> • Lead to different decompositions.
> • (A,B,C,D,E) generates R_1=(A,C,D) and R_2=(A,B,E)
>
> ➢ We need to decompose R_1=(A,C,D) because of the FD {C} → {D}, so (A,C,D) is replaced with R_3=(A,C) and R_4=(C,D).
>
> ➢ Final decomposition: R_2=(A,B,E), R_3=(A,C), R_4=(C,D)

- Algorithm to find all Candidate keys

```
1. Find the attributes that are neither on the left or right side

2. Find attributes that are only on the right side

3. Find attributes that are only on the left side

4. Combine the attributes on step 1 and 3 (not 2)

// by now, step 4 contains all attributes that must be included into the candidate key

5. Test if the closures of attributes on step 4 constitutes all the attributes. If yes it is a candidate key.

6. If not, find the relation exteriors, that is the attributes not included in step 4 and step 2.

7. Now test the closures of attributes on step 4 + one attribute in step 6 one at a time. All those combinations are candidate keys if their closures constitute all the attributes.
```

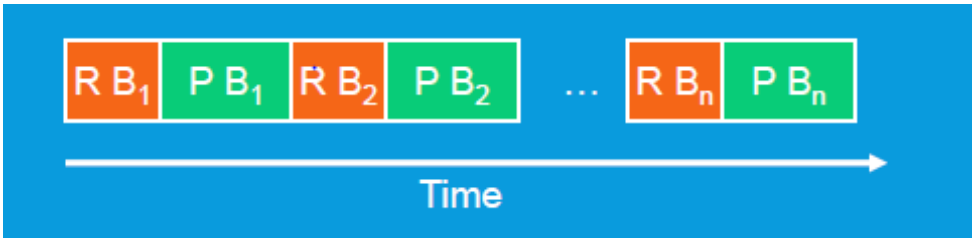
Lecture 07 Files and Hash Files

- Storage Medium for Databases:
  - Memory hierarchy
    - CPU cache > main memory > flash memory/Phase change memory > magnetic disks/optical disks
    - Slower in access delay but larger in memory size (less expensive)
  - Primary storage (volatile)
    - The storage media that can be operated **directly by the CPU**
    - Include main memory and cache memory
  - Secondary and tertiary storage (non-volatile)
    - Slower in access
    - Include magnetic **disks**, optical **disks** and flash memory
  - A database could be huge in size (several hundred GB or even larger) : Need to be resided in secondary/tertiary storage (non-volatile/persistent storage)
- Disk Storage Devices (Preferred secondary storage device for **high storage capacity and low cost**)

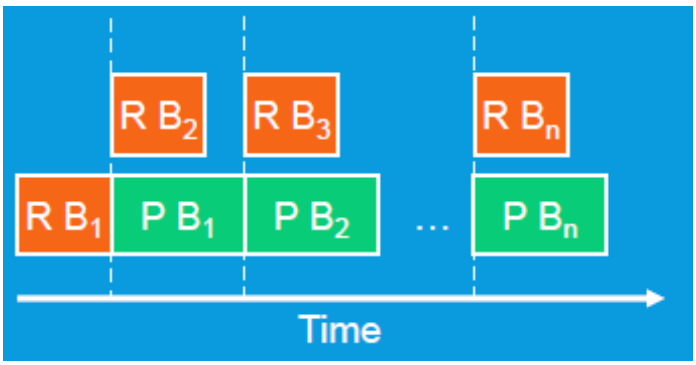


- Data are stored as **magnetized areas** on magnetic disk surfaces
- A disk pack contains several magnetic disks connected to a rotating spindle
- Disks are divided into concentric **circular tracks** on each disk surface
  - Track capacities vary typically from 4 to 50 Kbytes or more
- A track is divided into **fixed size sectors and then into blocks**
  - Typical block sizes range from B=512 bytes to B=4096 bytes
  - Whole blocks are transferred between disk and main memory for processing
- Disk pack -> Disk -> Track -> Sectors -> Blocks

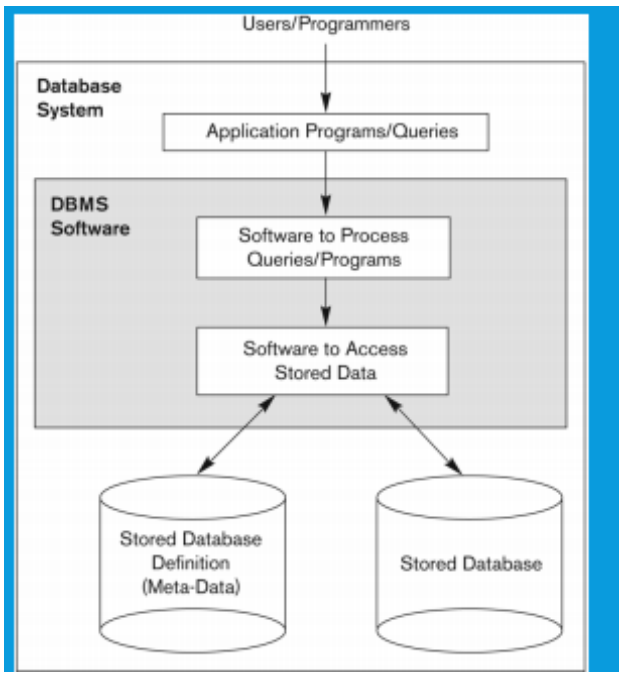
- Disk Storage Devices
  - A read-write head moves to find the track contains the block to be transferred, and **disk rotation moves the block** under the read-write head
  - To access a physical disk block:
    - identify track number (**seek time** e.g., 3 to 8 ms)
    - The block number (within the cylinder) (**rotation delay**, e.g., 2ms)
    - get the block data (**transfer delay**)
  - **Disk access delay = seek time(track) + rotational delay(block) + transfer delay(load block)**
  - First 2 delays are time counsuming
  - **Double buffering** can be used to speed up the transfer of contiguous disk blocks
- Double buffering
  - Problem : Have a file with a sequence of n blocks $B_1, B_2, \ldots, B_n$ to be **fetched and executed**, suppose R is time to read each block and P is time to process each block
  - Single buffer solution: Read and process doesn't overlap => Total time = $n(R+P)$



- Double buffer solution(**assume** $P > R$): Except the first period, other period can do the work at the same time => Total time = $R + nP$



- Simplified Database System Environment



  - DBMS(database manage system): A collection of programs that enables users to created and maintain a database
  - General-purpose software that facilitates the process of defining **constructing and manipulating databases** for **various applications**
  - Database System = DBMS Software + Database
- Database Records:
  - Database: data file(records) + meta-data(database definition)
  - **fixed/variable(changeable or not) length records**
  - The file records are of the same record type, but one or more of the fields may have multiple values for individual records; such a field is called a **repeating field** and a group of values for the field is often called a repeating group.
  - Records contains fields (attributes), field can contain **multiple attributes**
  - Fields may be fixed length or variable length: Different with fixed or variable length record, it represents the single field instead of the record
  - **Variable length fields can be mixed** into one record => Separator characters or length fields are needed so that the record can be “parsed” , **specify the variable length**
    - variable means the length is different among different records instead of meaning it can be

changed at run time
    - In (b), there are 2 variable, and three fixed contained in the variable-length fields. Fixed-length fields are fixed because they already have the specific value
    - In (c), the terminate record is a sign of variable record

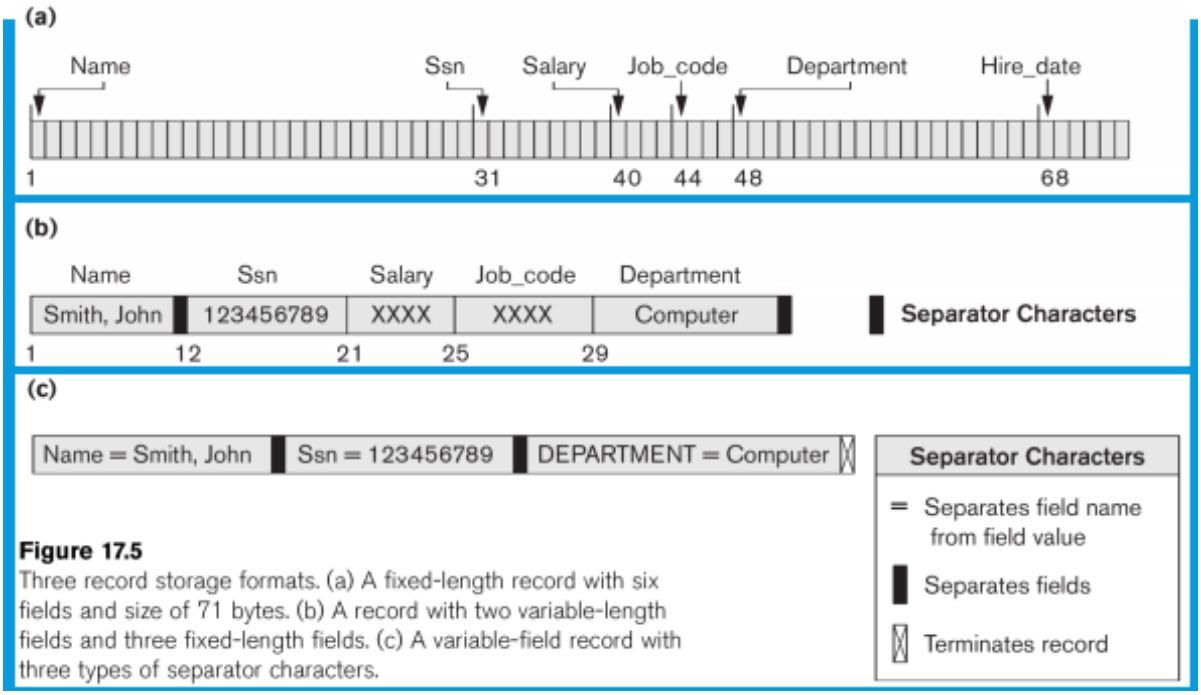
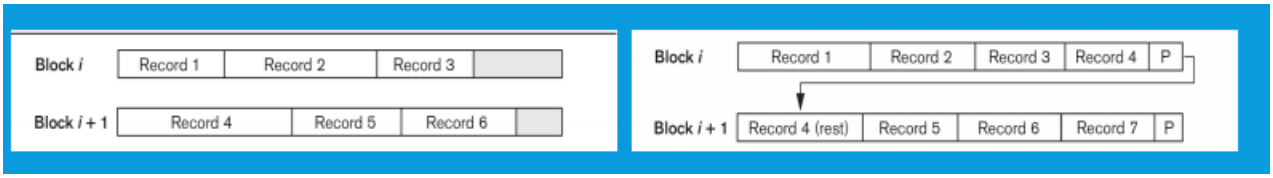


Figure 17.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

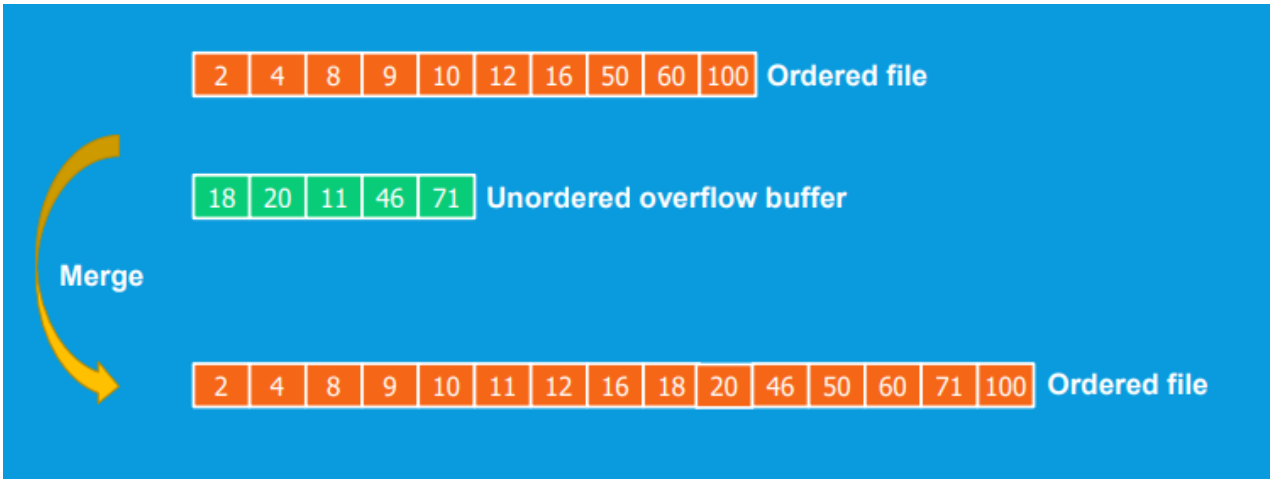
- **Blocking** : Refer to storing a number of records into one block on the disk
- Blocking factor (bfr) refers to the number of records per block
- There may be empty space in a block if an **integral number of records do not fit into one block**
- Suppose the block size is B for fixed-length records of size R with $B >= R$
- $bfr = floor(B/R)$ block per record
- The unused space in each block = $B–(bfr * R)$ bytes

- Files of Records:
    - A file (e.g., table) is a sequence of records (e.g., tuples), where each record is a collection of data values (fields)
    - A file can have fixed-length records or variable-length records
    - A file descriptor (or file header) includes information that describes the file, such as the **field names and their data types**, and the addresses of the file blocks on disk
    - File records are stored on disk blocks: The physical disk blocks that are allocated to hold the records of a file can be contiguous (one by one), linked (using pointers), or indexed (a table to describe their locations)
    - File records can be unspanned or spanned



    - Unspanned: no record can span two blocks (usually used in fixed-length records => fix length determine the concrete block)
    - Spanned: a record can be stored in **more than one block** (usually used in variable-length records => need additional information to be stored for variable-length records, such as separator characters)

- Typical operations on Files:
  - OPEN: makes the file ready for access, and associates a pointer that will refer to a current file record at each point in time
  - FIND: searches for the first file record that satisfies a certain condition, and makes it the current file record
  - FINDNEXT: searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record
  - READ: reads the current file record into **a program variable**
  - INSERT: inserts a **new record into the file**, and makes it the current file record
  - DELETE: **removes the current file record** from the file, usually by marking the record to indicate that it is **no longer valid**
  - MODIFY: changes the values of some fields of the current file record
  - CLOSE: terminates access to the file
  - REORGANIZE: reorganizes the file records. For example, the records marked “deleted” are physically removed from the file or a new organization of the file records is created
  - READ_ORDERED: reads the **file blocks in order** of a specific field of the file
- Unordered Files (heap file => Records are unordered)
  - New record inserted at the end of the file (O(1)) : Arranged in their insertion sequence
  - Linear search to find (worst case O(n), average O(n/2))
  - Reading the records in order need to sort
- Ordered Files (Sequential file)
  - File records are kept sorted by the values of an ordering field
  - insertion is expensive => can be impr
  - oved by hold a unordered temporary buffer for insertion
    - store to the buffer at first, periodically merged into the file records



  - Find (O(log(n)) => binary search

Algorithm 17.1. Binary Search on an Ordering Key of a Disk File

```
l ← 1; u ← b; (* b is the number of file blocks *)
while (u ≥ l ) do
    begin i ← (l + u) div 2;
    read block i of the file into the buffer;
    if K < (ordering key field value of the first record in block i )
        then u ← i – 1
    else if K > (ordering key field value of the last record in block i )
        then l ← i + 1
    else if the record with ordering key field value = K is in the buffer
        then goto found
    else goto notfound;
    end;
goto notfound;
```

- binary tree data structure is also available

- Summary of accessing time:

Table 17.2 Average Access Times for a File of b Blocks under Basic File Organizations

Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

- Hashed Files: A way to hash different block and combine them as a file
  - External hashing: Hashing for disk files
  - File blocks are divided into M equal-sized buckets, numbered $bucket_0, bucket_1, \ldots, bucket_{m-1}$
  - **Hashed field**: One of the file fields is designated to be the hash key of the file
  - **Bucket array**: an array(bucket) holds the header of m list
    - A bucket is either one disk block or a cluster of contiguous disk blocks. The hashing function **maps a key into a relative bucket number**, rather than assigning an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address (key <=>array <=> list <=> disk block address)
  - If a record has search key K, we store the record **by linking it** (separate chaining with fixed size bucket) to the bucket list for the bucket numbered h(K) where h is the hash function
  - Hash function take the hash key as an argument to compute an integer between 0 and m-1

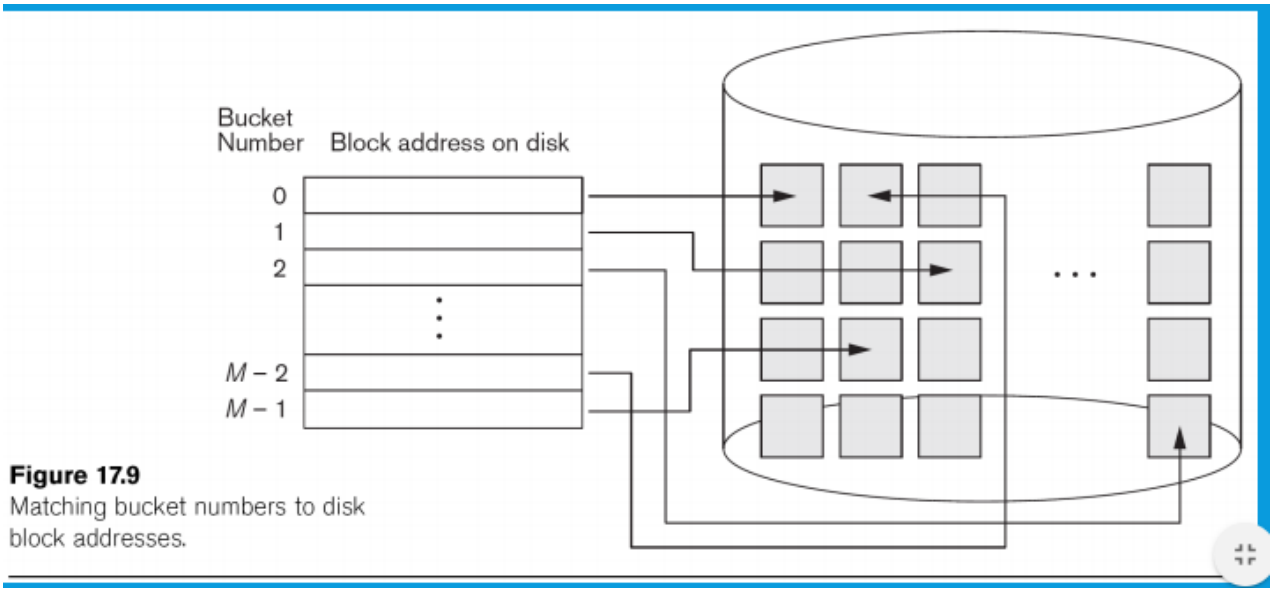


Figure 17.9
Matching bucket numbers to disk block addresses.

