Database

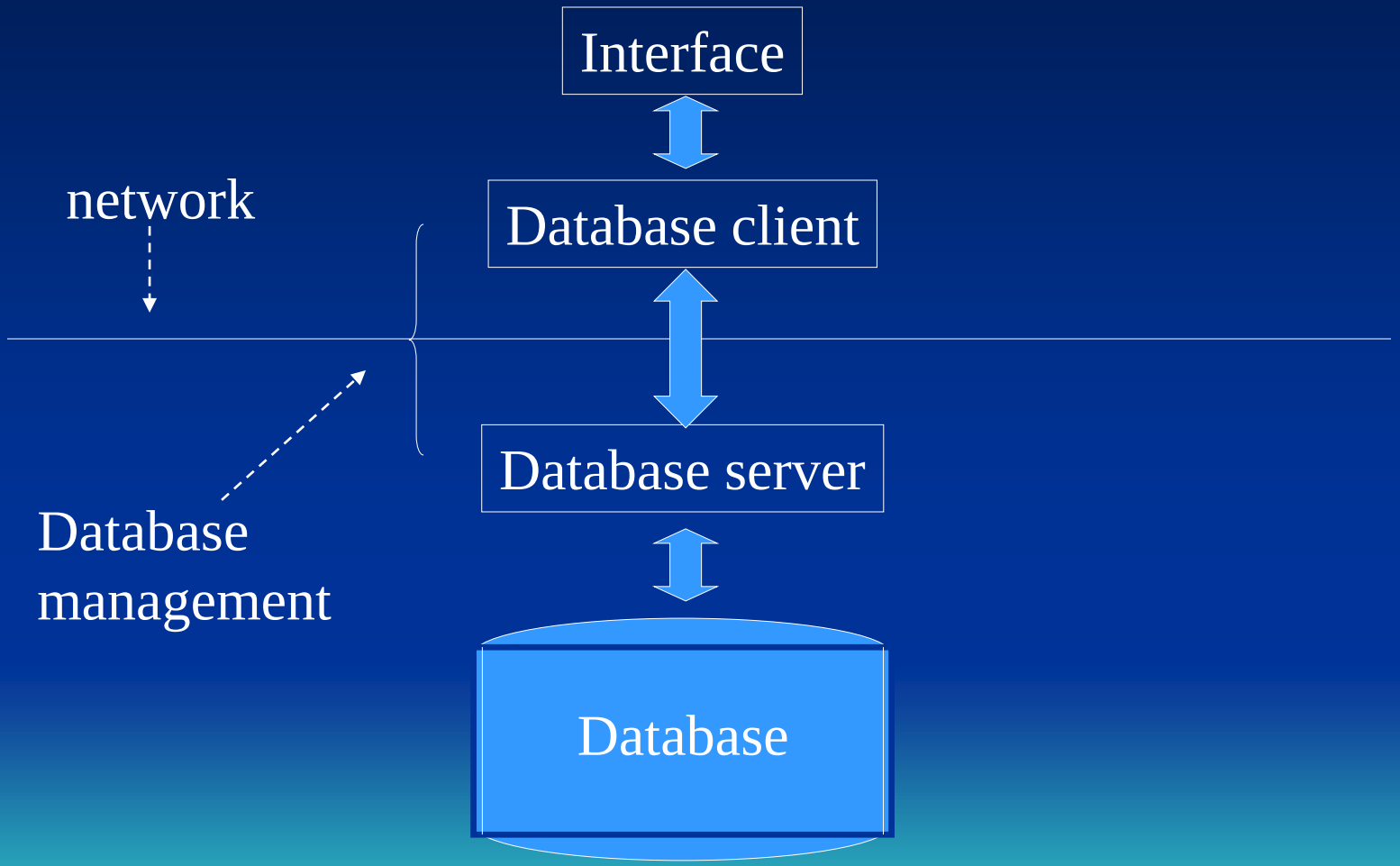
- Database Basics
- SQL Language and JDBC
- Static Hashing and Dynamic Hashing
- Index Techniques
- WEB Databases
- Semi-Structured Data Model
- Programming Language for XML

Not covered

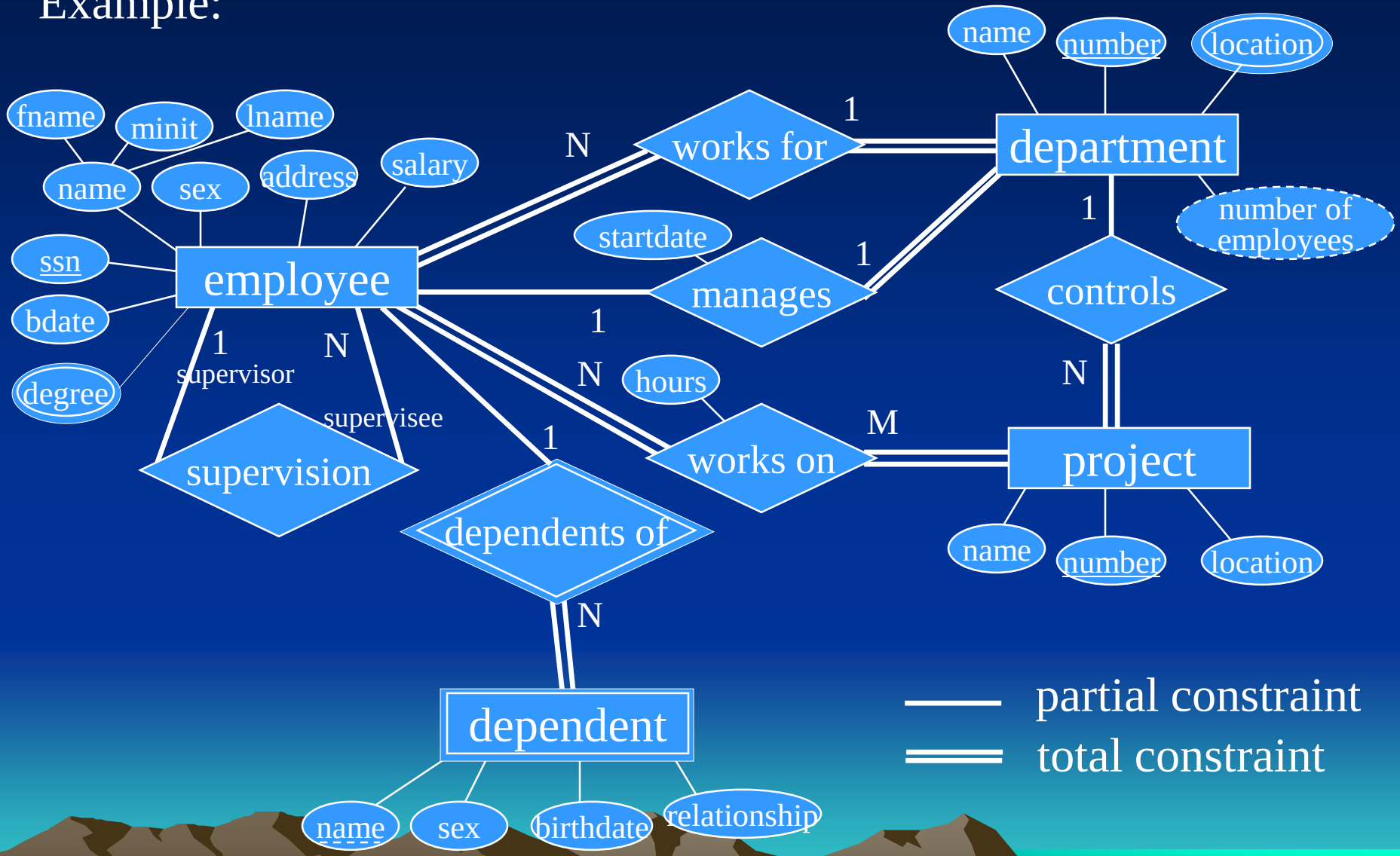
Database Basics

- Client-Server Computer Architecture
- Entity-Relationship Data Modeling
- ER-to-Relation-Mapping

- Client-server database system Architecture



Example:


name
number
location
fname
minit
lname
name
sex
address
salary
N
works for
1
department
1
number of employees
startdate
manages
1
controls
ssn
employee
bdate
1
N
degree
supervisor
supervisee
N
hours
N
1
supervision
works on
M
project
dependents of
name
number
location
N
dependent
name
sex
birthdate
relationship
partial constraint
total constraint

- **ER-to-Relational mapping**

1. Create a relation for each strong entity type
   - For each atomic attribute associated with the entity type, an attribute in the relation will be created.
   - Composite attributes are not included. However the atomic attributes comprising the composite attribute must appear in the pertinent relation.
2. Create a relation for each weak entity type
   - include primary key of owner (an FK - foreign key)
   - owner's PK + partial key becomes PK
3. For each binary *1:1* relationship choose an entity and include the other's PK in it as an FK. Include any attributes of the relationship

4. For each binary *1:n* relationship, choose the *n*-side entity and include an FK with respect to the other entity. Include any attributes of the relationship

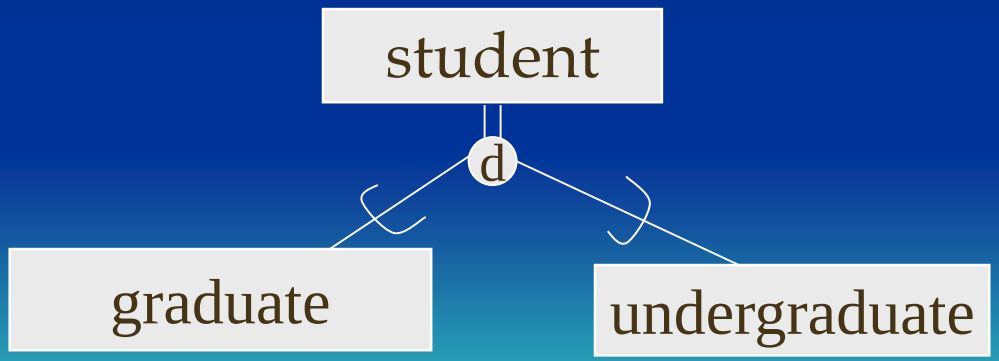
5. For each binary *M:N* relationship, create a relation for the relationship
   - include PKs of both participating entities and any attributes of the relationship
   - PK is the concatenation of the participating entity PKs

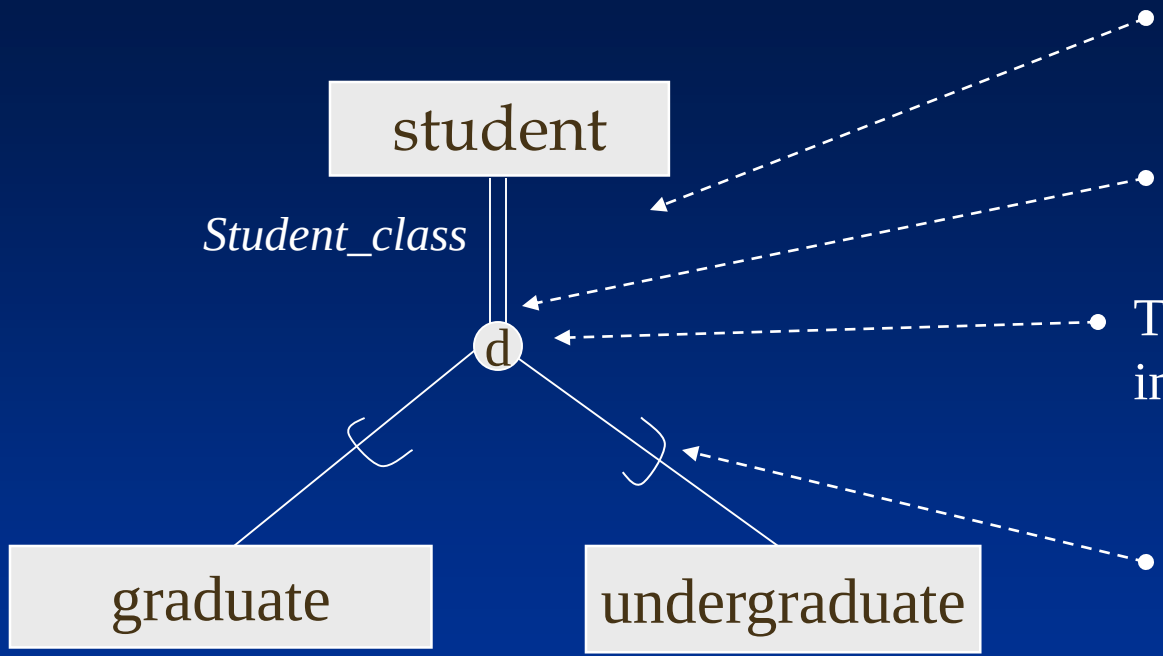
6. For each multivalued attribute create a new relation
   - include the PK attributes of the entity type
   - PK is the PK of the entity type and the multivalued attribute

7. For each *n*-ary relationship, create a relation for the relationship
   - include PKs of all participating entities and any attributes of the relationship
   - PK is the concatenation of the participating entity PKs

- **Specialization and Generalization**

- Specialization is the process of defining a set of sub-entities of some entity type. Generalization is the opposite approach/process of determining a supertype based on certain entities having common characteristics.
  - e.g. employees may be paid by the hour or a salary (part vs full-time)
  - e.g. students may be part-time or full-time; graduate or undergraduate
- these are similar to 1:1 relationships, but they always involve entities of one (super)type
- these are *'is-a'* relationships



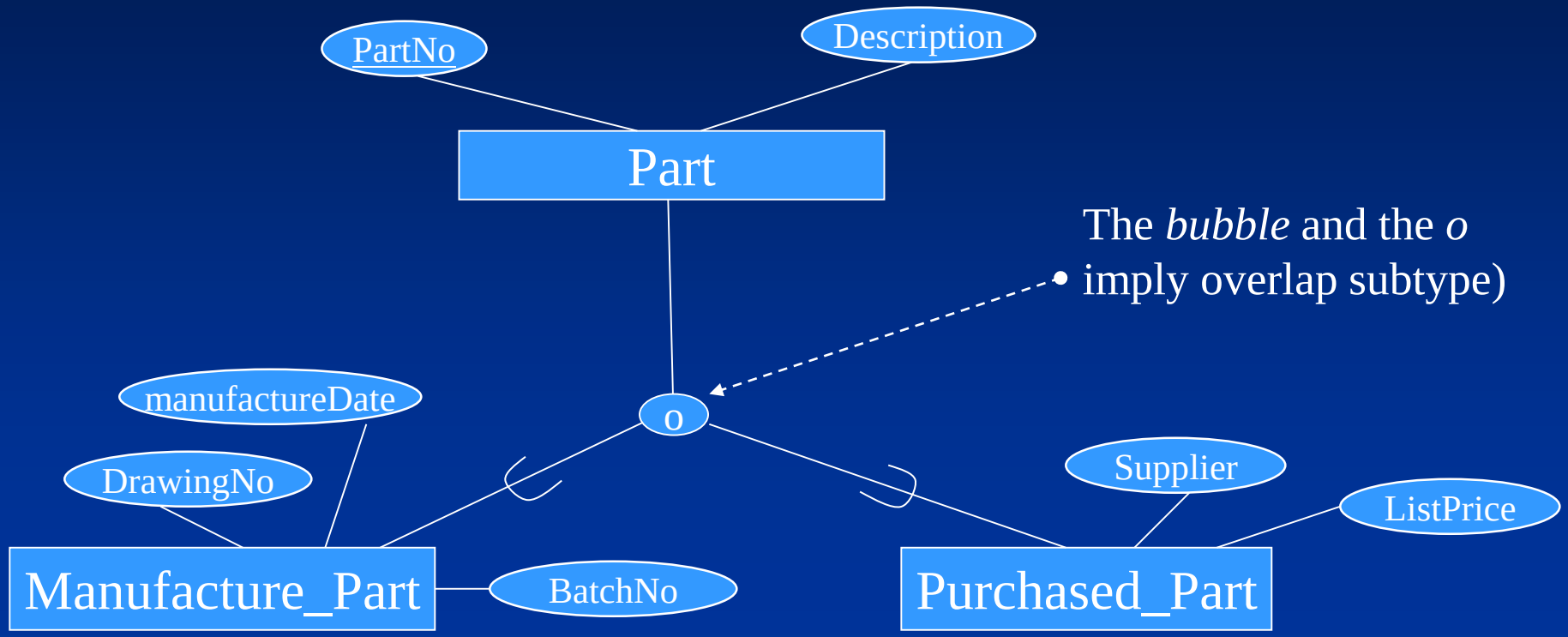


The *bubble* and the *d* imply disjoint subtypes

The arc implies *graduate* and *undergraduate* are subtypes of *student*

- Participation of supertype may be mandatory or optional
- Subtypes may be disjoint or overlapping
- a predicate (on an attribute) determines the subtype: e.g. attribute Student_class

  Student_class = ‘graduate’; Student_class = ‘undergraduate’


PartNo
Description
Part
The *bubble* and the *o* imply overlap subtype)
manufactureDate
o
DrawingNo
Supplier
ListPrice
Manufacture_Part
BatchNo
Purchased_Part

- **Mapping to a relational database**

  4 choices:

  1. Create separate relations for the supertype and each of the subtypes.

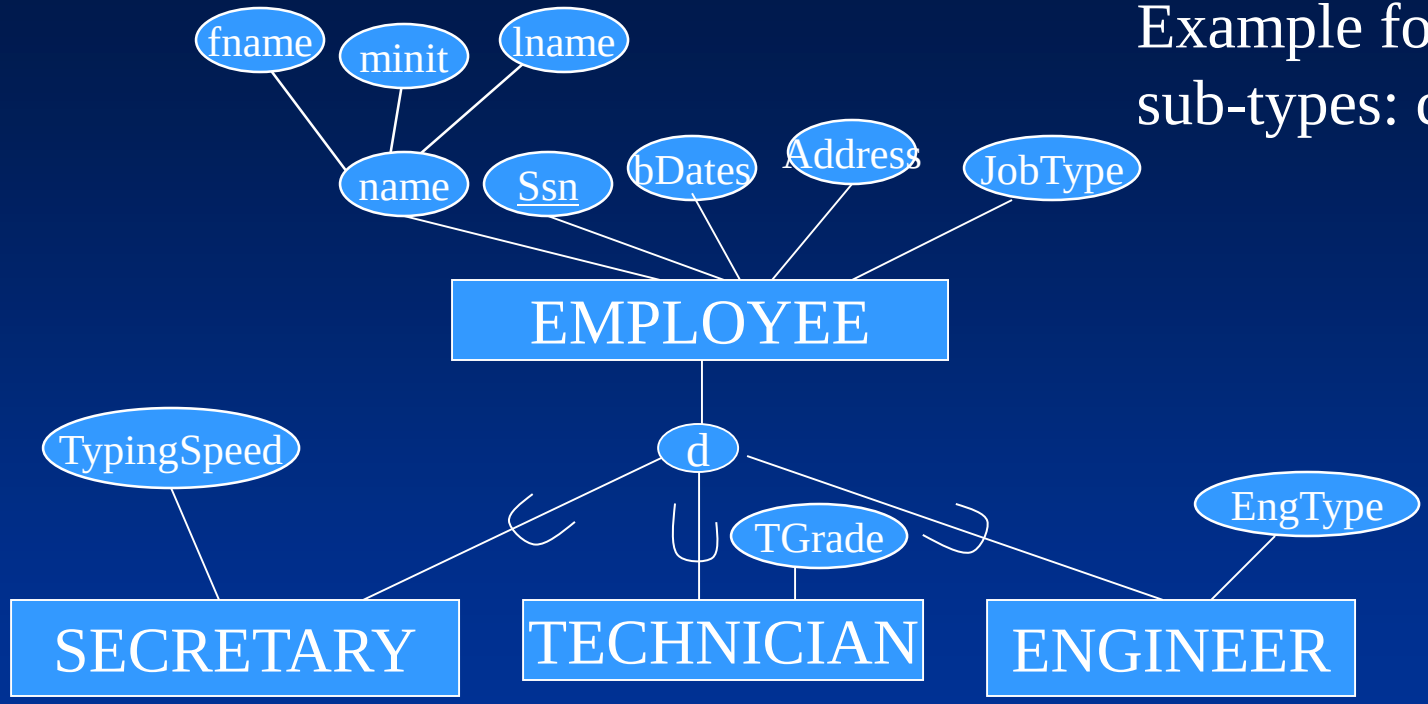
  2. Create relations for the subtypes only - each contains attributes from the supertype.

  3. (**disjoint** subtypes) Create only one relation - includes all of the attributes for the supertype and all for the subtypes, and one discriminator attribute.

  4. (**overlapping** subtypes) Create only one relation - includes all of the attributes for the supertype and all for the subtypes, and one logical discriminator attribute per subtype.

  PK is always the same - determined from the supertype

Example for super- & sub-types: choice 1



EMPLOYEE

fname, minit, lname, ssn, bdate, address, JobType

SECRETARY

Essn, TypingSpeed

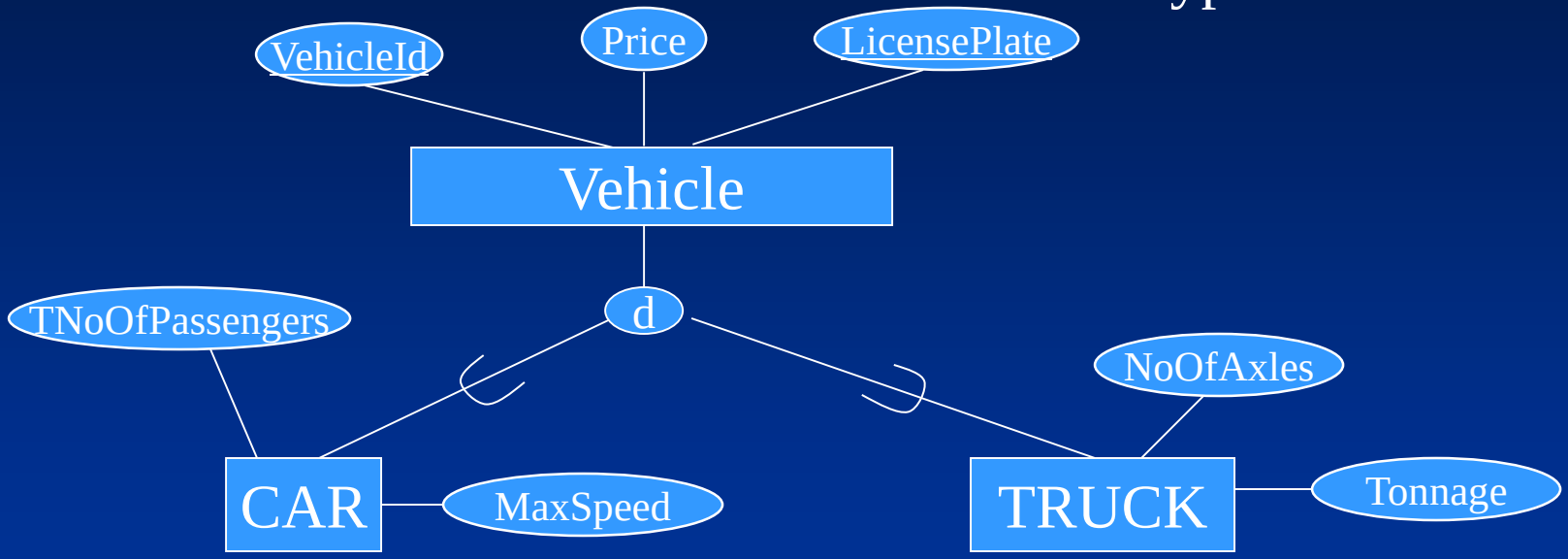
TECHNICIAN

Essn, TGrade

ENGINEER

Essn, EngType

Example for super- & sub-types: choice 2



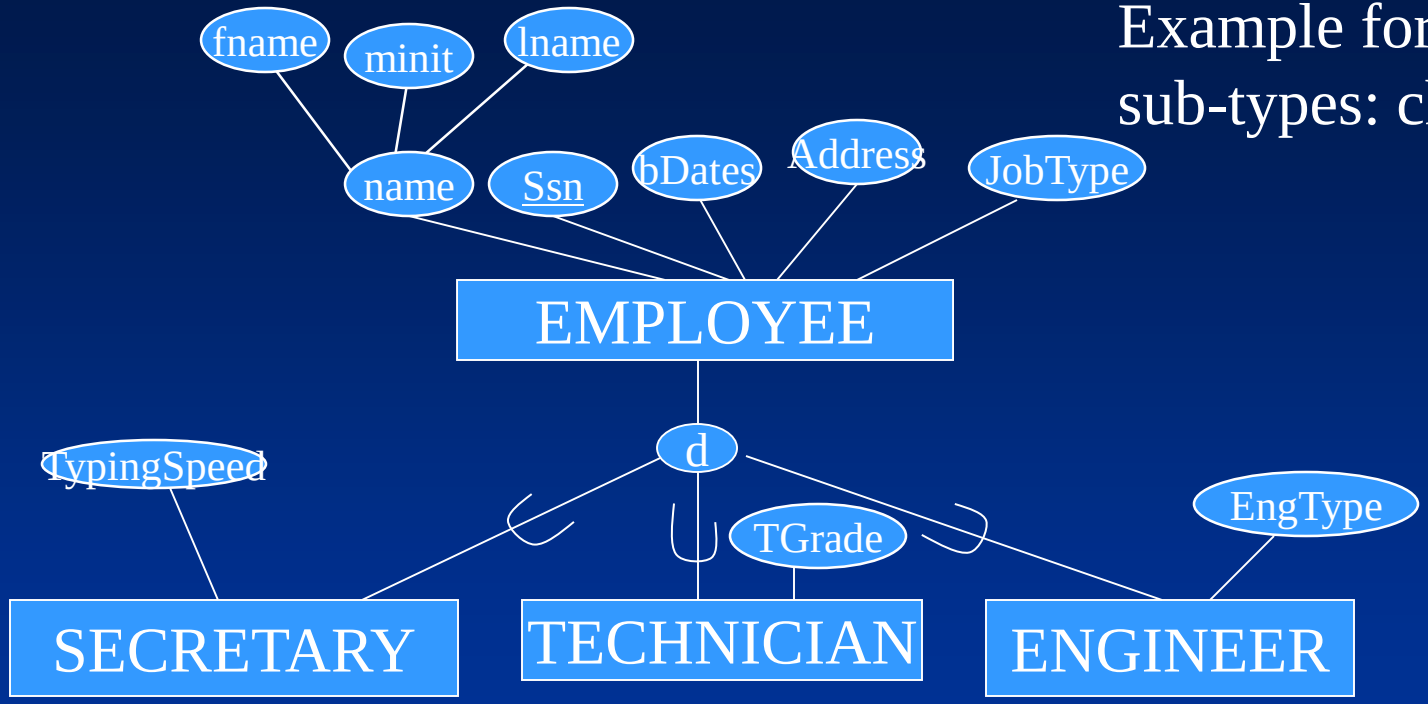
CAR

VehicleId, LicensePlate, Price, MaxSpeed, NoOfPassenger

TRUCK

VehicleId, LicensePlate, Price, NoOfAxles, Tonnage

Example for super- & sub-types: choice 3

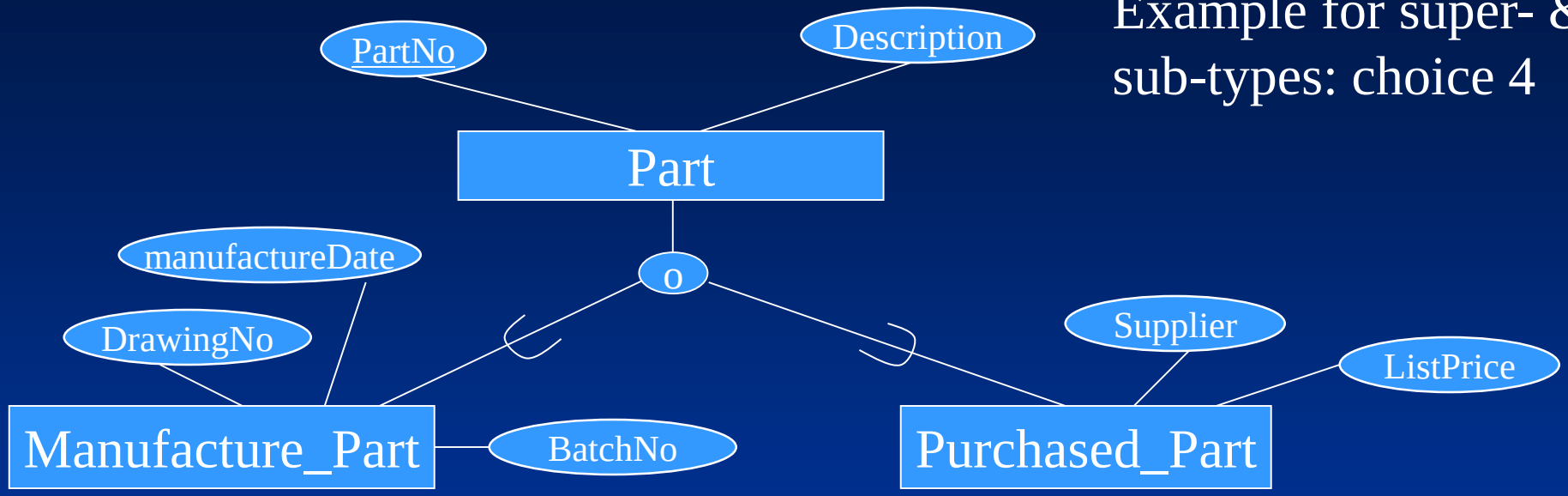


EMPLOYEE

fname, minit, lname, ssn, bdate, address, JobType, TypingSpeed, Tgrade, EngType

12345	…	…	1	…	…		
56463	…	…	2	…	…		
55554	…		3	…	…		

Example for super- & sub-types: choice 4

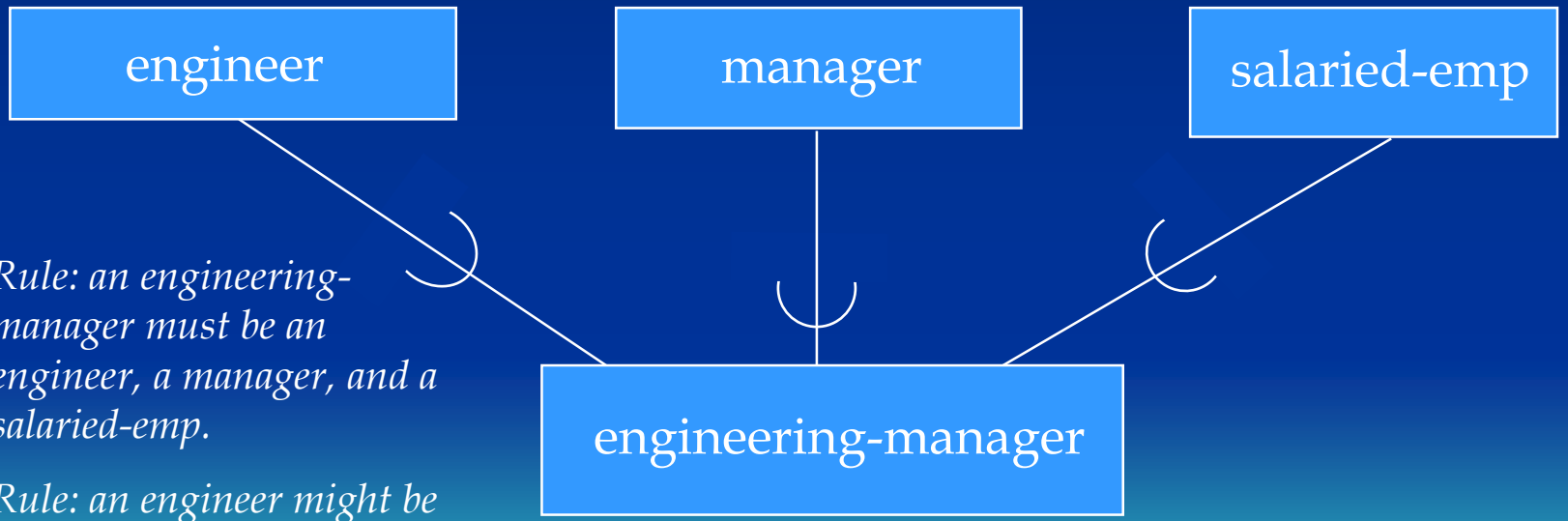


Part

PartNo, Desription, MFlag, Drawing, ManufactureDate, BatchNo, Pflag, Supplier, ListPrice

1	screw	1	…	…		…	…
2	Bolt				1	…	…
3	Axes	1			1		

- Shared SubClass
  - a subclass with more than one superclass
  - leads to the concept of multiple inheritance: engineering manager inherits attributes of engineer, manager, and salaried employee

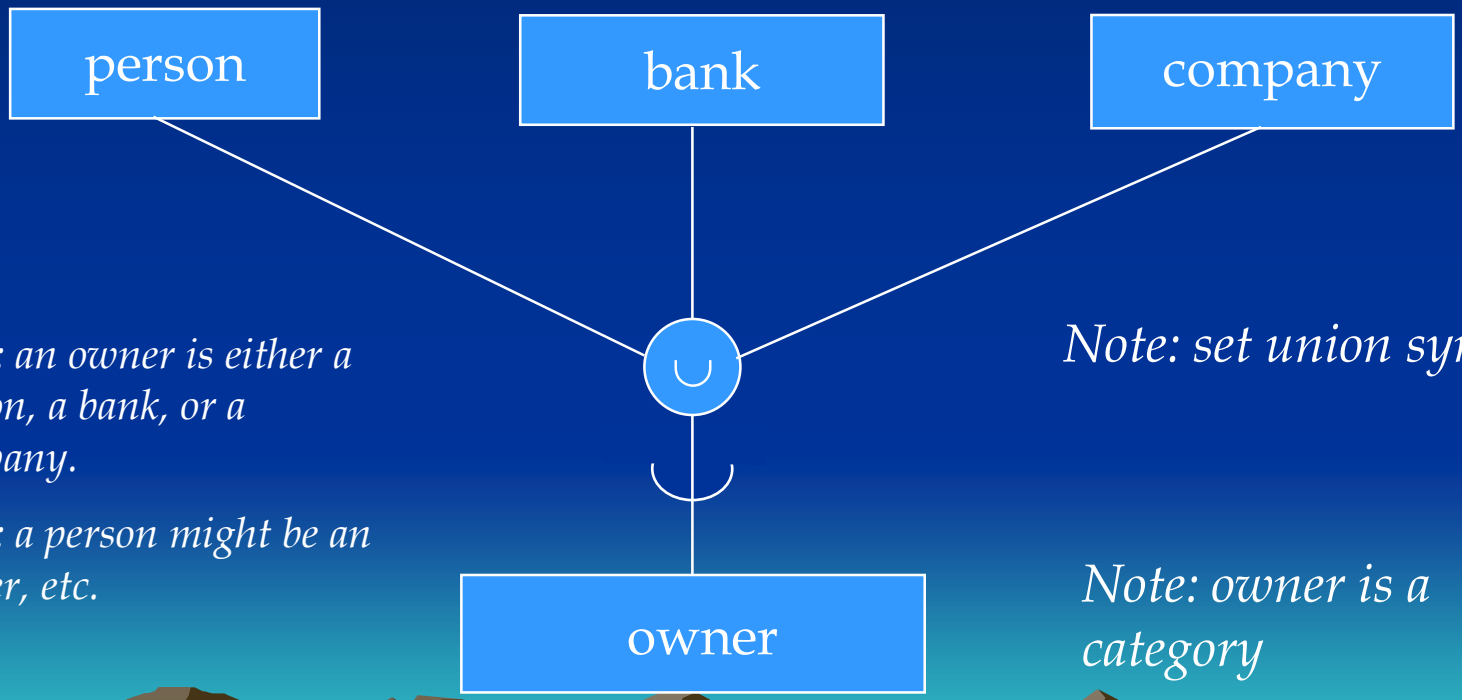


Rule: an engineering-manager must be an engineer, a manager, and a salaried-emp.