- Collision:
  - Each bucket has a capacity => if full collision happens
  - solve:
    - open addressing (search for empty slots):
      - Linear probing: Each time increase one => bucket_id +1 , bucket_id +2 , ... , bucket_id+n
      - Quadratic probing: Each time increase to square => bucket_id + 1, bucket_id + 4, .. , bucket_id + n^2
    - Chaining:
      - Extending the array with a number of overflow positions, each bucket has a pointer to the header of overflowed list, each overflowed block will also have pointer of next overflowed block
      - placing the new record in an unused overflow bucket and setting the pointer of the occupied hash address bucket to the address of that overflow bucket

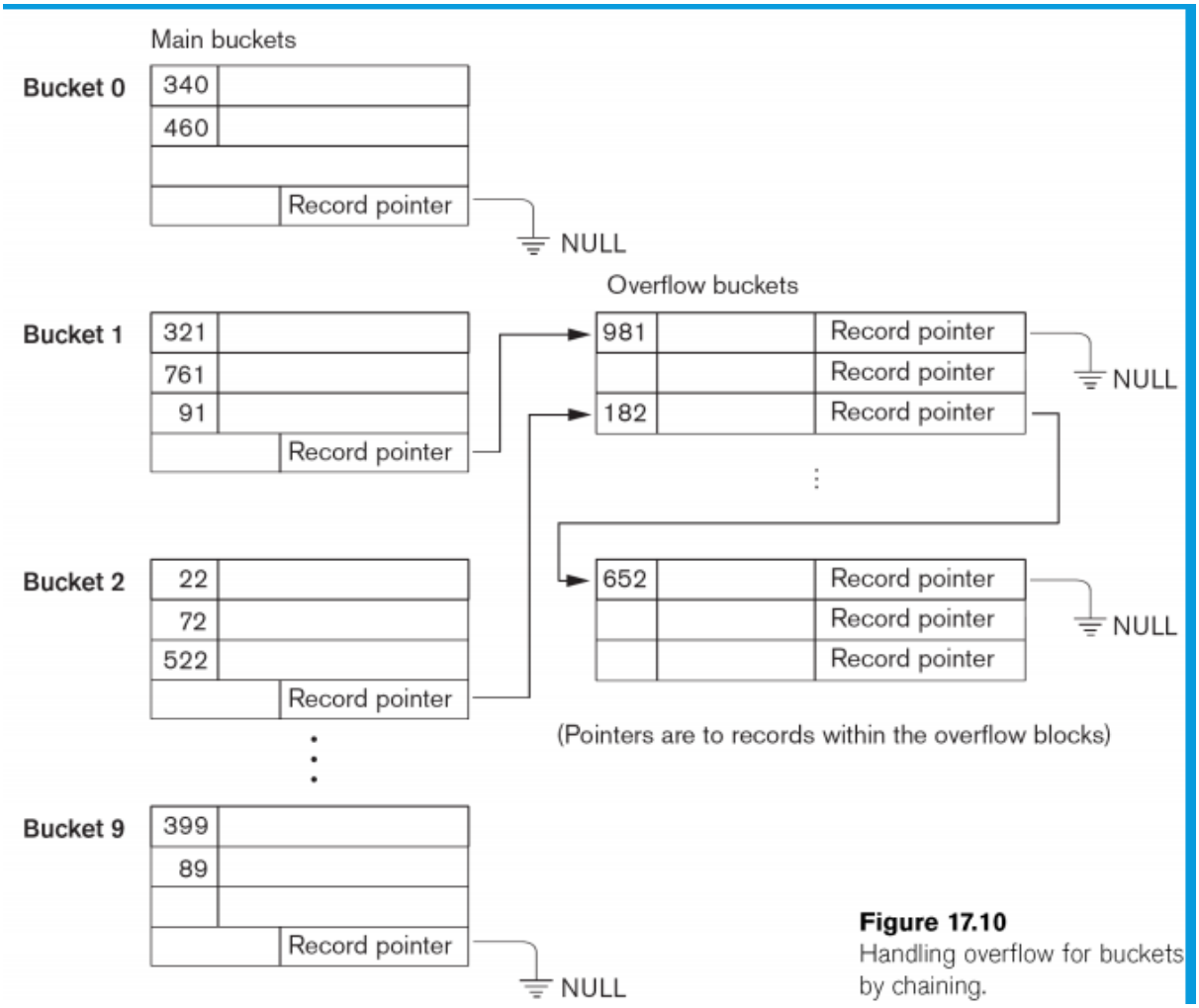


Figure 17.10
Handling overflow for buckets by chaining.

        - Multiple hashing:
            - The program applies a second hash function if the first results in a collision
            - If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary
    - To reduce collision hash file is typically kept 70% - 80% full
    - The hash function h should distribute the records **uniformly among the buckets**
        - Otherwise, search time will be increased because many overflow records will exist (Searching overflow records are more expensive)
    - Main disadvantages of static external hashing:
        - Fixed number of buckets M is a problem if the number of records in the file grows or shrinks
        - Ordered access on the hash key is quite inefficient (requires sorting the records)
- Extendible and Dynamic Hashing (extend hashing tech to allow dynamic growth and shrinking of file records)
    - Hash key: Use binary representation of the hash value h(K) in order to access the directory
    - Data structure: dynamic hashing(the directory is a binary tree); extendible hashing (use an array of size 2^d as the directory (d is global depth))
        - difference between the data structure and B-tree: B-tree actually is the truth data structure to store the data, but hash is use the tree to store the bucket of address
        - The depth of the tree for hashing will not be so large since it is actually a bucket
    - **Directory**: The directories can be stored on disk, and they expand or shrink dynamically: Directory entries point to the disk blocks that contain the stored records
    - Insertion to a full block => split the block, records are redistributed among the two blocks

- difference with tree structure: key is a hashed value

- Extendible hashing:
  - **Extendible hashing** is a type of hash system which treats a hash as a bit string and uses a trie for bucket lookup.

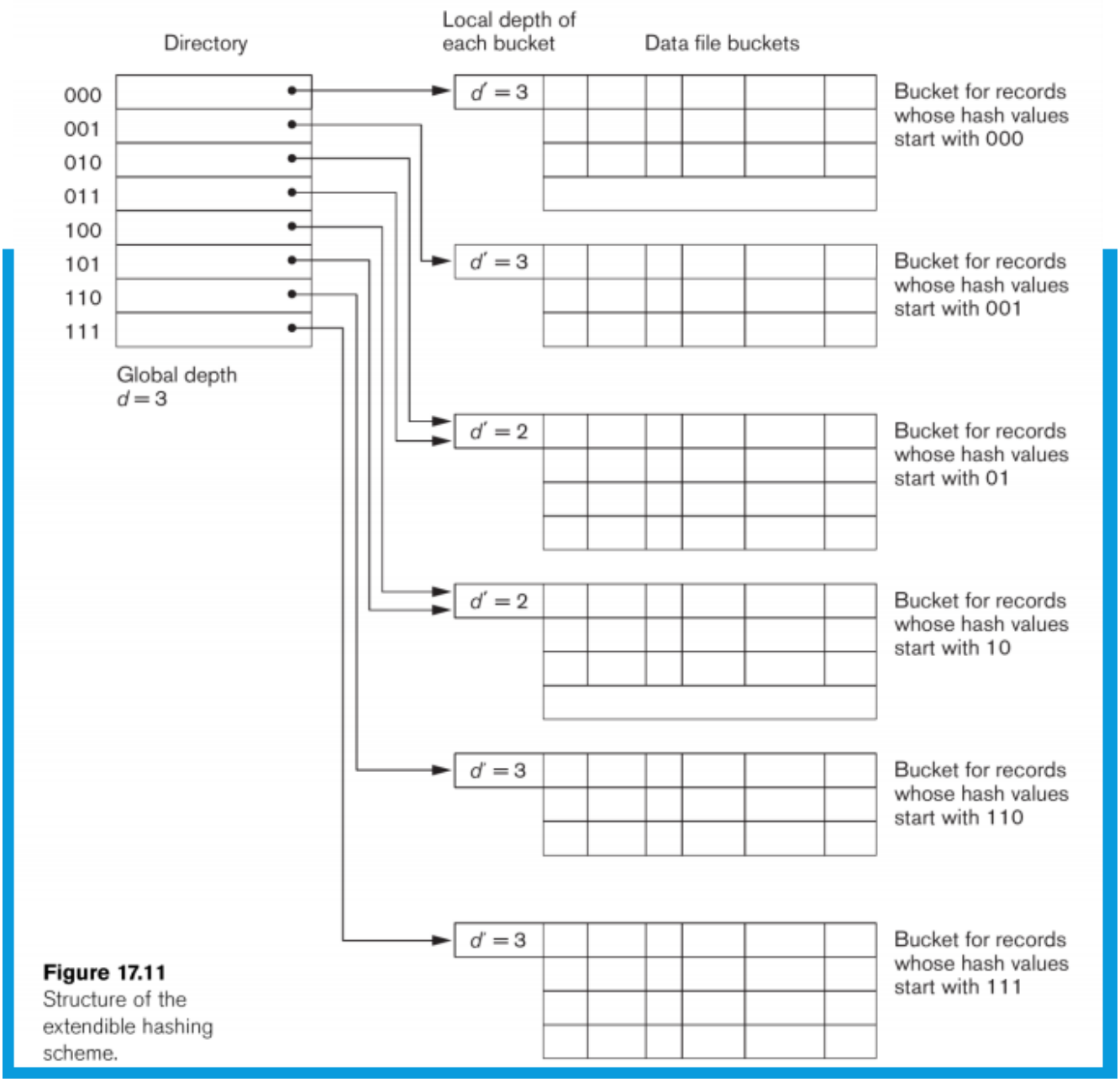
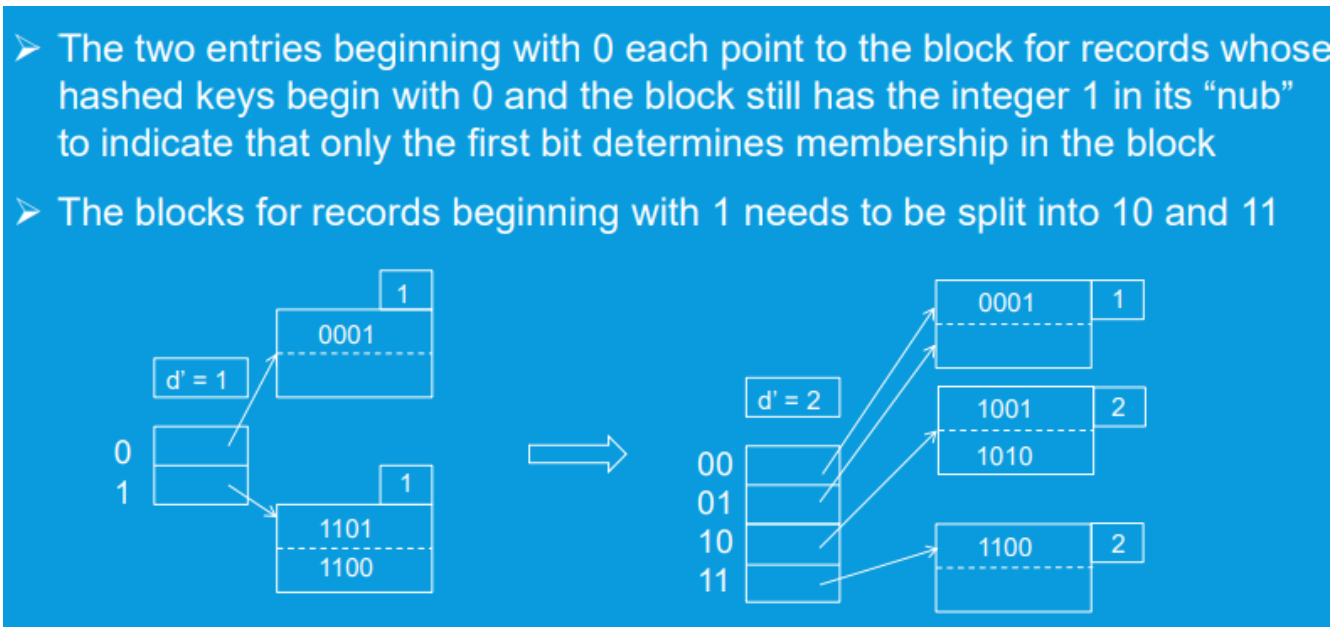


Figure 17.11
Structure of the extendible hashing scheme.

  - A directory consisting of an array of 2^d bucket addresses is maintained, d is called the global depth of the directory (maximum possible depth of the trie)
  - The integer value corresponding to the first (high-order) d bits of a hash value is used as an **index to the array** to determine a directory entry and the address in that entry determines the bucket storing the records
  - **A location d'** (called, local depth stored with each bucket) specifies the number of bits on which the bucket contents are based
  - The value of d' can be increased or decreased by one at a time to handle overflow or underflow respectively
    - At first d' is 1, there is only 2 buckets
    - bucket point to the memory address

    - The size of each bucket is also fixed at first
- e.g.: d = 4, at first d' = 1 => there are only 2 buckets each have 2 entries => if one is full and insert => split to 2



- Dynamic Hashing (similar concept with b-tree but different location of data)
    - Dynamic and extendible hashing do not require an overflow area in general
    - Dynamic hashing maintains tree-structured directory with two types of nodes
        - Internal nodes that have two pointers: the left pointer corresponding to the 0 bit (in the hash address) and a right pointer corresponding to the 1 bit
        - Leaf nodes: these hold a pointer to the actual bucket with records

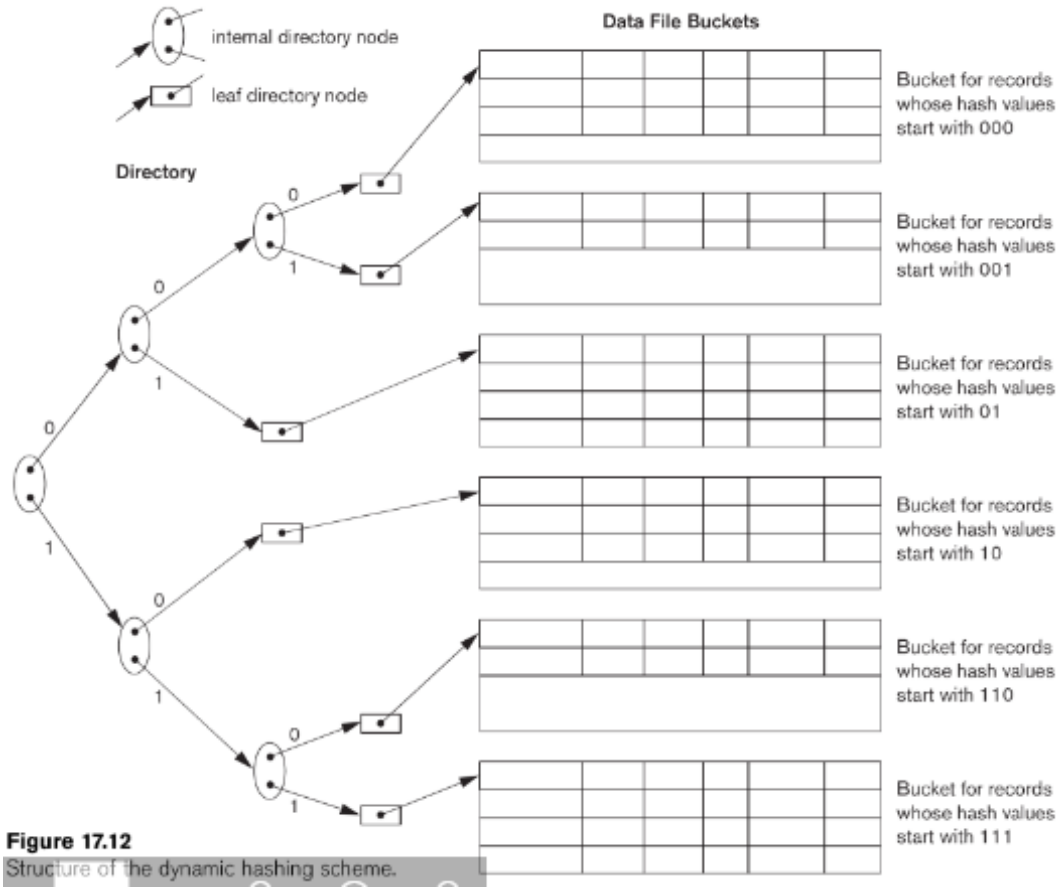
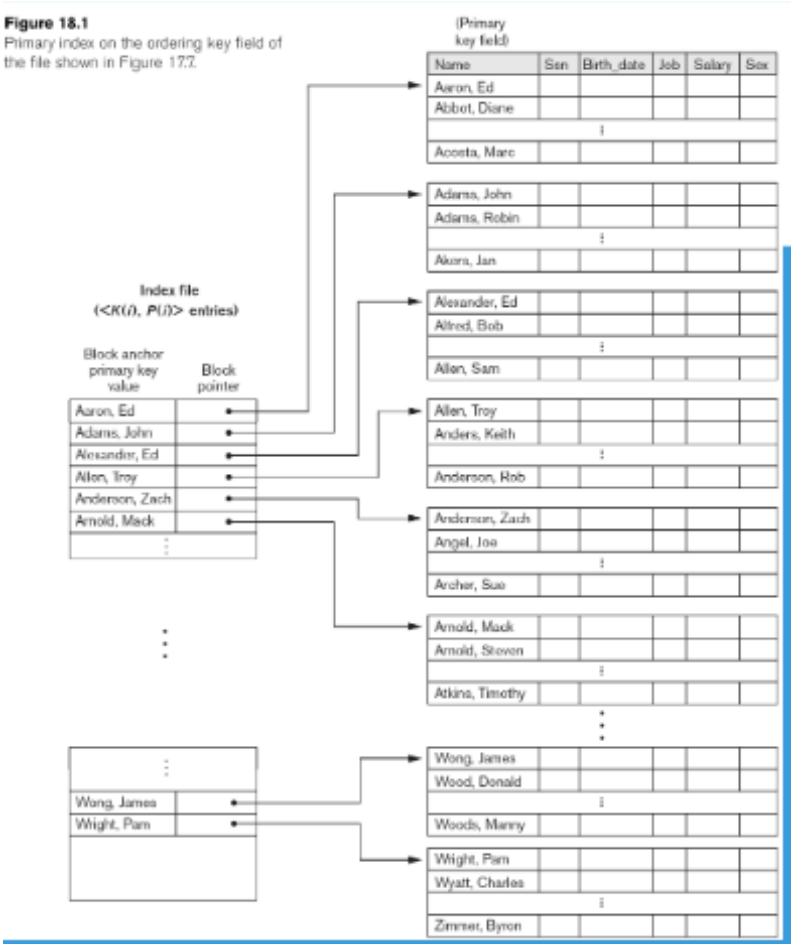


Figure 17.12
Structure of the dynamic hashing scheme.

Lecture 8: Indexing Techniques

- Indexes as access path:
  - An index is an auxiliary file to provide fast access to a record in a data file (index file + data file)
  - index file is a file that have partial information of the data file, it is pointers to data sorted by some attributes
  - Entries are ordered in the index file, each entry consists of 2 parts: field value(the index) and pointer to record (address of one entry in the data file)
  - An index is usually specified on one field (called key field which may not be a key) of the data file (although it could be specified on several fields)
  - The operation:
    - Access index file in the secondary disk (do binary search)
    - Load the searched block into the main memory, since the size of index file is much smaller that the data file, the delay is shorter
    - Since it is faster to search in memory, overhead of searching a certain entry in the loaded block can be ignored
    - Algorithm is still important because there is an overhead when retrieve the data record according to the index file
- Simple level index: only one index file directly point to data file
  - dense/sparse indexes
    - dense: An index entry for every search key value in the data file
    - sparse: One entry for multiple search keys => smaller index size
  - Primary Index: Use primary key as the base of sorting order and order the data file



- **the data file itself is physically sorted on primary key**
- Sparse index : The entry is smaller than the number of record (a entry has many records in a block), one entry corresponds to one block of records
- Each entry hold the first record of the block in the data file as the key field value(index), we call it **block anchor**
- The way to calculate: The binary search is actually done on the index file and multiple entries of index file are also arranged in blocks. After that, the corresponding block i whose block anchor $K(i) < K < K(i+1)$ where K is the searched value will be loaded into the main memory
- B is block size, R is record size, r is number of records, following is the cost with out the primary index (load the data file blocks for $log_2 b$ times)

$$
\begin{aligned}
Bfr(record\ per\ block) &= \frac{B}{R} \\
b(block\#) &= \frac{r}{Bfr} \\
cost &= log_2 b
\end{aligned} \tag{1}
$$

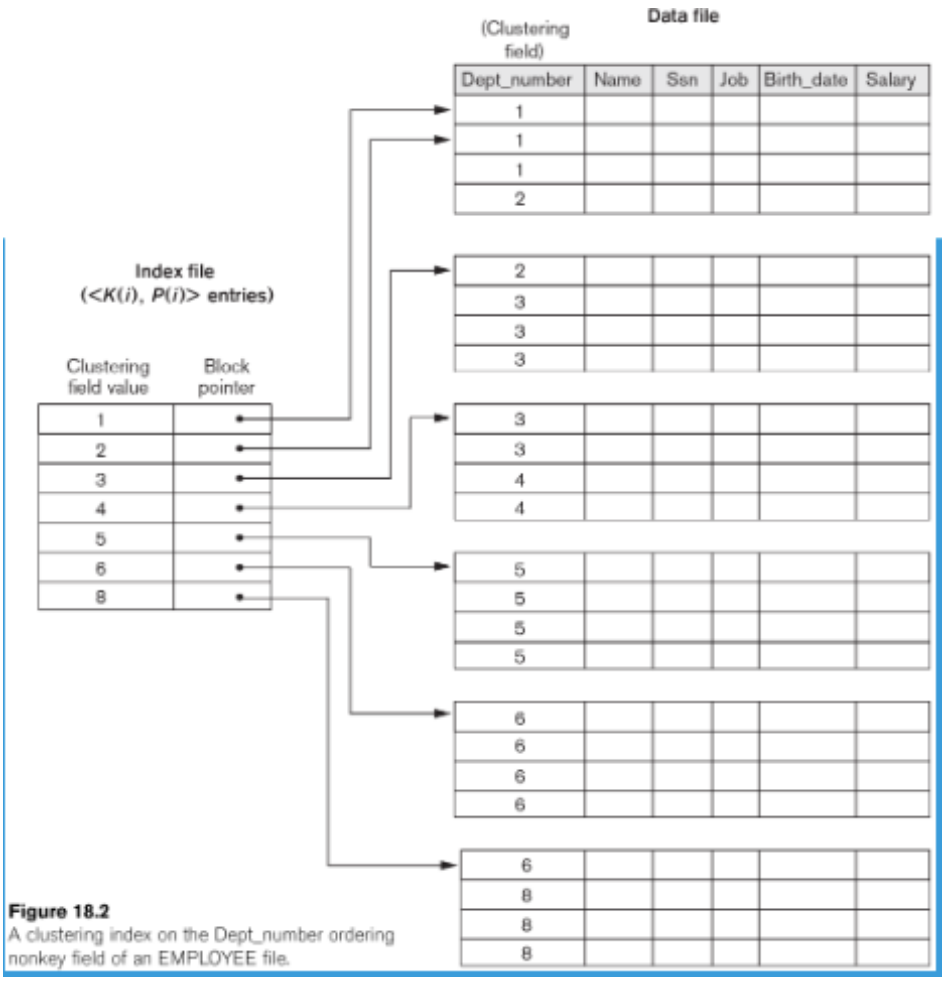
- Following is the cost with primary index

$$
\begin{aligned}
r_1 &= b \\
R &= size(index) + size(pointer) \\
Bfr_1 &= \frac{B}{R} \\
b_1 &= \frac{r_1}{Bfr_1} \\
b_1 &= \frac{(size(index) + size(pointer)) * data\ block\#}{block\ size} \\
cost &= log_2 b_1((load\ index\ \ file\ into\ memory)) + 1(load\ data\ \ file\ into\ memory)
\end{aligned} \tag{2}
$$

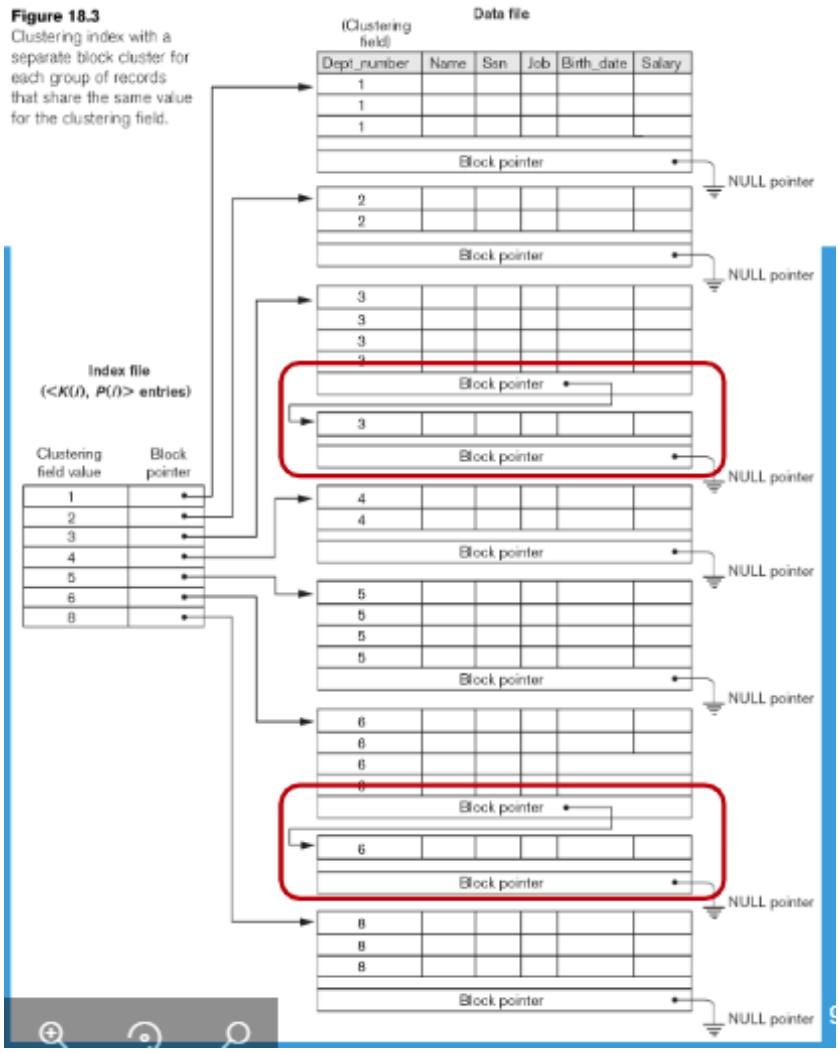
- Suppose the data record size R is 100 bytes , block size B is 1024 bytes, the number of records r is 30,000, pointer size P_R is 6 bytes, field size V is 9 bytes, calculate the block access number respectively

$$
\begin{aligned}
b_1 &= \frac{R * r}{B} = 30000 * 100/1024 = 3000\ bytes \\
b_2 &= \frac{b_1 * (P_R + V)}{B} = 3000 * 15/1024 = 45\ bytes \\
a_1 &= log_2 b_1 = 12 \\
a_2 &= log_2 b_2 + 1 = 6 + 1 = 7
\end{aligned} \tag{3}
$$

- Cluster index: Use the non-key field as the base of order => duplicate possible
  - The pointer point to the first **unoccupied** block in the file that includes the first existence of the index value within this block, one pointer for each distinct value of the field
    - reason of just pointing to the block => the cost of search within a block is lower than block access
  - pointer may not point to the first entry of the block because the size of a block is fixed and may not equal to the size of repeated entries
  - Another example of sparse index => one index entry for multiple records
  - each entry = key + block pointer

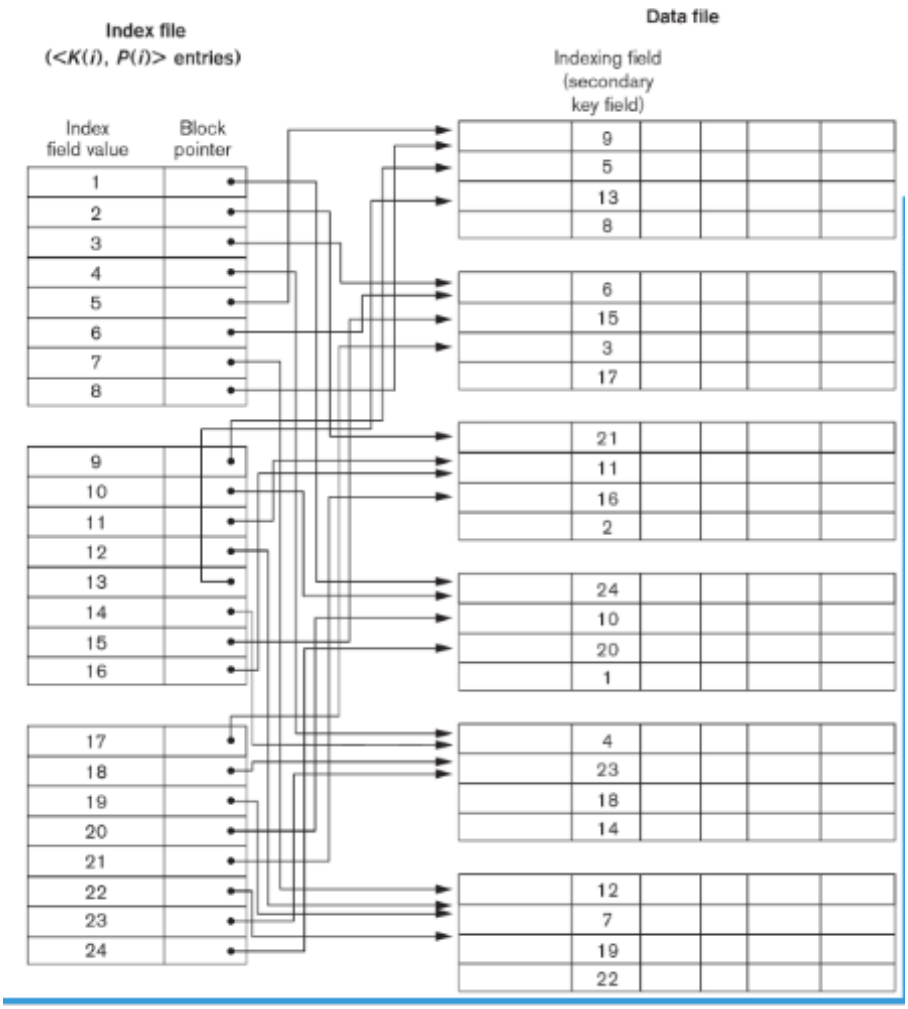


- ordered data file -> overflow problem -> use the overflow pointer point to next block if the current one is full

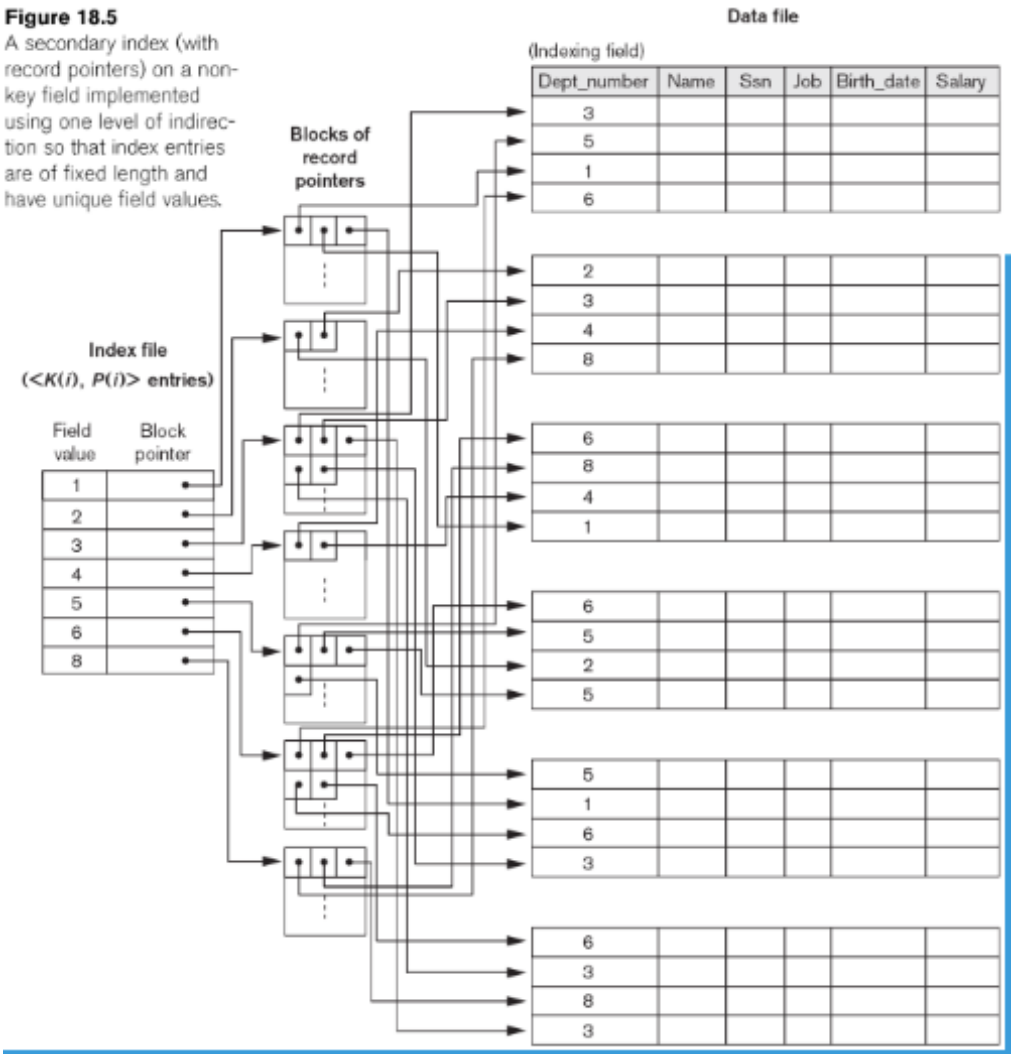


- Secondary index: When the index file is already physically sorted on some key by clustering or primary key, we cannot reorder the data file by another key or non-key => use secondary, the index field is specified on any non-ordering field of a data file
  - may on key or non-key fields
  - entry = index field + a block pointer or a record pointer
  - when the field is key => one to one and dense

Figure 18.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



- several secondary indexes can be applied to the same file when the field is non-key



- Have two types of pointers constructs two levels
  - Sparse: Each index entry points to a block of pointershas multiple index pointers pointing to the records with the same value
  - Dense: Each pointer within a the pointer block
- Access time with ordered index file: 12 as calculated
- Access time of unordered without indexing

$$b_1 = \frac{r * R}{B} = 30000 * 100/1024 = 3000\ blocks$$
$$cost = b_1/2 = 1500 \tag{4}$$

- Access time with index (dense key)

$$b_2 = \frac{r * (P_R + V)}{B} = 30000 * 15/1024 = 442\ blocks$$
$$cost = log_2 b_2 + 1 = 10 \tag{5}$$

- Advantage of secondary index file compare to unordered file
  - smaller entry size: 15 is less than 100
  - binary search is available

- summary of simple indexing

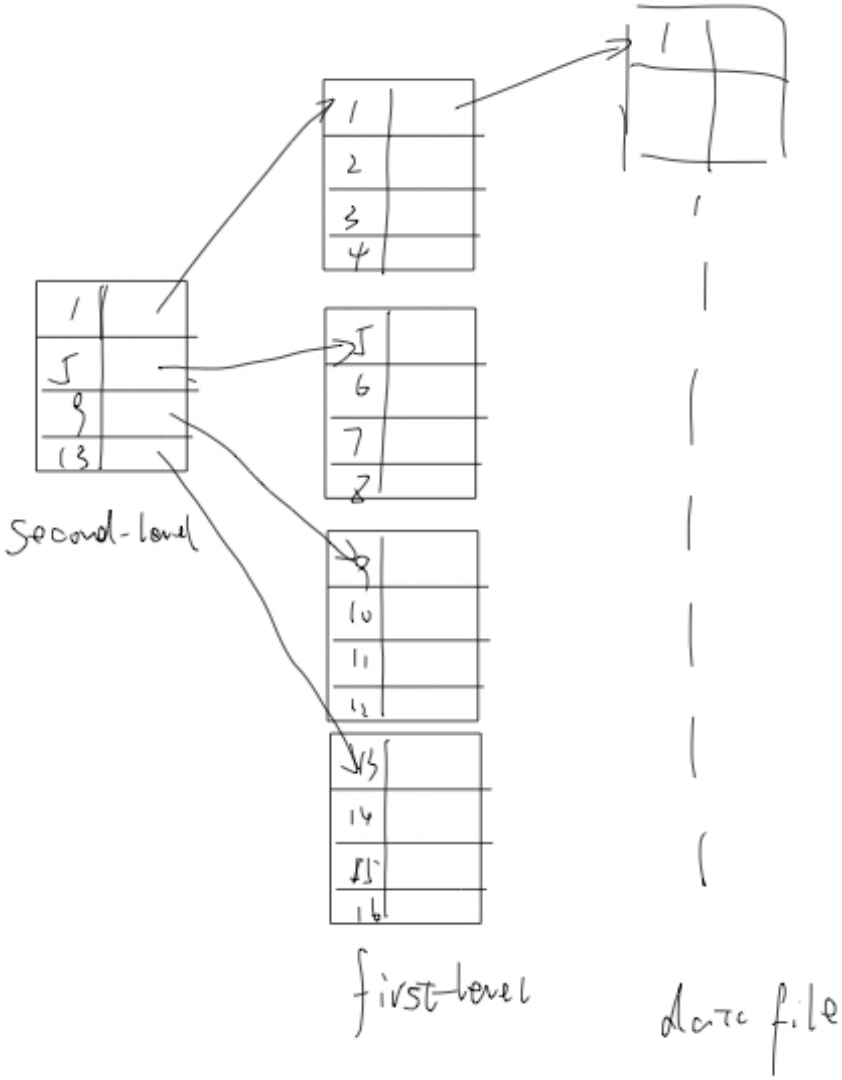
Table 18.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Table 18.2 Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