Rule: an engineer might be an engineering manager, etc.

Categories

- Models a single class/subclass with more than one super class of different entity types



Rule: an owner is either a person, a bank, or a company.

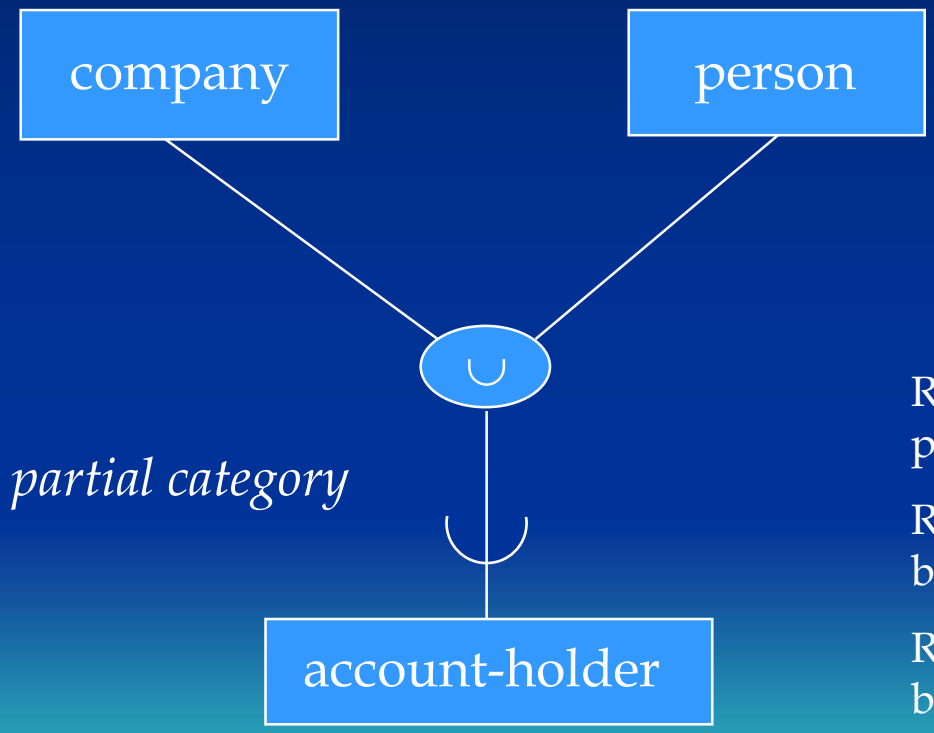
Rule: a person might be an owner, etc.

Note: set union symbol

Note: owner is a category

Categories

- A category can be either total or partial



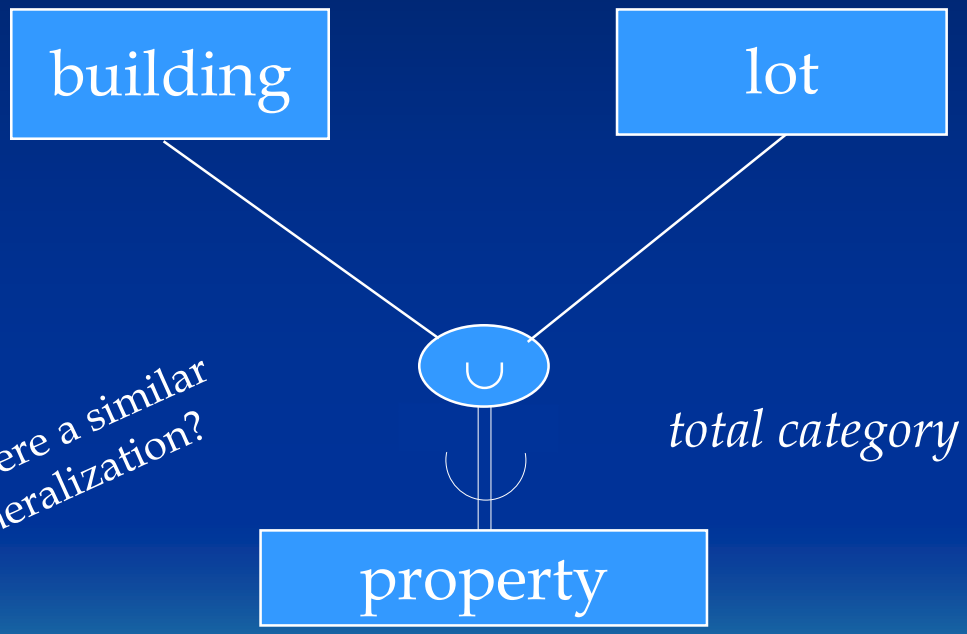
Rule: an account holder is either a person or a company.

Rule: a person may, or may not, be an account owner

Rule: a company may, or may not, be an account holder

Categories

- A category can be either total or partial



Rule: a property is either a building or a lot

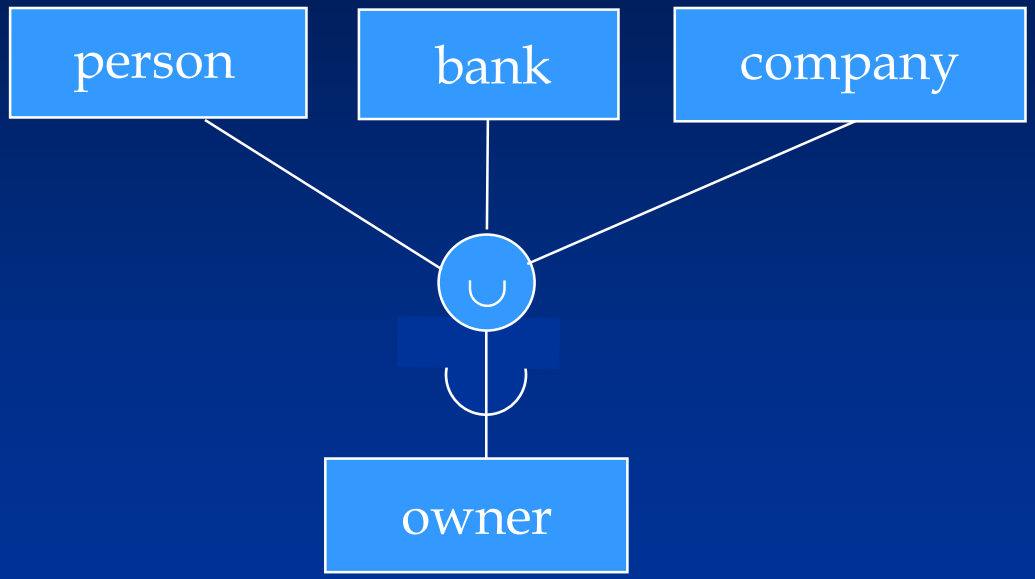
Rule: a building is a property

Rule: a lot is a property

- Mapping of Categories
  - Generate a table for each entity type involved
  - Superclasses with different key
  - Specify a new key called surrogate key for the category, which will also be included in the tables for the superclasses as foreign keys
  - Superclasses with the same keys

    No need of a surrogate key

- Categories - Superclasses with different keys

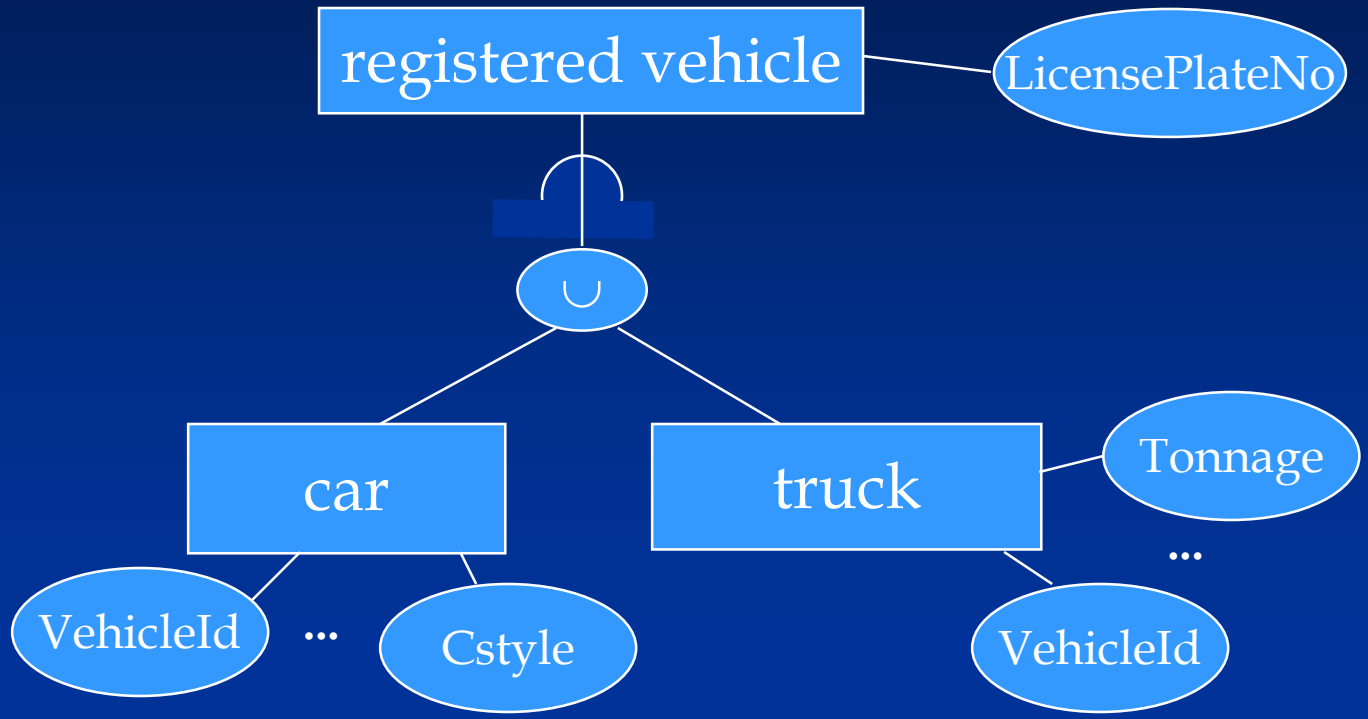


Person (SSN, DrLicNo, Name, Address, *Ownerid*)
Bank (Bname, BAddress, *Ownerid*)
Company (CName, CAddress, *Ownerid*)
Owner (*Ownerid*)

Note the Foreign Keys

Surrogate key

- Categories - Superclasses with the same keys



Registered Vehicle (*VehicleID*, LicensePlateNo,)

Car (*VehicleID* , Cstyle, CMake, CModel,CYear)

Truck (*VehicleID* , TMake, TModel,TYear, Tonnage)

Note there are no Foreign Keys

Outline: SQL and JDBC

- DDL
  - creating schemas
  - modifying schemas
- DML
  - select-from-where clause
  - group by, having, order by
  - update
  - view
- JDBC – Java Database Connectivity

DDL - creating schemas

- **Create schema** *schemaname* **authorization** *user*;
- **Create table** *tablename* …
  - attributes, data types
  - constraints:
    - primary keys
    - foreign keys
      - on delete set null|cascade|set default
      - on update set null|cascade|set default
      - on insert set null|cascade|set default
    - uniqueness for secondary keys
- **Create domain** *domainname* …

DDL - Examples:

- **Create schema:**

  Create schema COMPANY **authorization** JSMITH;

- **Create table:**

```
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));
```

DDL - Examples:

•**Specifying constraints:**

```
Create table EMPLOYEE
(…,
DNO       INT                    NOT NULL          DEFAULT 1,
CONSTRAINT EMPPK
     PRIMARY KEY(SSN),
CONSTRAINT EMPSUPERFK
FOREIGN KEY(SUPERSSN) REFERENCES EMPLOYEE(SSN)
     ON DELETE SET NULL  ON UPDATE CASCADE,
CONSTRAINT EMPDEPTFK
FOREIGN KEY(DNO) REFERENCES DEPARTMENT(DNUMBER)
     ON DELETE SET DEFAULT  ON UPDATE CASCADE);
```

•**Create domain:**

```
CREATE DOMAIN SSN_TYPE AS CHAR(9);
```

set null or cascade: strategies to maintain data consistency

Employee

ssn	… …	supervisor
123456789		234589710
… …		
234589710		null

delete

Employee

ssn	… …	supervisor
123456789		234589710
… …		
234589710		null

not reasonable

delete

cascade

delete

set null or cascade: strategies to maintain data consistency

Employee

ssn	… …	supervisor
123456789		234589710
… …		
234589710		null

delete

Employee

ssn	… …	supervisor
123456789		null
… …		
234589710		null

reasonable

set null

delete

set default: strategy to maintain data consistency

Department

DNUMBER	… …	… …
1		… …
… …		
4		… …

delete

Employee

ssn	… …	DNO
123456789		4
… …		
234589710		… …

change this value to the default value 1.

Cascade – a strategy to enforce referential integrity

Employee

ssn	…	
123456789		
…		

delete

Works_On

ssn	pno	hours
123456789	…	20
…		

Works_On

ssn	pno	hours
…		

Delete cascading.

DML - Queries (the Select statement)

select *attribute list*

from *table list*

where *condition*

group by *expression*

having *expression*

order by *expression ;*

Select *fname, salary* **from** *employee* ***where*** *salary > 30000* ⇨

$\pi_{\text{fname, salary}}(\sigma_{\text{salary}>30000}(\text{Employee}))$

Select *salary* **from** *employee*;

Salary
30000
40000
25000
43000
38000
25000
25000
55000

See Fig. 7.6 for the relation employee.

Duplicates are possible!

Select *fname*, *salary* **from** *employee* ***where*** *salary > 30000*;

Fname	***Salary***
Franklin	40000
Jennifer	43000
Ramesh	38000
James	55000

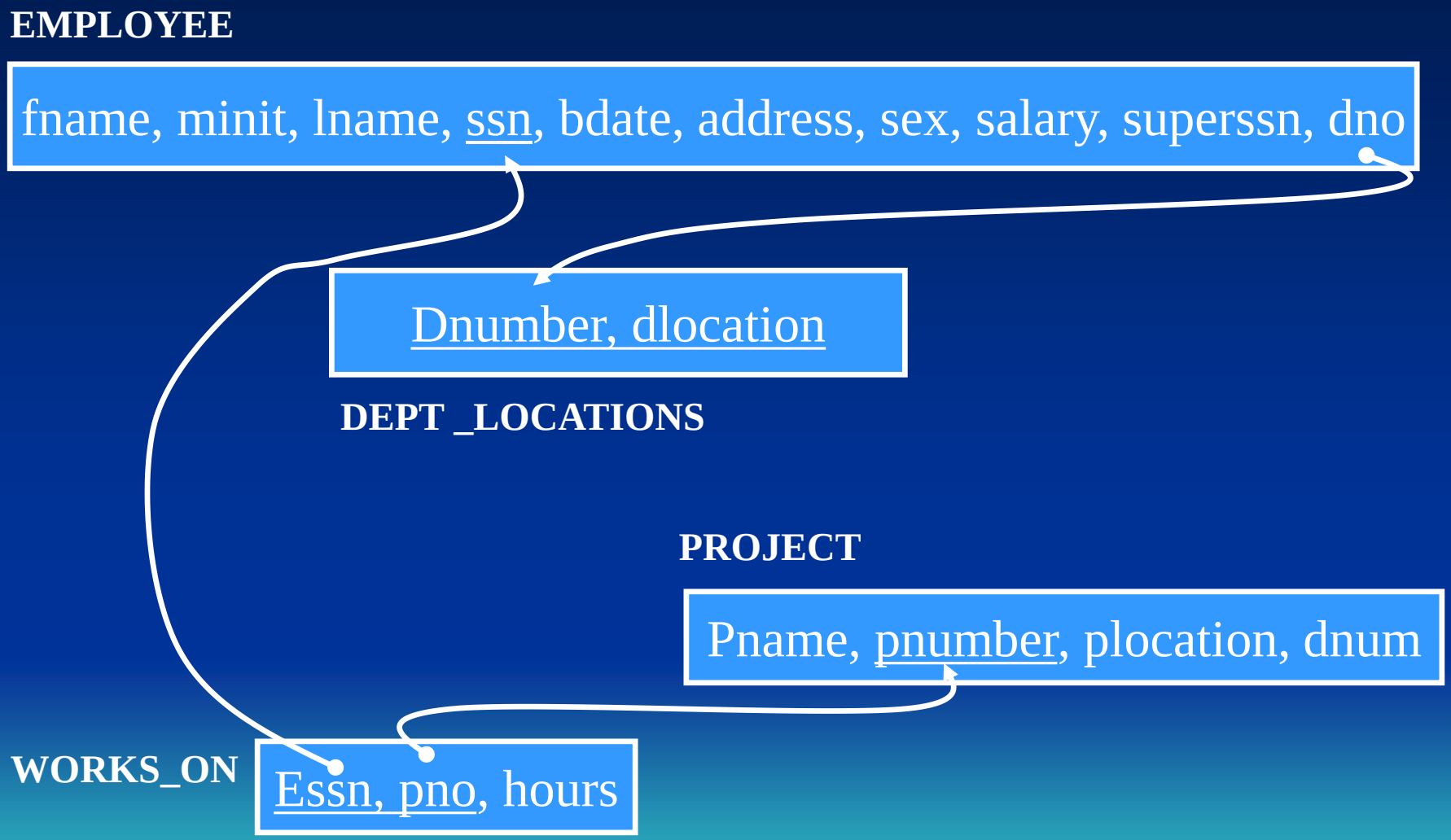
Correlated Subquery example:

Suppose we want to find out who is working on a project that is not located where their department is located.

- Note that the Project table has the location for the project
- Note that the Works_on relates employees to projects
- Note that the Employee table has the department number for an employee, and that Dept_locations has the locations for the department

We'll do this in two parts:

- a join that relates employees and projects (via works_on)
- a subquery that obtains the department locations for a given employee


EMPLOYEE
fname, minit, lname, ssn, bdate, address, sex, salary, superssn, dno
Dnumber, dlocation
DEPT _LOCATIONS
PROJECT
Pname, pnumber, plocation, dnum
WORKS_ON
Essn, pno, hours

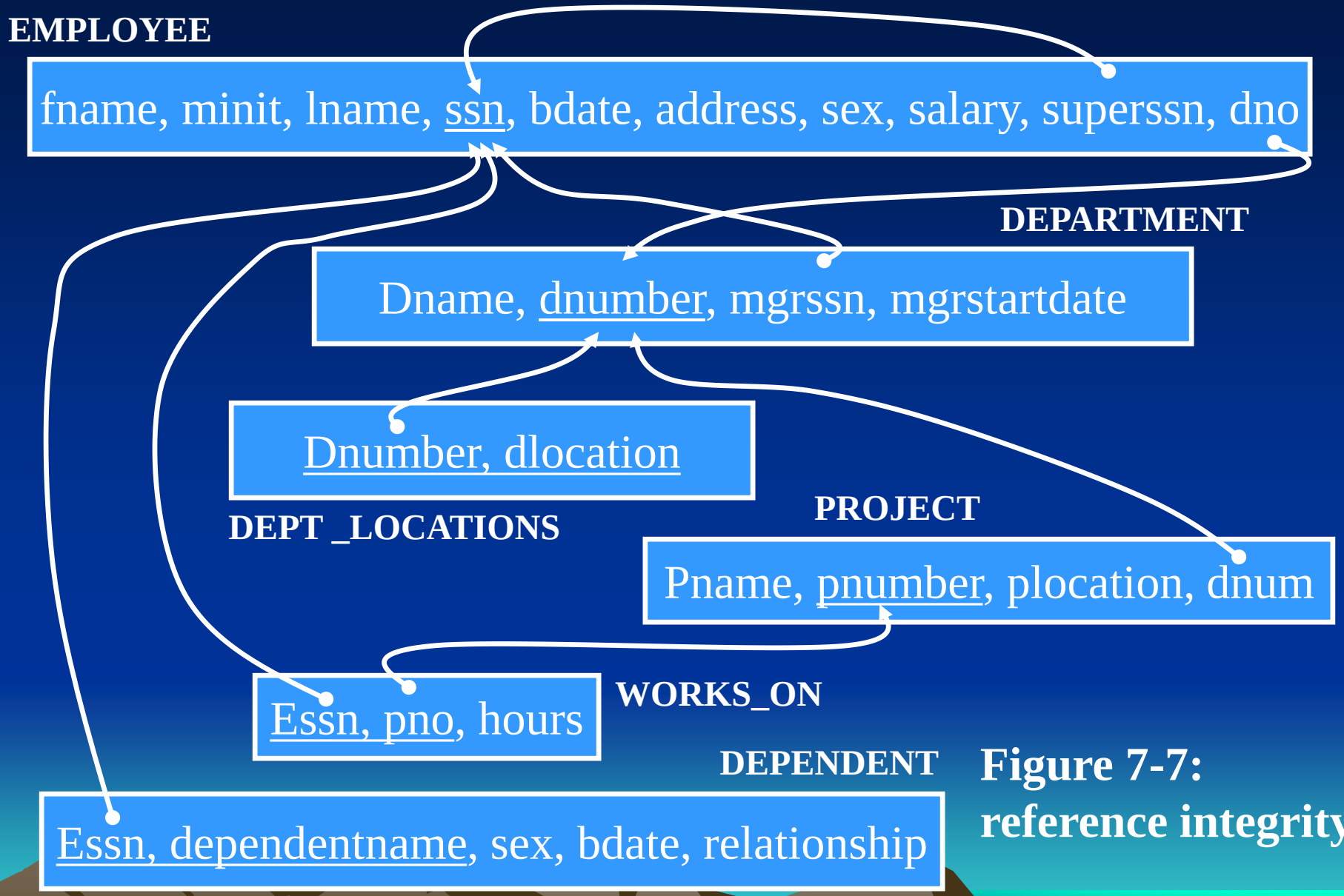


Figure 7-7: reference integrity

Correlated Subqueries:

A 3-way join to bring related employee and project data together:

```
SELECT employee.ssn, employee.fname, employee.lname,
        project.pnumber, project.plocation
FROM employee, project, works_on
WHERE
employee.ssn = works_on.essn and
project.pnumber = works_on.pno
```

A 3-way join

We'll see this join again where Inner Joins are discussed

Correlated Subqueries:

Now we incorporate a correlated subquery to restrict the result to those employees working on a project that is not where their department is located:

```
SELECT employee.ssn, employee.fname, employee.lname,
        project.pnumber, project.plocation
FROM employee, project, works_on
WHERE
employee.ssn = works_on.essn and
project.pnumber = works_on.pno and
plocation NOT IN
        (SELECT dlocation FROM dept_locations WHERE
        dnumber=employee.dno);
```

Correlated Subqueries:

Now we incorporate a correlated subquery to restrict the result to those employees working on a project that is not where their department is located:

```
SELECT employee.ssn, employee.fname, employee.lname,
        project.pnumber, project.plocation
FROM employee x, project, works_on
WHERE
employee.ssn = works_on.essn and
project.pnumber = works_on.pno and
plocation NOT IN
        (SELECT dlocation FROM dept_locations y WHERE
        y.dnumber = x.dno);
```

Subqueries with Exists and Not Exists:

Who is working on every project?

This is not a simple query!

```
SELECT e.ssn, e.fname, e.lname
FROM employee AS e
WHERE
NOT EXISTS
    (SELECT * FROM project AS p WHERE
    NOT EXISTS
        (SELECT * FROM works_on AS w WHERE w.essn=e.ssn
        AND w.pno=p.pno));
```

There is no project that the employee does not work on.

Example:

EMPLOYEE

ssn	fname	lname
1	… …	… …
2	… …	… …
3	… …	… …

WORK_ON

essn	PNo	hours
1	1	...
1	2	...
2	3	...
3	1	...
3	2	...
3	3	...

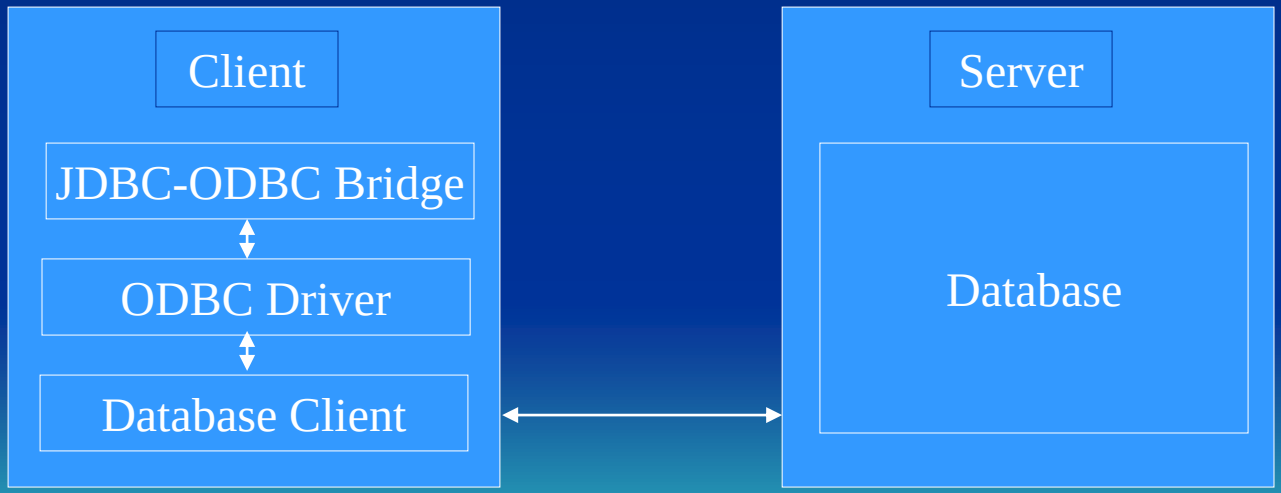
PROJECT

PNo	Pname	…
1	… …	…
2	… …	…
3	… …	…

To develop a database application, **JDBC** or **ODBC** should be used.

JDBC – JAVA Database Connectivity

ODBC – Open Database Connectivity



Connection to a database:

1. Loading driver class

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

2. Connection to a database

```
String url = "jdbc:odbc:<databaseName>";
Connction con =
DriverManager.getConnection(url, <userName>,
<password>)
```

3. Sending SQL statements

```
Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT *
FROM Information WHERE Balance >= 5000");
```

4. Getting results

```
while (rs.next())
    { …
    }
```

a table name

```
import java.sql.*;

public class DataSourceDemo1
{ public static void main(String[] args)
  { Connection con = null;
    try
    {//load driver class
     Class.forName{"sun.jdbs.odbs.JdbcOdbcDriver");

     //data source
     String url = "jdbs:odbc:Customers";

     //get connection
     con = DriverManager.getConnection(url,
     "sa", " ")
```

```
//create SQL statement
Statement stmt = con.createStatement();

//execute query
Result rs = stmt.executeQuery("SELECT *
FROM Information WHERE Balance >= 5000");

String firstName, lastName;
Date birthDate;
float balance;
int accountLevel;
```

```
        while(rs.next())
        {firstName = rs.getString("FirstName");
         lastName = rs.getString("lastName");
         balance = rs.getFloat("Balance");

         System.out.println(firstName + " " +
         lastName + ", balance = " + balance);
        }
      }
      catch(Exception e)
      {e.printStackTrace();}
      finally
      {try{con.close();}
       catch(Exception e){ }
      }
    }
  }
```

Programming in a dynamical environment:

Disadvantage of *DataSourceDemo1*:

If the JDBC-ODBC driver, database, user names, or password are changed, the program has to be modified.

Solution:

Configuration file:

config.driver=sun.jdbc.odbc.JdbcOdbcDriver
config.protocol=jdbc
config.subprotocol=odbc
config.dsname=Customers
config.username=sa
config.password=… …

file name: datasource.config

<property> = <property value>

config – datasource name

```
import java.sql.*;
import java.io.*;
import java.util.Properties;

public class DatabaseAccess
{ private String configDir;
  //directory for configuration file
  private String dsDriver = null;
  private String dsProtocol = null;
  private String dsSubprotocol = null;
  private String dsName = null;
  private String dsUsername = null;
  private String dsPassword = null;
```

```
public DatabaseAccess(String configDir)
{ this.configDir = configDir; }

public DatabaseAccess()
{ this(".."); }

//source: data source name
//configFile: source configuration file

public Connection getConnection(String source,
String configFile) throws SQLException, Exception
{ Connection con = null;

  try
  {Properties prop = loadConfig(ConfigDir, ConfigFile);
```

getConnection("config", "datasource.config");

```
  if (prop != null)
  {dsDriver = prop.getProperty(source + ".driver");
   dsProtocol = prop.getPropert(source + ".protocol");
   dsSubprotocol = prop.getPropert(source +
   ".subprotocol");
   if (dsName == null)
     dsName = prop.getProperty(source +
     ".dsName");
   if (dsUsername == null)
dsUsername = prop.getProperty(source +
".username");
   if (dsPassword == null)
     dsPassword = prop.getProperty(source +
".password");
```

```
    //load driver class
    Class.forName(dsDriver);

    //connect to data source
    String url = dsProtocol + ":" + dsSubprotocol + ":"
    + dsName;
    con = DriverManager.getConnection(url, dsUsername,
    dsPassword)
  }
  else
    throw new Exception("* Cannot find property file +
    configFile);

  return con;
}
catch (ClassNotFoundException e)
{ throw new Exception("* Cannot find driver class " +
  dsDriver + "!"); }
}
```

```
//dir: directory of configuration file
//filename: file name
public Properties loadConfig(String dir, String filename)
throws Exception
{ File inFile = null;
  Properties prop = null;

  try
  { inFile = new File(dir, filename);
    if (inFile.exists()
    {  prop = new Properties();
       prop.load(new FileInputStream(inFile));
    }
    else throw new Exception("* Error in finding " +
              inFile.toString());
  }
  finally {return prop;}
  }
}
```

Using class DatabaseAccess, DataSourceDemo1 should be modified a little bit:

```
DatabaseAccess db = new databaseAccess();

con = db.getConnection("config",
"datasource.config");
```

Database updating:

```
import java.sql.*;

public class UpdateDemo1
{ public static void main(String[] args)
  { Connection con = null;
    try
    {
     //get connection
     Databaseaccess db = new DatabaseAccess();
     con = db.getConnection("config",
     "datasource.config");
```

```
      //execute update
      Statement stmt = con.CreateStatement();
      int account = stmt.executeUpdate("UPDATE
      Information SET Accountlevl = 2 WHERE
      Balance >= 50000");
      System.out.println(account + " record has been
   updated");

      //execute insert
      account = stmt.executeUpdate("INSERT INTO
      Information VALUE ('David', 'Feng', '05/05/1975',
      2000, 1)");
      System.out.println(account + " record has been
      inserted");
    }
    catch (Exception e) {e.printStackTrace(); }
    finally {try{con.close(); catch(Exception e){ }}
  }
}
```

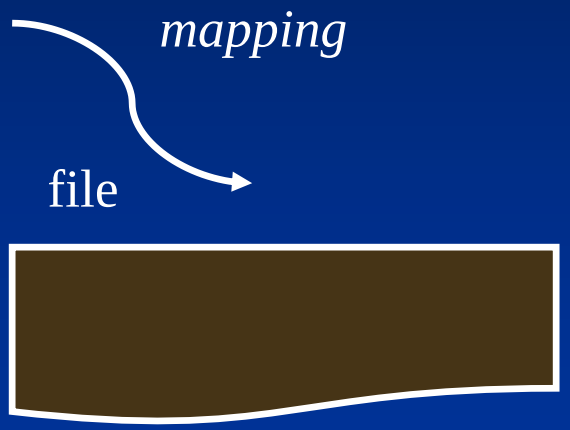
Outline: Hashing (5.9, 5.10, 3rd. ed.; 13.8, 4th, 5th ed.; 17.8, 6th ed.)

- external hashing
- static hashing & dynamic hashing
- hash function
  - mathematical function that maps a key to a bucket address
  - collisions
    - collision resolution scheme
      - open addressing
      - chaining
      - multiple hashing
- linear hashing

Mapping a table into a file

Employee

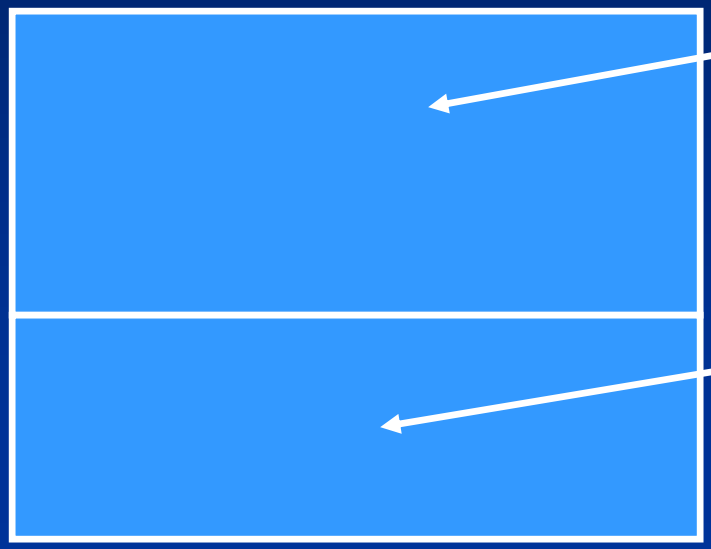
ssn	name	bdate	sex	address	salary
	… …				



- Block (or page)
  - access unit of operating system
  - block size: range from 512 to 4096 bytes
- Bucket
  - access unit of database system
  - A bucket contains one or more blocks.
- A file can be considered as a collection of buckets. Each bucket has an address.

External Hashing

- Consider a file comprising a primary area and an overflow area



Records hash to one of many primary buckets

Records not fitting into the primary area are relegated to overflow

- Common implementations are *static* - the number of primary buckets is fixed - and we expect to need to reorganize this type of files on a regular basis.

External Hashing

- Consider a static hash file comprising M primary buckets
- We need a hash function that maps the key onto {0, 1, … M-1}
- If M is prime and Key is numeric then

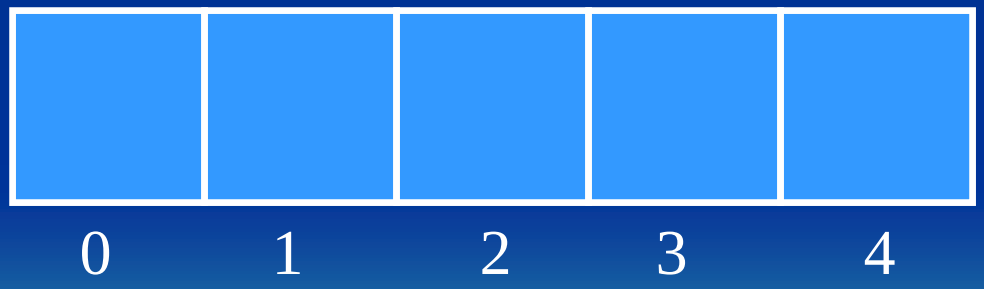
$$Hash(Key) = Key \bmod M$$

  can work well

- A collision may occur when more than one records hash to the same address
- We need a collision resolution scheme for overflow handling because the number of collisions for one primary bucket can exceed the bucket capacity
  - open addressing
  - chaining

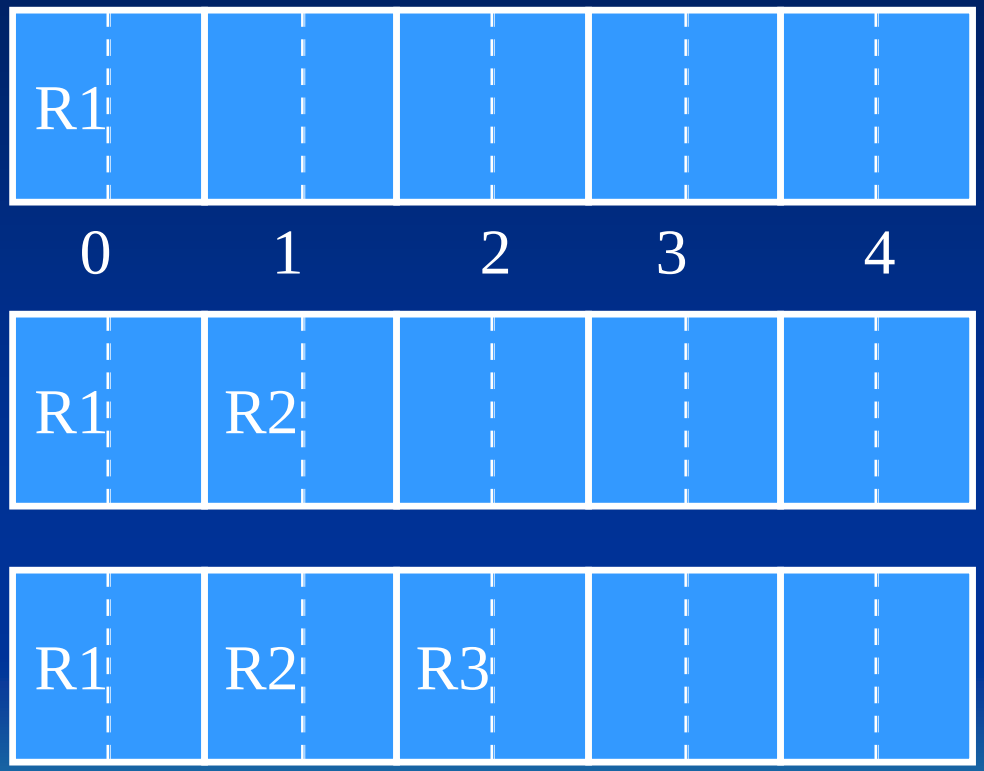
Overflow handling

- Open addressing
  - subsequent buckets are examined until an open record position is found
  - no need for an overflow area
  - consider records being inserted R1, R2, R3, R4, R5, R6, R7 with bucket capacity of 2 and hash values 0, 1, 2, 1, 1, 0, 3

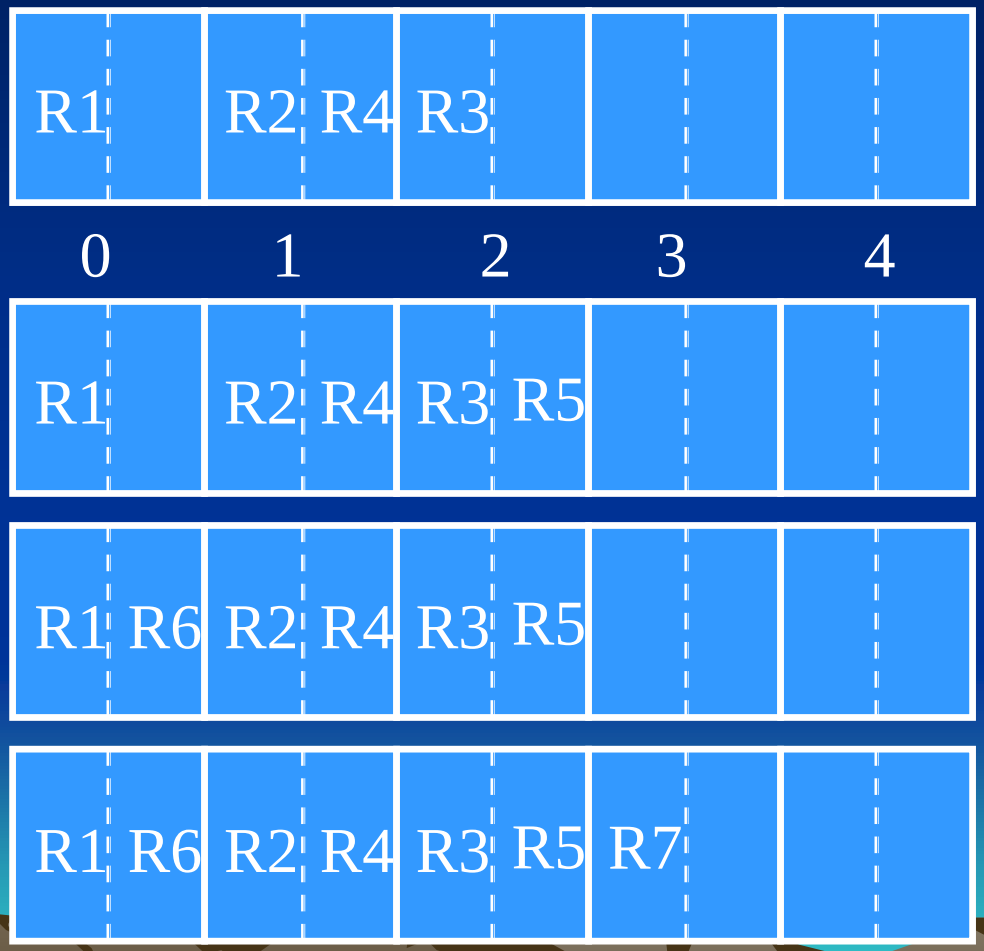


How do we handle retrieval, deletion?

- consider records being inserted R1, R2, R3, R4, R5, R6, R7 with bucket capacity of 2 and hash values 0, 1, 2, 1, 1, 0, 3

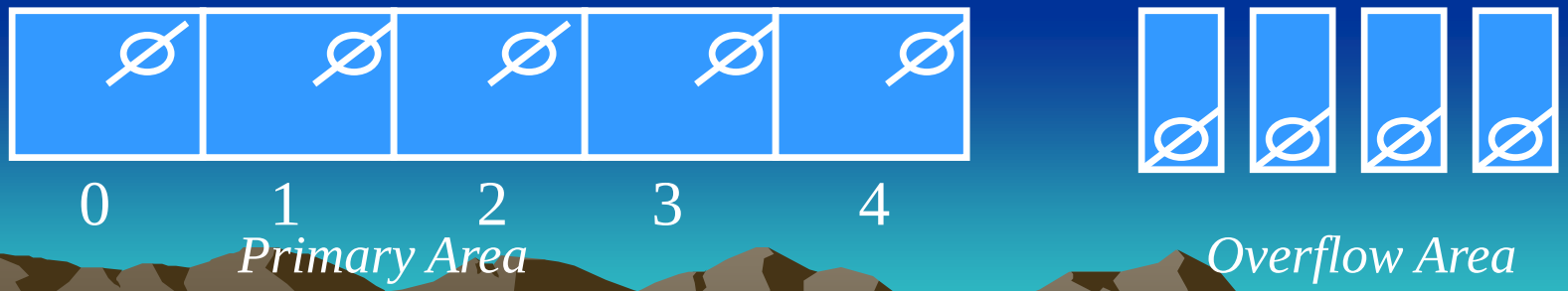


R1, R2, R3, R4, R5, R6, R7
hash values: 0, 1, 2, 1, 1, 0, 3



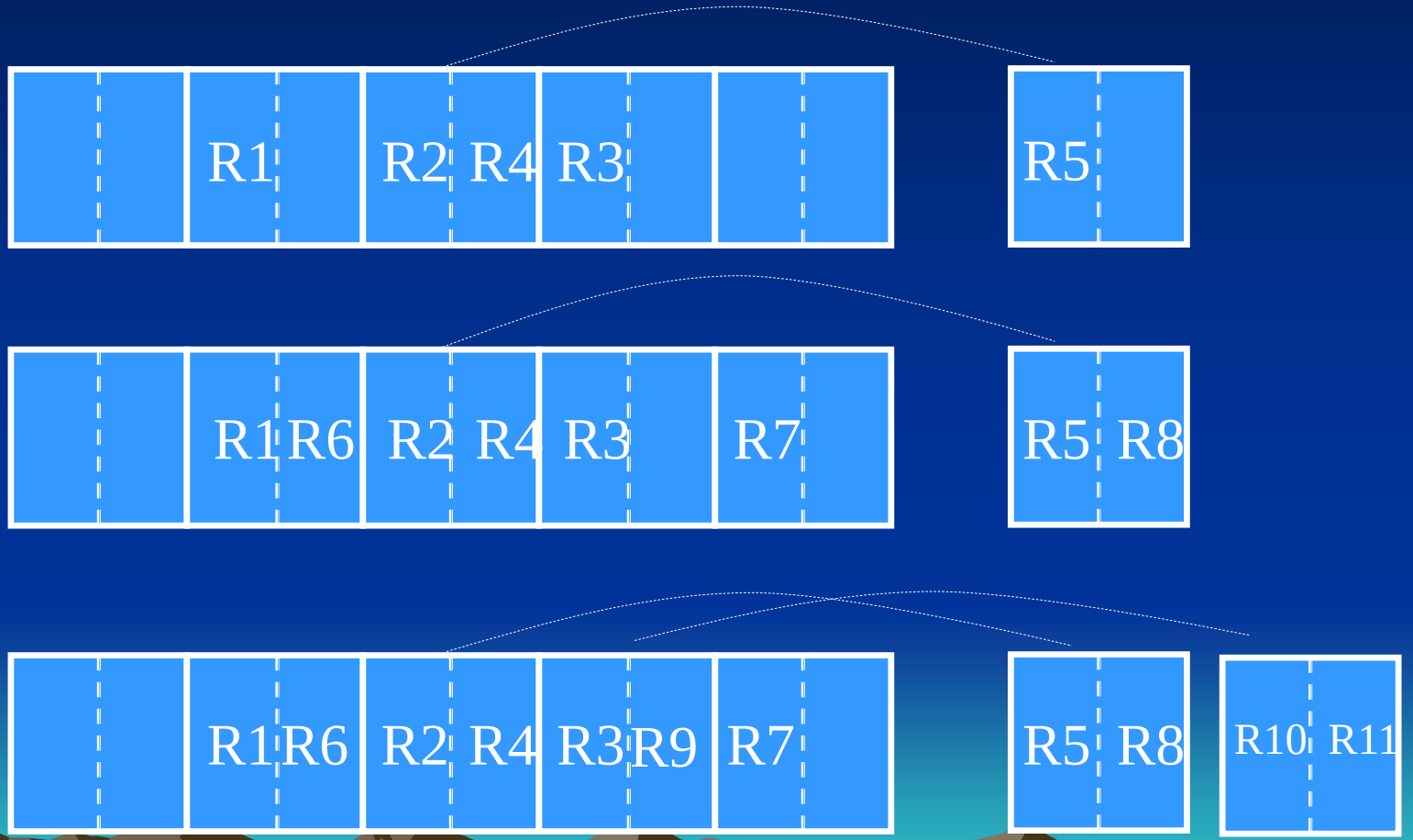
Overflow handling

- Chaining
  - a pointer in the primary bucket points to the first overflow record
  - overflow records for one primary bucket are chained together
  - consider records being inserted R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11.
  - with bucket capacity of 2 and hash values 1, 2, 3, 2, 2, 1, 4, 2, 3, 3, 3.
  - deletions?



R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11

1, 2, 3, 2, 2, 1, 4, 2, 3, 3, 3



Overflow handling

- Multiple Hashing
  - when collision occurs a next hash function is tried to find an unfilled bucket
  - eventually we would resort to chaining
  - note that open addressing can suffer from poor performance due to islands of full buckets occurring and having a tendency to get even longer - using a second hash function helps avoid that problem

Linear Hashing

- A dynamic hash file:

  grows and shrinks gracefully

- initially the hash file comprises M primary buckets numbered 0, 1, … M-1
- the hashing process is divided into several phases (phase 0, phase 1, phase 2, …). In phase j, records are hashed according to hash functions h_j(key) and h_{j+1}(key)
- h_j(key) = key mod (2^j*M)

phase 0: h_0(key) = key mod (2^0*M), h_1(key) = key mod (2^1*M)
phase 1: h_1(key) = key mod (2^1*M), h_2(key) = key mod (2^2*M)
phase 2: h_2(key) = key mod (2^2*M), h_3(key) = key mod (2^3*M)
… …

Linear Hashing

- h_j(key) is used first; to split, use h_{j+1}(key)
- splitting a bucket means to redistribute the records into two buckets: the original one and a new one. In phase j, to determine which ones go into the original while the others go into the new one, we use h_{j+1}(key) = key mod 2^{j+1}*M to calculate their address.
- splitting buckets

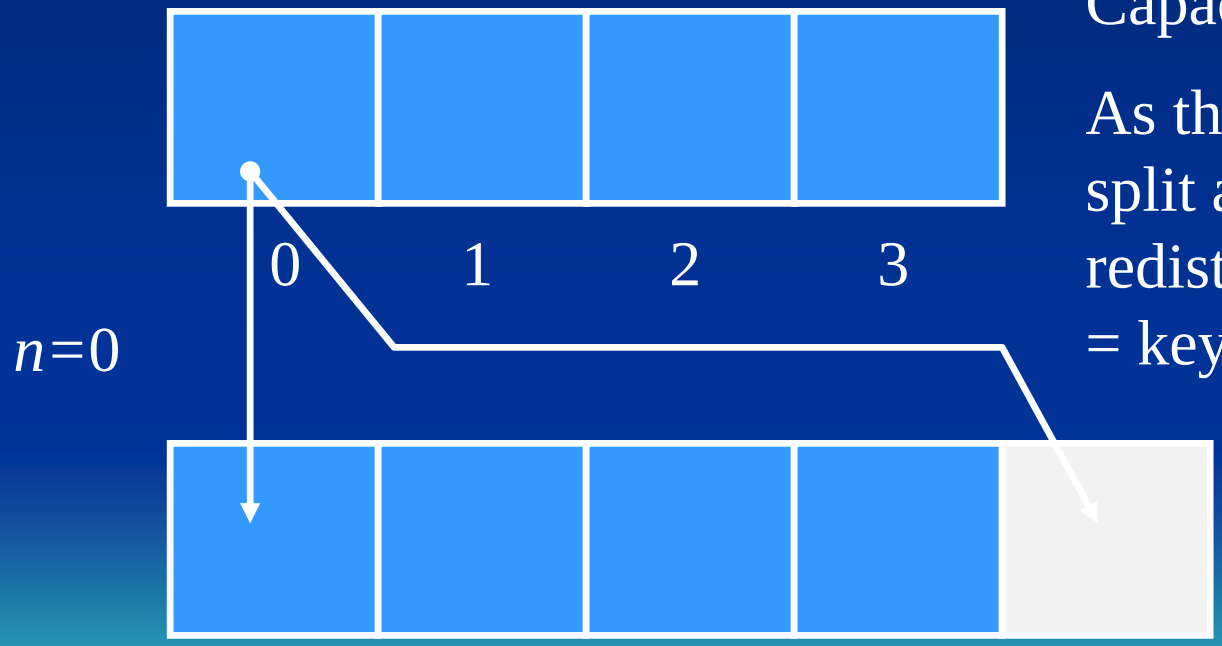
  splitting occurs according to a specific rule such as
  - an overflow occurring, or
  - the load factor reaching a certain value, etc.
- a split pointer keeps track of which bucket to split next
- split pointer goes from 0 to 2^j*M - 1 during the j^{th} phase, j= 0, 1, 2, … ...

Linear Hashing

1. What is a phase?
2. When to split a bucket?
3. How to split a bucket?
4. What bucket will be chosen to split next?
5. How do we find a record inserted into a linear hashing file?

Linear Hashing, example

- initially suppose M=4
- h_0(key) = key mod M; i.e. key mod 4 (rightmost 2 bits)
- h_1(key) = key mod 2*M



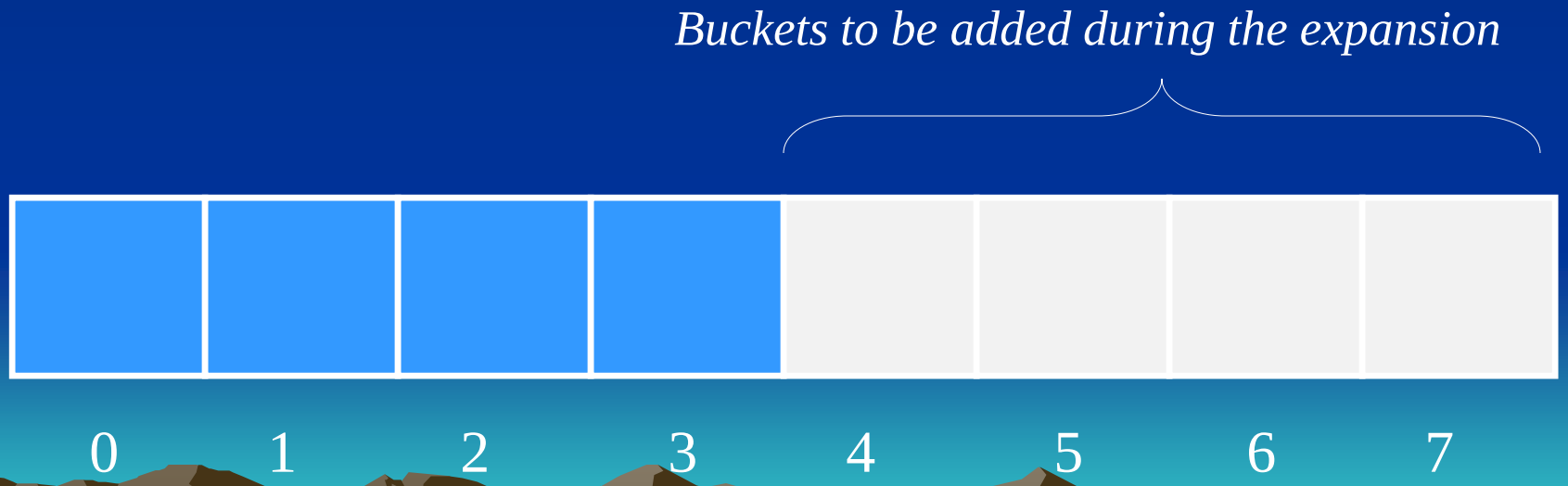
Capacity of a bucket is 2.

As the file grows, buckets split and records are redistributed using h_1(key) = key mode 2*M.

Linear Hashing, example

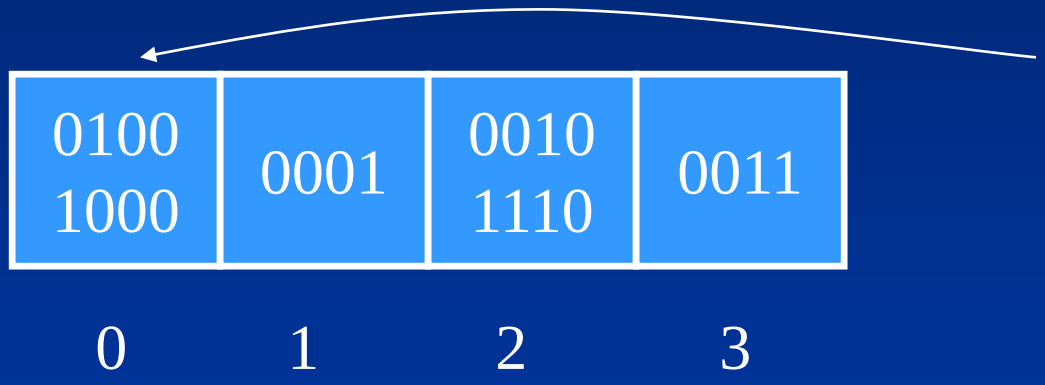
- collision resolution strategy: chaining
- split rule: if load factor > 0.70
- insert the records with key values:

  0011, 0010, 0100, 0001, 1000, 1110, 0101, 1010, 0111, 1100



Linear Hashing, example

- when inserting the sixth record (using h_0 = Key mod M) we would have



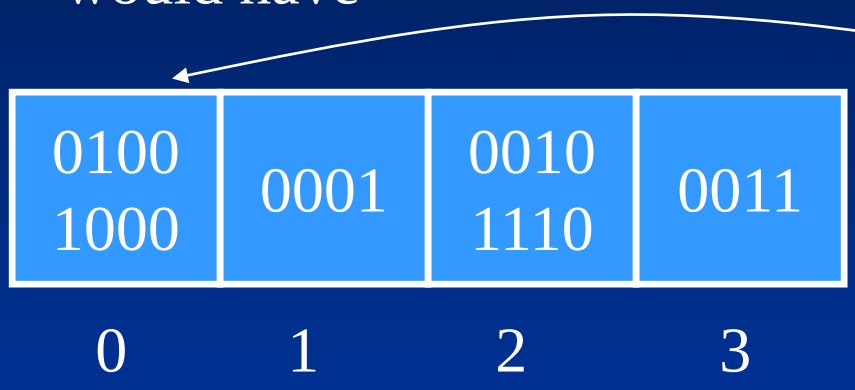
n=0 before the split

(n is the split point, i.e., the point to the bucket to be split.)

0011, 0010, 0100, 0001, 1000, 1110, 0101, 1010, 0111, 1100

Linear Hashing, example

- when inserting the sixth record (using h_0 = Key mod M) we would have



n=0 before the split

(n is the point to the bucket to be split.)