- Multiple-level indexes: Index file for the previous index file, the first-level index file is directly point to the data file



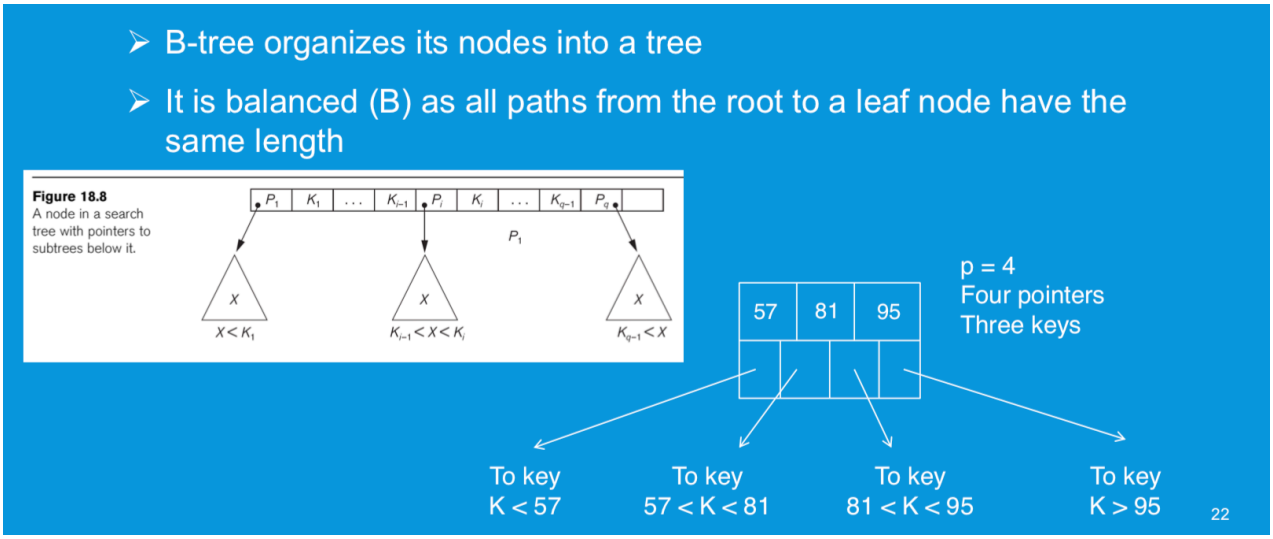
- Multi-level index can be created for any type of first-level index as long as the previous level has multiple blocks
- Because index is ordered and identical, all the non-first-level index can use primary index
- bfr is fixed besides the first layer, because record size fixed and equals to size of pointer plus size of key
- block number b is equals to the previous b divide by bfr
- An example from the previous example

$$
\begin{aligned}
b_1 &= \frac{r * (P_R + V)}{B} = 30000 * 15/1024 = 442\ blocks \\
b_2 &= \frac{b_2 * (P_R + V)}{B} = 442 * 15/1024 = 7\ blocks \\
b_3 &= \frac{b_3 * (P_R + V)}{B} = 7 * 15/1024 = 1\ blocks
\end{aligned} \tag{6}
$$

  - Thus, the total level of indexing is 3
  - one accessing for each level and final access for data block

$$
cost = level\ of\ blocks + 1 \tag{7}
$$

- Tree index structure: Multiple index actually form a tree => insert into deletion becomes a problem because every level of the index is an *ordered file*
  - some definitions:
    - except root, each node has a parent and zero or more child
    - zero child: leaf
    - non-zero child: internal
    - sub-tree of a node: itself and all of its descendant
    - balanced: leaf nodes are at the same level
  - Common things of B-tree and B+-tree
    - Reserve spaces in each tree node to allow new index entries
    - Each node correspond a disk block
    - Each node is kept between full(split => which will propagate to higher level) and half full(merge)
    - insert into non-full or delete into more than half full is efficient
  - B-tree
    - In a B-tree, pointers to data records exist at all levels of the tree (P_i points to the lower level, K_i points to data)
      - at most p tree pointers and at least p/2
      - Only the middle value is kept in the root node and the rest of the values are split evenly between the other two nodes
      - When a non-root node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes (parent full also split parent)



- Each internal node in a B-tree is of the form <P_1, <K_1,Pr_1>, P_2, <K_2,Pr_2>, …, <K_{q-1},Pr_{q-1}>,P_q> where $q \le p$ (each node has at most p tree pointers)
- Each P_i is a tree pointer to another node in the B-tree
- Each Pr_i is a data pointer points to the record whose search key field value is equal to K_i
- Within each node, $K_1 < K_2 < \ldots < K_{q-1}$
- For all search key field values X in the subtree pointed by P_i, we have (i) $K_{i-1} < X < K_i$ for $1 < i < q$; (ii) $X < K_i$ for $i = 1$; and (iii) $K_{i-1} < X$ for $i = q$
- Each node has at most p tree pointers (order of the tree)
- Each node except the root and leaf nodes, has at least p/2 tree pointers
- All leaf nodes are at the same level (balanced tree) and have the same structure as internal nodes except that all of their tree pointers of P_i are NULL

23

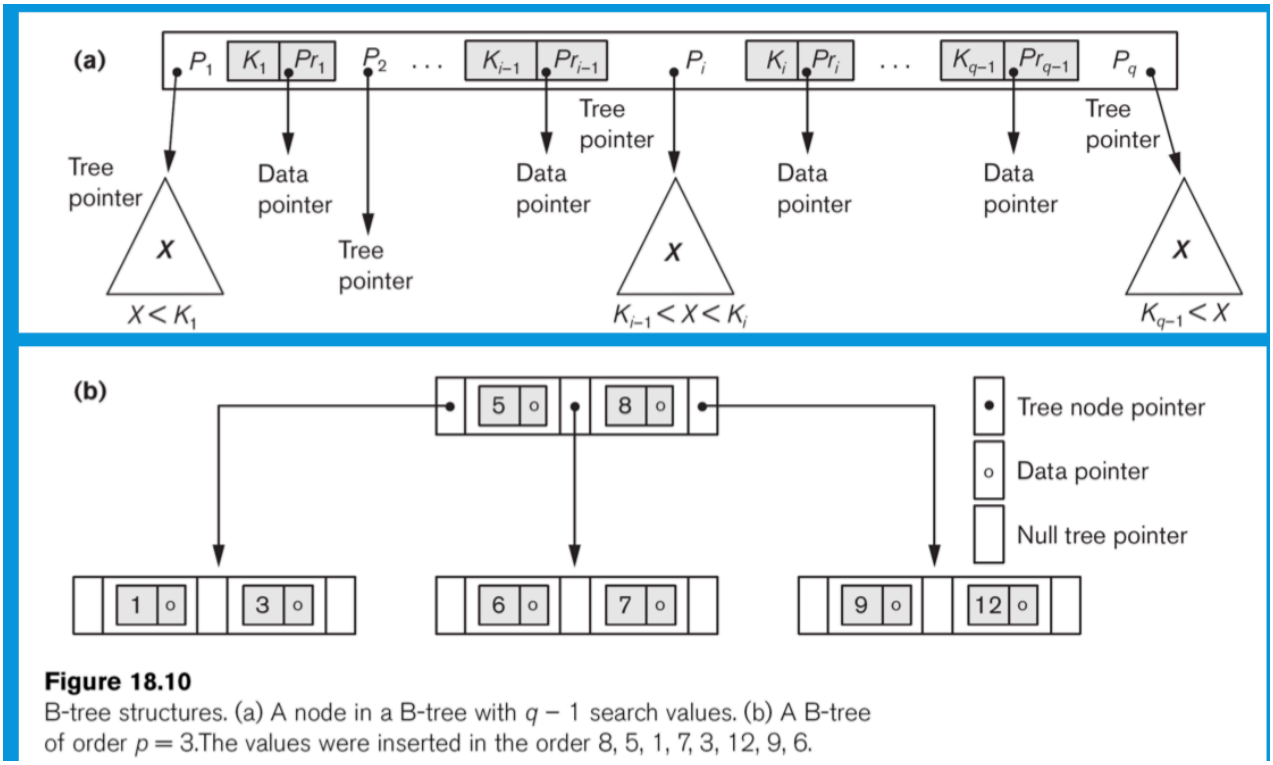


Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

- Insertion: B-tree starts with a single root node at level 0
  - Once the root is full with p –1 search key values and we attempt to insert another entry into the tree, the root node splits into two nodes at level 1
  - Only the middle value is kept in the root node and the rest of the values are split evenly between the other two nodes => **middle node doesn't stay at previous level anymore**
  - When a non-root node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes
  - If the parent node is full, it is also split
- index size: suppose the pointer limit is 23 and each of the node is 69% full, thus there are $0.69 * 23 = 16$ pointers to next level, therefore on level k

$$
\begin{aligned}
Node\# &= pointer\#^{k} \\
current\ level\ pointer\# &= pointer\#^{k+1} \\
index\ entries &= current\ level\ pointer\# - pointer\#
\end{aligned}
\tag{8}
$$

Root:	1 node	15 key entries	16 pointers
Level 1:	16 nodes	240 key entries	256 pointers
Level 2:	256 nodes	3840 key entries	4096 pointers
Level 3:	4096 nodes	61,440 key entries	

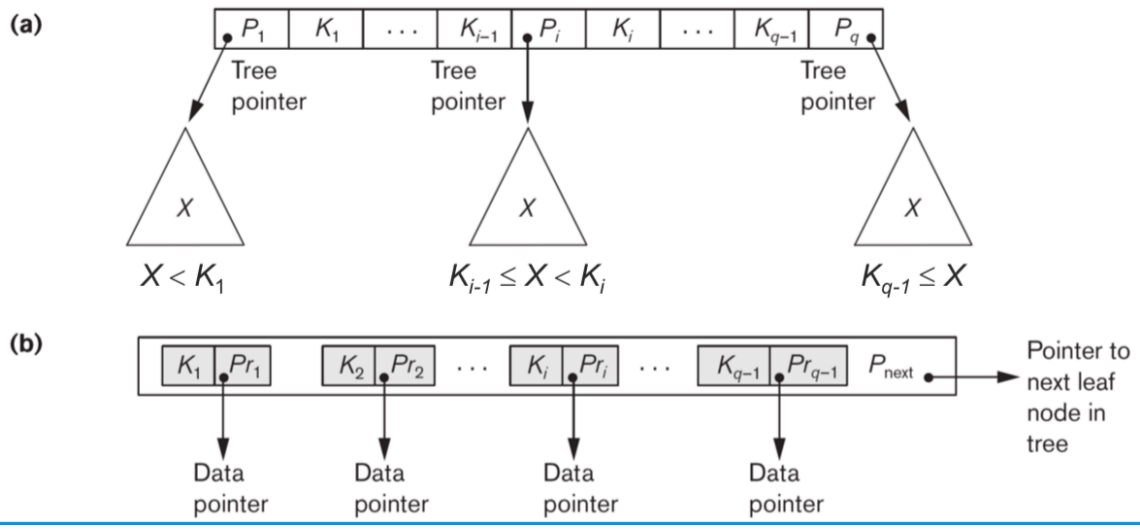
- B+-tree: all pointers to data records exists at the leaf-level nodes => less levels because its entry is smaller in size (only store pointer to tree node, no need to store the data pointer => more space to store key => larger capacity and smaller level)
  - Notice P_{leaf} is the order of the tree minus one

- Each internal node in a B$^+$-tree is of the form <P_1, K_1, P_2, K_2, …,P_{q-1}, K_{q-1},P_q> where q ≤ p and P_i is a tree pointer and is also called the tree order
- Each node has at most p tree pointers
- Within each node, $K_1 < K_2 < … < K_{q-1}$, for all search key field values X in the subtree pointed by P_i, we have
  - $K_{i-1} \leq X < K_i$ for $1 < i < q$;
  - $X < K_i$ for $i = 1$; and
  - $K_{i-1} \leq X$ for $i = q$
- Each node except the root and leaf nodes has at least $\lfloor p/2 \rfloor$ tree pointers

The Structure of the Leaf Nodes of a B$^+$-tree

- Each leaf node is of the form <<K_1, Pr_1> ,<K_2, Pr_2>, …,<K_{q-1}, Pr_{q-1}>, P_{next}> and Pr_i is a data pointer and P_{next} points to the next leaf node of the tree
- Within each node, $K_1 < K_2 < … < K_{q-1}$
- The maximum number of data pointers in a leaf node is the tree order (or degree) minus 1
- Each Pr_i is a data pointer points to the record whose search field is K_i
- Each leaf node has at least $\lfloor p_{leaf}/2 \rfloor$ data records
- All leaf nodes are at the same level

Figure 18.11
The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.



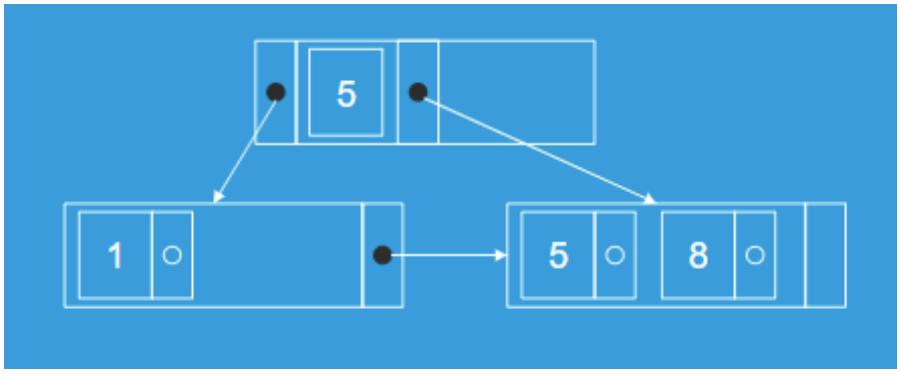
- Suppose the search key field is V = 9 bytes, block size B = 512 bytes, record pointer (leaf pointer) size P_r = 7 bytes, block pointer is P = 6 bytes => determine the pointer number limit p

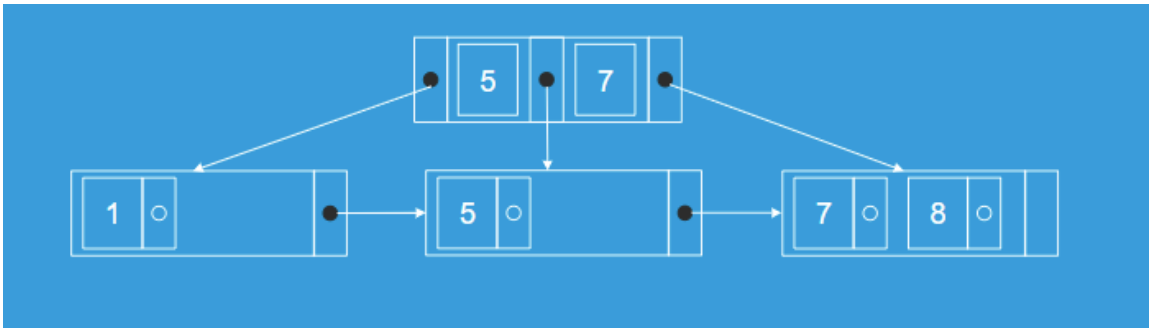
$$
\begin{aligned}
p * P + (p-1) * V &\leq B \\
6p + 9p - 9 &\leq 512 \\
p_{leaf} * (P_r + V) + P &\leq B \\
p_{leaf} * (7+9) + 6 &\leq 512 \\
p &= 34 \\
p_{leaf} &= 31
\end{aligned} \tag{9}
$$

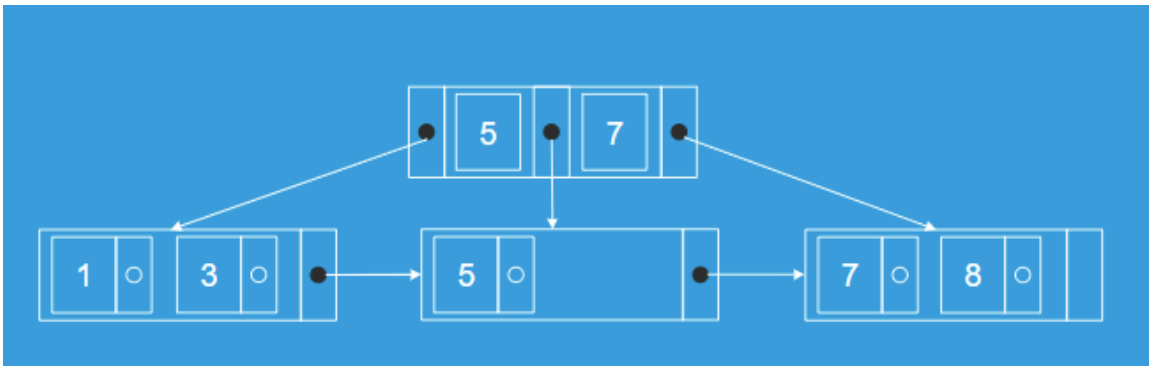
- The way to calculate the index size is the same => $0.69 * 34 = 23$

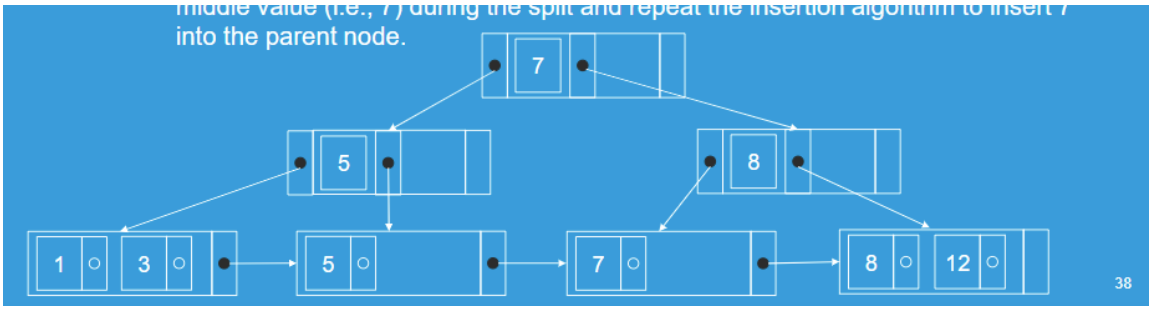
Root:	1 node	22 key entries	23 pointers
Level 1:	23 nodes	506 key entries	529 pointers
Level 2:	529 nodes	11,638 key entries	12,167 pointers
Level 3:	12,167 nodes	255,507 data record pointers	

- Insertion & deletion of B+ tree
  - Insertion (search corresponding leaf at first): if full
    - split into two, first node have $floor((n+1)/2)$ keys (where n is the size before insertion)
    - on the leaf: split and copy the minimum key of second child to upper level (also keep the original key)
    - on the internal: split and insert the minimum key of second child to upper level, delete the corresponding node of current level (don't need the original key)
  - Example (notice that the P is start with P_1, and the new value is assumed to be inserted into the node and not split at first), the sequence is 8, 5, 1, 7, 3, 12 the number of key in internal and leaf nodes are both 2:

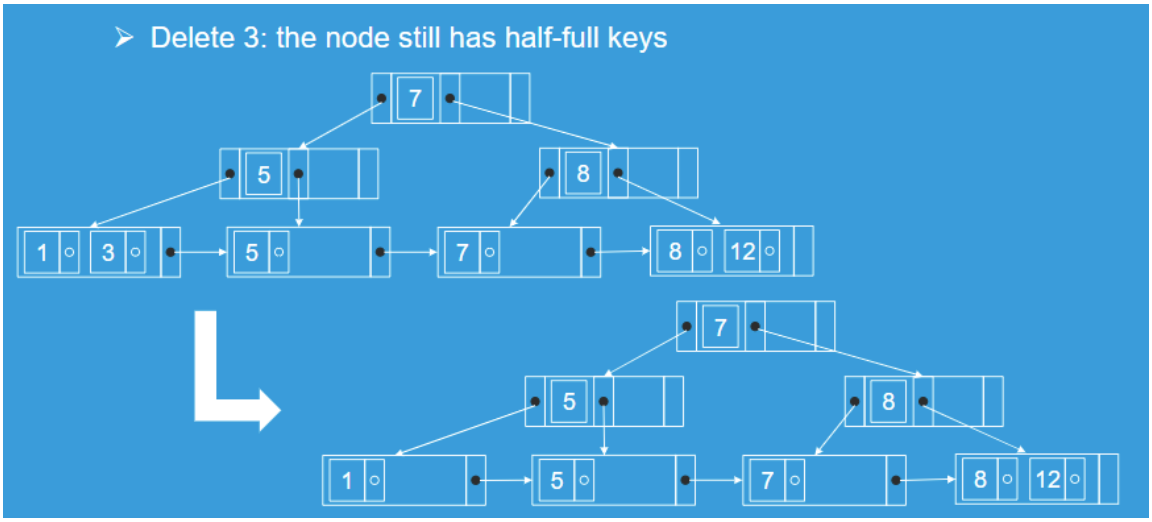


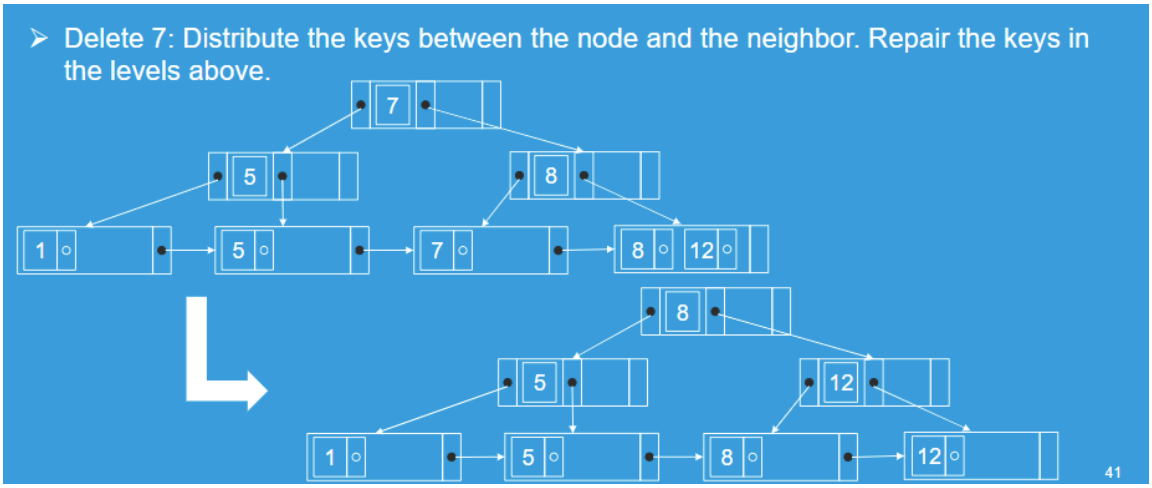


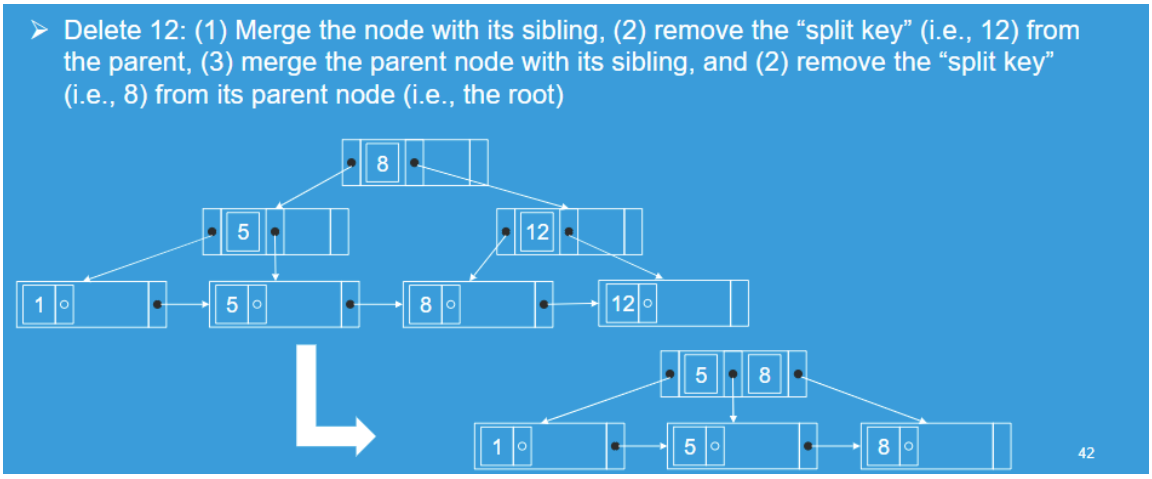




- Deletion: if not half full: overall, need to remove the corresponding key in the parent level
  - if neighbor sibling has more than necessary(if both ok, choose larger one): distribute the key between current node and neighbor, change the corresponding key in the parent => always consider borrow from neighbor at first
  - Merge the node with its sibling; if the node is a non-leaf, we will need to **incorporate the "split key" from the parent into our merging** (if a parent have any one child is empty, then one key from parent will be sent back). In either case, we will need to repeat the removal algorithm on the parent node to remove the "split key" that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).
- Example:



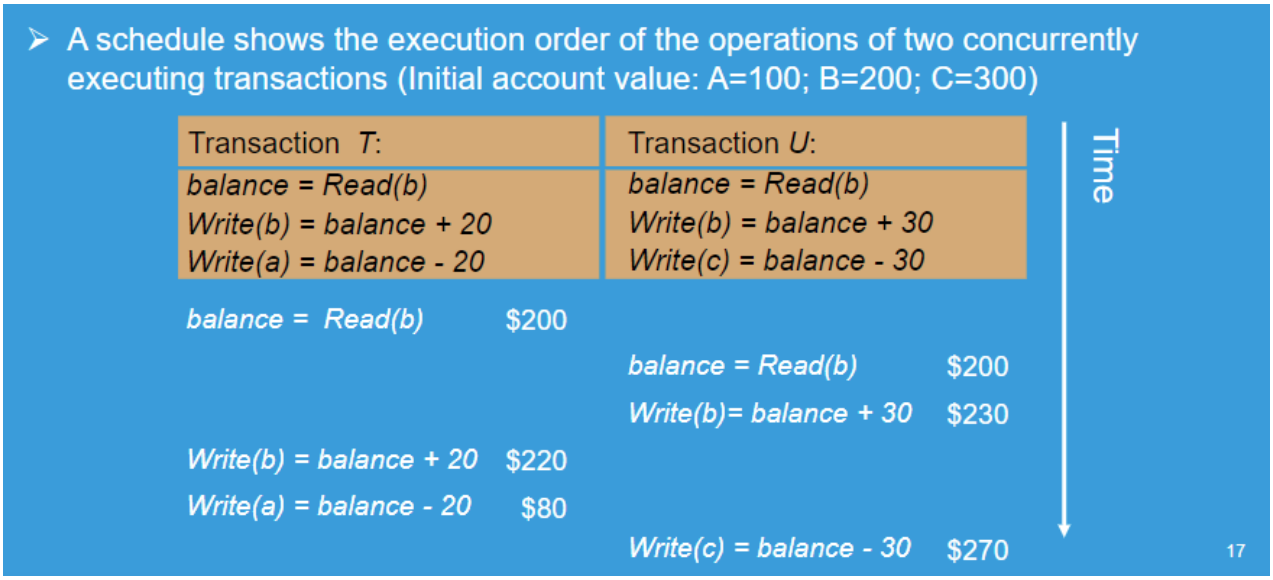




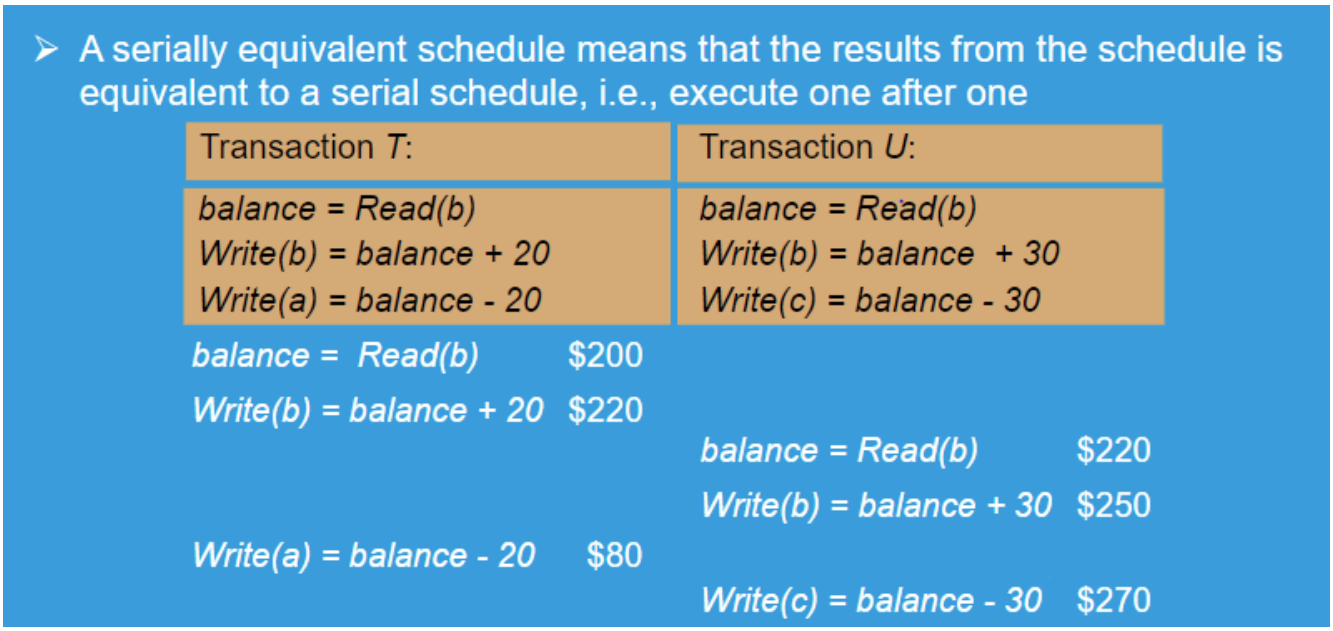
Lecture 9: Transaction

- ACID principle, a transaction is :
  - Atomicity: A transaction is either performed completely or not performed at all
  - Consistency: A correct execution of a transaction must take the database from one consistent state to another
  - Isolation: Only after a transaction is committed, it can be visible to other transactions (no partial results)
  - Durability: Once a transaction is committed, these changes must never be lost because of subsequent failure (committed and permanent results)

- Transaction (the execution of a program/application on behalf of the user to perform a specific user function by accessing data items maintained in a database management system) = DB operations + transaction operations
    - operation : atomic steps within each transaction
    - DB ops:
        - Read: SELECT
        - Write: UPDATE
        - Delete
        - Insert
    - transaction ops: Make sure atomic => partial results are not allowed
        - Begin (may have a lock)
        - End (Commit/Abort)
            - commit: new value of transaction will become permanent
            - abort: fail and recover
    - State: Begin(consistent state) => executing DB operations (maybe inconsistent temporarily) => End(consistent state)
        - whole transaction is atomic
        - multiple steps => single user application
- read/write description
    - Data are resided on disk and the basic unit of data transferring from the disk to the main memory is one disk block
    - In general, a data item (what is read or written) will be the field/fields of some records in the database (in a disk block)
    - read_item(X) command includes the following steps:
        - Find the address of the disk block that contains item X
        - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer)
        - Search for the required value in the buffer
        - Copy item X from the buffer to the program variable named X
    - write_item(X) command includes the following steps:
        - Find the address of the disk block that contains item X
        - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer)
        - Search for the required value in the buffer
        - Copy item X from the program variable named X into its correct location in the buffer
        - Store the updated block from the buffer back to disk (either immediately or at some later point in time)
        - Note that we DO NOT need to read an item before update it
- Problems of transaction processing performance:
    - slow disk fast CPU
    - undo redo of long transactions to maintain atomicity in cases of failures
    - concurrency problem for multiuser system and interleaving/parallel processing
        - high concurrency => better performance, more difficult to maintain ACID properties

- Transaction schedule: When transactions are executing concurrently in an interleaved fashion or serially, the order of execution of operations from the transactions forms a transaction schedule
    - A schedule S of a sequence of operations is an ordering of the operations of the transactions **if it is under the constraint that the relative order of the operations in the same transaction is not changed**
    - Concurrent schedule: interleave of different transactions exists => another transaction start BEFORE the other has finished
    - Serial schedule: otherwise
- Consistency problems:
    - Lost update problem (write/write conflicts) => Because the effect of a pair of write operations depends on the order of their execution
    - 

    - Inconsistent retrieval problem (read/write conflicts) => Because the effect of a read and a write operation depends on the order of their execution
    - read read don't have conflict
- Serializable schedule: A schedule S which is **equivalent** to serial schedule (no interleaving => itself is serializable schedule) => guarantee the consistency and have better performance (serial equivalence)



- Conflict serializable schedule: A schedule S is conflict equivalent to the serial schedule

- conflict equivalent schedule: RW, WW on same data item are in relatively same order with some serial schedule
- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'

S1

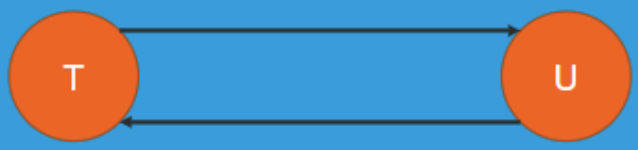
T1	T2
Read(A)	
Write(A)	
	Read(A)
Read(B)	

S2

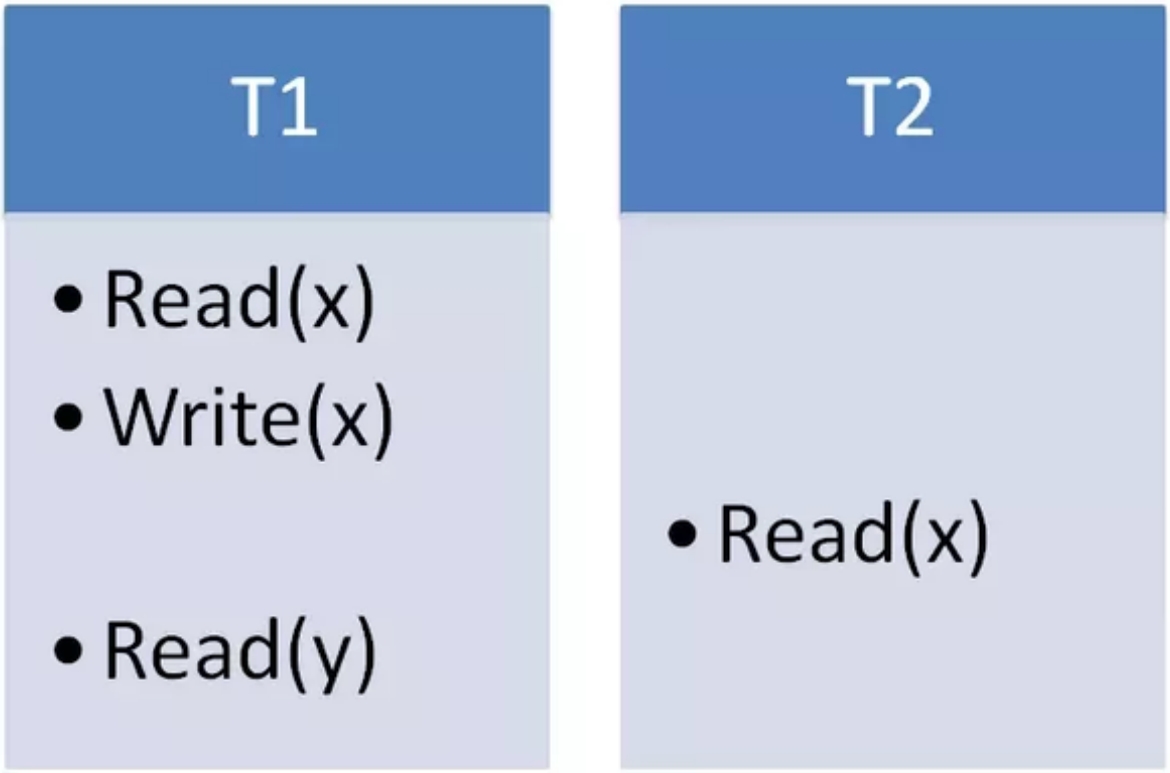
T1	T2
Read(A)	
Write(A)	
Read(B)	
	Read(A)

- Serialization Graphs: A direct edge $T_i -> T_j$ can drawn if j is after i, and
  - i is write, j is read or
  - i is read, j is write or
  - i is write, j is write
  - It is serializable iff the graph is acyclic(there are 2 nodes represent i & j, no bidirectional edge)
  - all of one node's operations is before the other one

Transaction *T*:	Transaction *U*:
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read(i)