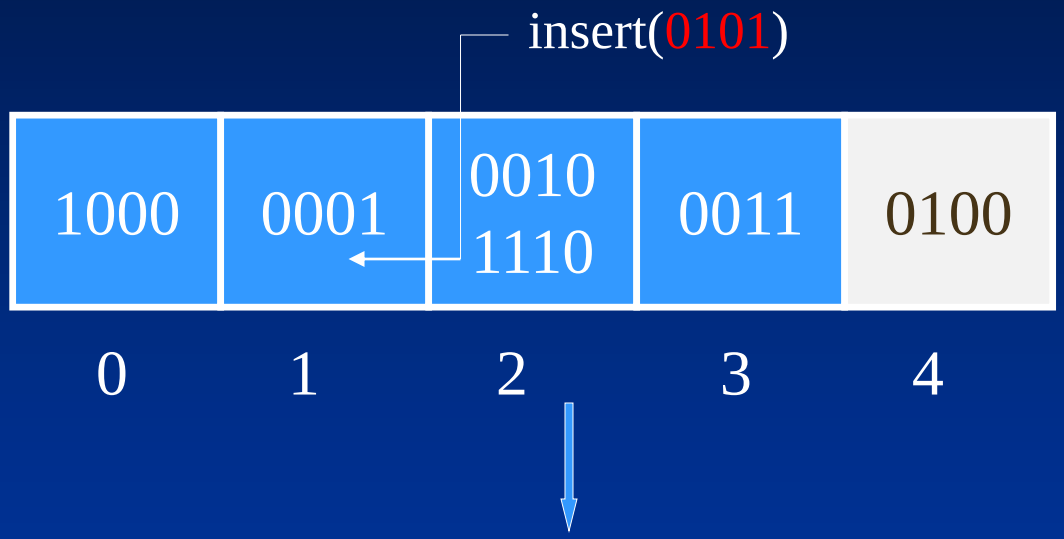
- but the load factor 6/8= 0.75 > 0.70 and so bucket 0 must be split (using h_1 = Key mod 2M):

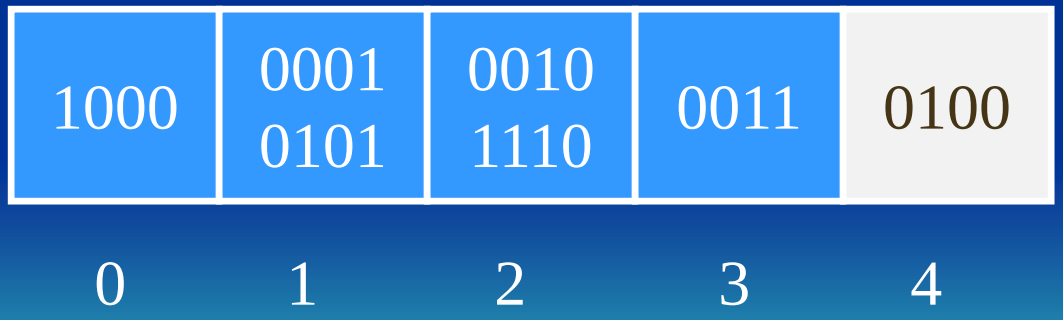
n=1 after the split

load factor: 6/10=0.6

no split

Linear Hashing, example



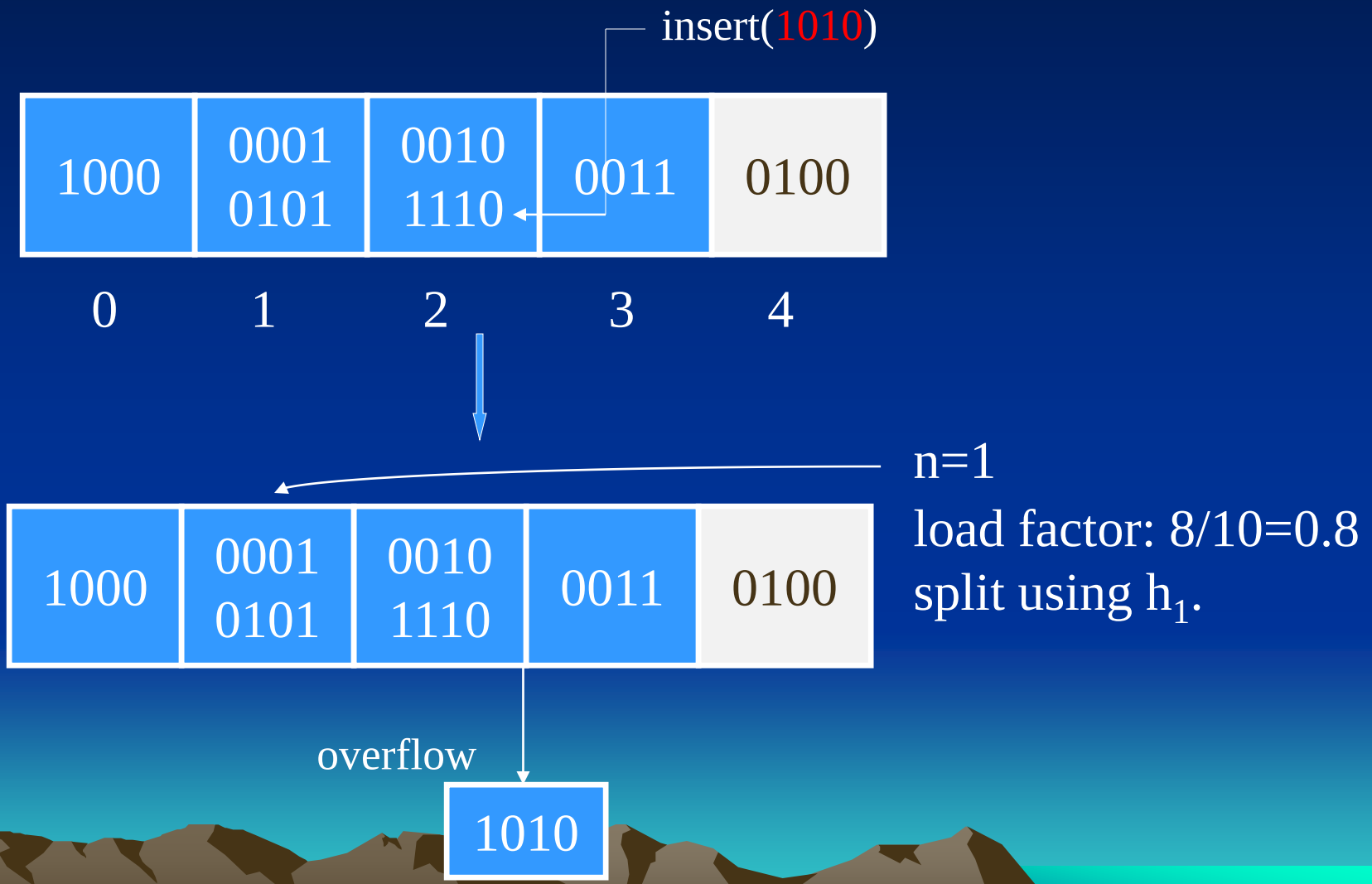


n=1

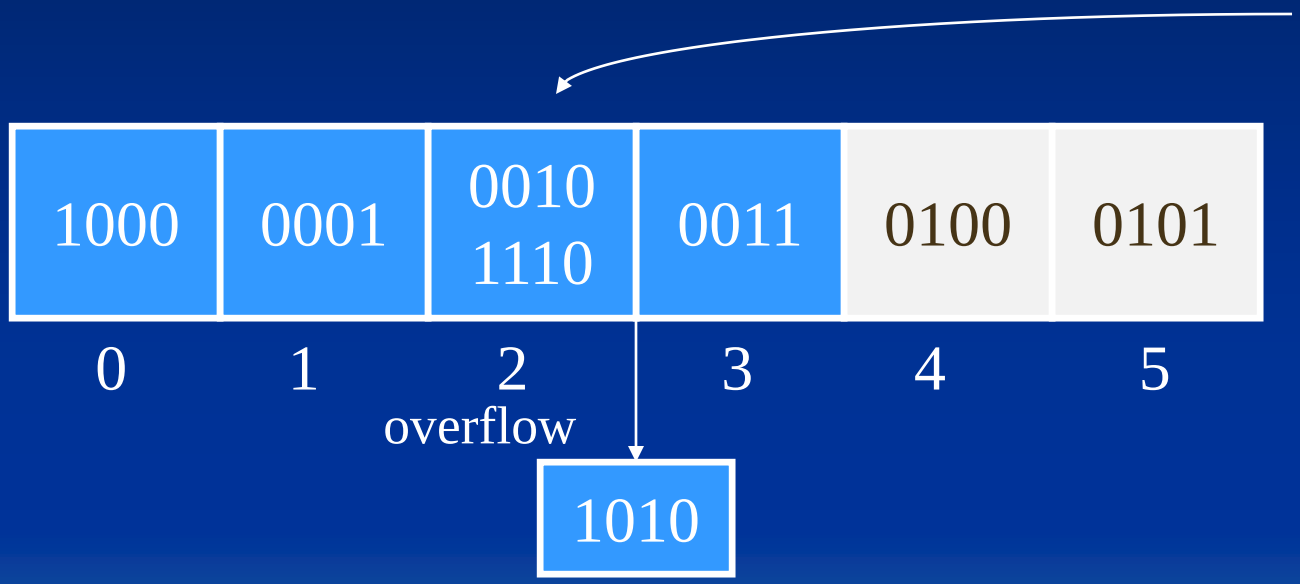
load factor: 7/10=0.7

no split

Linear Hashing, example



Linear Hashing, example



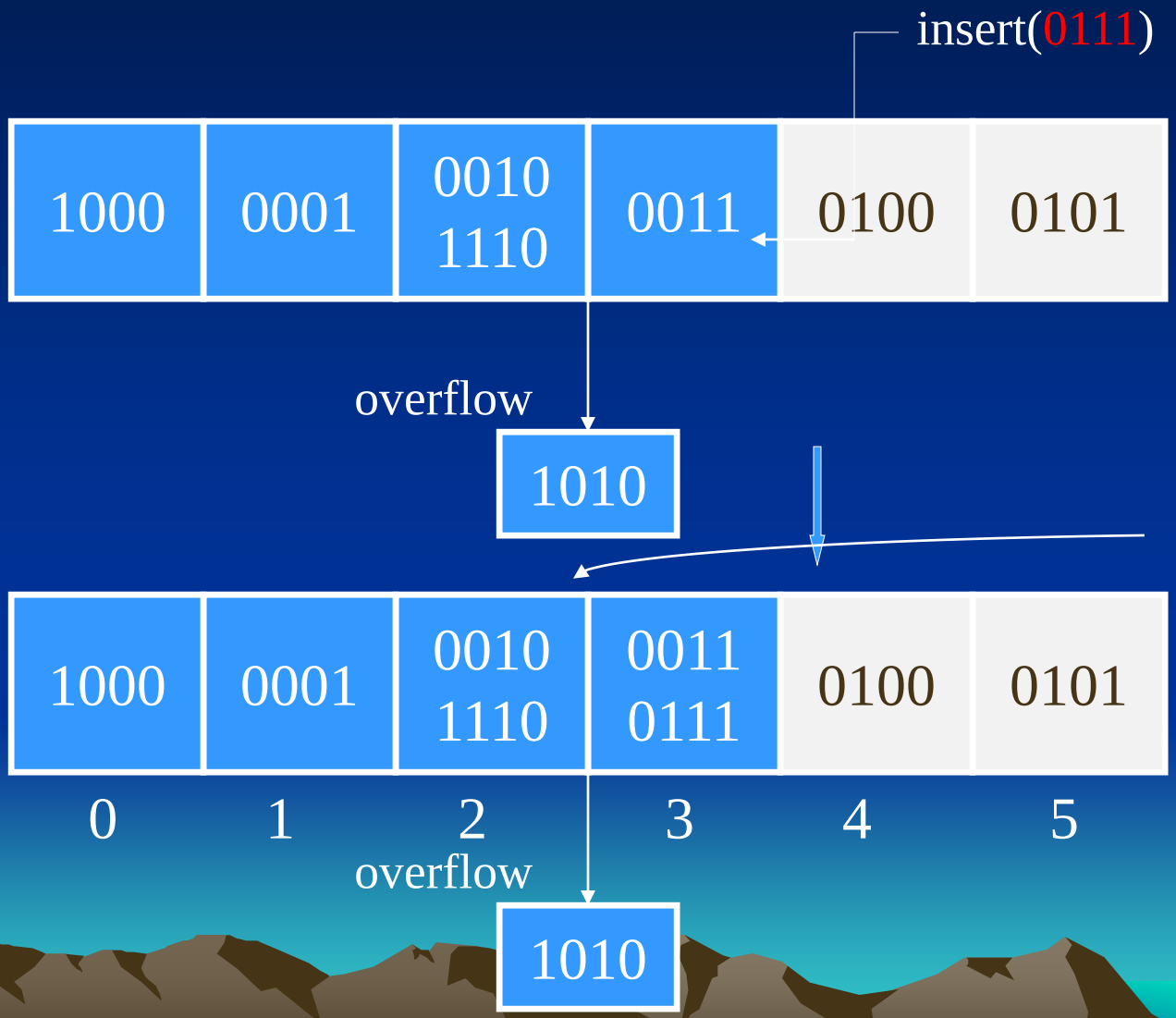
n=2

load factor:

8/12=0.66

no split

Linear Hashing, example



n=2

load factor:

9/12=0.75

split using h_1.

Linear Hashing, example

n=3

load factor: 9/14=0.642

no split.

insert(1100)

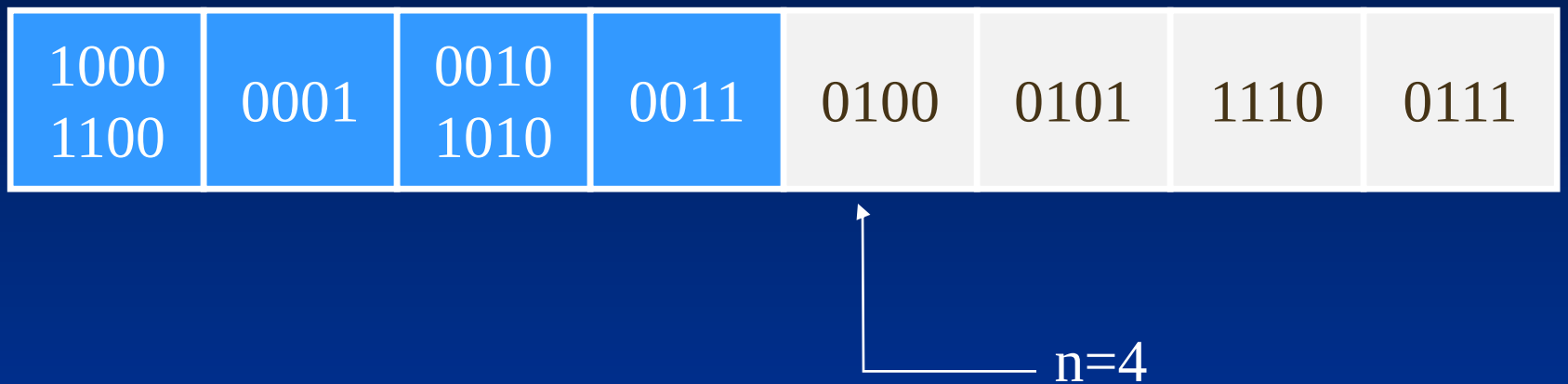
Linear Hashing, example

n=3

load factor: 10/14=0.71

split using h_1.

Linear Hashing, example



load factor: 10/16=0.625
no split.

- At this point, all the 4 (M) buckets are split. The size of the primary area becomes 2M. n should be set to 0. It begins a second phase.
- In the second phase, we will use h_1 to insert records and h_2 to split a bucket.
  - note that $h_1(K) = K \bmod 2M$ and $h_2(K) = K \bmod 4M$.

Database Index Techniques

- B^+ - tree
- Multiple-key indexes
- kd – tree
- Quad - tree
- R – tree
- Bitmap
- Inverted files

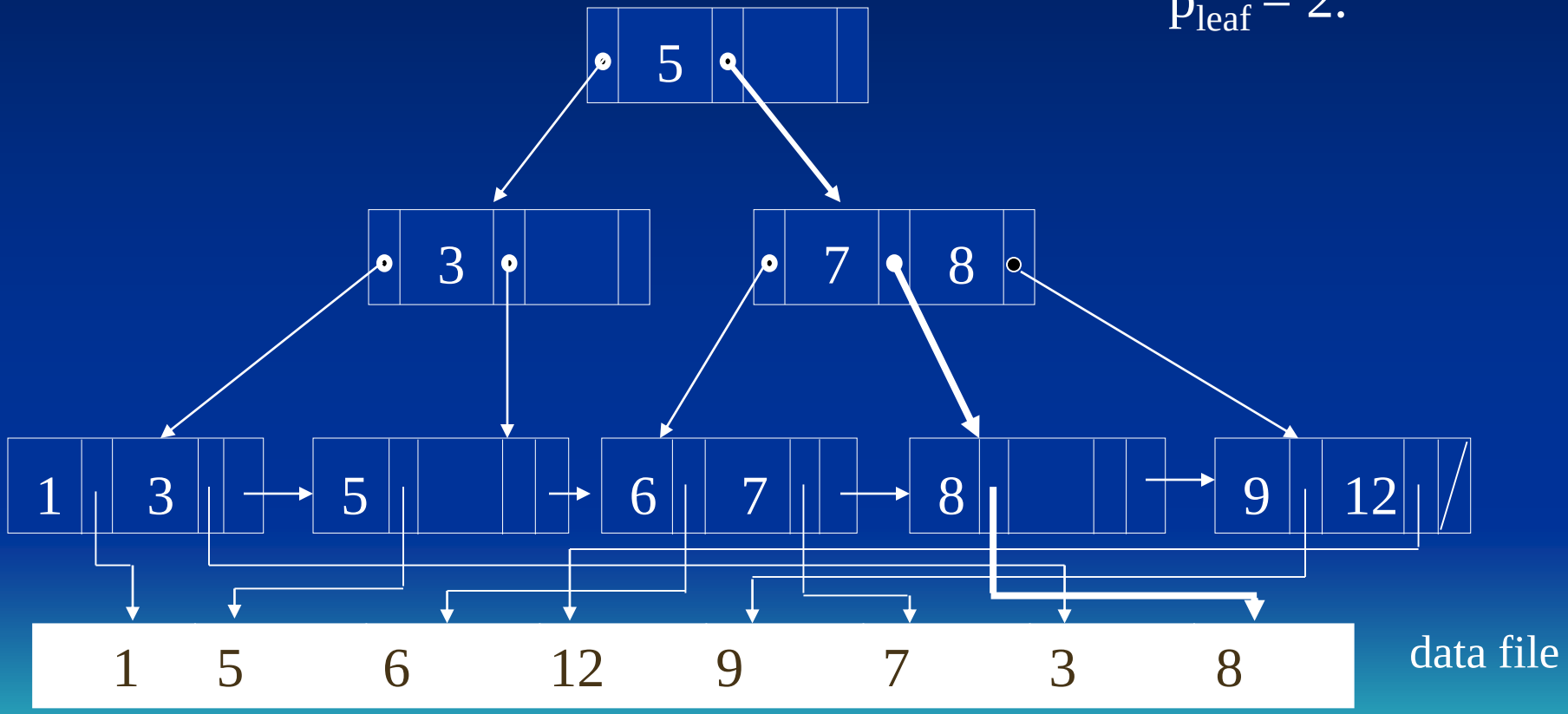
B^+-tree Structure

non-leaf node (internal node or a root)

- $< P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q >$ $(q \le p_{internal})$
- $K_1 < K_2 < ... < K_{q-1}$ (i.e. it's an ordered set)
- For any key value, X, in the subtree pointed to by P_i
  - $K_{i-1} < X \le K_i$ for $1 < i < q$
  - $X \le K_1$ for $i = 1$
  - $K_{q-1} < X$ for $i = q$
- Each internal node has at most $p_{internal}$ pointers.
- Each node except root must have at least $\lceil p_{internal}/2 \rceil$ pointers.
- The root, if it has some children, must have at least 2 pointers.

A B$^+$-tree

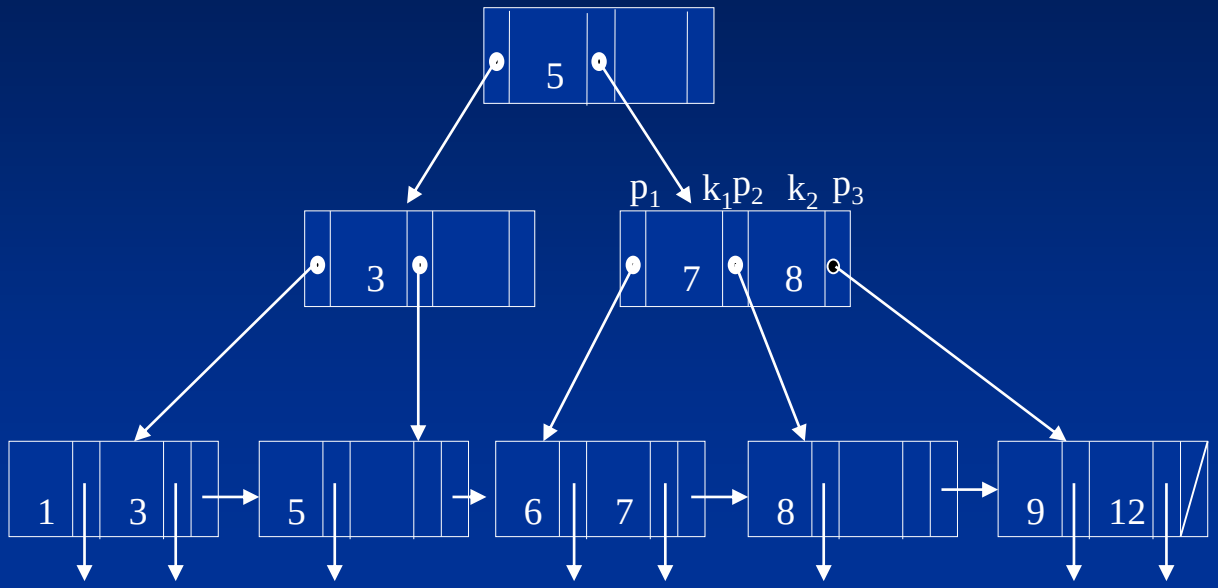
$p_{internal} = 3$,
$p_{leaf} = 2$.



B^+-tree Structure

leaf node (terminal node)

- $< (K_1, Pr_1), (K_2, Pr_2), \ldots, (K_{q-1}, Pr_{q-1}), P_{next} >$
- $K_1 < K_2 < ... < K_{q-1}$
- Pr_i points to a record with key value K_i, or Pr_i points to a page containing a record with key value K_i.
- Maximum of p_{leaf} key/pointer pairs.
- Each leaf has at least $\lceil p_{leaf}/2 \rceil$ keys.
- All leaves are at the same level (balanced).
- P_{next} points to the next leaf node for key sequencing.



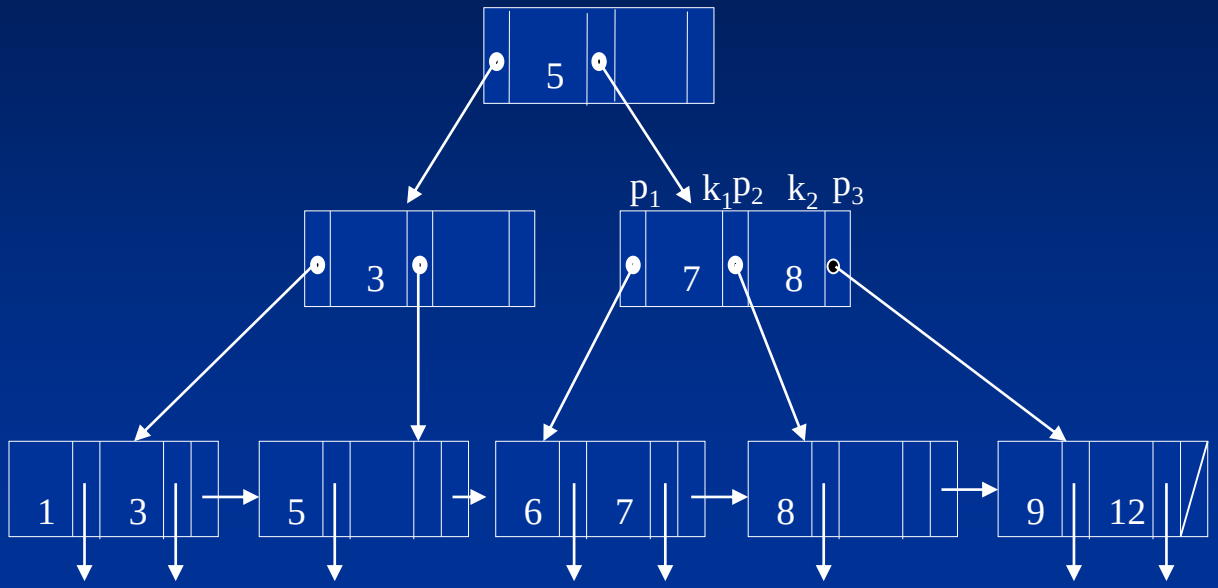
B+-tree stored in a file:

0	5			
1	3			
2	1	3		
3	5			
4	7	8		
5	6	7		
6	8			
7	9	12		

Data file:

0	1	2	3
1 5	6 12	9 7	3 8

B+-tree stored in a file:



0	1	**5**	4					
1	2	**3**	3					
2	**1**	0	**3**	3				
3	**5**	0						
4	5	**7**	6	**8**	7			
5	**6**	1	**7**	2				
6	**8**	3						
7	**9**	2	**12**	1				

Data file:

0	1	2	3	4	5	6	7
1	5	6	12	9	7	3	8

Store a B+-tree on hard disk

Algorithm:

```
push(root, -1, -1);
while (S is not empty) do
{     x := pop( );
      store x.data in file F;
      assume that the address of x in F is ad;
      if x.address-of-parent ≠ -1 then {
            y := x.address-of-parent;
            z := x.position;
            write ad in page y at position z in F;
      }
      let x1, …, xk be the children of x;
      for (i = k to 1) {push(xi, ad, i)};
}
```

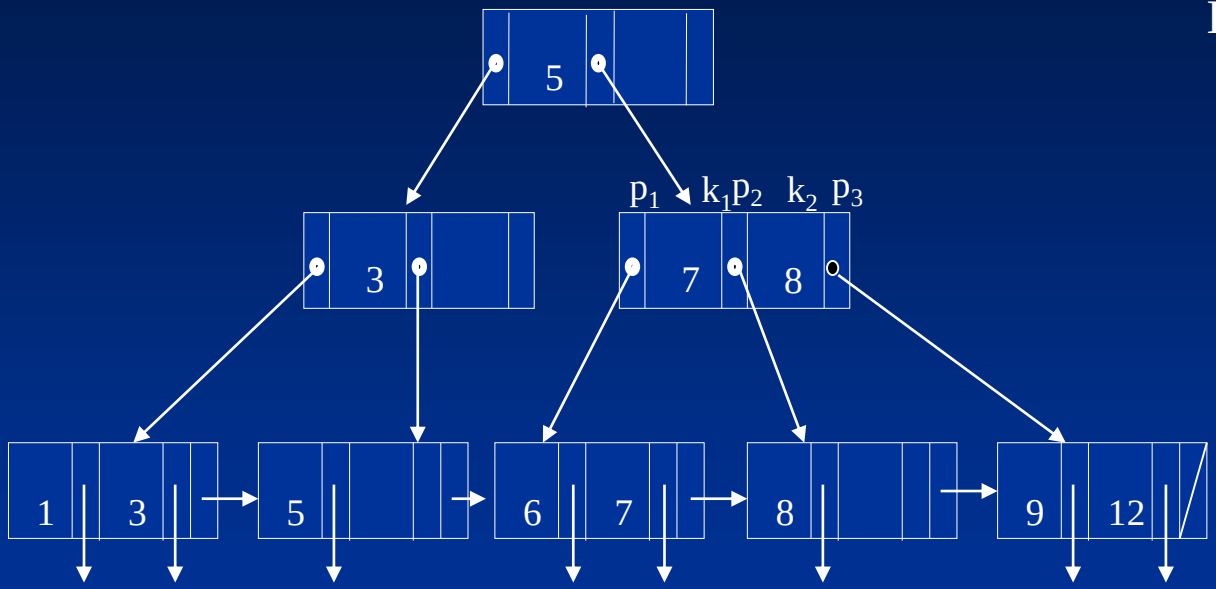
data	address-of-parent	position

stack: *S*

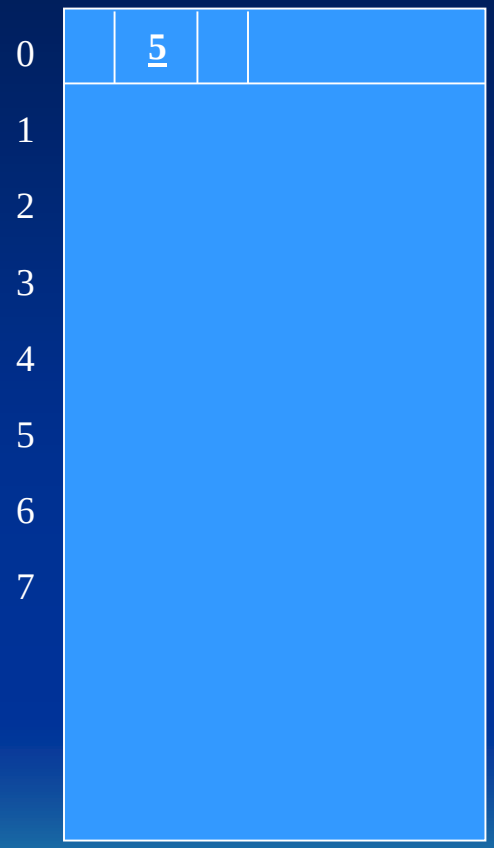
data: all the key values in a node

address-of-parent: a page number in the file *F*, where the parent of the node is stored.

position: a number indicating what is the ranking of a child. That is, whether it is the first, second, …, child of its parent.



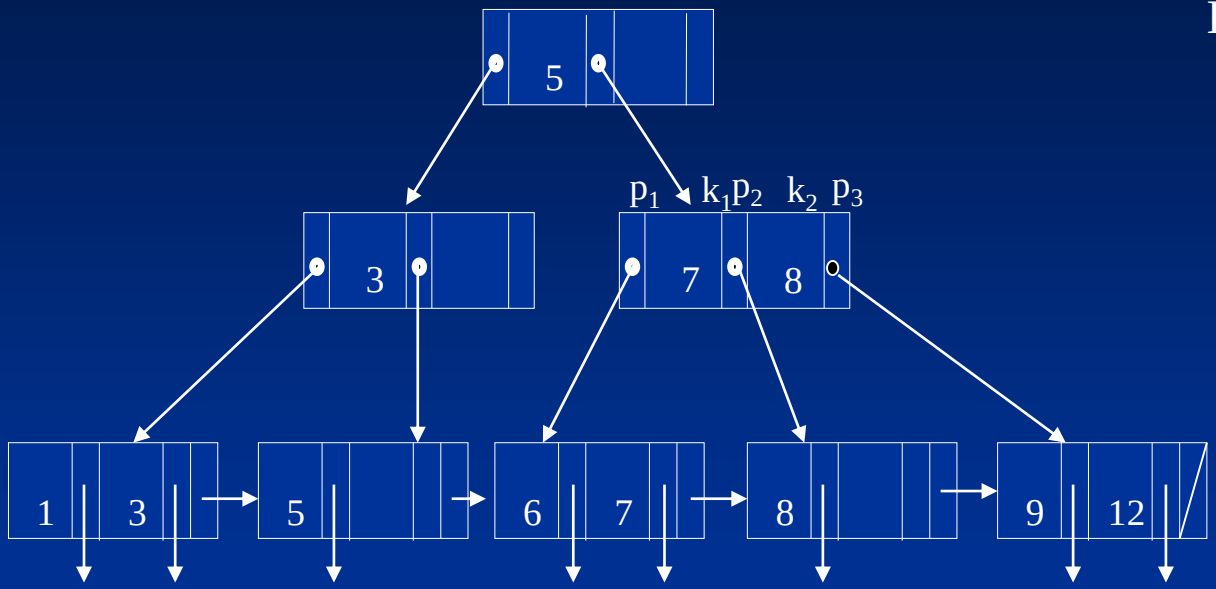
B+-tree stored in a file:



Data file:

1	5	6	12	9	7	3	8	
0		1		2		3		

Stack:



B+-tree stored in a file:

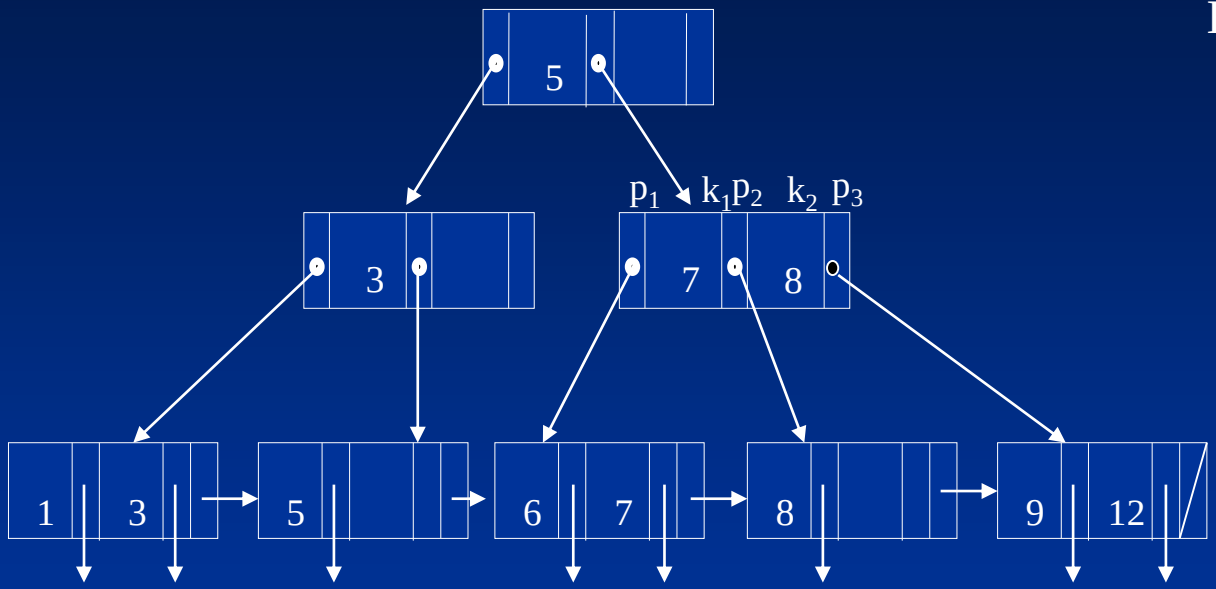
0	1	5		
1		3		
2				
3				
4				
5				
6				
7				

Data file:

1	5	6	12	9	7	3	8	

	0	1
3	0	1
7, 8	0	2

	2	3
1, 3	1	1
5	1	2
7, 8	0	2



B+-tree stored in a file:

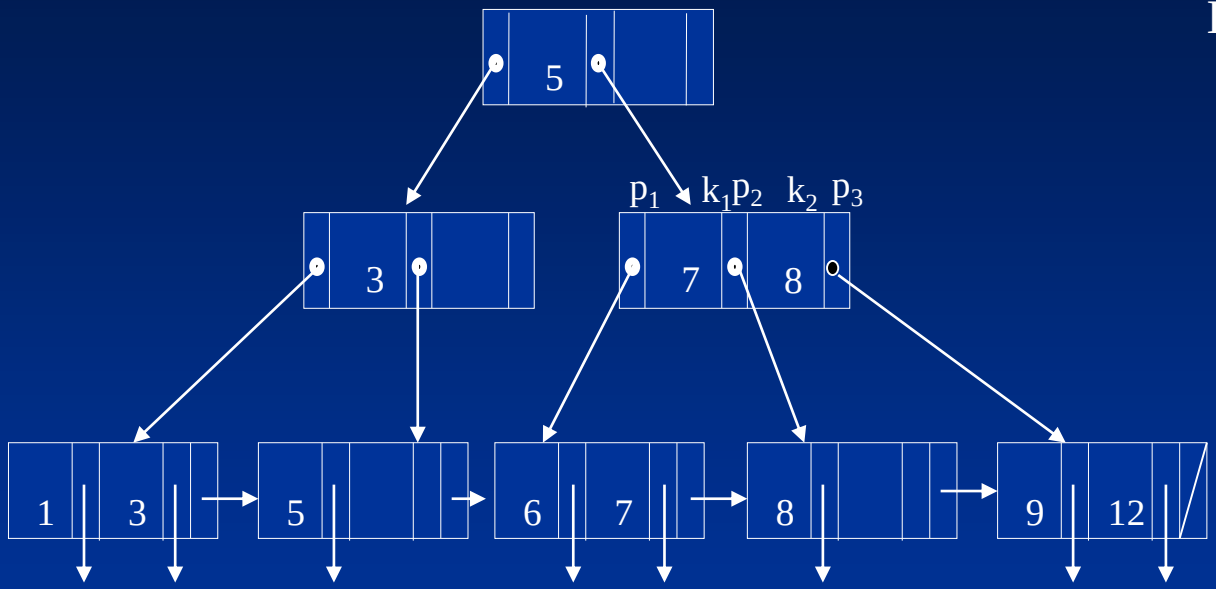
0	1	**5**		
1	2	**3**		
2	**1**	0	**3**	3
3				
4				
5				
6				
7				

Data file:

1	5	6	12	9	7	3	8

	0	1
1, 3	1	1
5	1	2
7, 8	0	2

	2	3
5	1	2
7, 8	0	2



B+-tree stored in a file:

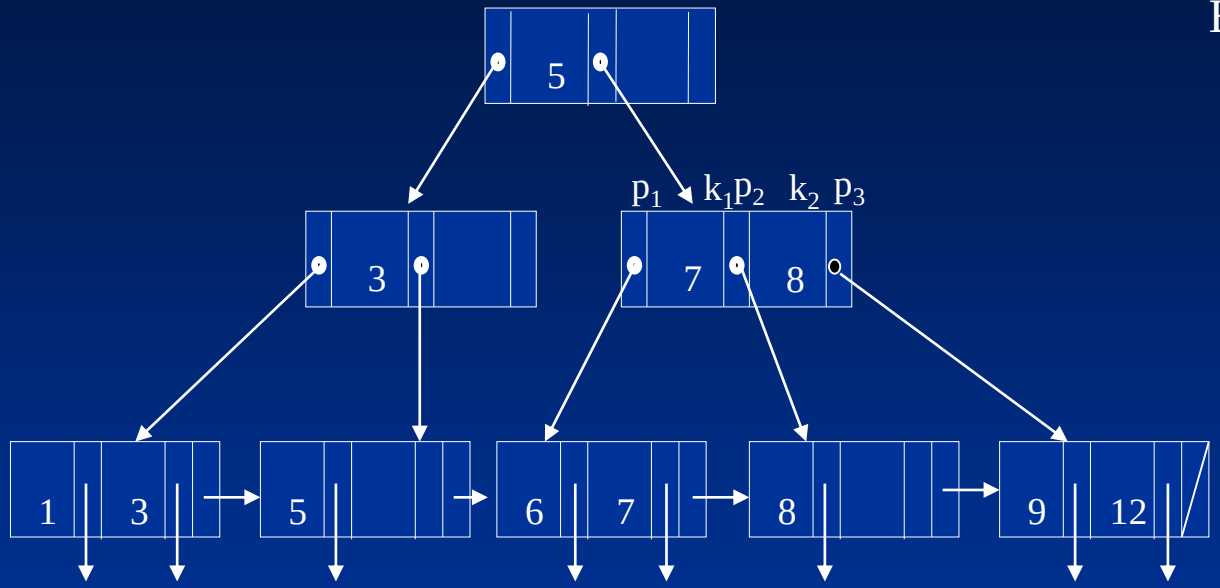
0	1	**5**		
1	2	**3**	3	
2	**1**	0	**3**	3
3	**5**	0		
4				
5				
6				
7				

Data file:

0	1	2	3
1	5	6	12
9	7	3	8

5	1	2
7, 8	0	2

7, 8	0	2



B+-tree stored in a file:

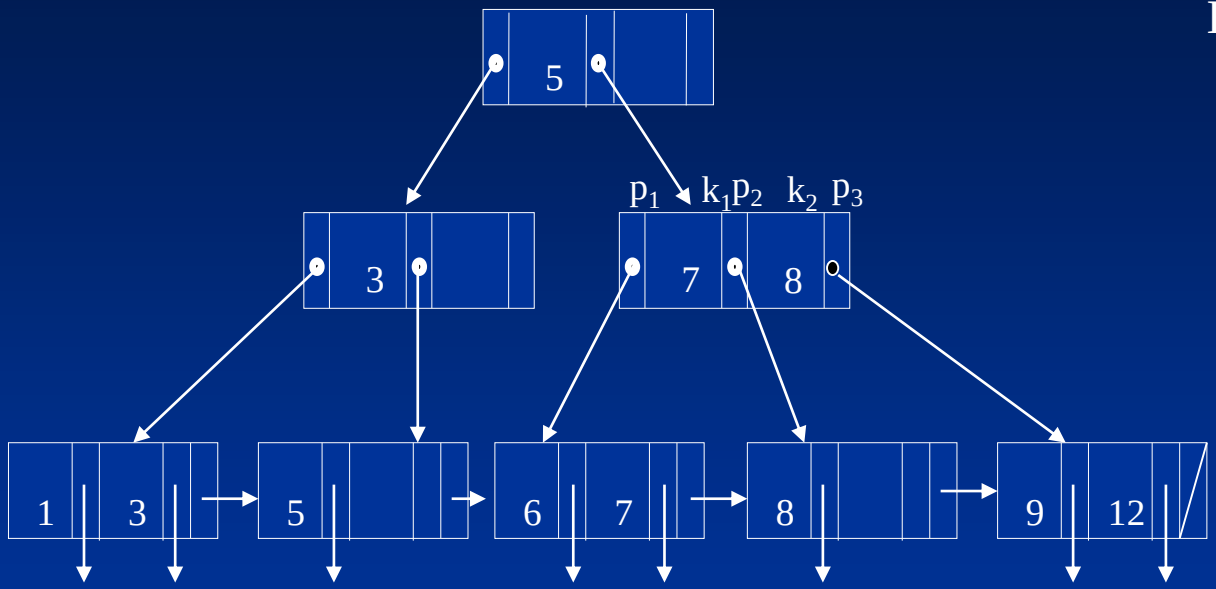
0	1	**5**	4	
1	2	**3**	3	
2	**1**	0	**3**	3
3	**5**	0		
4		**7**		**8**
5				
6				
7				

Data file:

0	1	2	3				
1	5	6	12	9	7	3	8

7, 8	0	2

6, 7	4	1
8	4	2
9,12	4	3



B+-tree stored in a file:

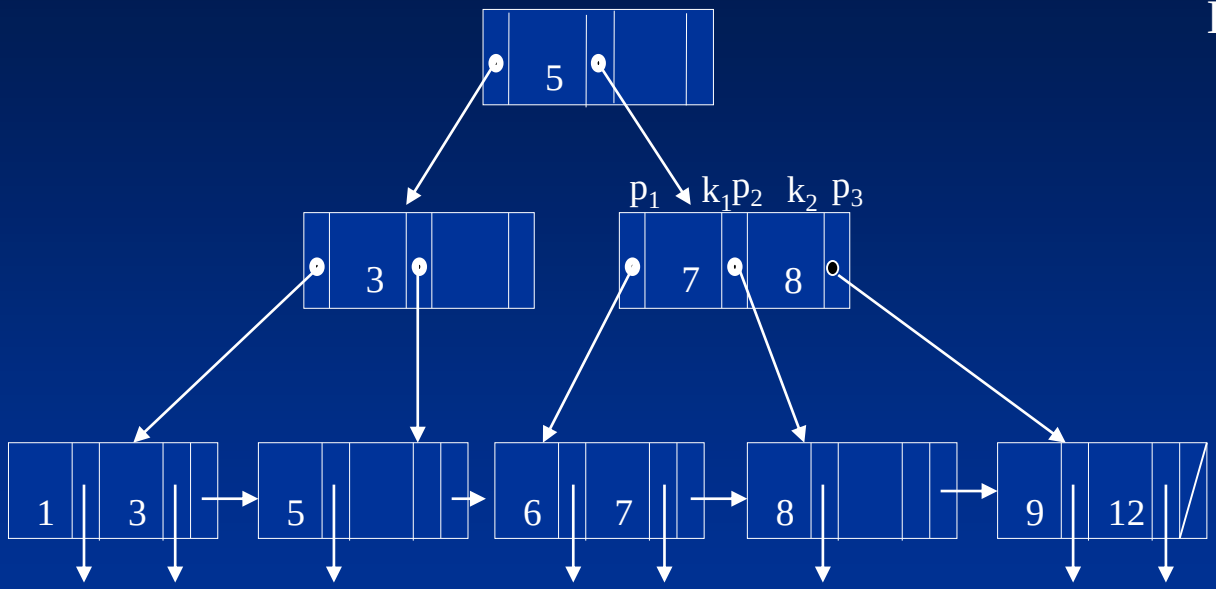
0	1	**5**	4	
1	2	**3**	3	
2	**1**	0	**3**	3
3	**5**	0		
4	5	**7**		**8**
5	**6**	1	**7**	2
6				
7				

Data file:

1	5	6	12	9	7	3	8	

	0	1	2
6, 7	4	1	
8	4	2	
9,12	4	3	

	0	1	2
8	4	2	
9,12	4	3	



B+-tree stored in a file:

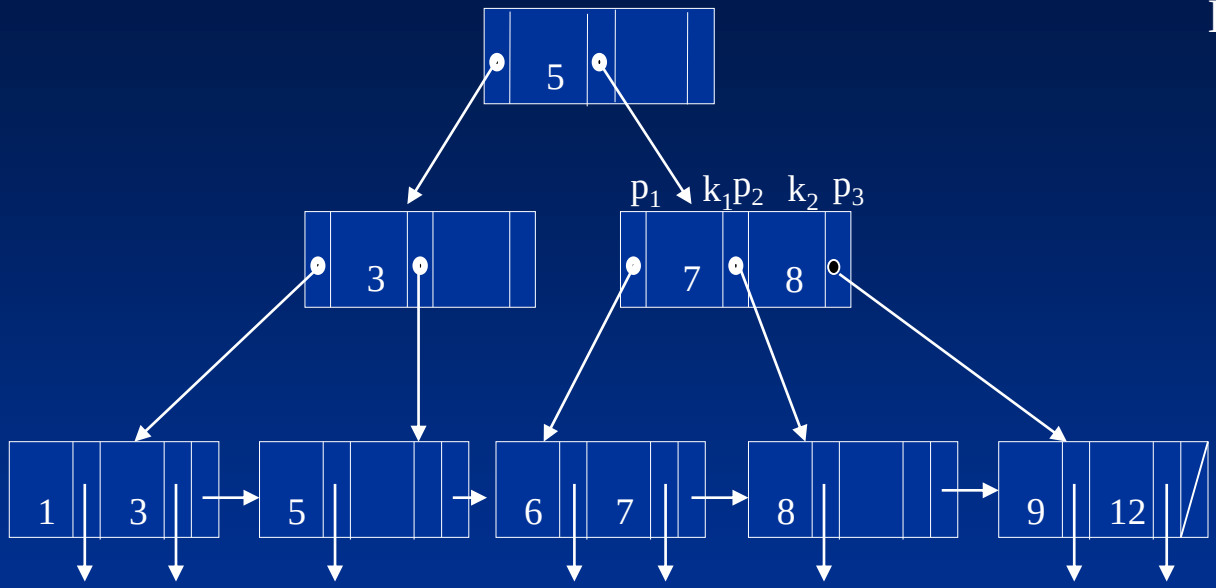
0	1	**5**	4					
1	2	**3**	3					
2	**1**	0	**3**	3				
3	**5**	0						
4	5	**7**	6	**8**				
5	**6**	1	**7**	2				
6	**8**	3						
7								

Data file:

0	1	2	3				
1	5	6	12	9	7	3	8

8	4	2
9,12	4	3

9,12	4	3



B+-tree stored in a file:

0	1	**5**	4			
1	2	**3**	3			
2	**1**	0	**3**	3		
3	**5**	0				
4	5	**7**	6	**8**	7	
5	**6**	1	**7**	2		
6	**8**	3				
7	**9**	2	**12**	1		

Data file:

0	1	2	3
1	5	6	12	9	7	3	8

9,12	4	3

empty stack

Index Structures for Multidimensional Data

- Multiple-key indexes
- *kd*-trees
- Quad trees
- R-trees
- Bit map

Indexes over texts

- Inverted files

Multiple-key indexes

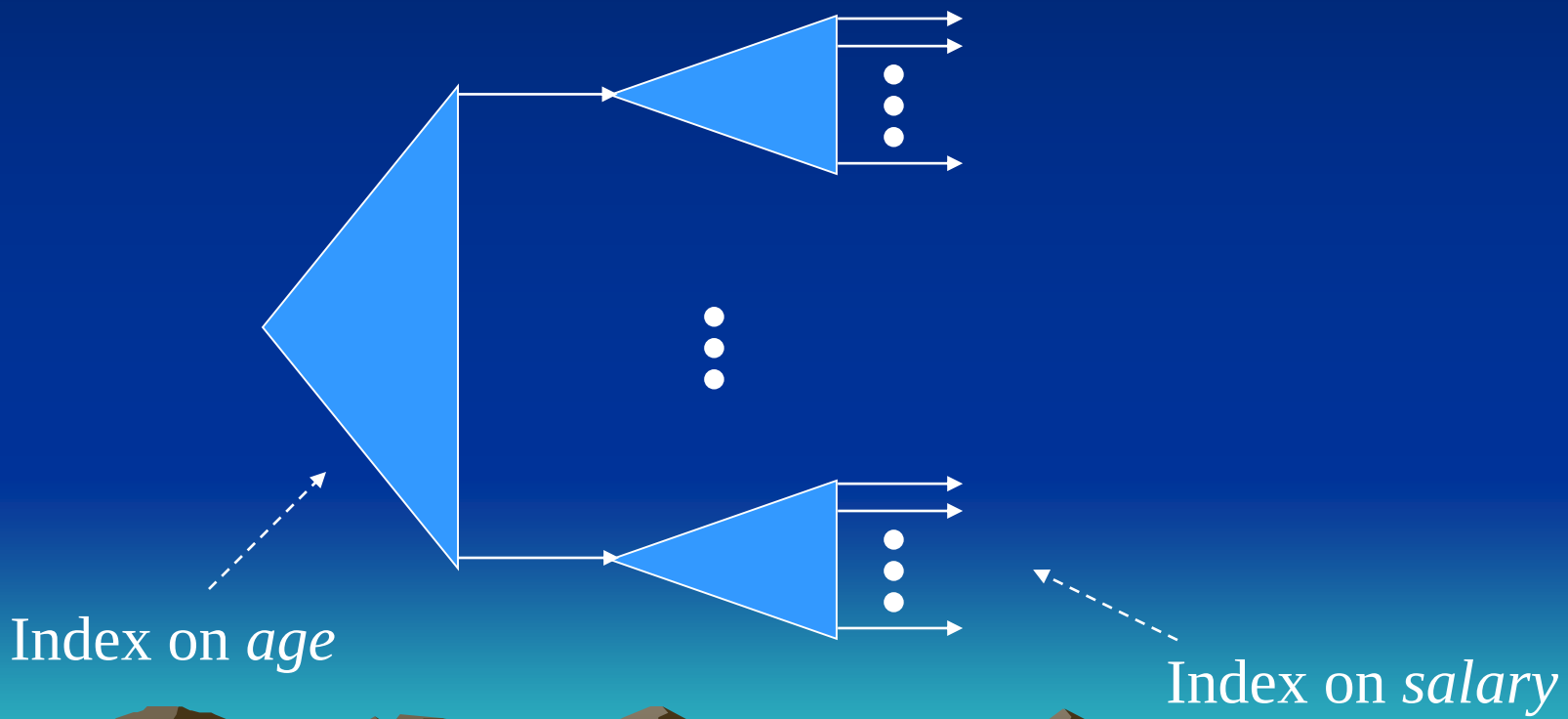
(Indexes over more than one attributes)

Employee

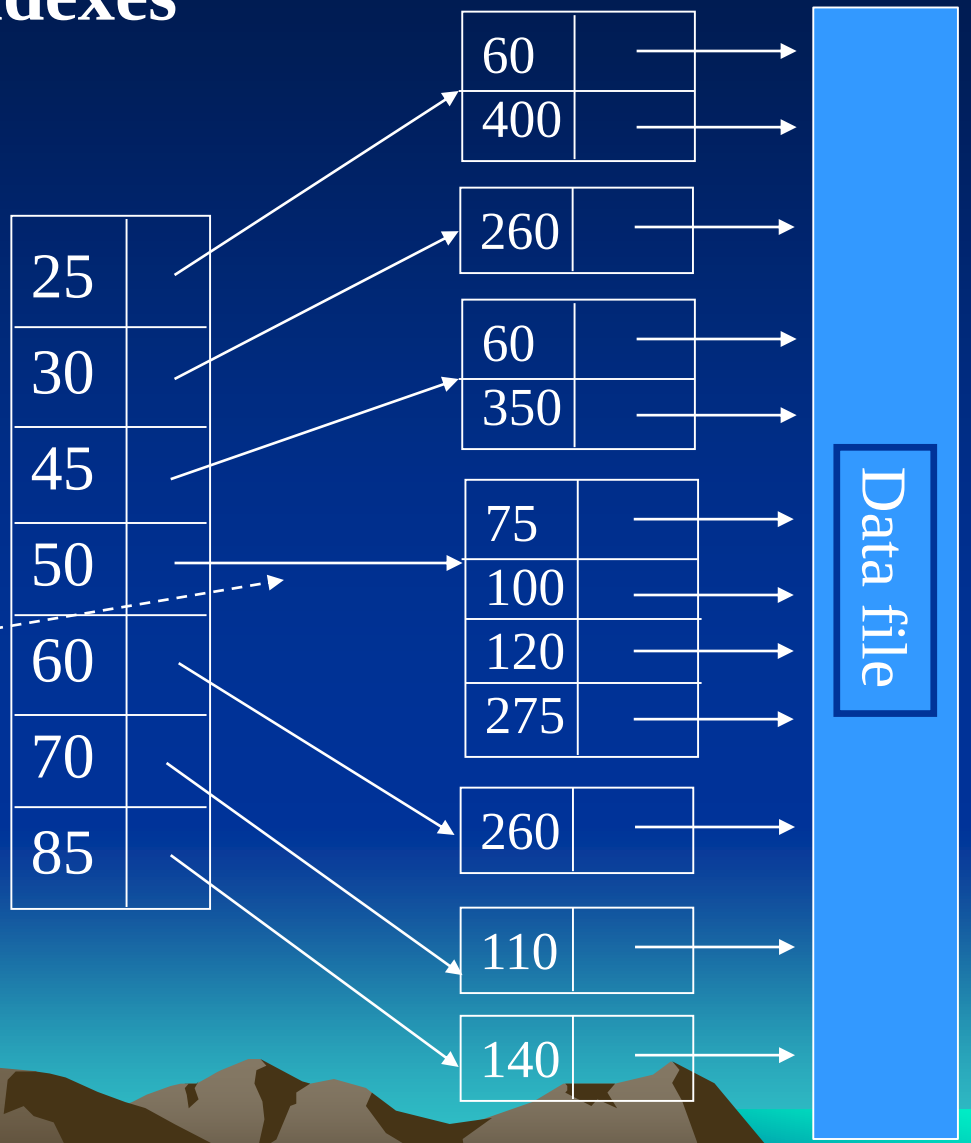
ename	ssn	age	salary	dnumber
Aaron, Ed				
Abbott, Diane				
Adams, John				
Adams, Robin				

Multiple-key indexes

(Indexes over more than one attributes)



Multiple-key indexes



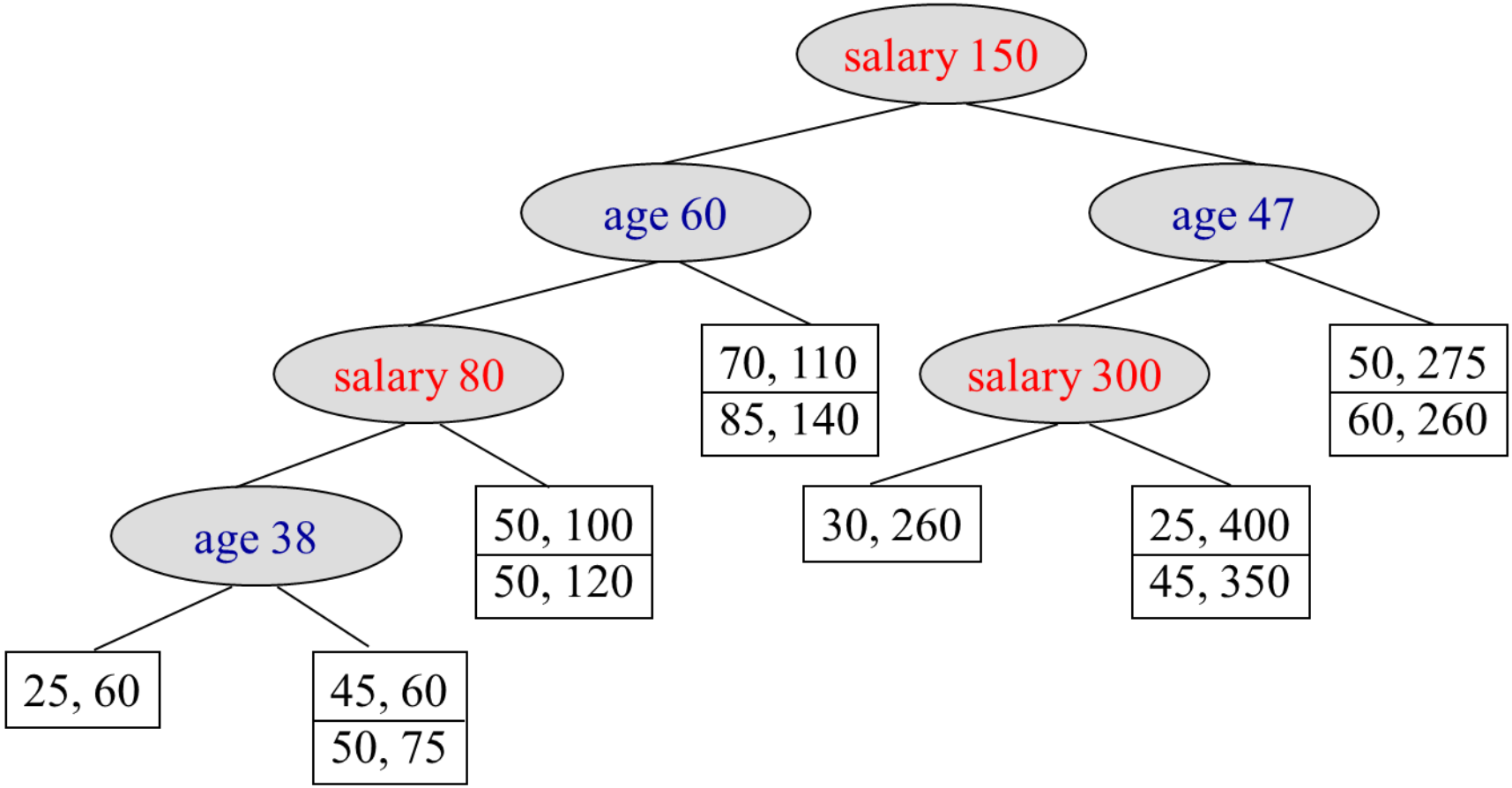
pointer to an index over a set of salaries of all those employees of age between 50 and 60

kd-Trees

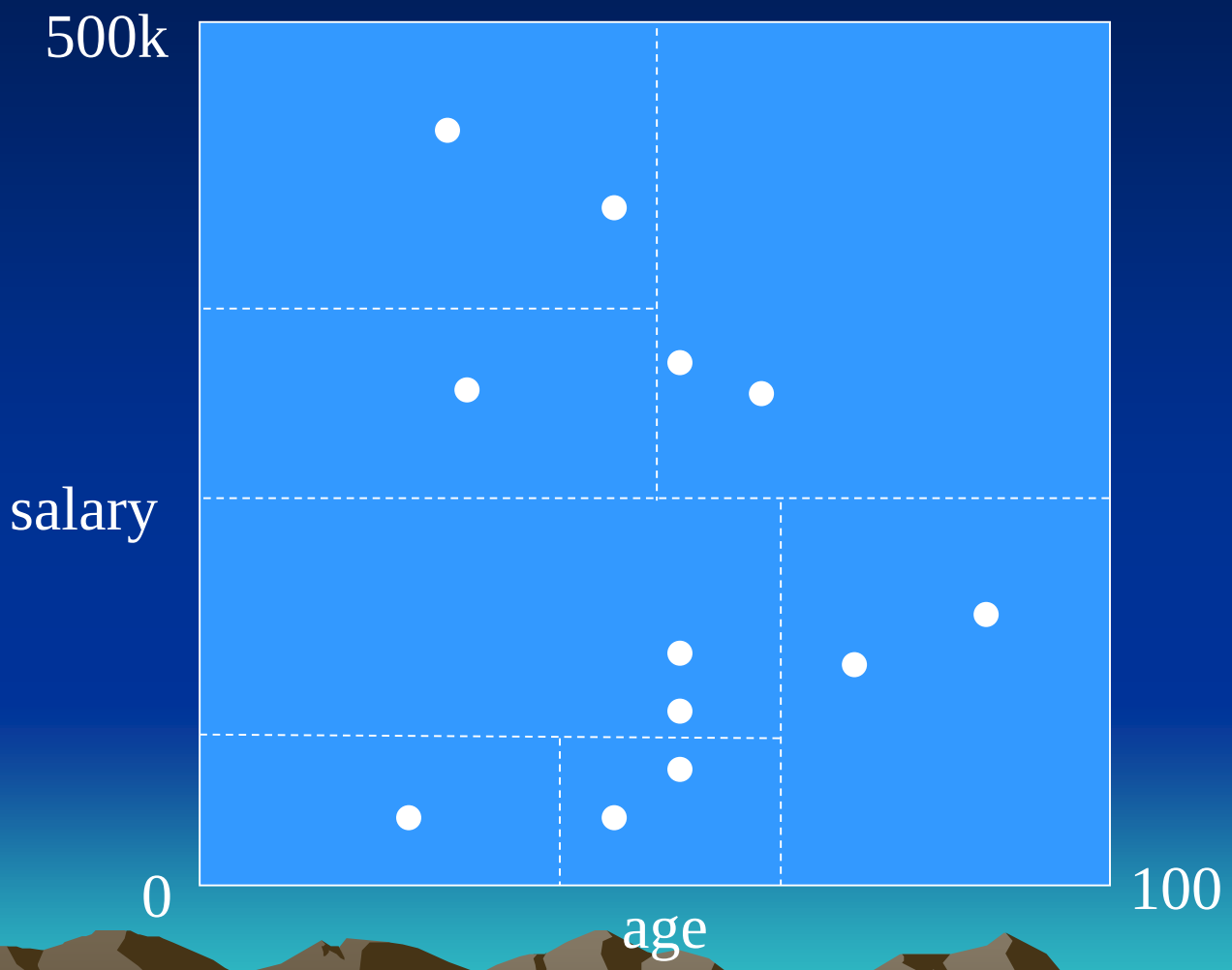
(A generalization of binary search trees)

A *kd*-tree is a binary tree in which interior nodes have an associated attribute *a* and a value *v* that splits the data points into two parts: those with *a*-value less than *v* and those with *a*-value equal to or larger than *v*.

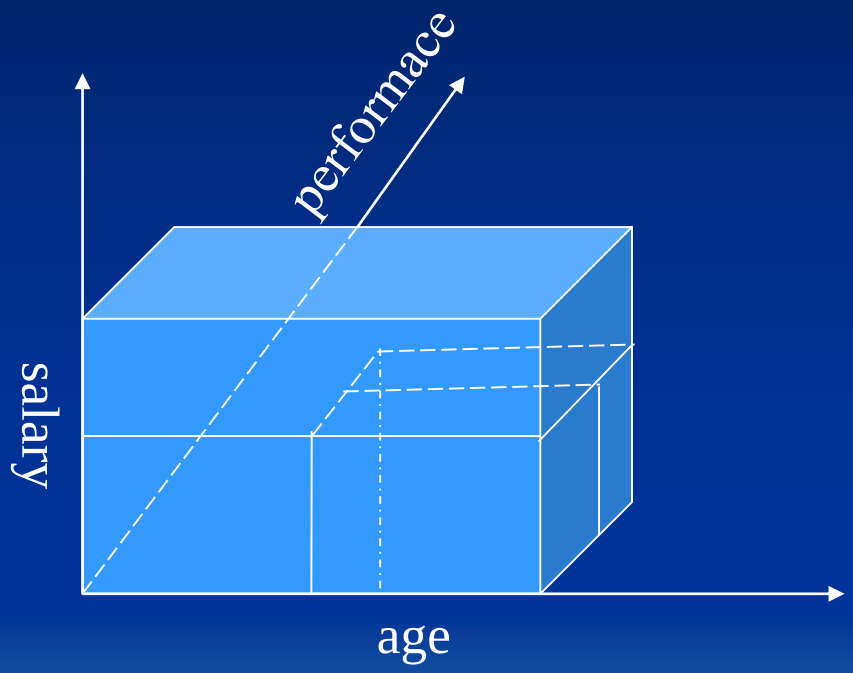
kd-Trees


salary 150
age 60
age 47
salary 80
70, 110
85, 140
salary 300
50, 275
60, 260
age 38
50, 100
50, 120
30, 260
25, 400
45, 350
25, 60
45, 60
50, 75