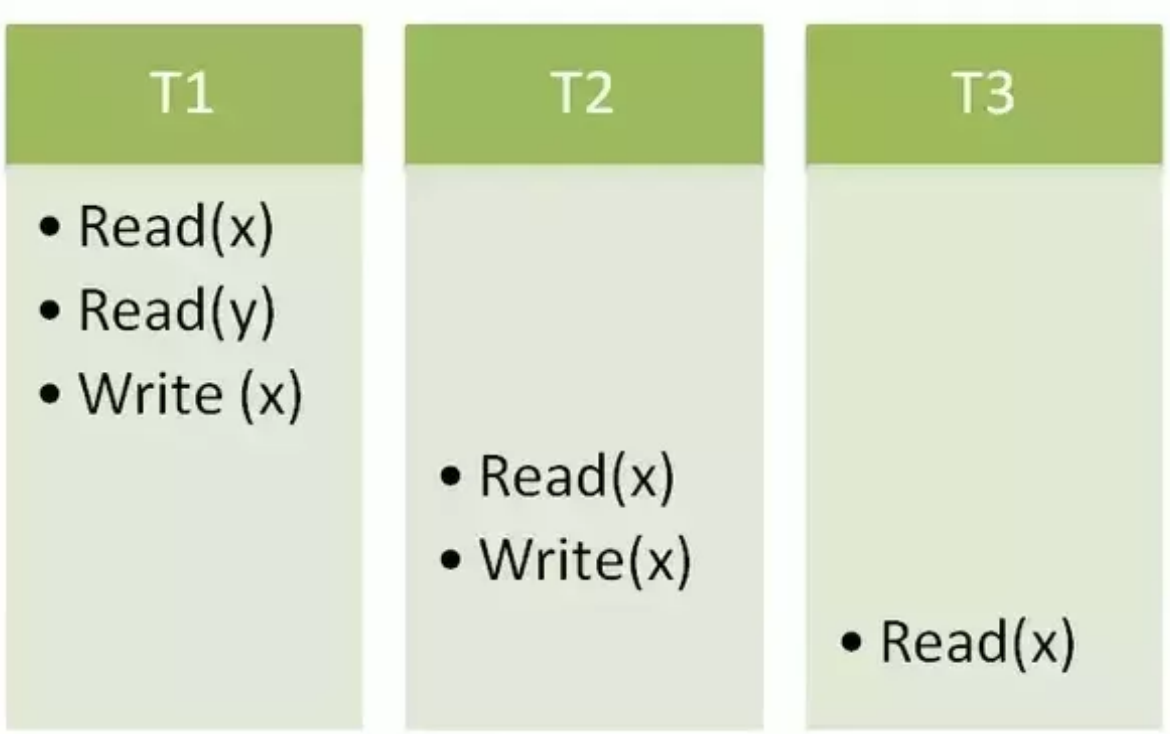


- Recoverable schedule: more information
  - recover: transaction is aborted => need to undo those processed operations of an aborted transaction to maintain consistency and all or nothing property
  - **recoverable schedule** (read commit after write commit): A recoverable schedule is one where, for each pair of Transaction Ti and Tj such that if Tj reads data item previously written by Ti , then the **commit operation of Ti appears before the commit operation Tj.**



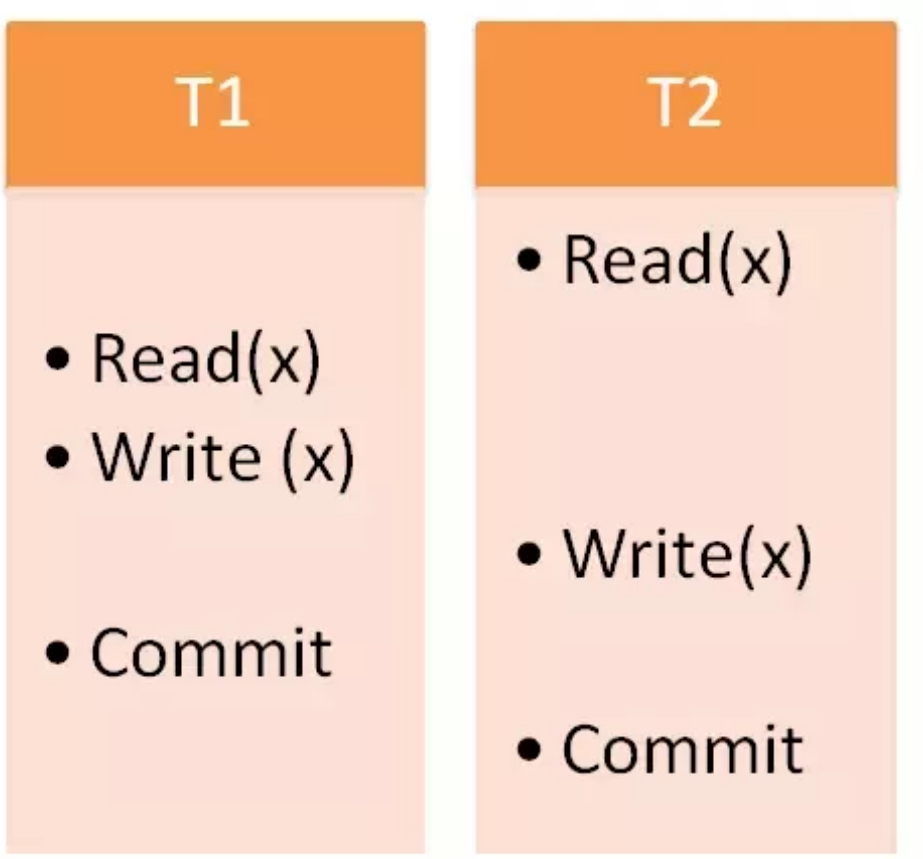
- Suppose after T2 Read(x) operation, it commits. And then somehow T1 fails. So the transaction T2 must be aborted so as to ensure atomicity. However, since T2 is commited , and can't be aborted . Hence a situation arrives where it is impossible to recover.

- **CASCADELESS SCHEDULE** (read start after write commit)



- Transaction T1 writes x that is read by Transaction T2. Transaction T2 writes x that is read by Transaction T3. Suppose at this point T1 fails. T1 must be rolled back, since T2 is dependent on T1, T2 must be rolled back,and since T3 is dependent on T2, T3 must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks is called *Cascading rollback.*

        - Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.
        - It is desirable to restrict the schedules to those where cascading rollbacks cannot occur, Such schedules are called Cascadeless Schedules.
        - Formally, a cascadeless schedule is one where for each pair of transaction Ti and Tj such that Tj reads data item, previously written by Ti the **commit operation of Ti appears before the read operation of Tj .**
    - Cascadeless schedules are also recoverable schedules, because recoverable recommend commit of read is after commit of previous write, cascadeless recommends **start of read is after commit of previous write**
- **STRICT SCHEDULE** (write and read start after pre write commit)



    - In this case, the Write(x) of the transaction T2 overwrites the previous value written by T1 , and hence overwrite conflicts arise . This problem is taken care in **Strict Schedule**. Strict Schedule is a schedule in which a transaction can neither Read(x) nor Write(x) until the last transaction that wrote x has committed or aborted.

T_1	T_2
read(A)	
write(A)	
	read(A)
	commit/abort
read(b)	
commit/abort	

- Non-recoverable
  - If T_2 commits and then T_1 aborts

T1	T2
read(A)	
write(A)	
	read(A)
read(b)	
commit/abort	
	commit/abort

- Recoverable

T_1	T_2	T_3
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
commit/abort		
	commit/abort	
		commit/abort

- Recoverable but when T_1 fails, T_2 and T_3 should rollback

Dirty Read

Transaction *T*:		Transaction *U*:	
balance = read (a)	$100		
write(a) = balance + 10	$110		
		balance = read(a)	$110
		write(a) = balance + 20	$130
		commit	
abort			

Dirty read

- To ensure recoverability:
  - no dirty read: **reading and commit read** uncommitted data items => read but not commit before write is committed is acceptable
  - no premature write: no update on a data item if another transaction updated it has not been committed
  - Strict execution: delay the reading or updating until the previous transaction that has updated the

same data item has committed/aborted

- Reason of recover:
  - A computer failure (system crash)
    - A hardware or software error occurs in the computer system during transaction execution
    - If the hardware crashes, the contents of the computer's internal memory may be lost.
  - A transaction error
    - Some operation in the transaction may cause it to fail, such as integer overflow or division by zero
    - Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.
  - Local errors or exception conditions detected by the transaction
    - Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found
  - Concurrency control enforcement
    - The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock
  - Disk failure
    - Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
  - Physical problems and catastrophes
    - This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, overwriting disks
- Transaction state:
  - atomic unit of work that is either completed in its entirety or not done at all => for recovery need to keep track of states
  - States: Active, Partially committed, committed, failed, terminated

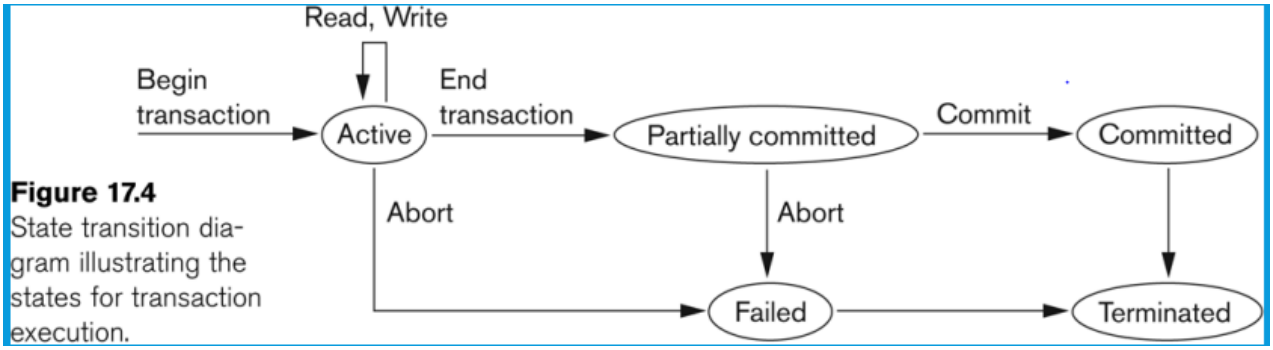


Figure 17.4
State transition diagram illustrating the states for transaction execution.

- Primitive Operations of Transactions:
  - X is the field in memory block, t is a register
  - input(X) : copy the disk block contains X from the disk, output(X): store the disk block contains X to the disk
  - Read(X, t) : run Input(X) and copy the X into local variable t
  - Write(X, t) : run Input(X) and copy t to X

- Transaction
  - Read(A, t); t:= t * 2; Write(A, t); Read(B, t); t:= t * 2; Write(B, t)

Action	t	MEM A	MEM B	DISK A	DISK B
Read(A, t)	8	8		8	8
t:= t * 2;	16	8		8	8
Write(A, t);	16	16		8	8
Read(B, t);	8	16	8	8	8
t:= t * 2;	16	16	8	8	8
Write(B, t)	16	16	16	8	8
Output(A)	16	16	16	16	8
Output(B)	16	16	16	16	16

- Log file for recovery
  - log file: telling something about what some transaction has done by records in the log each important event (e.g. write operations), stored in main memory instead of disk (disk I/O takes a lot of time)
  - flush-log: copy the log file to the disk (when system failure, use the log file in nonvolatile storage to recover)
  - Type of record

    - <START T>: This record indicates that transaction T has begun
    - <COMMIT T>:
      - Transaction T has completed successfully and will make no more changes to database.
      - Because we cannot control when the buffer manager chooses to copy blocks from memory to disk, we cannot be sure that the changes are already on disk when we see <COMMIT T>
    - <ABORT T>: Transaction T could not complete successfully. If transaction T aborts, no change it made can be copied to disk, and it is the job of the transaction manager to make sure that such changes never appear on disk

    - **Notice that commit doesn't means data is already on the disk, but different logging will have further requirement**
  - undo-logging rules:
    - U1: if transaction T modifies database element X, the log need to record in the form <T,X,v> , where
      - T :the transaction
      - X is the variable to change
      - v is the original value
    - U2: <commit> log record must be written to disk ONLY after all DB elements changed by the transaction have been written back to disk

- order: log records indicating changes(and flush log) => actual changes(only OUTPUT) => commit(and flush log)
- example:

Action	t	MEM A	MEM B	DISK A	DISK B	Log
						<START T>
Read(A, t)	8	8		8	8	
t:= t * 2;	16	8		8	8	
Write(A, t);	16	16		8	8	<T, A, 8>
Read(B, t);	8	16	8	8	8	
t:= t * 2;	16	16	8	8	8	
Write(B, t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						(1)
Output(A)	16	16	16	16	8	(2)
Output(B)	16	16	16	16	16	(2)
						<COMMIT T> (3)
FLUSH LOG						

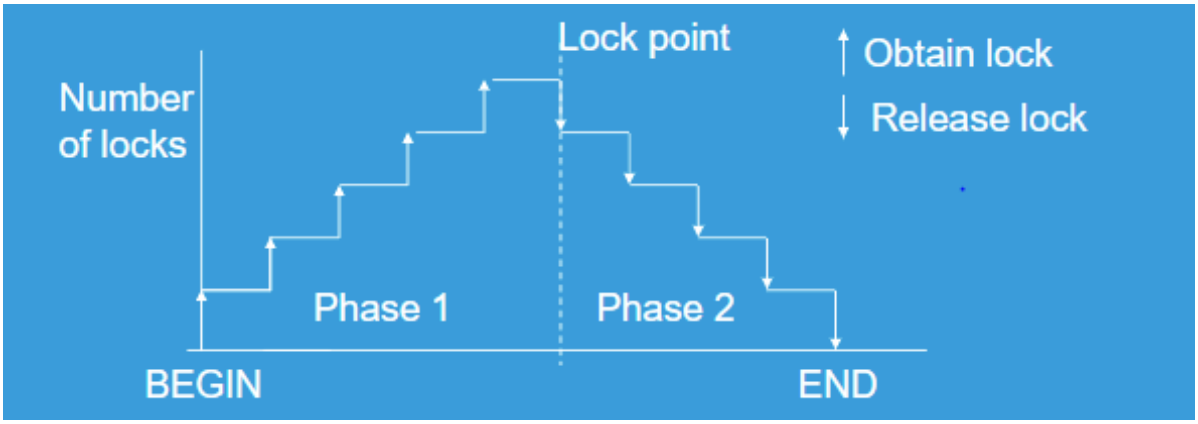
- recovery using undo-logging:
  - When system failure, there might be partial change, which need to be moved during recovery
  - if we find <COMMIT T>, all changes are on the disk before it
  - If we find <START T> but no commit T => might be error, need undo this transaction
  - Rule U1 assures all changes are recorded as <T,X,v>, we can write v to X.
  - Steps:
    - Recovery manager scans the log from the end and remember all transactions T with <COMMIT T> record or an <ABORT T> record
    - if it sees <T,X,v>
      - if there is <COMMIT T> after it, do nothing;
      - Otherwise, write v to X;
      - After making the changes, the manager must write a log record <ABORT T> for each incomplete transaction that was not previously aborted and then flush the log.
- redo logging: Redo the uncertain(not committed) operations
  - undo logging may have the problem that **we cannot commit a transaction without first writing all changed data to disk**
  - Difference between undo and redo:
    - undo ignore committed ones and cancel incomplete transaction; redo ignores incomplete and repeat the changes made by committed ones (overwrite the possible partial result by the previous operations)
    - undo requires COMMIT after write to disk; redo requires COMMIT before write to disk(OUTPUT)
  - redo-logging rules:
    - R1: Any change to X must recorded as <T, X, v > followed by <COMMIT T> where v is the **new value**
    - order: log file indicates changes -> COMMIT(and flush log) -> change (OUTPUT)
    - e.g.

Action	t	MEM A	MEM B	DISK A	DISK B	Log
						<START T>
Read(A, t)	8	8		8	8	
t:= t * 2;	16	8		8	8	
Write(A, t);	16	16		8	8	<T, A, 16>
Read(B, t);	8	16	8	8	8	
t:= t * 2;	16	16	8	8	8	
Write(B, t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
Output(A)	16	16	16	16	8	
Output(B)	16	16	16	16	16	

- recover redo log:
  - Identify the committed transactions
  - Scan the log forward from the beginning, if meet <T, X, v>
    - if T is not a committed transaction, do nothing
    - if T is a committed transaction, write value v for database element X
    - For each incomplete transaction T, the manager must write a log record <ABORT T> for each incomplete transaction that was not previously aborted and then flush the log.\

Lecture 10: Concurrency Control

- Purposes of Concurrency control:
  - Preserve database consistency to ensure all schedules are serializable
  - Maximize the system performance
  - e.g.: If in concurrent execution environment, if T_1 conflicts with T_2 over a data item A, then the existing concurrency control decides whether T_1 or T_2 should get the A and whether the other transaction is rolled-back or waits
- Two-Phase Locking tech
  - locking is an operation which secures both permission to read and write a data item for a transaction
  - unlocking is an operation which removes these permissions from the data item
  - both are atomic operations
- B2PL (Basic two phase locking)
  - each data item has a lock associated with it (e.g. a lock entry in the lock table)
  - the scheduler creates a lock operation $ol_i[x]$ for each received operation $o_i[x]$
  - Rules: When the scheduler receives an operation $p_i[x]$, it tests
    - if $pl_i[x]$ conflicts with some $ql_i[x]$ that is already set. If so $p_i[x]$ is delayed and T_i is forced to wait until it can get the lock
    - Else $pl_i[x]$ is set and $P_i[x]$ is sent to DM (data manager)
    - **Once the scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction (on any data item) because other is holding it**
  - The two phases : growing and shrinking
  - Can guarantee that all pairs of conflicting operations of 2 transactions are scheduled in the same order (T_1 -> T_2 or T_2 -> T_1 and No T_1 <-> T_2)
    - The graph records the number of locks within **one transaction**



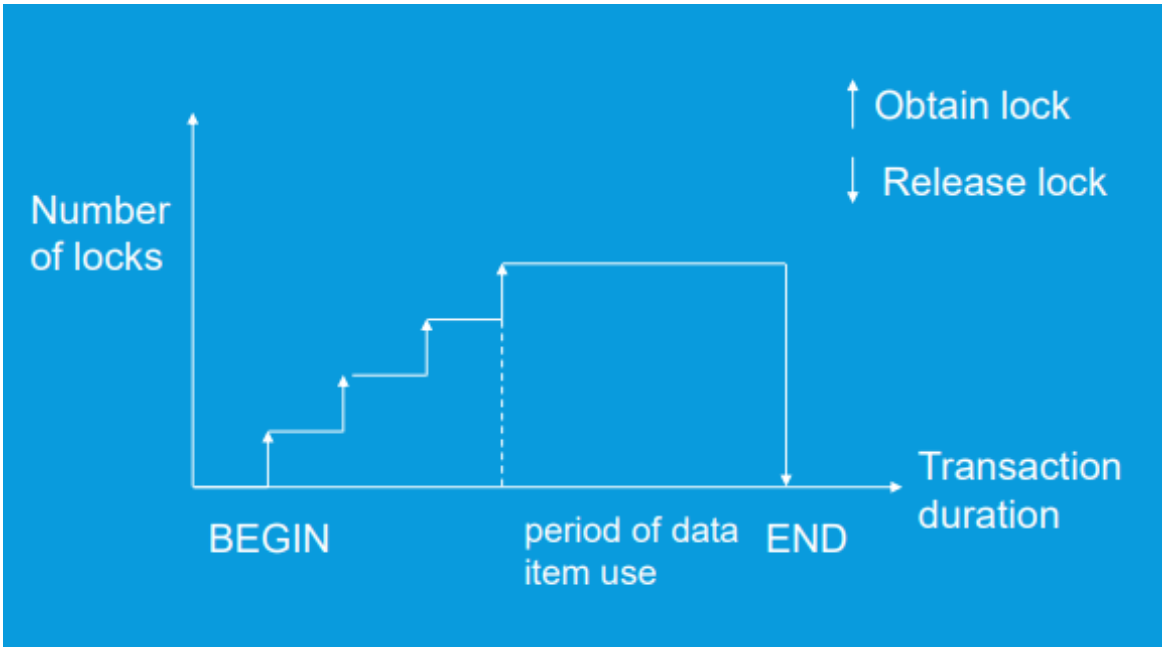
- C2PL (Conservative two phase locking) => like the prevention of hold and wait
  - Avoid deadlocks and abort of transactions by requiring each transaction to obtain all of its lock **before any of its operations are submitted to the DM**
  - Each transaction pre-declares its read-set and write-set of data items to the scheduler
    - What are the locks (data items) to be accessed by the transaction?
  - The scheduler tries to set all of the locks needed by the transaction **ALL at ONCE**
  - Set lock of a transaction in one step and lock release in another step => the graph is like a rectangle
    - If all the locks can be set, the operations will be submitted to the DM for processing
    - After the DM acknowledges the processing of Ti's last database operation, the scheduler may release all of Ti's locks
    - If any of the locks cannot be requested => does not grant any of T_i's lock, and T_i is inserted into a waiting queue, any of the locks of a complete transaction, it examines the waiting queue to see if it can grant all the lock requests of any waiting transactions
    - In Conservative 2PL, if a transaction Ti is waiting for a lock held by Tj, Ti is holding no locks (no hold and wait situation → no deadlock)