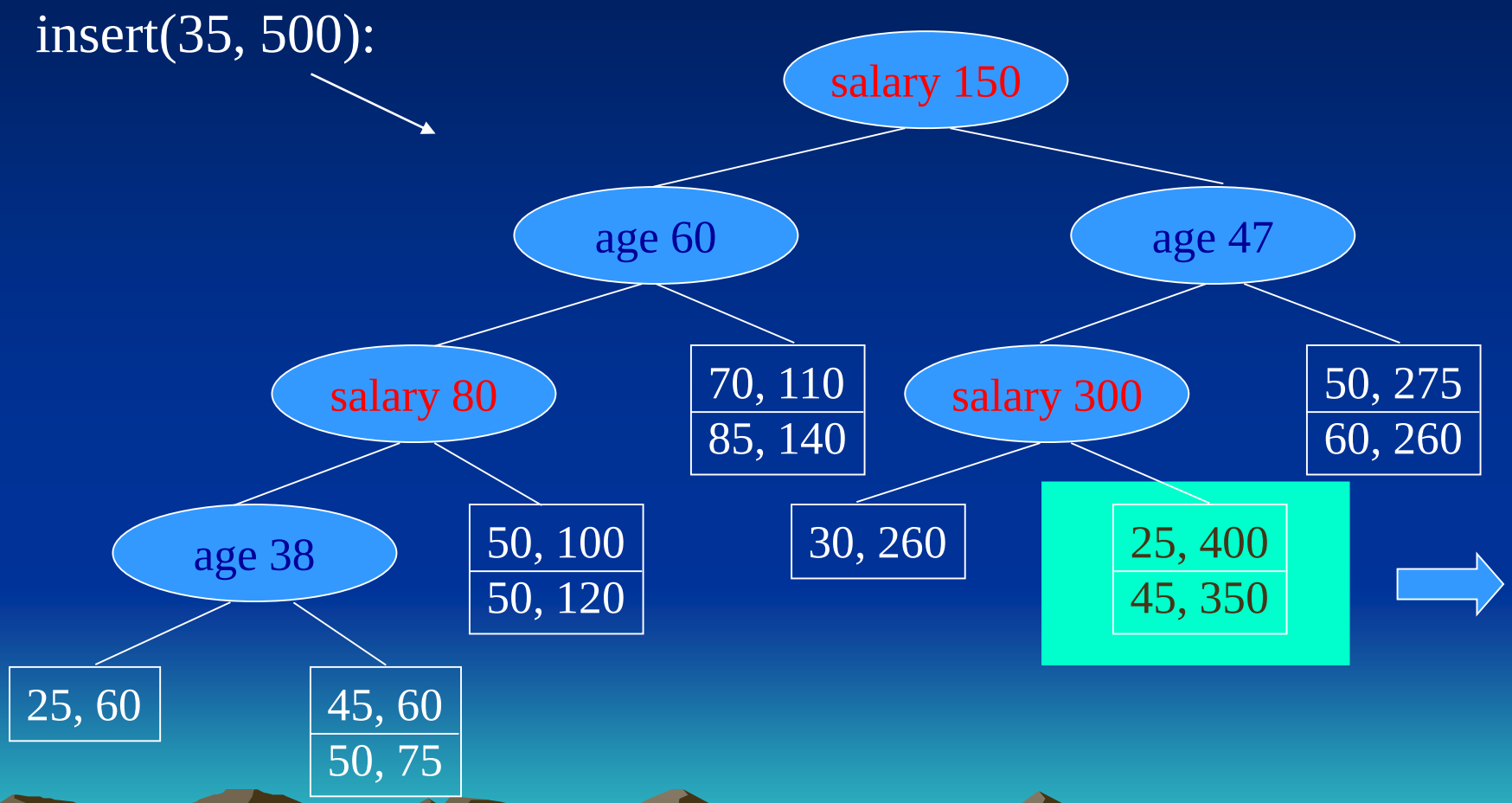
kd-trees



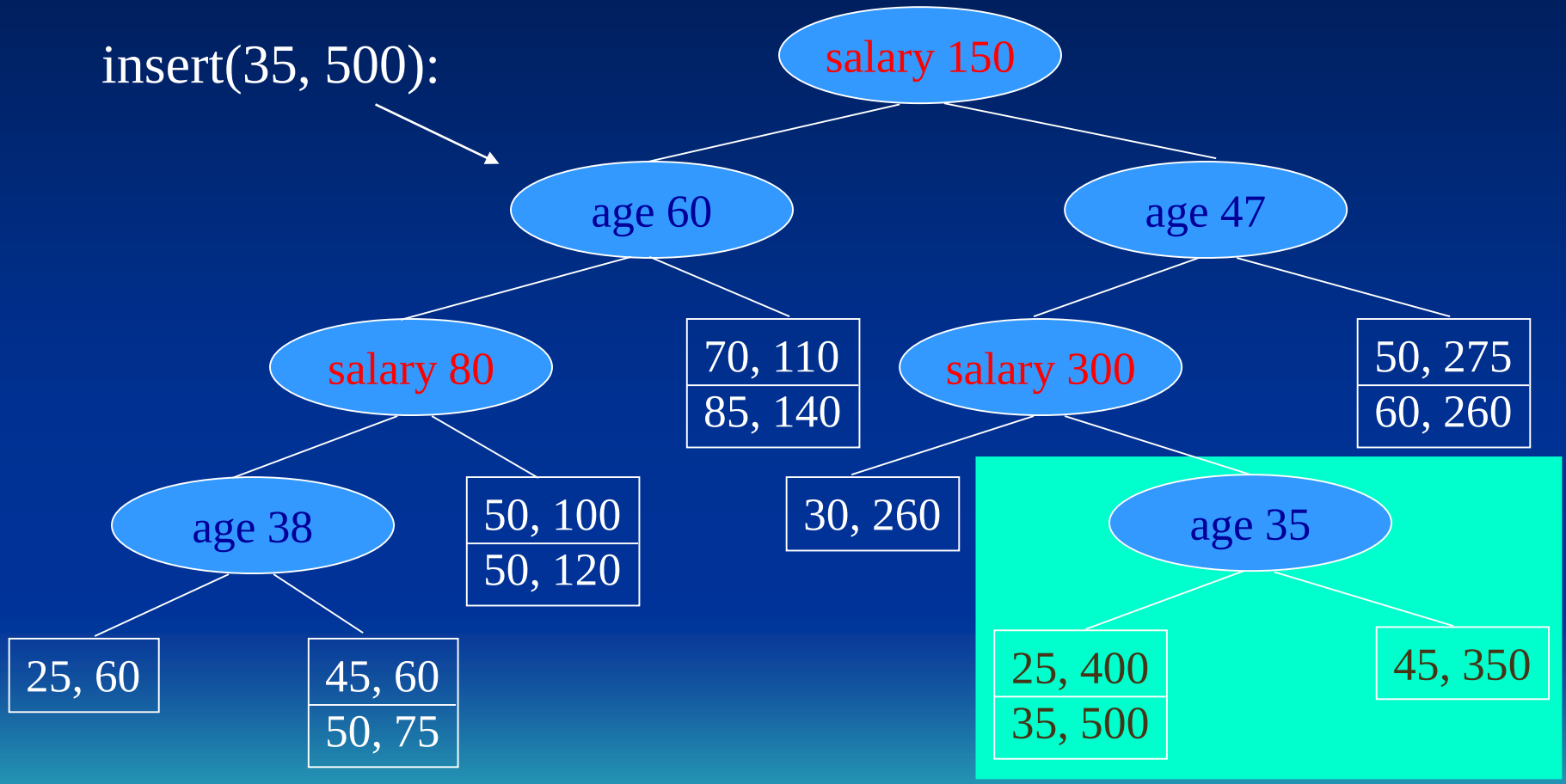
3-dimensional data division



Insert a new entry into a *kd*-tree:



Insert a new entry into a *kd*-tree:



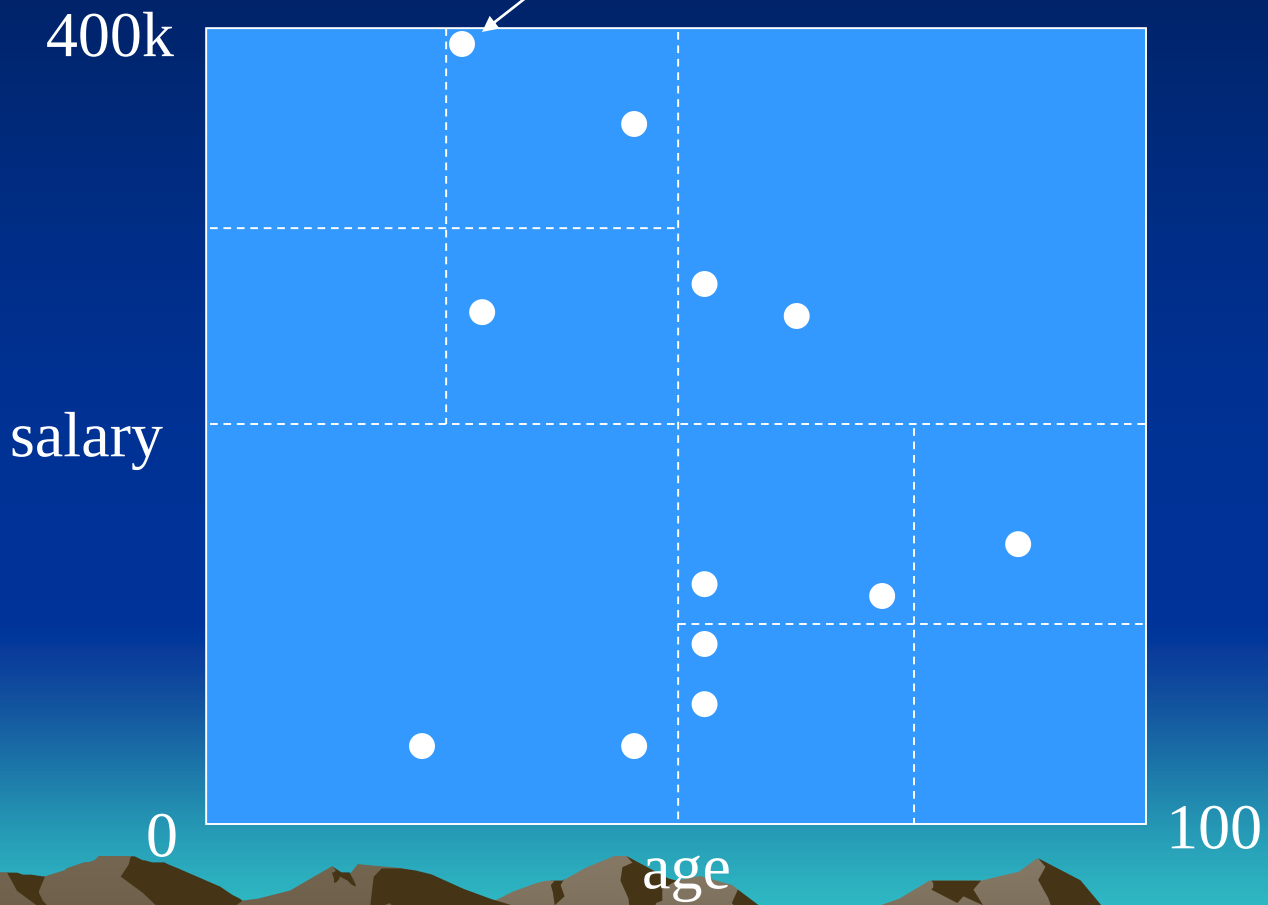
Quad-trees

In a Quad-tree, each node corresponds to a square region in two dimensions, or to a *k*-dimensional cube in *k* dimensions.

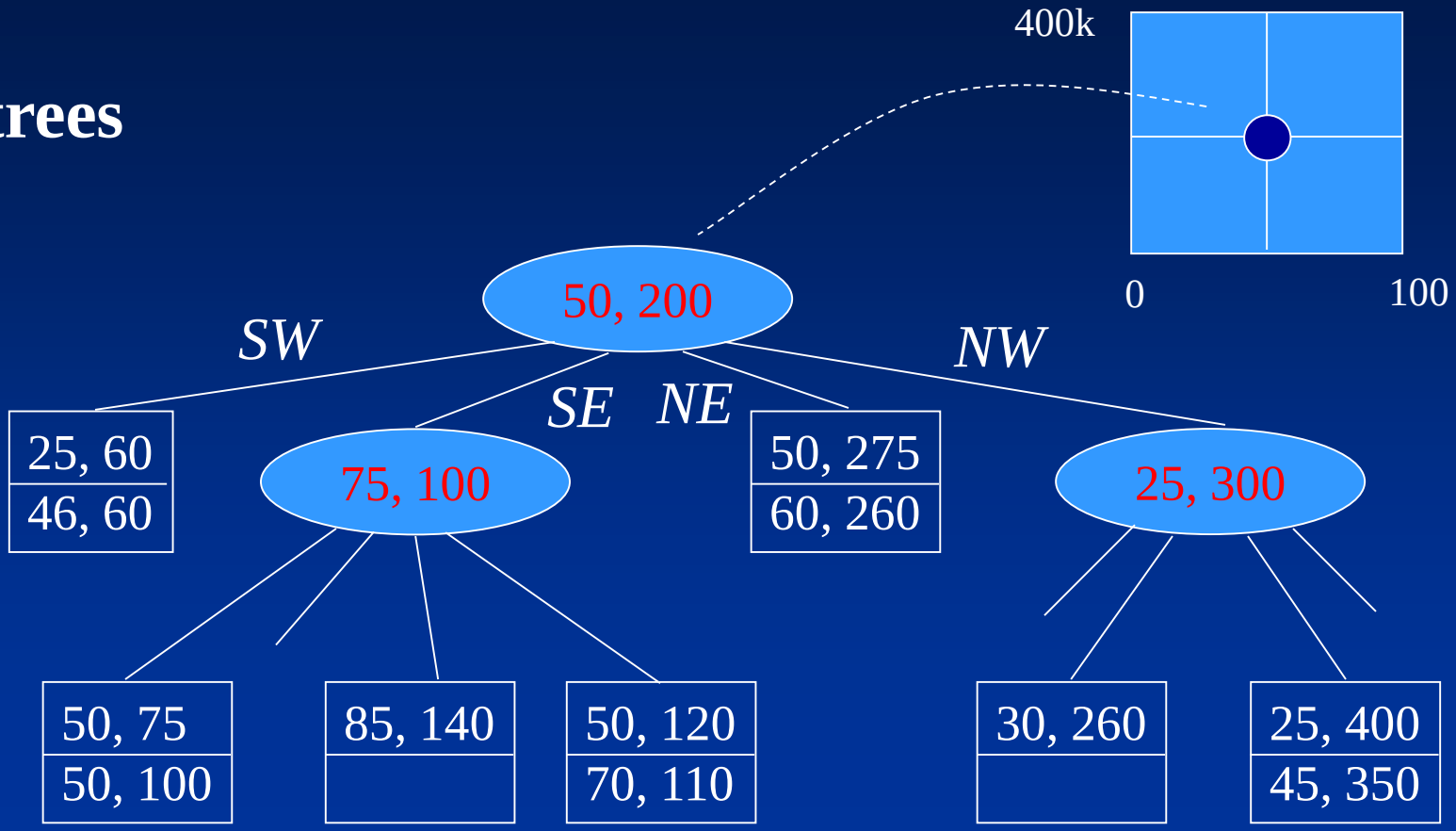
- If the number of data entries in a square is not larger than what will fit in a block, then we can think of this square as a leaf node.
- If there are too many data entries to fit in one block, then we treat the square as an interior node, whose children correspond to its four quadrants.

Quad-trees

name	age	…	salary	…
…	25	…	400	…



Quad-trees



NW – north-west
NE – north-east
SW – south-west
SE – south-east

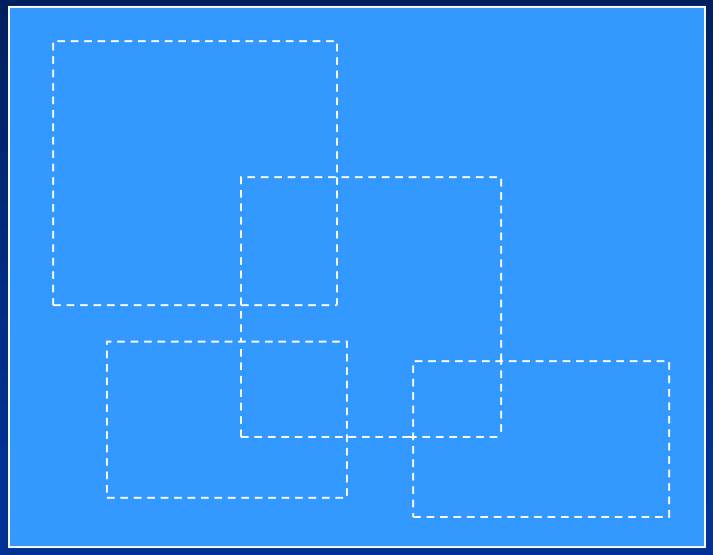
R-trees

An R-tree is an extension of B-trees for multidimensional data.

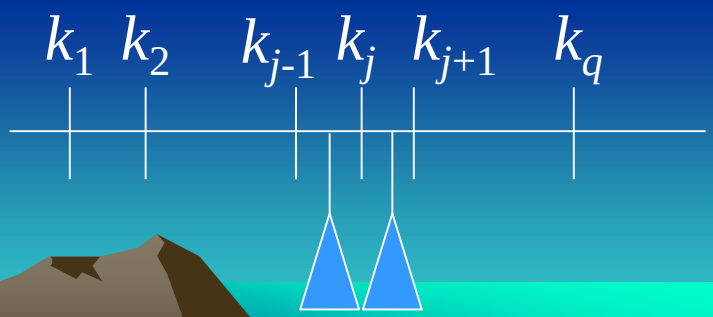
- An R-tree corresponds to a whole area (a rectangle for two-dimensional data.)
- In an R-tree, any interior node corresponds to some interior regions, or just regions, which are usually a rectangle
- Each region *x* in an interior node *n* is associated with a link to a child of *n*, which corresponds to all the subregions within *x*.

R-trees

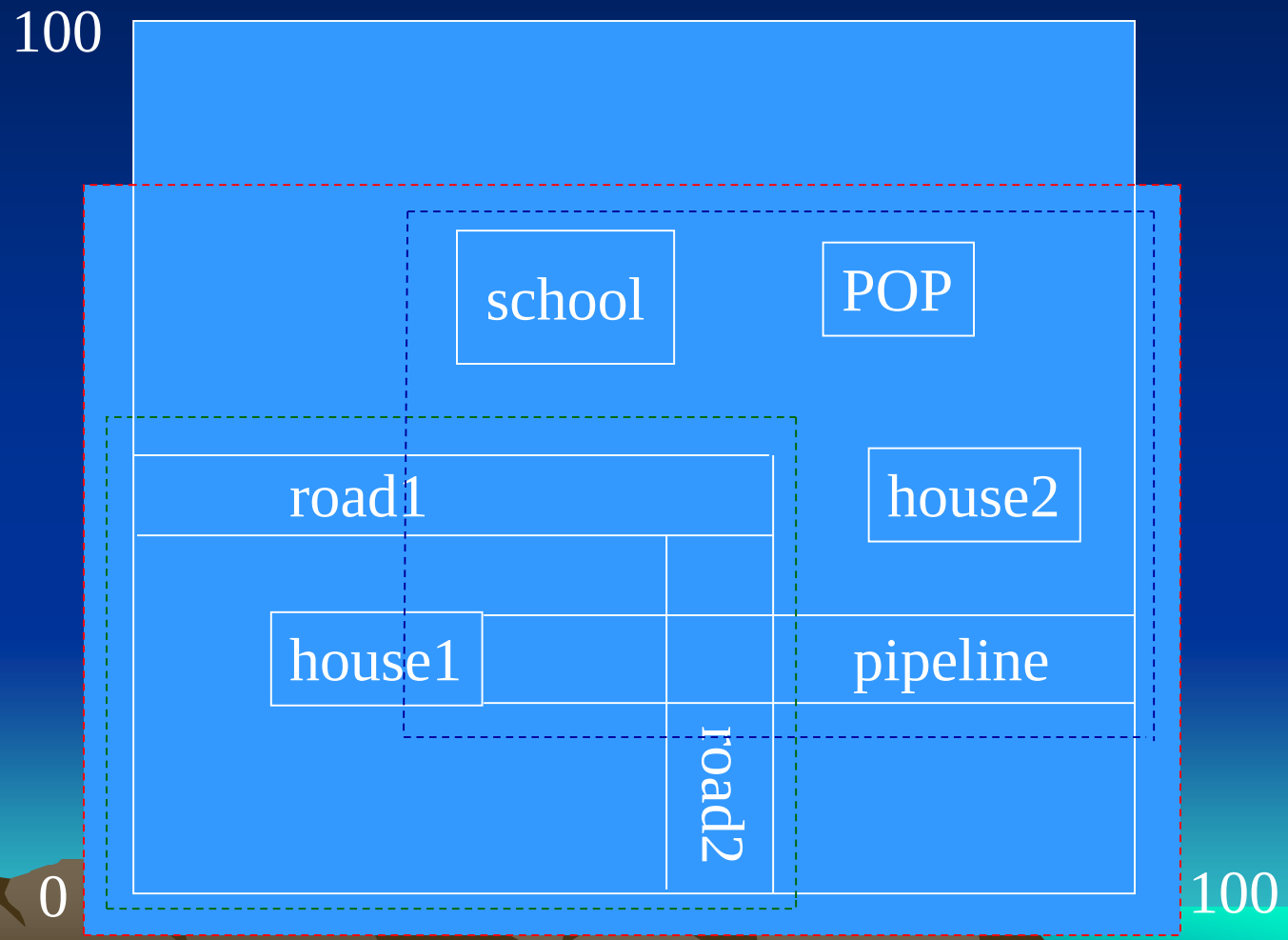
In an R-tree, each interior node contains several subregions.



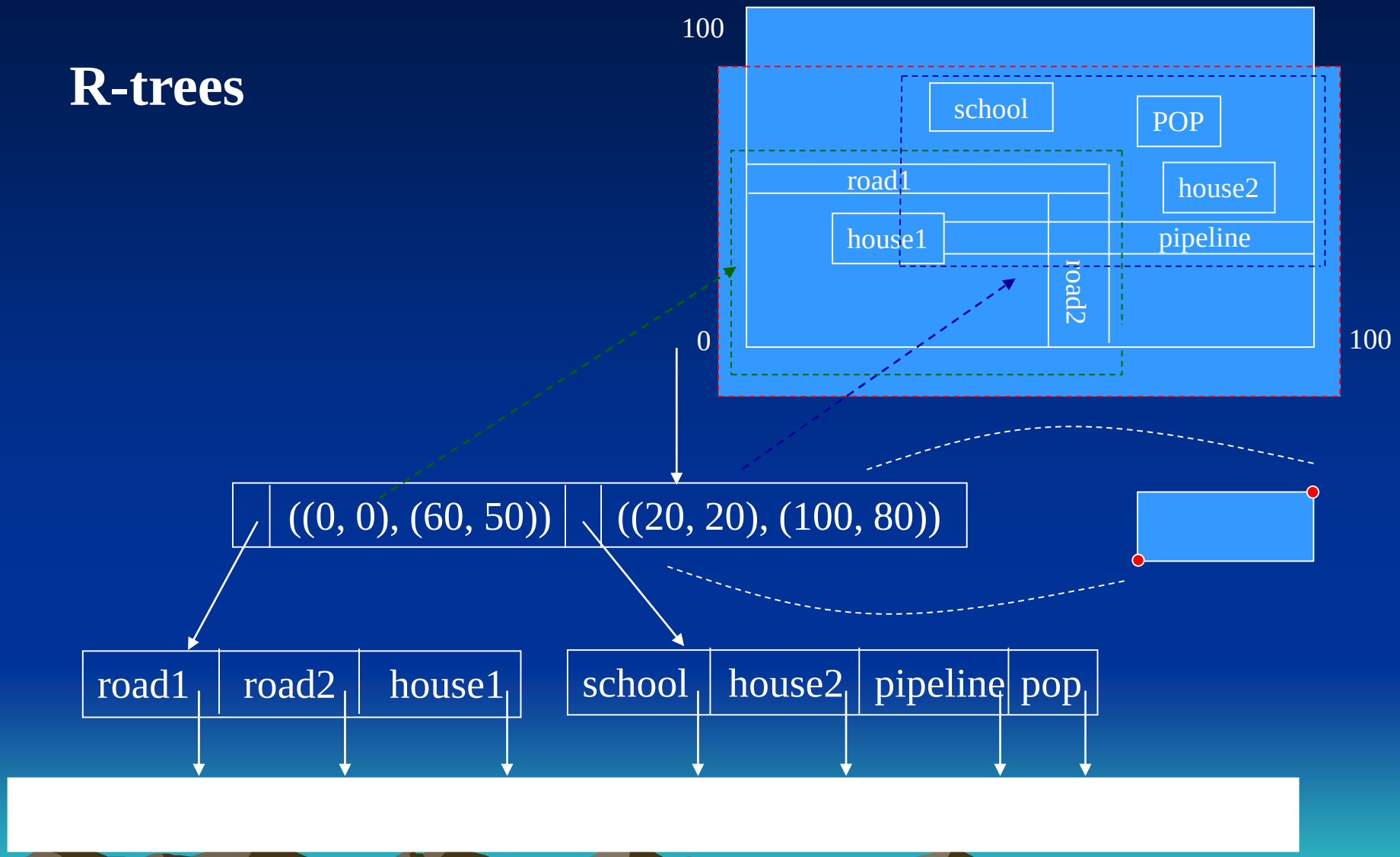
In a B+-tree, each interior node contains a set of keys that divides a line into segments.



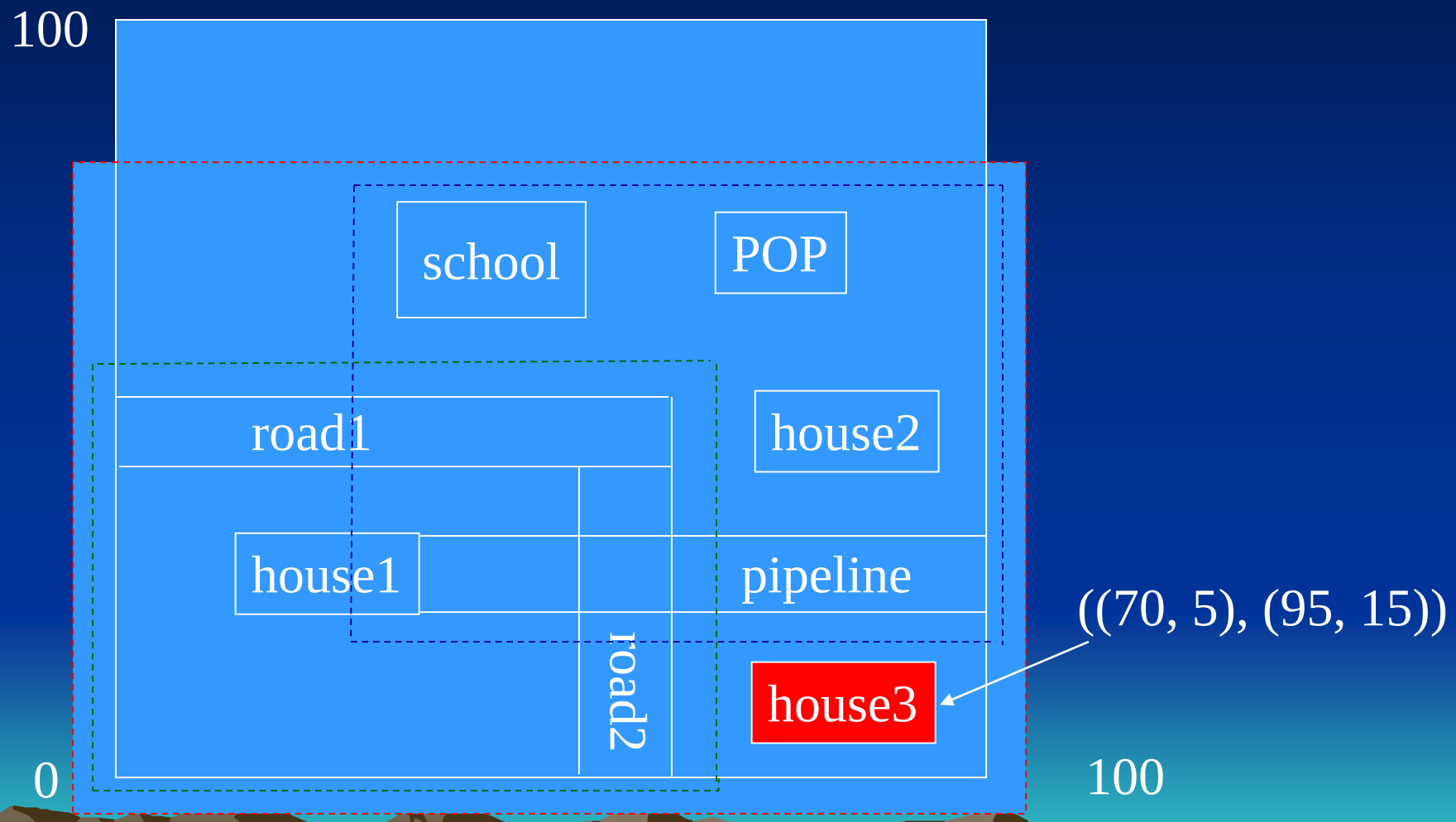
Suppose that the local cellular phone company adds a POP (point of presence, or base station) at the position shown below.



R-trees


100
school
POP
road1
house2
house1
pipeline
road2
0
100
((0, 0), (60, 50))
((20, 20), (100, 80))
road1
road2
house1
school
house2
pipeline
pop

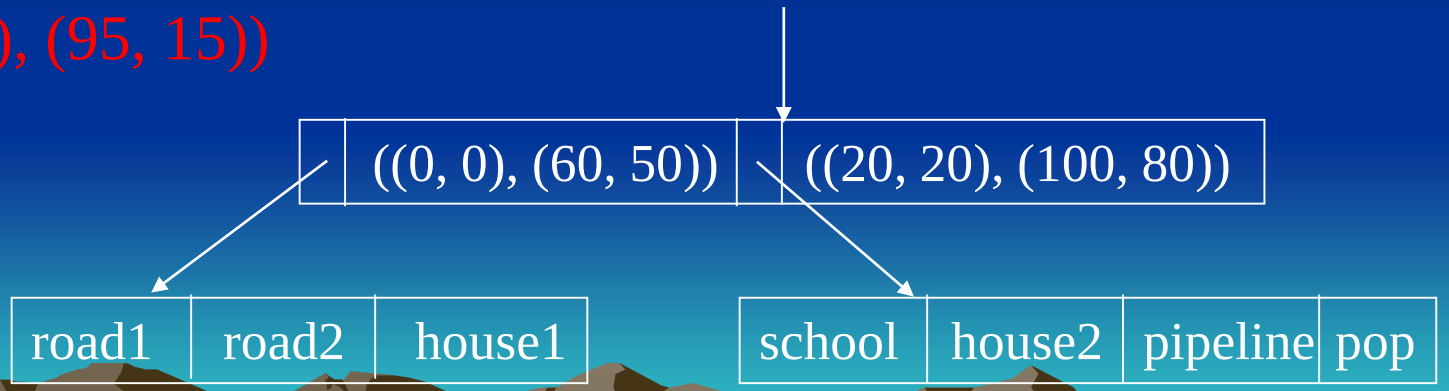
Insert a new region *r* into an R-tree.



Insert a new region *r* into an R-tree.

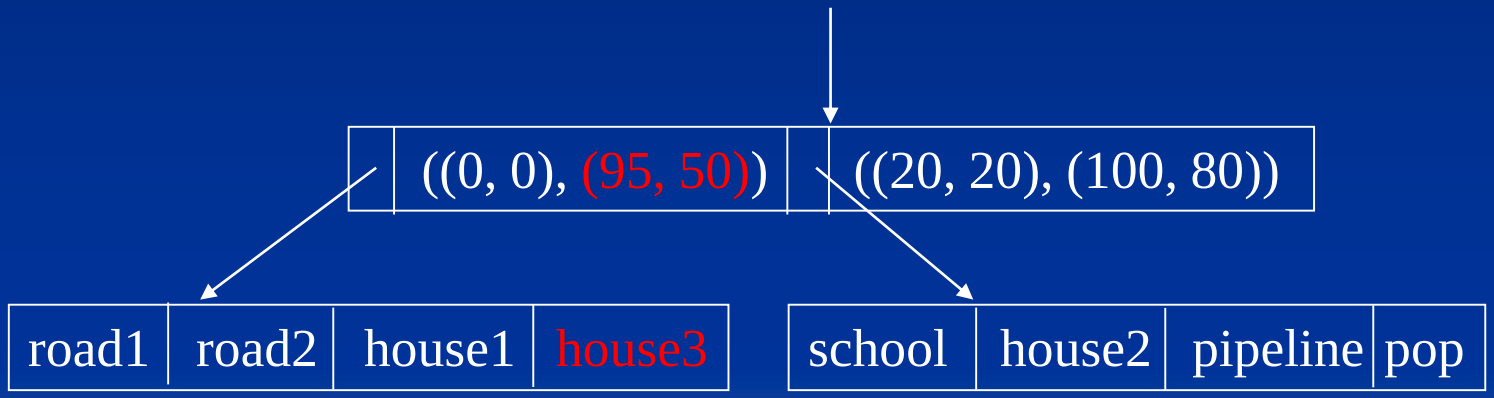
1. Search the *R*-tree, starting at the root.
2. If the encountered node is internal, find a subregion into which *r* fits.
   - If there is more than one such region, pick one and go to its corresponding child.
   - If there is no subregion that contains *r*, choose any subregion such that it needs to be expanded as little as possible to contain *r*.

((70, 5), (95, 15))

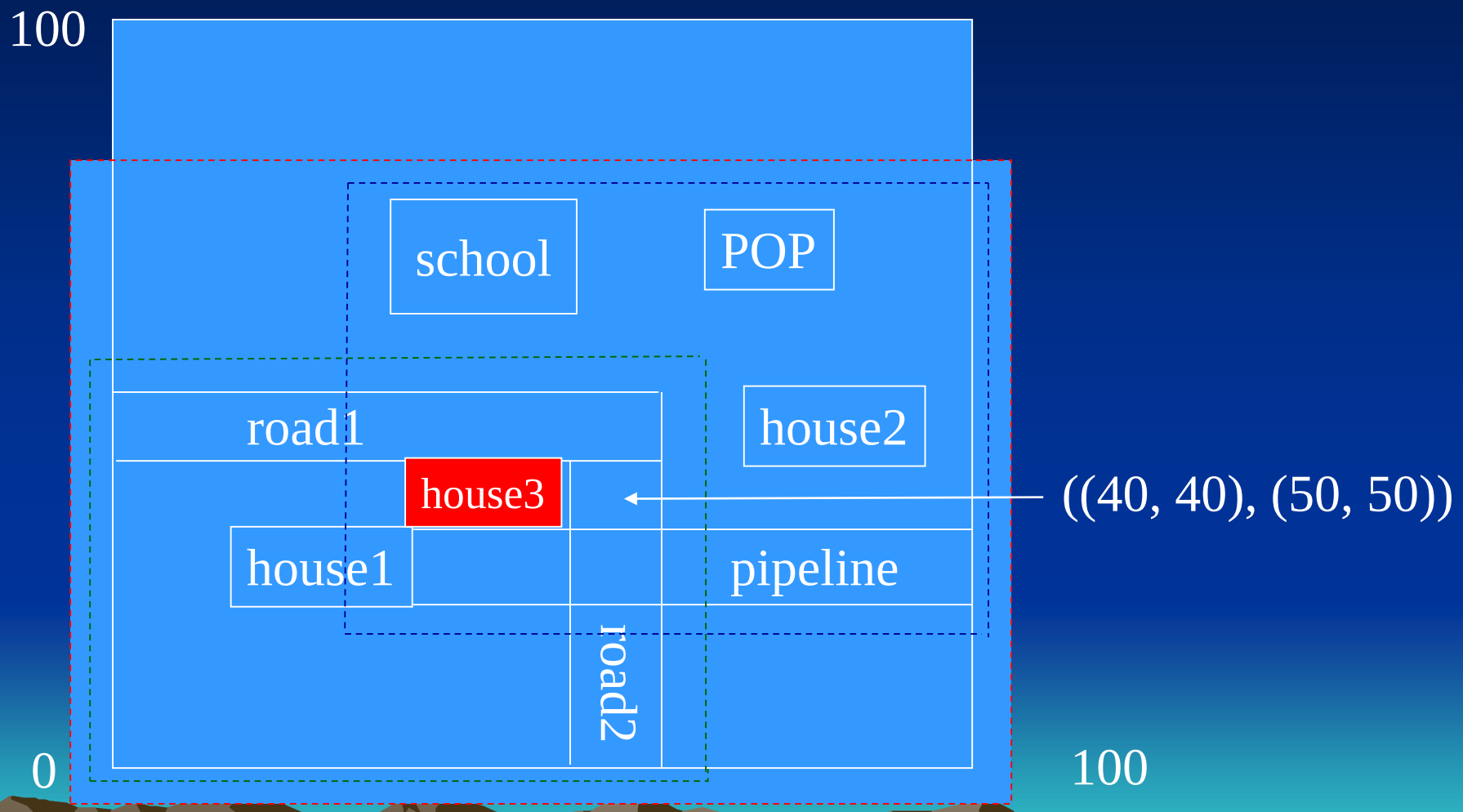


Two choices:

- If we expand the lower subregion, corresponding to the first leaf, then we add 1050 square units to the region.
- If we extend the other subregion by lowering its bottom by 15 units, then we add 1200 square units.



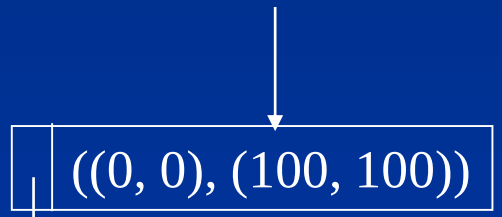
Insert a new region *r* into an R-tree.



Insert a new region *r* into an R-tree.

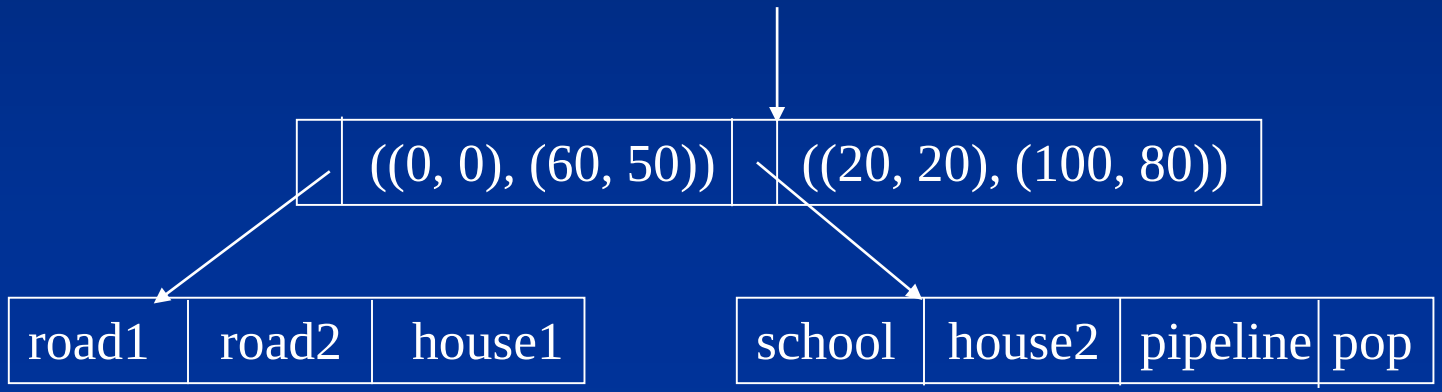
3. If the encountered node *v* is a leaf, insert *r* into it. If there is no room for *r*, split the leaf into two and distribute all subregions in them as evenly as possible. Calculate the 'parent' regions for the new leaf nodes and insert them into *v*'s parent. If there is the room at *v*'s parent, we are done. Otherwise, we recursively split nodes going up the tree.

Suppose that each leaf has room for 6 regions.



Add POP (point of presence, or base station)

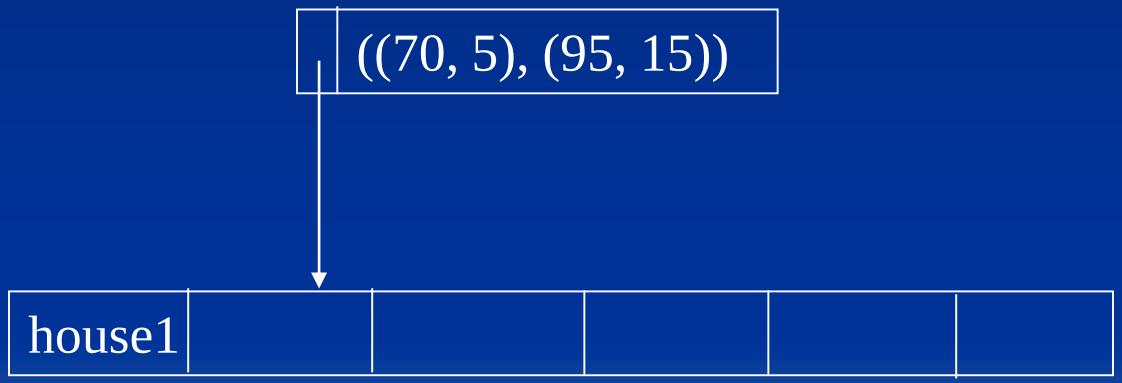
- Split the leaf into two and distribute all the regions evenly.
- Calculate two new regions each covering a leaf.



Insert the first object into an R-tree:

house1
((70, 5), (95, 15))

$R = \varnothing$



Bit map

1. Imagine that the records of a file are numbered 1, …, *n*.
2. A bitmap for a data field *F* is a collection of bit-vectors of length *n*, one for each possible value that may appear in the field *F*.
3. The vector for a specific value *v* has 1 in position *i* if the *i*th record has *v* in the field *F*, and it has 0 there if not.

Example

Employee

ename	ssn	age	salary	dnumber
Aaron, Ed		30	60	
Abbott, Diane		30	60	
Adams, John		40	75	
Adams, Robin		50	75	
Brian, Robin		55	78	
Brian, Mary		55	80	
Widom, Jones		60	100	

Bit maps for *age*:

30: 1100000
40: 0010000
50: 0001000
55: 0000110
60: 0000001

Bit maps for *salary*:

60: 1100000
75: 0011000
78: 0000100
80: 0000010
100: 0000001

Example

Employee

ename	ssn	age	salary	dnumber
Aaron, Ed		30	60	
Abbott, Diane		30	60	
Adams, John		40	75	
Adams, Robin		50	75	
Brian, Robin		55	78	
Brian, Mary		55	80	
Widom, Jones		60	100	

Bit maps for *age*:

30: 1100000
40: 0010000
50: 0001000
55: 0000110
60: 0000001

Bit maps for *salary*:

60: 1100000
75: 0011000
78: 0000100
80: 0000010
100: 0000001

Range query evaluation

```
Select ename
From Employee
Where 40 ≤ age ≤ 50 and 50 ≤ salary ≤ 78
```

We first find the bit-vectors for the age values in (30, 50); there are only two: 0010000 and 0001000 for 40 and 50, respectively.

Take their bitwise OR: $0010000 \vee 0001000 = 0011000$.

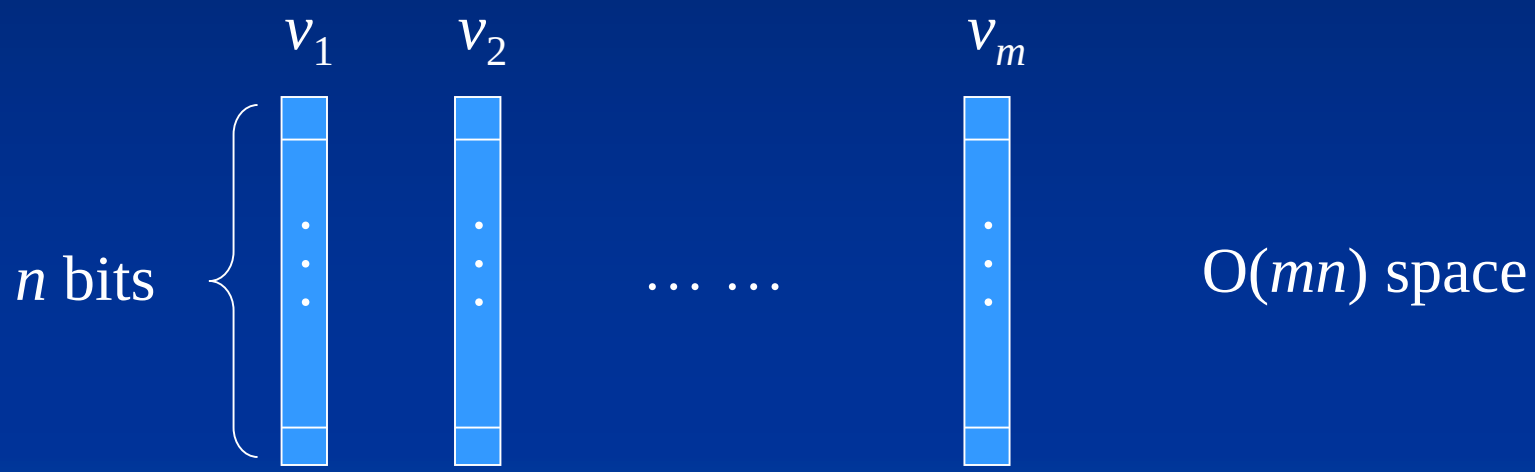
Next find the bit-vectors for the salary values in (50, 78) and take their bitwise OR: $1100000 \vee 0011000 \vee 0000100 = 1111100$.

$$\begin{array}{rr} & 0011000 \\ \wedge & 1111100 \\ \hline & 0011000 \end{array}$$

The 3rd and 4th tuples are the answer.

Compression of bitmaps

Suppose we have a bitmap index on field *F* of a file with *n* records, and there are *m* different values for field *F* that appear in the file.



Compression of bitmaps

Run-length encoding:

Run in a bit vector: a sequence of i 0's followed by a 1.

000000010001 ⟵ This bit vector contains two runs.

Run compression: a run r is represented as another bit string r' composed of two parts.

part 1: i expressed as a binary number, denoted as $b_1(i)$.
part 2: Assume that $b_1(i)$ is j bits long. Then, part 2 is a sequence of $(j - 1)$ 1's followed by a 0, denoted as $b_2(i)$.

$r' = b_2(i)b_1(i)$.

Compression of bitmaps

Run-length encoding:

Run in a bit vector s: a sequence of i 0's followed by a 1.

000000010001 ← This bit vector contains two runs.

$r' = b_2(i)b_1(i)$.

$r_1 = 00000001$

$b_{11} = 7 = 111$, $b_{12} = 110$ ⇒ $r_1' = 110111$

$r_2 = 0001$

$b_{11} = 3 = 11$, $b_{12} = 10$ ⇒ $r_2' = 1011$

000000010001

$r_1'r_2' = 1101111011$

Starting at the beginning, find the first 0 at the 3rd bit, so $j = 3$. The next 3 bits are 111, so we determine that the first integer is 7. In the same way, we can decode1011.

Decoding a compressed sequence *s*:

1. Scan *s* from the beginning to find the first 0.
2. Let the first 0 appears at position *j*. Check the next *j* bits. The corresponding value is a run.
3. Remove all these bits from *s*. Go to (1).

Uncompression:

$r_1'r_2' = 1101111011$

$r_1 = 00000001$

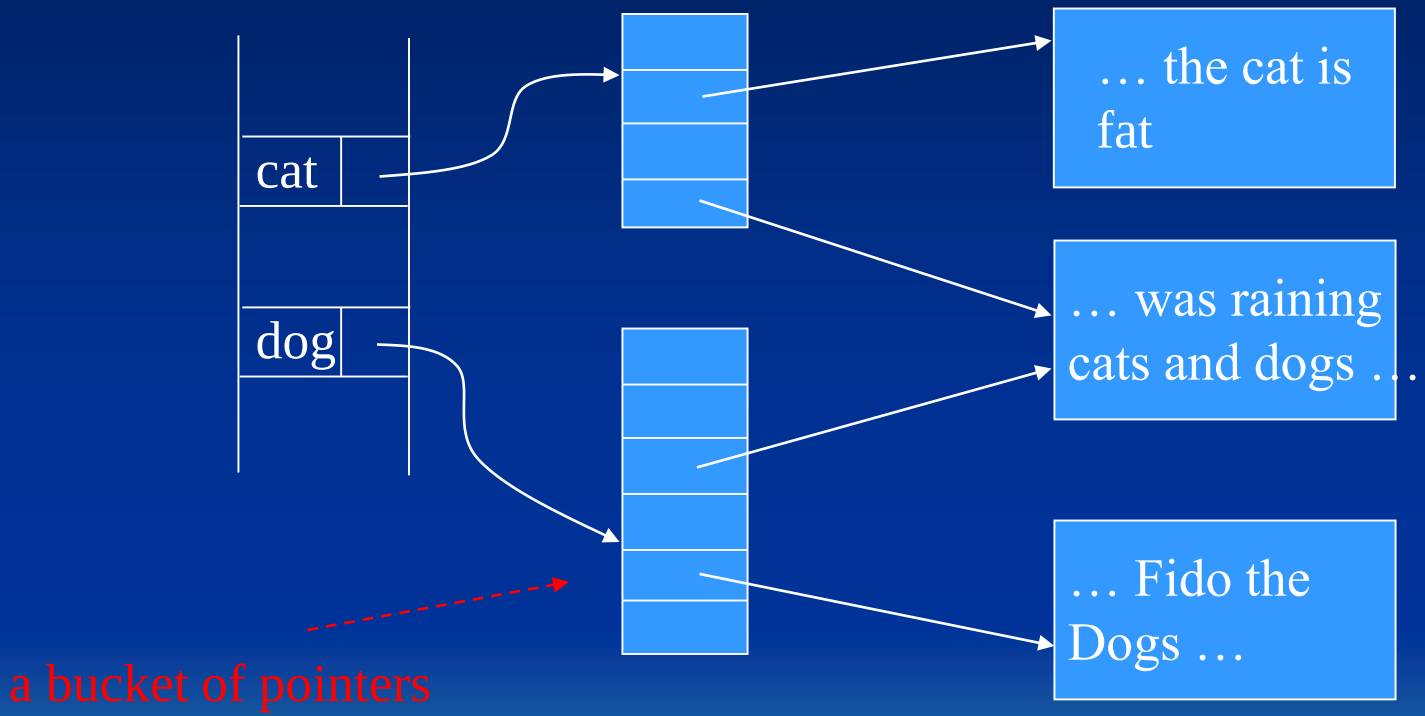
$r_2' = 1011$

$r_2 = 0001$

$r_1r_2 = 000000010001$

Inverted files

An inverted file - A list of pairs of the form: <key word, pointer>

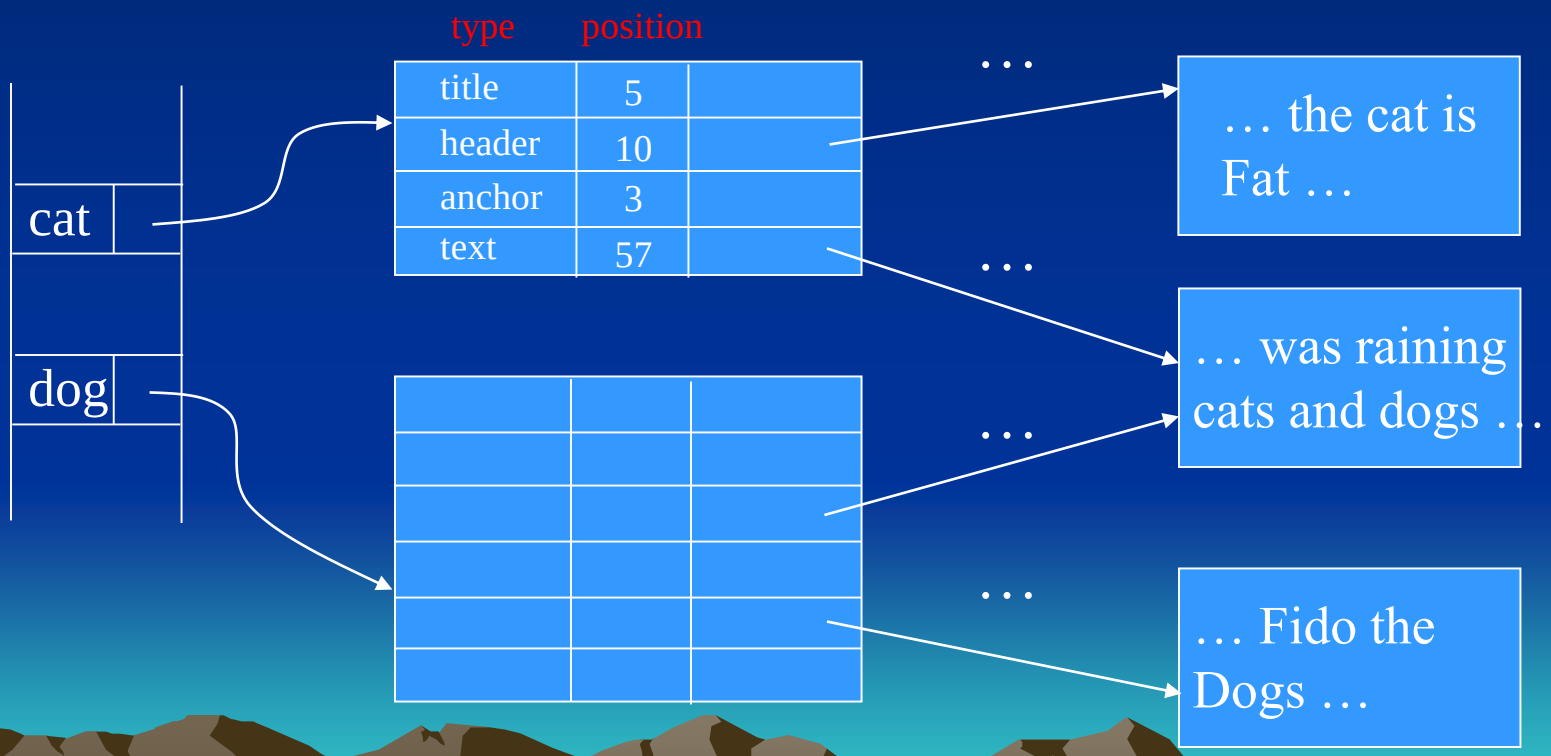


$L(\text{cat}) = \{1, 3, 5\}$ $L(\text{dog}) = \{3, 5, 8, 9\}$

$L(\text{cat} \wedge \text{dog}) = \{1, 3, 5\} \cap \{3, 5, 8, 9\} = \{3, 5\}$

Inverted files

When we use “buckets” of pointers to occurrences of each word, we may extend the idea to include in the bucket array some information about each occurrence.



Web Databases

Not included in the mid-term

- Web database
- System architecture
- Web programming language:
  - PHP
  - Node.js

- What is a web database?
  - A database accessed from the Internet
  - E-commence and other Internet applications are designed to interact with the user through *web interfaces*
  - An online flight ticket booking system

    web interface:

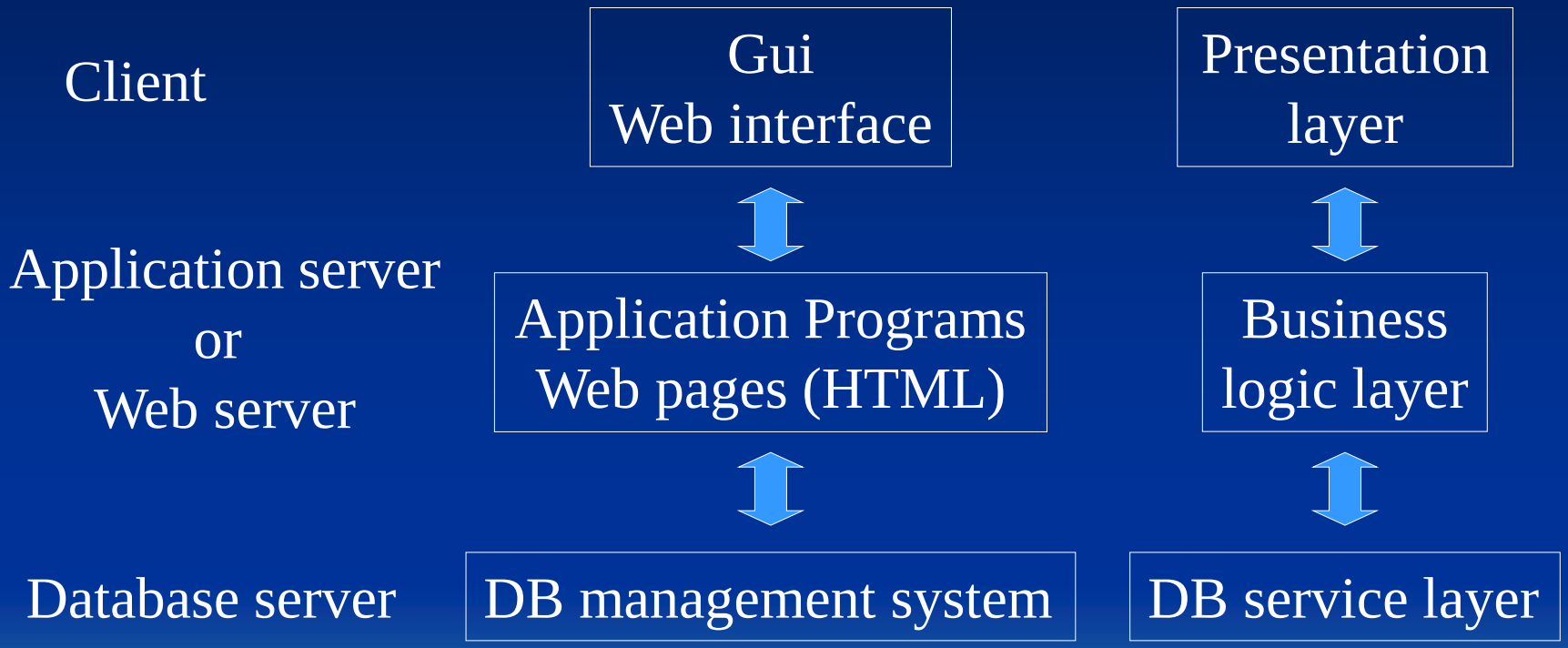
    input - customer information: time, location, airport, destination

    output – departure time, arrival time, flight number, price

    database access:

    query evaluation

- Three-tier architecture:



- **Web server language (script language): PHP**

  - PHP – a script language, used to generate
    dynamic HTML pages.
    PHP programs are executed on Web server computers.
    (This is in contrast to some scripting languages, such as JavaScript, which are executed on client computers.)

  - The official PHP website has installation instructions for PHP: http://PHP.org.net

  - PHP 5 and later versions can work with a MySQL database using:
    - MySQLi extension (the 'i' stands for improved)
    - PDO (PHP Data Objects)

- **A simple PHP example**
  - The program prompts a user to enter the first and last name and then prints a welcome message to that user.

```
<?PHP
    //Printing a welcome message if the user submitted his/her name
    //through the PHP form
    if ($_post['user_name']) {
            print("Welcome, ");
            print($_post['user_name']); }
    else { print <<<_HTML_
            <FORM method="post" action="$_SERVER['PHP_SELF']">
            Enter your name: <input type="text" name="user_name">
            <BR/>
            <INPUT type="submit" value="SUBMIT NAME"></FORM>
             _HTML_;
            }
?>
```

Enter your name:

SUBMIT NAME

Enter your name John Smith

SUBMIT NAME

Welcome, John Smith

- A PHP script is enclosed with a pair of tags:
  start tag: <?php
  end tag: ?>
  Stored in a file, named, for example,
  greeting.php

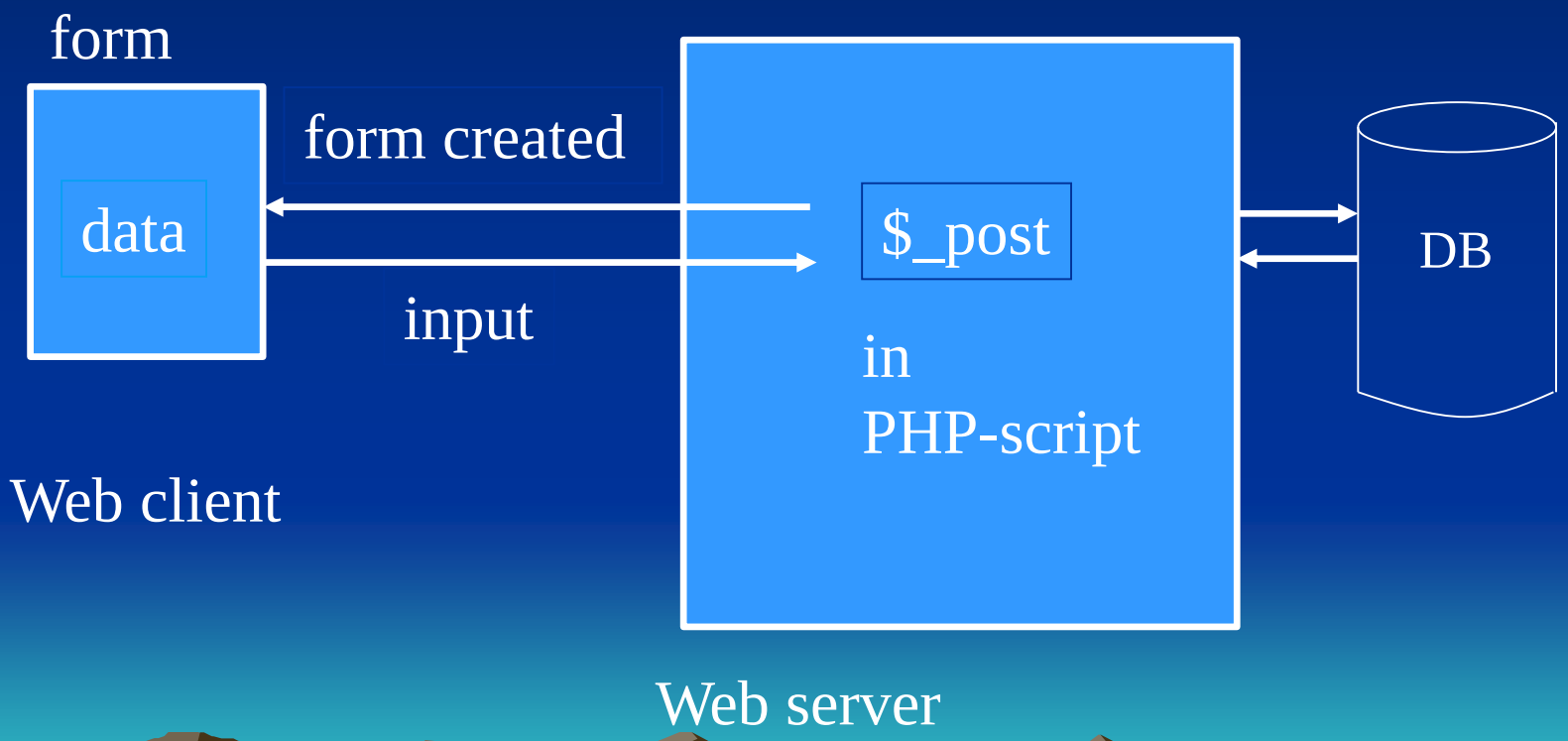
  and located in an address, for example,

  http://www.myserver.com/examples/greeting.php

- You can also put it is a HTML file.

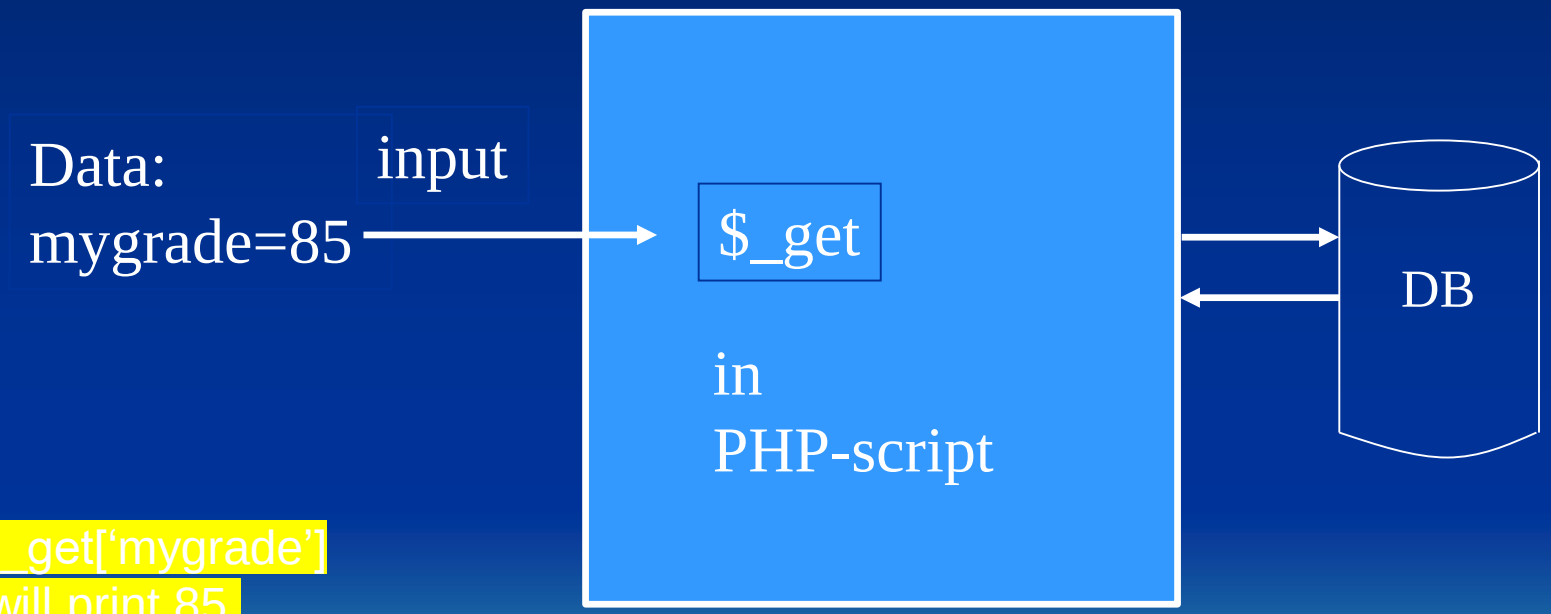
post:

http://www.myserver.com/examples/greeting.php



get:

http://www.myserver.com/examples/another.php?mygrade=85




```
echo $_get['mygrade']
//This will print 85.
```

- **Connecting to a database**

```
require ‘DB.php’
$d = DB::connect{‘mysqli://acct1:pass12@www.host.com/db1’};
if (DB:isError($d)){die(“cannot connect …”, $d->getMessage());}
    …
$q = $d->query(“CREATE TABLE EMPLOYEE
    (       Emp_id INT,
            Name VARCHAR(15),
            Job VARCHAR(10),
            Dno INT)”    );
if (DB:isError($q)) {die(“table creation not successful …”, $d->getMessage());
    …
$d ->setErrorHandling( );
    …          <----------------------
$eid = $d->nextID(‘EMPLOYEE’);
$q = $d->query(“INSERT INTO EMPLOYEE VALUES
    ($eid,$_post[‘emp_name’], $_post[‘emp_job’], $_post[‘emp_dno’])”);
```

A form should be displayed here to receive input data.

What is Node.js?