T_1	T_2
	Write(a)
Write(a)	
	Write (b)
	Commit
Write(b)	
Commit	

T_1	T_2
	WriteLock(a,b)
	Write(a)
WriteLock(a,b) → blocked	
	Write(b)
	Commit
	ReleaseLock(a,b)
WriteLock(a,b)	
Write(a)	
Write(b)	
Commit	
ReleaseLock(a,b)	

- S2PL (Strict two phase locking) => combination of B2PL and C2PL
  - B2PL only defines the earliest time when the schedule may release a lock for a transaction (i.e. after the operation is finished)
  - S2PL requires the scheduler to release all of transaction's locks altogether

        - The lock is released when the transaction is in state of COMMIT or ABORT
    - Compare with B2PL => longer holding time of lock and lower concurrency
    - Compare with C2PL => shorter holding time (don't need to request at one time) of lock and higher concurrency



T_1	T_2
	Write(a)
Write(a)	
	Write (b)
	Commit
Write(b)	
Commit	

T_1	T_2
	WriteLock(a)
	Write(a)
WriteLock(a) → blocked	
	WriteLock(b)
	Write(b)
	Commit
	ReleaseLock(a,b)
WriteLock(a)	
Write(a)	
WriteLock(b)	
Write(b)	
Commit	
ReleaseLock(a,b)	

- Performance: S2PL is better than C2PL when the transaction workload is not heavy since the lock holding time is shorter in S2PL, but when heavy workload C2PL is better because deadlock may occur in S2PL (T_1 locks b in the first operation of the above graph)
- Implementation Issue: Essential Components
    - Lock modes:
        - Shared mode for read: More than one transaction can apply shared lock on X **for reading** its value but **no write lock** can be applied on X by any other transaction

    - Exclusive mode for write: Only **one write lock on X** can exist at any time and **no shared lock can be applied by any other transactions** on X
    - Conflict: RW, WR, WW
- Lock Manager: managing locks on data items
- Lock table:
    - An array to show the lock entry for each data item
    - Each entry of the array stores the identify of transaction that has set a lock on the data item including the mode
    - The resource or data item is defined for one database? One record? One field? Lock granularity (粒度) (Coarse granularity vs. fine granularity) => how to define common resource
        - Larger granularity (smaller number of locks) → higher conflict probability but lower locking overhead
        - How to detect lock conflict for insertion operations from a transaction?
            - A transaction reads all data items and another one inserts a new item
- Lock and unlock

➢ The following code performs the lock operation:

```
01: if LOCK(X) = 0 (*item is unlocked*)
02: then LOCK(X) ← 1 (*lock the item*);
03: else begin
04:     wait (until LOCK(X) = 0 and the lock
          manager wakes up the transaction);
05:    go to Line 01;
06  end;
```

➢ The following code performs the unlock operation:

```
01: LOCK(X) ← 0 (*unlock the
      item*);
02: if any transactions are
      waiting then wake up one of
      the waiting transactions;
```

- Rules for shared/exclusive locking scheme:
    - read or write lock must be issued before any read operation in T
    - write lock must be issued before any write operation in T
    - unlock must be issued after all read operation or one write operation is finished
    - read and write lock will not be issued if T already holds a read lock or a write lock (this may be relaxed)
    - unlock is not issued unless it already holds a read lock or a write lock
- Read lock

```
01: if LOCK(X) = "unlocked" then
02:    begin
03:       LOCK(X) ← "read-locked";
04:       no_of_reads(X) ← 1;
05:    end;
06: else if LOCK(X) = "read-locked" then
07:    no_of_reads(X) ← no_of_reads(X) +1;
08: else begin
09:    wait (until LOCK(X) = "unlocked" and the lock manager wakes up the
          transaction);
10:    go to Line 01;
11: end;
```

- Write lock

```
01: if LOCK(X) = "unlocked" then
02:    LOCK(X) ← "write-locked";
03: else begin
04:    wait (until LOCK(X) = "unlocked" and the lock manager wakes up
          the transaction);
05:    go to Line 01
06: end;
```

- Unlock

```
01: if LOCK(X) = "write-locked" then
02:    begin
03:       LOCK(X) ← "unlocked";
04:       wakes up one of the transactions, if any;
05:    end
06: else if LOCK(X) = "read-locked" then
07:    begin
08:       no_of_reads(X) ← no_of_reads(X) - 1;
09:       if  no_of_reads(X) = 0 then
10:          begin
11:             LOCK(X) = "unlocked";
12:             wake up one of the transactions, if any;
13:          end;
14:    end;
```

- Lock Conversion (read lock to write lock)
  - Rule 4,5 are relaxed to do the lock conversion

- Lock conversion (read lock to write lock)
  - Lock upgrade: existing read lock to write lock

```
01: if Ti has a read-lock(X) and Tj has no read-lock(X) (i ≠ j) then
02:     convert read-lock(X) to write-lock(X);
03: else
04:     force Ti to wait until Tj unlocks X;
```

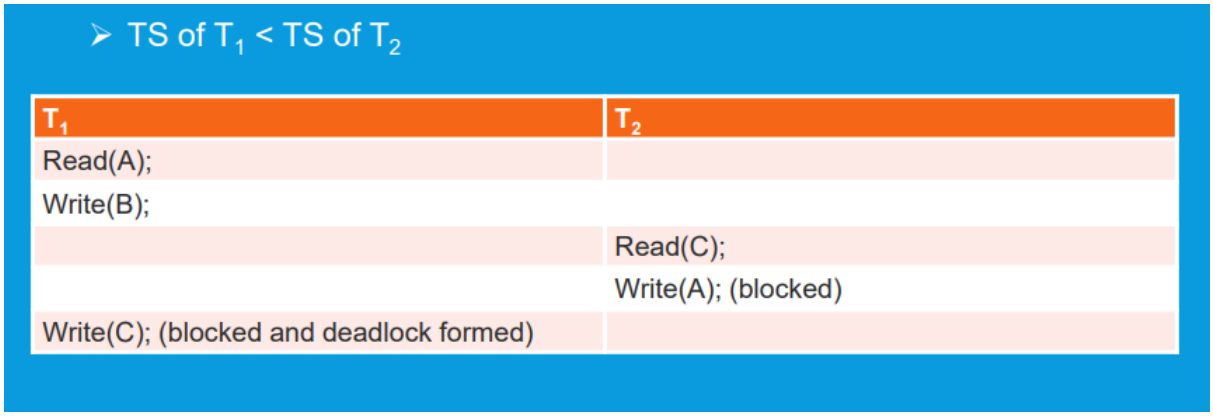
  - Lock down grade: existing write lock to read lock

```
01: if Ti has a write-lock(X) (*no transaction can have any lock on X*)
02:     convert write-lock(X) to read-lock(X);
```

- Dead lock

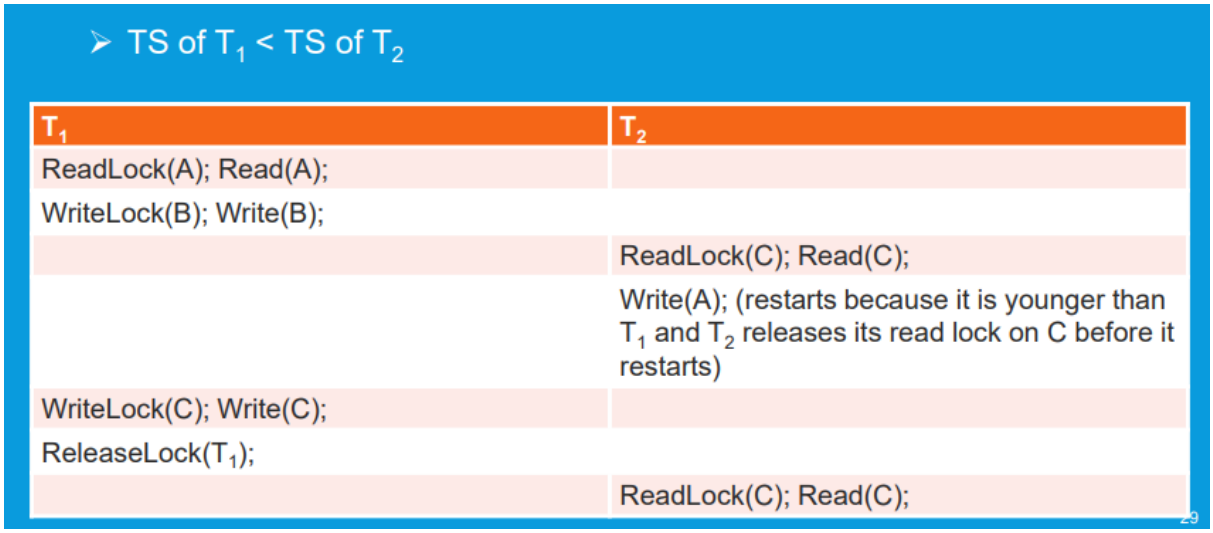
T_1	T_2
Read_lock(Y);	
Read_item(Y);	
	Read_lock(X);
	Read_item(X);
Write_lock(X); (waits for X)	
	Write_lock(Y); (waits for Y)

- Dead lock prevention:
  - Hold and wait : locks all items before it begins execution, used in C2PL
  - cyclic wait => use timestamp
    - Each transaction is assigned a unique time stamp (its creation time or creation time + site ID for distributed databases)
    - wait(old)-die(young) rule (non-preemptive): A transaction can be allowed to wait for a lock iff it is older than the holder, otherwise it is restarted with **the same timestamp** (second time the lock might be available)
    - wound(old)-wait(young) rule (preemptive): A transaction can be allowed to wait for a lock iff it is younger than the holder, otherwise the holder is restarted with **the same timestamp** and the lock is granted to the requester
    - cyclic wait is broken by both of two methods (T_1->T_2->T_3->...->T_n -> T_1)
      - wait-die => the last request in the chain will restart T_n
      - wound-wait => the older one will always execute at first
    - Examples:



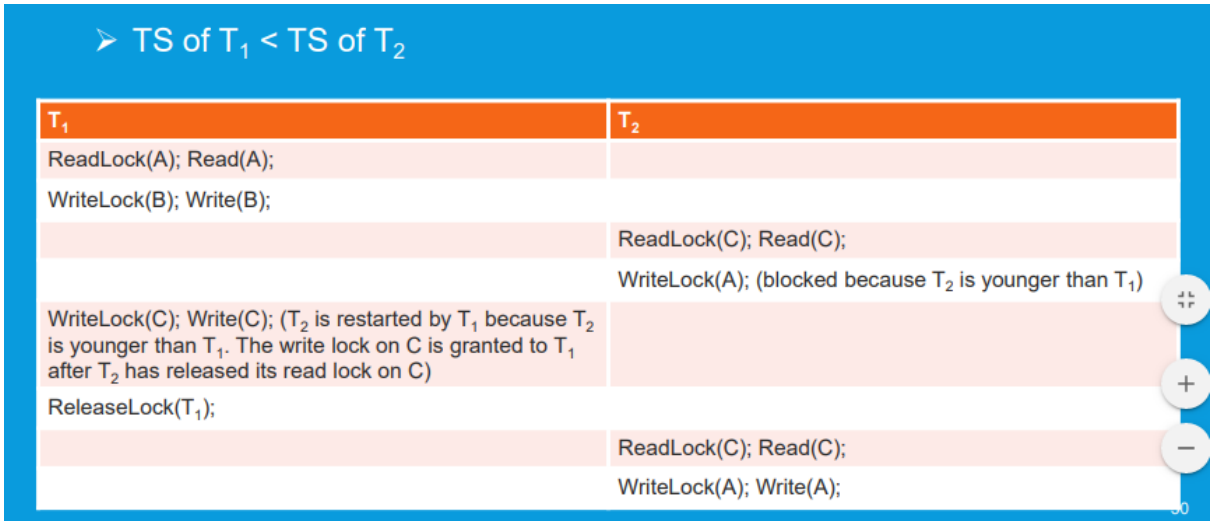
➢ TS of T_1 < TS of T_2

T_1	T_2
Read(A);	
Write(B);	
	Read(C);
	Write(A); (blocked)
Write(C); (blocked and deadlock formed)	



➢ TS of T_1 < TS of T_2

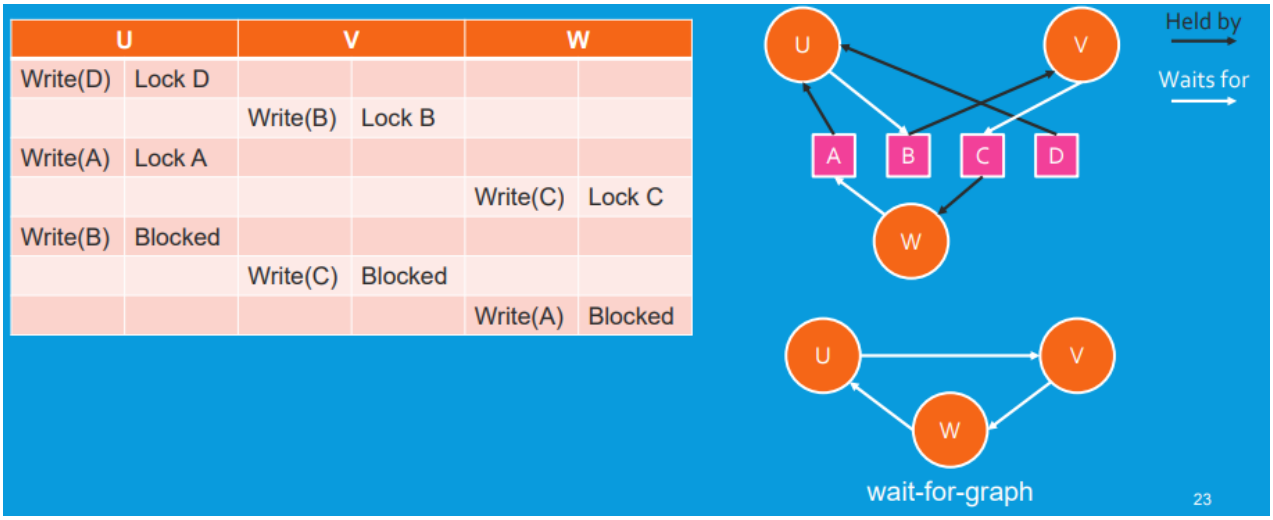
T_1	T_2
ReadLock(A); Read(A);	
WriteLock(B); Write(B);	
	ReadLock(C); Read(C);
	Write(A); (restarts because it is younger than T_1 and T_2 releases its read lock on C before it restarts)
WriteLock(C); Write(C);	
ReleaseLock(T_1);	
	ReadLock(C); Read(C);



➢ TS of T_1 < TS of T_2

T_1	T_2
ReadLock(A); Read(A);	
WriteLock(B); Write(B);	
	ReadLock(C); Read(C);
	WriteLock(A); (blocked because T_2 is younger than T_1)
WriteLock(C); Write(C); (T_2 is restarted by T_1 because T_2 is younger than T_1. The write lock on C is granted to T_1 after T_2 has released its read lock on C)	
ReleaseLock(T_1);	
	ReadLock(C); Read(C);
	WriteLock(A); Write(A);

- Dead lock detection and resolution
  - detection: in some approaches, deadlocks are allowed (e.g. in S2PL), maintain a wait-for-graph, if cycle exists, one transaction involved in the cycle is selected and rolled-back
  - Resolution: A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj, Tj waits for Tk and Tk waits for Ti occurs, this creates a cycle. One of the transactions will be chosen to abort.

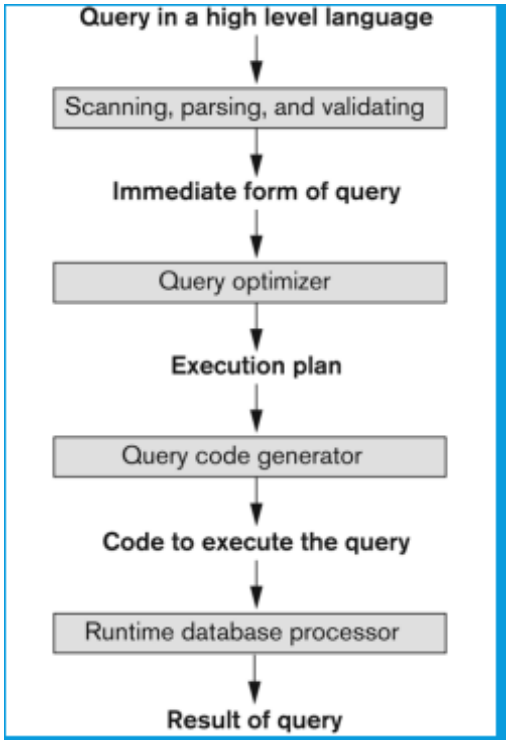


- Starvation
  - Occurs when a particular transaction consistently waits for restarts and never gets a chance to proceed further
  - For e.g., in dead lock, one of the transaction is always selected to roll back

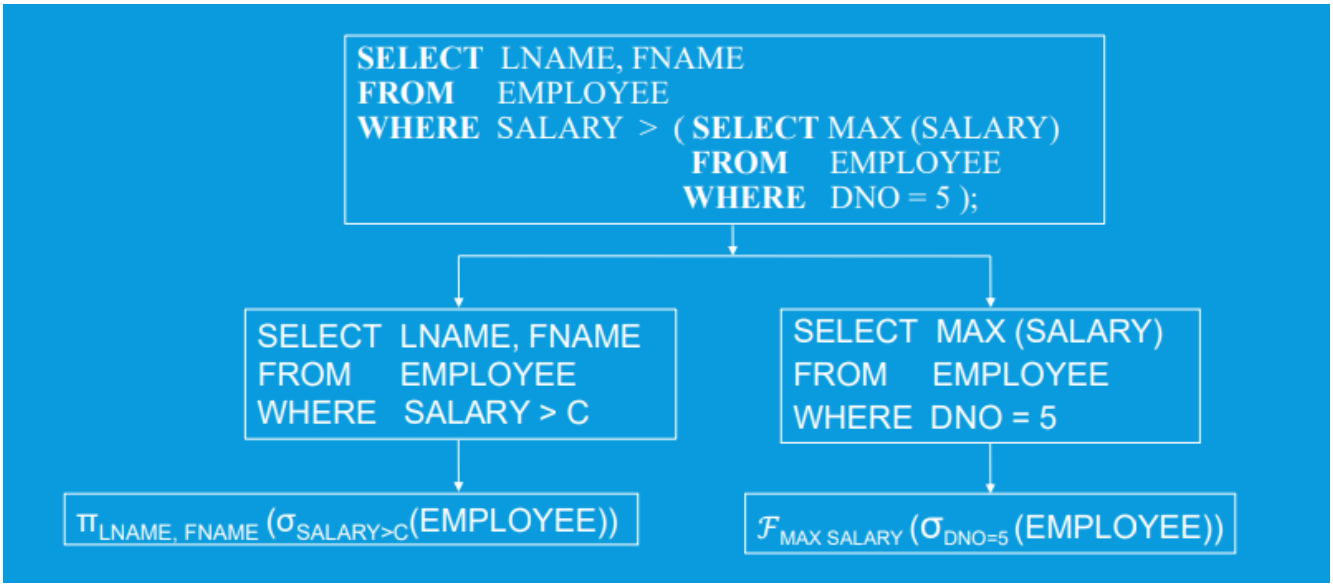
---

Lecture 11: Query Optimization

- Introduction to Query Optimization
  - Query optimization: process of choosing a suitable execution strategy for processing a query
  - May not optimal but is a reasonably efficient strategy
  - Step of executing a query
    - Scan: Scanner identifies the query tokens (keywords, attribute names, relation names)
    - Parse: Parser checks the query syntax
    - Validate: Checks that all attributes and relation names are valid
  - two internal representations of a query: Query Tree and Query Graph
  - Code can be
    - interpreted mode: executed directly
    - compiled mode: stored and executed later whenever needed
  - Typical steps when processing a high-level query:



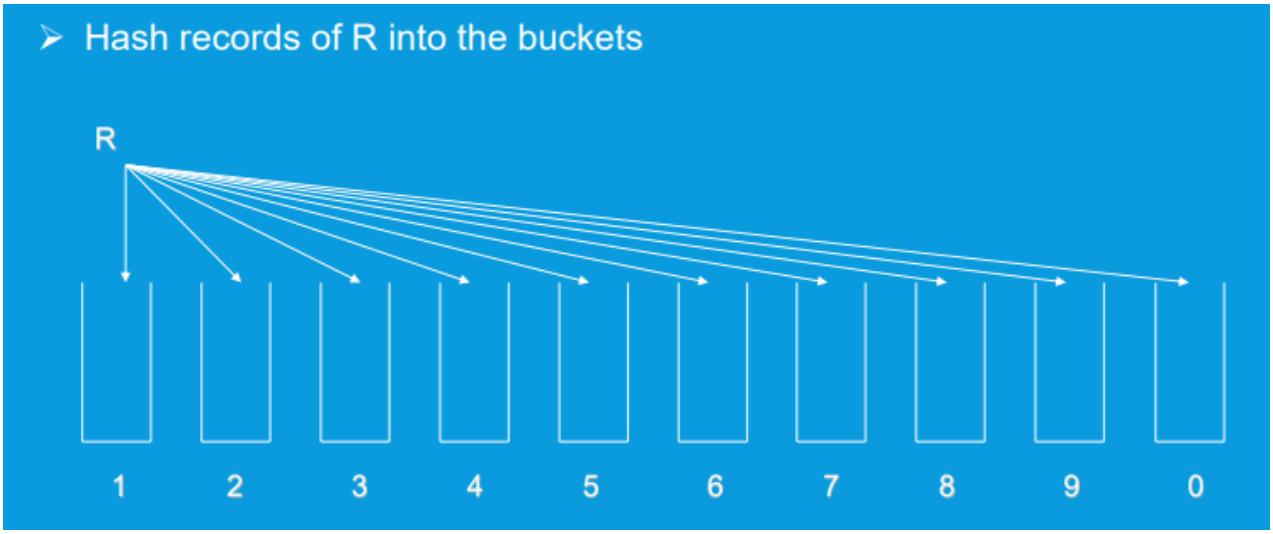
- Translating SQL Queries into Relational Algebra
  - Query block is the basic unit that can be translated into the algebraic operators
  - Query block contains a single SELECT-FROM-WHERE expression as well as GROUP BY and HAVING clause if these are part of the block
  - Nested queries within a query are identified as separate query blocks



- Select operation
  - Types
    - simple SELECT: select condition is only one boolean fomula
    - conjunctive: simple condition connected by AND
    - disjunctive: simple condition connected by OR
  - Search methods
    - S1: Linear search(unordered file): retrieve record sequentially test one by one
    - S2: Binary search(ordered file): Condition: The file is ordered according to the value of the key

        - S3: Using a primary index or hash key (which tells the location) to retrieve a single record (comparation on key and directly retrieve): Condition: the selection condition involves an equality comparison on a key attribute with a primary index
        - S4: Using a primary key to retrieve multiple records (condition: the comparison condition is >, <, >=, <= on a key field with primary index): Compare to find the equal one and retrieve all the preceding or subsequent records
        - S5: Using a clustering index to retrieve multiple records, condition: the selection condition involves an equality comparison on a non-key attribute with a clustering index
            - For example , select the employee with department number 5
        - S6: Using a secondary index or B+-tree
            - The secondary index may be created on a field that is a candidate key and has a unique value in every record, or a non-key field with duplicated values
            - The index tells the location of the records
                - We can retrieve records on conditions involving >,>=, <, or <=. (e.g., range queries)
                - Range query example: 30000<=salary<=35000
        - S7: Conjunctive selection:
            - A conjunctive condition contains several simple conditions connected with AND
            - Use an attribute has an access path that permits the use of one of the methods S2 to S6 to retrieve the records and then check whether each of retrieved record satisfies the remaining simple conditions in the conjunctive condition
        - S8: Conjunctive selection using a composite index
            - if an index has been created on the composite key we can use it directly
        - S9: Conjunctive selection by intersection of record pointers
            - Condition: secondary indexes are available on all (or some of because one can be the physical order key) the fields involved in equality comparison conditions in the conjunctive condition and the indexes include record pointers (rather than block pointers)
            - Index pointing to the records (dense index)
            - Method:
                - Each index can be used to retrieve the record pointers that satisfy the individual condition
                - The intersection (common) of these sets of record pointers gives the record pointers that satisfy the conjunctive condition
                - E.g., use the indexes to get dno > 5 and salary > 30000
    - Selection Optimization
        - Disjunctive condition is much harder to process and optimize
        - If any one of the conditions does not have an access path, we have to use the brute force linear search approach
        - If an access path exists on every condition, we can optimize the selection by retrieving the records satisfying each condition and then applying the union operation to remove duplicate records
        - If the appropriate access paths that provide record pointers exist for every condition, we can union record pointers instead of records => pointer is smaller and the time to get the data block is longer
- Implementing the Join operation
    - The join operation is one of the most time-consuming operations in query processing
    - Background

    - Size of the main memory n_B: 7 blocks
        - 1 block to store the result
        - 1 block to read one block of inner loop
        - 5 blocks for out loop
    - DEPARTMENT file consists of r_D = 50 records in b_D = 10 blocks
    - EMPLOYEE file consists of r_E = 6000 records in b_E = 2000 blocks
- Join selection factor: The fraction of records in a file that will be joined with records in the other file
    - The join: $DEPARTMENT *_{MGRSSN=SSN} EMPLOYEE$
    - Suppose the **secondary index** exist for both of the files on MGR__SSN and SSN with the level X_{SSN} = 4 and X_{MGRSSN} = 2 and the selection factor of SSN = MGR_SSN is 1
    - First retrieves each EMPLOYEE record and the uses the index on MGR_SSN of DEPARTMENT to find a matching DEPARTMENT record
        - no. of block access = $b_E + (r_E * (X_{MGRSSN} + 1)) = 2000 + 6000 * 3 = 20000$
    - Retrieve each DEPARTMENT at first
        - no. of block access = $b_D + (r_D * (X_{MGRSSN} + 1)) = 10 + 50 * 5 = 260$
- J1: Nested (inner-outer) loop approach (brute force)
    - For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition t[A] = s[B]
    - Choice of outer-loop:
        - choose employee as outer loop: $(n_B - 2)$ blocks of employee file each time
        - No. of times to retrieve $(n_B - 2)$ blocks: $b_E/(n_B - 2)$
        - No. of time to retrieve inner loop blocks: $b_D * b_E/(n_B - 2)$
        - Total number of block read access = $b_E + b_D * b_E/(n_B - 2)$
        - choose the one with less block as the outer loop
- J2: Using an access structure to retrieve the matching records
    - if an index exists for one of the two join attr, say, B of S,
    - retrieve each record t in R, one at a time
    - use access structure to get all matching records s from S satisfy s[B] = t[A]
- J3: Sort-merge join
    - Condition: the records of R and S are physically sorted by value of join attr A and B respectively
    - Method:
        - Both files are scanned in order of the join attributes, matching the records that have the same values for A and B
        - In this method, if the joining attribute is sorted, the records of each file are scanned only once each for matching with the other file
        - Otherwise, sort the records first before matching. Sorting cost = O(n log n)
- J4: Hash join
    - The records of files R and S are hashed by the same function on A and B
    - Step 1 (partitioning phase). A single pass through the file with fewer records (say, R) hashes its records to the hash file buckets
    - Step 2 (probing phase). A single pass through the other file (S) then hashes each of its records to the appropriate bucket, where the record is combined with all matching records from R



- Implementing aggregate operation
  - SUM, COUNT, and AVG
  - Methods
    - For a dense index: apply the computation to the values in index
    - for a non-dense index: actual number of records associated with each index entry are used for computation (multiple records indexed by an index entry)
  - GROUP BY: This operator is applied to subsets of a table. Employee relation is hashed or sorted to partition the file into groups such that each group has the same grouping attribute
    - With clustering index on the grouping attribute: records are already partitioned (grouped) on that attribute
- Using Heuristics (啟發式) in Query Optimization
  - Heuristic rule may be applied to modify the internal representation of a query (e.g., a query tree) to improve performance
  - Process for heuristics optimization
    - The parser generates an initial internal representation
    - Apply heuristics rules to optimize the internal representation
    - A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query
  - Main heuristic is to **apply the operations that reduce the size of intermediate results first**
    - **SELECT and PROJECT** (reduce size operation) before **JOIN** or other binary operations
    - The size of the resulting file from a binary operation(JOIN) is usually a multiplicative function of the sizes of input files
- Query Tree: A tree data structure that corresponds to a relational algebra expression
  - query as leaf nodes represents the input relations
  - order of execution of operations starts at the leaf and ends at the root node
  - General Transformation Rules for Relational Algebra Operations.
    - There are many rules for transforming relational algebra operations into equivalent ones. (Here we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a different order but the two relations represent the same information, we consider the relations equivalent.)

1. Cascade of σ : A conjunctive selection condition can be broken up into a cascade (sequence) of individual s operations:

$$\sigma_{c1\ AND\ c2\ AND\ \ldots\ AND\ cn}(R) = \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$$

2. Commutativity of σ : The σ operation is commutative:

$$\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$$

3. Cascade of π : In a cascade (sequence) of π operations, all but the last one can be ignored:

$$\pi\ List_1(\pi\ List_2(\ldots(\pi\ List_n(R))\ldots)) = \pi\ List_1(R)$$

4. Commuting σ with π : If the selection condition c involves only the attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:

$$\pi\ A_1, A_2, \ldots, A_n(\sigma\ c(R)) = \sigma\ c(\pi\ A_1, A_2, \ldots, A_n(R))$$

5. Commutativity of * (or ⋈): The * operation is commutative:

$$R * S = S * R$$

Notice that, although the order of attributes may not be the same in the relations resulting from the two joins, the "meaning" is the same because order of attributes is not important in the alternative definition of *relation* that we use here. The ⋈ (and $\bowtie_c$) operation is commutative in the same sense as the * operation.

6. Commuting σ with * (or X): If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:

$$\sigma\ c(R * S) = (\sigma\ c(R)) * S$$

Alternatively, if the selection condition c can be written as (c1 and c2), where condition c1 involves only the attributes of R and condition c2 involves only the attributes of S, the operations commute as follows:

$$\sigma\ c(R * S) = (\sigma\ c1(R)) * (\sigma\ c2(S))$$

The same rules apply if the * is replaced by a X operation. These transformations are very useful during heuristic optimization.

7. Commutativity of set operations: The set operations ∪ and ∩ are commutative, but – is not

8. Commuting π with $\bowtie_c$ (or X): Suppose that the projection list is $L = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$, where $A_1, \ldots, A_n$ are attributes of R and $B_1, \ldots, B_m$ are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

$$\pi\ L(R \bowtie_c S) = (\pi\ A1, \ldots, An(R)) \bowtie_c (\pi\ B1, \ldots, Bm(S))$$

If the join condition c contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed. For example, if attributes $A_{n+1}, \ldots, A_{n+k}$ of R and $B_{m+1}, \ldots, B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

$$\pi_L(R \bowtie_c S) =$$

$$\pi\ L((\pi\ A_1,\ldots,A_n,A_{n+1},\ldots,A_{n+k}(R)) \bowtie_c (\pi\ B_1,\ldots,B_m,B_{m+1},\ldots,B_{m+p}(S)))$$

9. Associativity of *, X, U, and ∩ (X: cross product): These four operations are individually associative; that is, if q stands for any one of these four operations (throughout the expression), we have

   (R q S) q T = R q (S q T)

   E.g., (R U S) U T = R U (S U T)

10. Commuting σ with set operations: The σ operation commutes with U, ∩, and –. If q stands for any one of these three operations, we have

    σ c (R q S) = (σ c (R)) q (σ c (S))

11. The π operation commutes with U. If q stands for U, we have

    π_L (R q S) = (π_L (R)) q (π_L (S))

12. Other transformations: There are other possible transformations. For example, a selection or join condition c can be converted into an equivalent condition by using the following rules (known as DeMorgan's laws):

    NOT (c1 AND c2) ≡ (NOT c1) OR (NOT c2)

    NOT (c1 OR c2) ≡ (NOT c1) AND (NOT c2)

- Optimization algorithm
  - Using rule 1, break up conjunctive SELECT to cascade(nested) of SELECT operations (permits a greater degree of freedom in moving select operations down different branches of the tree)
  - Using rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as possible
  - Using rule 9 concerning associativity of binary operations, rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive SELECT operations are executed first in the query tree representation
  - Combine a CROSS PRODUCT operation with a subsequent select operation whose condition represents a join condition into a join operation
  - Using rules 3, 4, 8, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed.
  - Identify subtrees that represent groups of operations that can be executed by a single algorithm
- e.g.: for every project located in "Stafford", retrieve the project number, the controlling DEPTNO and the manager's last name, address and birthdate

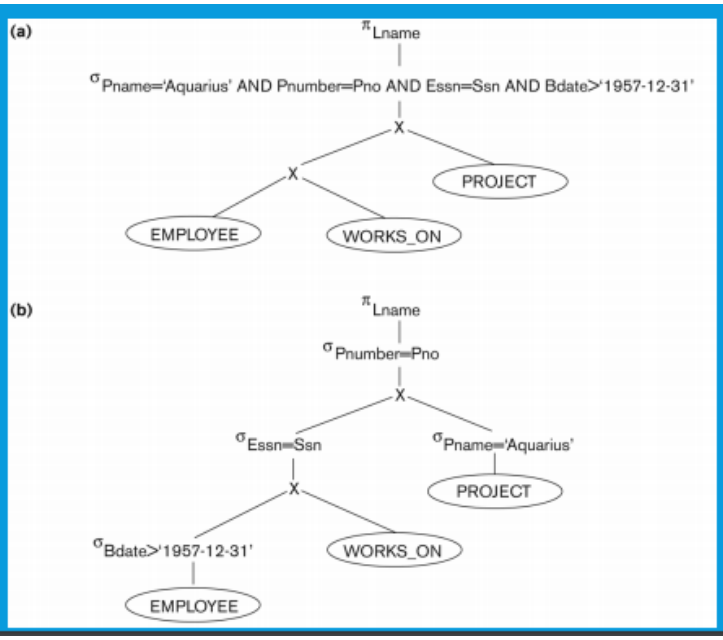
➢ Relation algebra:

$\pi_{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}$ (((σ$_{PLOCATION='STAFFORD'}$(PROJECT)) ⋈ $_{DNUM=DNUMBER}$ (DEPARTMENT)) ⋈ $_{MGRSSN=SSN}$ (EMPLOYEE))

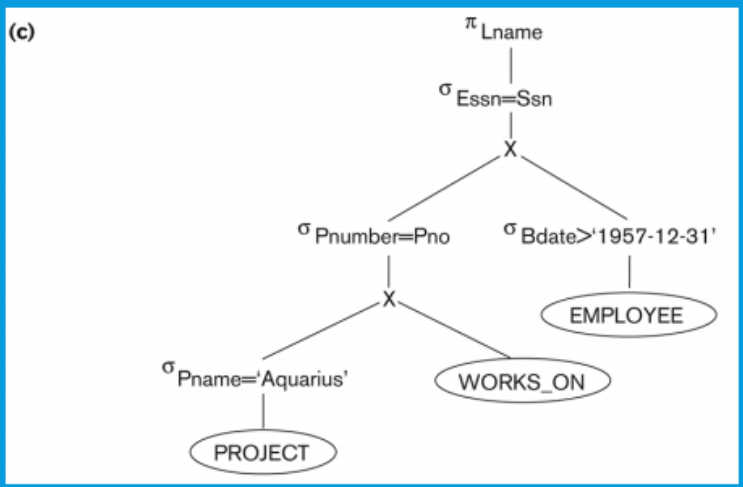
➢ SQL query:

```
Q2: SELECT   P.NUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE
    FROM     PROJECT AS P,DEPARTMENT AS D, EMPLOYEE AS E
    WHERE    P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN AND
             P.PLOCATION='STAFFORD';
```

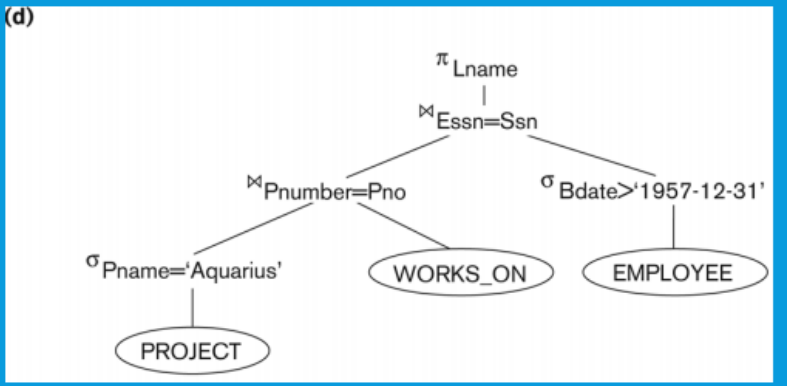
- Figure (a) sows the initial (canonical) query tree for a SQL query
- Figure (b) shows the query tree of after applying Steps 1 and 2 of the algorithm
  - Moving SELECT operations down the query tree



- Figure (c) shows the tree after applying Step 3
  - Applying the more restrictive SELECT operation first



- Figure (d) shows the query tree after applying Step 4
  - Replacing CROSS PRODUCT and SELECT with JOIN operation



- Figure (e) after applying Step 5
  - Moving PROJECT operations down the query tree