- Node.js is a script language
- Node.js is an open source server environment
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the serve

myfirst.js

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!'); //write a response and then
                                    //end the response
}).listen(8080);
```

Save the file on your computer:

C:\Users*Your Name*\myfirst.js

The code tells the computer to write "Hello World!" if anyone (e.g. a web browser) tries to access your computer on port 8080.

- **Command Line Interface**

  -Node.js files must be initiated in the "Command Line Interface" program of your computer.

  -Navigate to the folder that contains the file "myfirst.js", the command line interface window should look something like this:

  C:\Users*Your Name*>_

  C:\Users*Your Name*>node myfirst.js

- **Execution of myfirst.js**

  -Now, your computer works as a server!
  -If anyone tries to access your computer on port 8080, they will get a "Hello World!" message in return!
  -Start your internet browser, and type in address:

  http://localhost:8080

- MySQL databases in a web server
  - You can download a free MySQL database at http://www.mysql.com/downloads/
  - Install MySQL Driver
- Once you have MySQL up and running on your computer, you can access it by using Node.js.
- To access a MySQL database with Node.js, you need a MySQL driver.
- Install MySQL from nmp.

- To download and install the "mysql" module, open the Command Terminal and execute the following:

  C:\Users*Your Name*>npm install mysql

npm - a package manager for installing Node.js packages.

- Create Connection

demo_db_connection.js

```
var mysql = require('mysql');
var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword"
});
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
});
```

C:\Users*Your Name*>node demo_db_connection.js

- Creating a Database
  - Create a database named "mydb”

```
var mysql = require('mysql');
var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword"
});
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  con.query("CREATE DATABASE mydb", function (err, result) {if (err) throw err;
    console.log("Database created"); }); });
```

Save the code above in a file called "demo_create_db.js“

C:\Users*Your Name*>node demo_create_db.js

- Creating a table
  - Create a table named “customers”

```
var mysql = require('mysql');
var con = mysql.createConnection({
  host: "localhost", user: "yourusername“,
  password: "yourpassword", database: "mydb"});
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  var sql = "CREATE TABLE customers (name
  VARCHAR(255), address VARCHAR(255))";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("Table created");});});
```

```
var mysql = require('mysql');
var con = mysql.createConnection({
  host: "localhost",
  user: "yourusername",
  password: "yourpassword",
  database: "mydb"
});
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  var sql = "INSERT INTO customers (name, address)
            VALUES ('Company Inc', 'Highway 37')";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("1 record inserted");
  });
}).listen(8080);
```

- Query a Database

  - Use SQL statements to read from (or write to) a MySQL database

```
… …
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  database: “mydb”
  var sql = “select * from customers where  name = 'David'";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("Result: " + result);
  });
});
```

Semistructured-Data Model

- Semistructured data
- XML
- DTD (Document type definitions)
- XML schema

Semistructured Data

The semistructured-data model plays a special role in database systems:

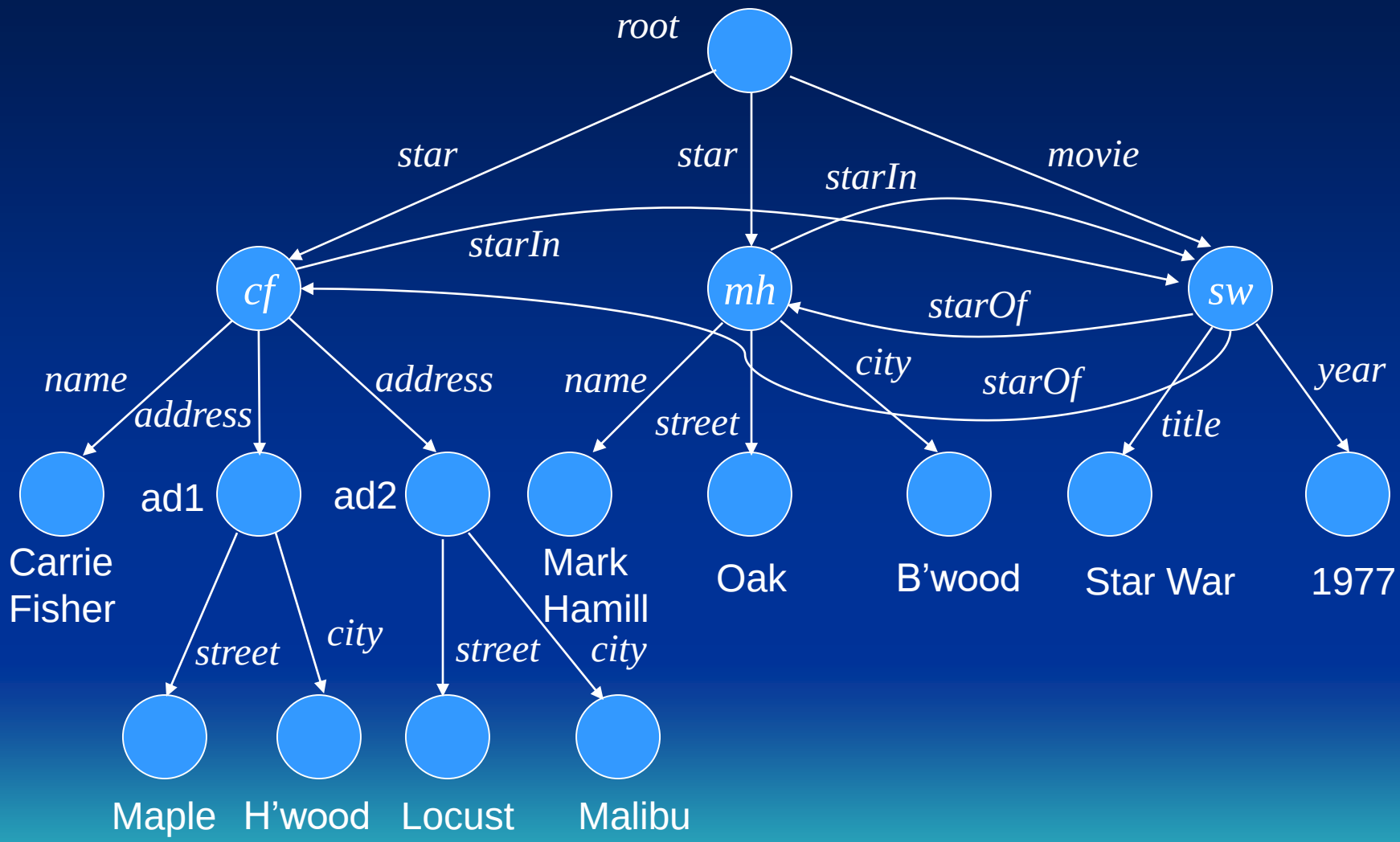
1. It serves as a model suitable for integration of databases, i.e., for describing the data contained in two or more databases that contain similar data with different schemas.
2. It serves as the underlying model for notations such as XML that are being used to share information on the web.

The semistructured data model can represent information more flexibly than the other models – E-R, UML, relational model, ODL (Object Definition Language).

Semistructured Data representation

A database of semistructured data is a collection of nodes.

- Each node is either a leaf or interior
- Leaf nodes have associated data; the type of this data can be any atomic type, such as numbers and strings.
- Interior nodes have one or more arcs out. Each arc has a label, which indicates how the node at the head of the arc relates to the node at the tail.
- One interior node, called the root, has no arcs entering and represents the entire database.


root
star
star
starIn
movie
starIn
cf
mh
sw
starOf
name
address
address
name
city
starOf
year
street
title
ad1
ad2
Carrie Fisher
Mark Hamill
Oak
B'wood
Star War
1977
street
city
street
city
Maple
H'wood
Locust
Malibu

Semistructured Data representation

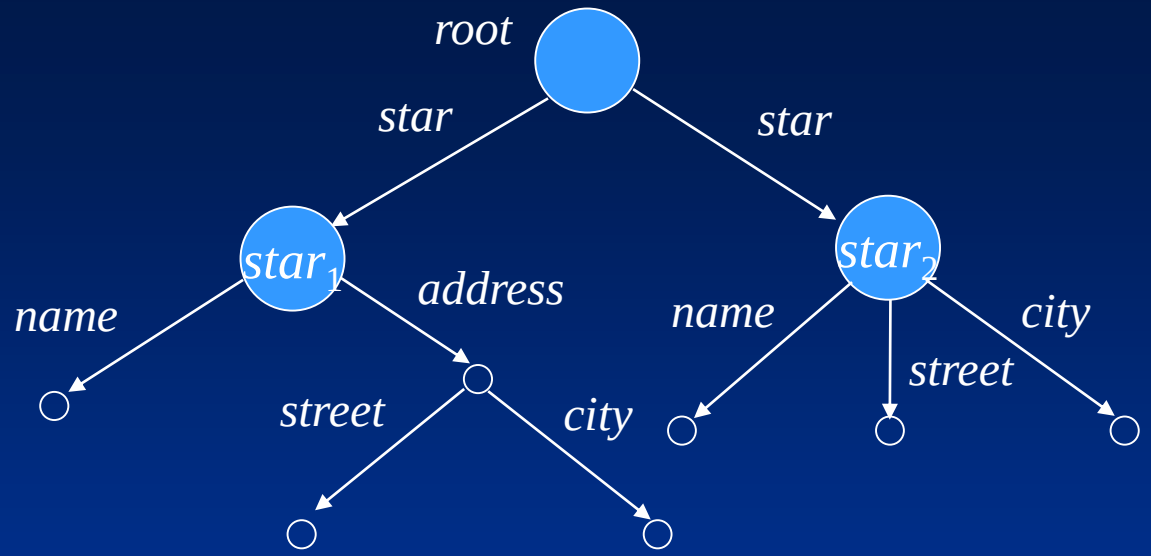
A label *L* on the arc from node *N* to node *M* can play one of two roles.

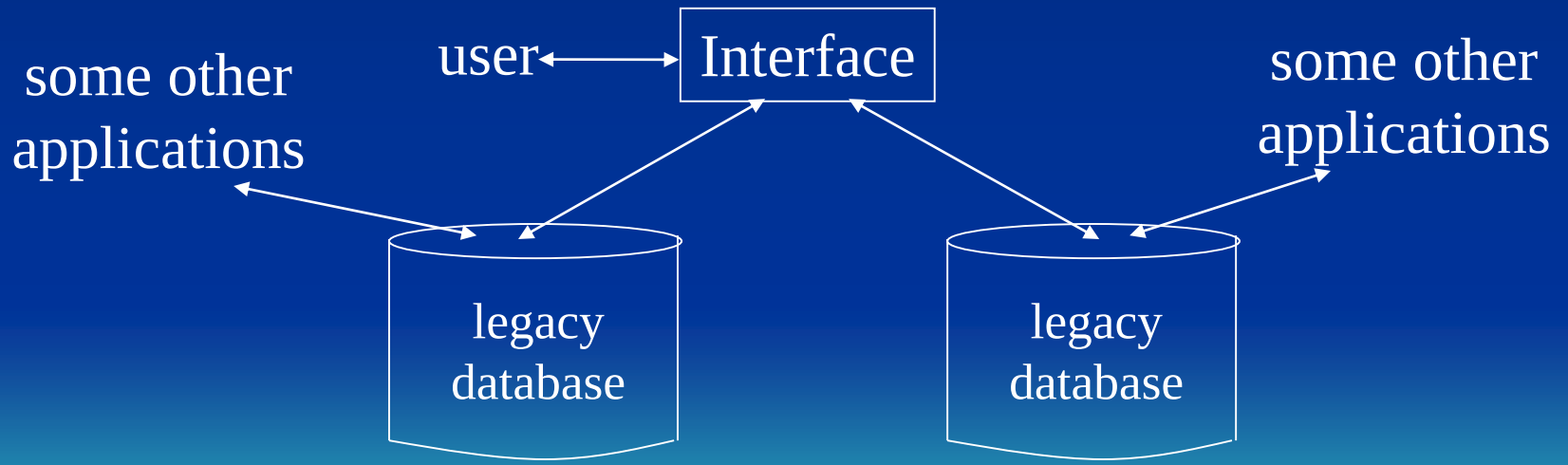
1. It may be possible to think of *N* as representing an object or entity, while *M* represents one of its attributes. Then, *L* represents the name of the attribute.
2. We may be able to think of *N* and *M* as objects or entities and *L* as the name of a relationship from *N* to *M*.

Semistructured Data model can be used to integrate information

Legacy-database problem: Databases tend over time to be used in so many different applications that it is impossible to turn them off and copy or translate their data into another database, even if we could figure out an efficient way to transform the data from one schema to another.

In this case, we will define a semistructured data model over all the legacy databases, working as an interface for users. Then, any query submitted against the interface will be translated according to local schemas.





Stars(name, address(street, city))

Stars(name, street, city)

XML (*Extensible Markup Language*)

XML is a tag-based notation designed originally for *marking* documents, much like HTML. While HTML's tags talk about the presentation of the information contained in documents – for instance, which portion is to be displayed in italics or what the entries of a list are – XML tags intended to talk about the meanings of pieces of the document.

Tags:

opening tag - < …. >, e.g., <Foo>
closing tag - </ … >, e.g., </Foo>

A pair of matching tags and everything that comes between them is called an *element*.

XML with and without a schema

XML is designed to be used in two somewhat different modes:

1. *Well-formed* XML allows you to invent your own tags, much like the arc-labels in semistructured data. But there is no predefined schema. However, the nesting rule for tags must be obeyed, or the document is not well-formed.
2. *Valid* XML involves a DTD (Document Type Definition) that specifies the allowed tags and gives a grammar for how they may be nested. This form of XML is intermediate between the strict-schema such as the relational model, and the completely schemaless world of semistructured data.

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<StarMovieData>
    <Star>
        <Name>Carrie Fishes</Name>
        <Address>
            <Street>123 Maple St.</Street><City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street><City>Malibu</City>
        <Address>
    </Star>
    <Star>
        <Name>Mark Hamill</Name><Street>456 Oak Rd.</Street>
        <City>Brentwood</City>
    </Star>
    <Movie>
        <Title>Star Wars</title><Year>1977</Year>
    </Movie>
</StarMovieData>
```

prologue

Attributes

As in HTML, an XML element can have attributes (name-value pairs) with its opening tag. An attribute is an alternative way to represent a leaf node of semistructured data. Attributes, like tags, can represent labeled arcs in a semisructured-data graph.

<Movie>
  <Title>“Star Wars”</title>
  <Year>1977</Year>
</Movie>

<Movie year = 1977>
  <Title>“Star Wars”</title>
</Movie>

Attributes that connect elements

An important use for attributes is to represent connections in a semistructured data graph that do not form a tree.

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<StarMovieData>
    <Star starID = “cf” starredIn = “sw”>
            … …
    </Star>
    <Star starID = “mh” starredIn = “sw”>
            …  …
    </Star>
    <Movie movieID = “sw” starsOf = “cf”, “mh”>
        <Title>Star Wars</title><Year>1977</Year>
    </Movie>
</StarMovieData>
```

Namespace

There are situations in which XML data involves tags that come from two or more different sources. So we may have conflicting names. For example, we would not want to confuse an HTML tag used in a text with an XML tag that represents the meaning of that text. To distinguish among different vocabularies for tags in the same document, we can use a *namespace* for a set of tags.

To indicate that an element's tag should be interpreted as part of a certain space, we use the attribute xmlns in its opening tag:

xmlns: name = <Universal Resource Identifier>

Example:

<md : StarMoviedata xmlns : md = http://infolab.stanford.edu/movies>

XML storage

There are three approaches to storing XML to provide some efficiency:

1. Store the XML data in a parsed form, and provide a library of tools to navigate the data in that form. Two common standards are called SAX (Simple API for XML), and DOM (Document Object Model).
2. MongoDB – non-tabular databases

In Mongo DB, a document is stored as a set of property-value pairs (JSON format).

```
[   { title : “post1”,
      body: “body of post 1”,
      category: “news”,
      time: Date( )
      }
      { title : “post2”,
      body: “body of post 2”,
      category: “events”,
      time: Date( )
      }   ]
```

3. Represent the document and their elements as relations, and use a conventional, relational DBMS to store them.

In order to represent XML documents as relations, we should give each document and each element of a document a unique ID. For each document, the ID could be its URL or its path in a file system.

A possible relational database schema:

DocRoot(docID, rootElmentID)
ElementValue(elementID, value)
SubElement(parentID, childID, position)
ElementAttribute(elementID, name, value)

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
< md : StarMovieData xmlns : md = http://infolab.stanford.edu/movies >
    <Star starID = “cf” starredIn = “sw”>
        <Name>Carrie Fishes</Name>
        <Address>
            <Street>123 Maple St.</Street><City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street><City>Malibu</City>
        <Address>
    </Star>
    <Star starID = “mh” starredIn = “sw”>
        <Name>Mark Hamill</Name><Street>456 Oak Rd.</Street>
        <City>Brentwood</City>
    </Star>
    <Movie movieID = “sw” starsOf = “cf”, “mh”>
        <Title>Star Wars</title><Year>1977</Year>
    </Movie>
</StarMovieData>
```

DocRoot

Doc-id	rootElementID
1	1

elementValue

Doc-id	element-id	value
1	1	starMovieData
1	2	Star
1	3	Star
1	4	movie
…	…	…

subElement

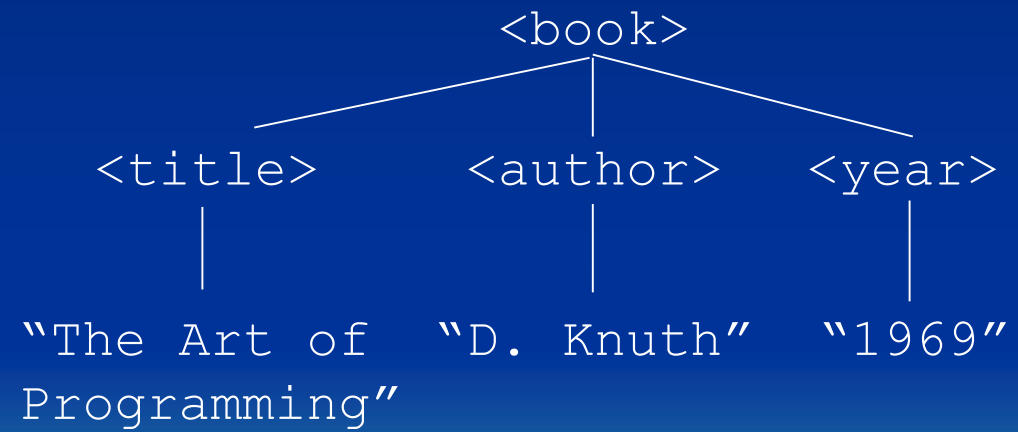
parentId	childId	position
1.1	1.2	1
1.1	1.3	2
1.1	1.4	4
… …	… …	… …

elementAttribute

elemenAttId	attName	value
1.1	xmlns : md	http://... …
1.2	starId	“mf”
1.2	starId	“mh”
1.3	starredIn	“sw”
1.3	starredIn	“sw”
1.4	movieId	“sw”
1.4	starsOf	“sf”, “mh”

Transform an XML document to a tree

```
<book>
     <title>
        "The Art of Programming"
     </title>
      <author>
         "D. Knuth"
      </author>
      <year>
         "1969"
      </year>
</book>
```



Transform an XML document to a tree

Read a file into a character array A:

<	b	o	o	k	>	<	t	i	t	l	e	>	"	T	h	e		A	r	t	…

stack S:

Transform an XML document to a tree

```
Algorithm:
Scan array A; Let A[i] be the character currently
encountered;
If A[i] is '<' and A[i+1] is a character then {
      generate a node x for A[i..j],
      where A[j] is '>' directly after A[i];
      let y = S.top().pointer_to_node;
      make x be a child of y; S.push(A[i..j], x);
If A[i] is ' " ', then {
      genearte a node x for A[i..j],
      where A[j] is ' " ' directly after A[i];
      let y = S.top().pointer_to_node;
      make x be a child of y;
If A[i] is '<' and A[i+1] is '/',
then S.pop();
```

Generating a node for an opening tag.

Generating a leaf node for a string value.

Popping out the stack when meeting a closing tag.

Document Type Definition (*DTD*)

A DTD is a set of grammar-like rules to indicate how elements can be nested.

DTD general form:

```
<!DOCTYPE root-tag [
    <!ELEMENT element-name (components)>
        … …
    ]>
```

Stars.dtd

```
<!DOCTYPE Stars [
    <!ELEMENT Stars (Star*)>
    <!ELEMENT Star (Name, Address+, Movies)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Address (Street, City)>
    <!ELEMENT Street (#PCDATA)>
    <!ELEMENT City (#PCDATA)>
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie (Title, Year)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

```
<Stars>
    <Star>
        <Name>Carrie Fishes</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Movies>
            <Movie>
                <Title>Star Wars</Title>
                <Year>1977</Year>
            </Movie>
            <Movie>
                <Title>Empire Striker</Title>
                <Year>1980</Year>
            </Movie>
            <Movie>
                <Title>Return of the Jedi</Title><Year>1983</Year>
            </Movie>
        </Movies>
    </Star>
```

```
<!DOCTYPE Stars [
    <!ELEMENT Stars (Star*)>
    <!ELEMENT Star (Name, Address+, Movies)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Address (Street, City)>
    <!ELEMENT Street (#PCDATA)>
    <!ELEMENT City (#PCDATA)>
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie (Title, Year)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

```
<Star>
    <Name>Mark Hamill</Name>
    <Address>
        <Street>456 Oak Rd.</Street>
        <City>Brentwood</City>
    </Address>
    <Movies>
        <Movie>
            <Title>Star Wars</Title>
            <Year>1977</Year>
        </Movie>
        <Movie>
            <Title>Empire Wars</Title>
            <Year>1980</Year>
        </Movie>
        <Movie>
            <Title>Return of the Jedi</Title>
            <Year>1983</Year>
        </Movie>
    </Movie>
</Star>
</Stars>
```

```
<!DOCTYPE Stars [
    <!ELEMENT Stars (Star*)>
    <!ELEMENT Star (Name, Address+, Movies)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Address (Street, City)>
    <!ELEMENT Street (#PCDATA)>
    <!ELEMENT City (#PCDATA)>
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie (Title, Year)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

```
<!DOCTYPE Stars [
    <!ELEMENT Stars (Star*)>
    <!ELEMENT Star (Name, Address+, Movies)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Address (Street, City)>
    <!ELEMENT Street (#PCDATA)>
    <!ELEMENT City (#PCDATA)>
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie (Title, Year)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<Stars>
    <Star>
        <Name>Carrie Fishes</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Malibu</City>
        <Address>
    </Star>
    <Star>
        <Name>Mark Hamill</Nam>
        <Street>456 Oak Rd.</Street>
        <City>Brentwood</City>
    </Star>
    <Movie>
        <Title>Star Wars</title><Year>1977</Year>
    </Movie>
</Stars>
```

This document does not confirm to the DTD. ⟶

Using a DTD

If a document is intended to conform to a certain DTD, we

a) Include the DTD itself as a preamble to the document, or

b) In the opening line, refer to the DTD, which must be stored separately in the file system accessible to the application that is processing the document.

<?xml version = “1.0” encoding = “utf-8” standalone = “no”?>

<!DOCTYPE Star SYSTEM “star.dtd”>

SYSTEM – keyword indicating that the DTD can be find in file star.dtd (this can also be a valid URL if the .dtd file is remote.)

```
<?xml version="1.0" ?>
 <!DOCTYPE r [
<!ELEMENT r ANY >
<!ELEMENT a ANY >
<!ELEMENT b ANY >
<!ELEMENT c (a*)>
<!ELEMENT d (b*)>
]>
<r>
        <a>
                <b>
                        <a></a><a></a><b></b>
                </b>
                <c>
                        <a>
                                <b></b>
                        </a>
                </c>
                <a>
                        <a></a><b></b>
                </a>
        </a>
</r>
```

<---------- A DTD is included as a preamble.

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no" ?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
        <name>Tanmay Patil</name>
        <company>TutorialsPoint</company>
        <phone>(011) 123-4567</phone>
</address>
```

Attribute Lists

An element may be associated with an attribute list:

<!ATTLIST *element-name attribute-name type*>

```
<!ELEMENT Movie EMPTY>
    <!ATTLIST       Movie
        title    CDATA        #REQUIRED
        year     CDATA        #REQUIRED
        genre    (comedy | drama | sciFi | teen) #IMPLIED
    >
```

<Movie title = “Star Wars” year = “1977” genre = “sciFi”/>

```
<!DOCTYPE StarMovieData [
   <!ELEMENT StarMovieData          (Star*, Movie*)>
   <!ELEMENT Star                   (Name, Address+)>
      <!ATTLIST Star
         starId            ID       #REQUIRED
         StarredIn         IDREFS   #IMPLIED
      >
   <!ELEMENT Name                   (#PCDATA)>
   <!ELEMENT Address                (Street, City)>
   <!ELEMNT Street                  (#PCDATA)>
   <!ELEMENT City                   (#PCDATA)>
   <!ELEMENT Movie                  (Title, Year)>
      <!ATTLIST      Movie
         movieId           ID       #REQUIRED
         startOf           IDREFS   #REQUIRED
      >
   <!ELEMENT Title                  (#PCDATA)>
   <!ELEMENT Year                   (#PCDATA)>
]>
```

Identifiers and Reference

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
    <Star starID = "cf" starredIn = "sw">
        <Name>Carrie Fishes</Name>
        <Address>
            <Street>123 Maple St.</Street><City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street><City>Malibu</City>
        <Address>
    </Star>
    <Star starID = "mh" starredIn = "sw">
        <Name>Mark Hamill</Name>
        <address>
            <Street>456 Oak Rd.</Street>
            <City>Brentwood</City>
        </address>
    </Star>
    <Movie movieID = "sw" starOf = "cf mh">
        <Title>Star Wars</title><Year>1977</Yea
    </Movie>
</StarMovieData>
```

```
<!DOCTYPE StarMovieData [
    <!ELEMENT StarMovieData          (Star*, Movie*)>
        <!ELEMENT Star               (Name, Address+)>
            <!ATTLIST Star
                starId      ID       #REQUIRED
                StarredIn   INREFS   #IMPLIED
            >
    <!ELEMENT Name                   (#PCDATA)>
    <ELEMENT Address                 (Street, City)>
    <!ELEMNT Street                  (#PCDATA)>
    <!ELEMENT City                   (#PCDATA)>
    <!ELEMENT Movie                  (Title, Year)>
        <!ATTLIST Movie
                movieIn              ID        #REQUIRED
                startOf              IDREFS    #REQUIRED
            >
    <!ELEMENT Title  (#PCDATA)>
    <!ELEMENT Year   (#PCDATA)>
]>
```

XML Schema

XML Schema is an alternative way to provide a schema for XML documents.

More powerful – give the schema designer extra capabilities.

- allow us to declare types, such as integers or float for simple elements.
- allow arbitrary restriction on the number of occurrences of subelements.
- give us the ability to declare keys and foreign keys.

The Form of an XML schema

- An XML schema description of a schema is itself an XML document. It uses the namespace at the URL

  http://www.w3.org/2001/XMLSchema

  that is provided by the World-Wide-Web Consortium.

•Each XML-schema document has the form:

```
<? xml version = '1.0" encoding = "utf-8" ?>
<xs: schema xmlns: xs = "http://www.w3.org/2001/
XMLSchema">
    ... ...
</xs: schema>
```

Elements

An important component in an XML schema is the element, which is similar to an element definition in a DTD.

The form of an element definition in XML schema is:

```
<xs: element name = element name type = element type>
      constraints and/or structure information
</xs: element>
```

<xs: element name = “Title” type = “xs: string” />
<xs: element name = “Year” type = “xs: integer” />

DTD

```
<!DOCTYPE root-tag [
        … …
    <!ELEMENT Title (#PCDATA)>
     <!ELEMENT Year (#PCDATA)>
        … …
    ]>
```

Complex Types

A *complex type* in XML Schema can have several forms, but the most common is a sequence of elements.

```
<xs: complexType name = type name >
      <xs: sequence>
            list of element definitions
      </xs: sequence>
</xs: complexType>
```

```
<xs: complexType name = type name >
            list of attribute definitions
      </xs: complexType>
```

DTD

```
<!DOCTYPE root-tag [
      <!ELEMENT element-name (components)>
            ... ...
      ]>
```

```
<? Xml version = “1.0” encoding = “utf-8” ?>
<xs: schema xmlns: xs = “http://www.w3.org/2001/XMLSchema”>

    <xs:complexType name = “movieType”>
        <xs: sequence>
            <xs: element name = “Title” type = “xs: string” />
            </xs: element name = “Year” type = “xs: integer” />
        </xs: sequence>
    </xs: complexType>

    <xs: element name = “Movies”>
        <xs: complexTyp>
            <xs: sequence>
                <xs: element name = “Movie” type = “movieType””
                    minOccurs = “0” maxOcurs = “unbounded” />
            </xs: sequence>
        </xs: complexTyp>
    </xs: element>
</xs: schema>
```

```
<xs: complexType name = type name >
    <xs: sequence>
        list of element definitions
    </xs: sequence>
</xs: complexType>
```

A schema for movies in XML schema. Itself is a document.

The above schema (in XML schema) is equivalent to the following DTD.

```
<!DOCTYPE Movies [
    <!ELEMENT     Movies (Movie*) >
    <!ELEMENT     Movie (Title, Year) >
    <!ELEMENT     Title (#PCDATA) >
    <!ELEMENT     Year (#PCDATA) >
]>
```

Attributes

A *complex type* can have *attributes*. That is, when we define a complex type *T*, we can include instances of element <xs: attribute>. Thus, when we use *T* as the type of an element *E* (in a document), then *E* can have (or must have) an instance of this attribute. The form of an attribute definition is:

<xs: attribute name = *attribute name* type = *type name*
other information about attribute />

<xs: attribute name = “title” type = “xs: integer” default = “0” />
<xs: attribute name = “year” type = “xs: integer” use = “required” />

```
<? Xml version = “1.0” encoding = “utf-8” ?>
<xs: schema xmlns: xs = “http://www.w3.org/2001/XMLSchema”>

    <xs: complexType name = “movieType”>
        <xs: attribute name = “title” type = “xs: string” use = “required” />
        <xs: attribute name = “year” type = “xs: integer” use = “required” />
    </xs: complexType>

    <xs: element name = “Movies”>
        <xs: complexTyp>
            <xs: sequence>
                <xs: element name = “Movie” type = “movieType”
                    minOccurs = “0” maxOcurs = “unbounded” />
            </xs: sequence>
        </xs: complexTyp>
    </xs: element>
</xs: schema>
```

```
<xs:complexType name = “movieType”>
    <xs: sequence>
        <xs: element name = “Title” type = “xs: string” />
        </xs: element name = “Year” type = “xs: integer” />
    </xs: sequence>
</xs: complexType>
```

A schema for movies in XML schema. Itself is a document.

The above schema (in XML schema) is equivalent to the following DTD.

```
<!DOCTYPE Movies [
    <!ELEMENT      Movies (Movie*) >
    <!ELEMENT      Movie EMPTY >
        <!ATTLIST  Movie
            Title      CDATA #REQUIRED
            Year       CDATA #REQUIRED
        >
]>
```

```
<!DOCTYPE Movies [
      <!ELEMENT      Movies (Movie*) >
      <!ELEMENT      Movie (Title, Year) >
      <!ELEMENT      Title (#PCDATA) >
      <!ELEMENT      Year (#PCDATA) >
]>
```

Restricted Simple Types

It is possible to create a restricted version of a simple type such as integer or string by limiting the values the type can take. These types can then be used as the type of an attribute or element.

1. Restricting numerical values by using minInclusive to state the lower bound, maxInclusive to state the upper bound.
2. Restricting values to an numerated type.

```
<xs: simpleType name = type name >
    <xs: restriction base = base type >
        upper and/or lower bounds
    </xs: restriction>
</xs: simpleType>
```

```
<xs: enumeration value = some value />
```

```
<xs: simpleType name = "movieYearType" >
    <xs: restriction base = "xs: integer" >
        <xs:minInclusive value = "1915" />
    </xs: restriction>
</xs: simpleType>
```

```
<xs: simpleType name = "genretype" >
    <xs: restriction base = "xs: string" >
        <xs: enumeration value = "comedy" />
        <xs: enumeration value = "drama" />
        <xs: enumeration value = "sciFi" />
        <xs: enumeration value = "teen" />
    </xs: restriction>
</xs: simpleType>
```

Keys in XML Schema

An element can have a *key declaration*, which is a field or several fields to uniquely identify the element among a certain class *C* of elements).

field: an attribute or a subelement.
selector: a path to reach a certain node in a document tree.

```
<xs: key name = key name >
      <xs: selector xpath = path description >
      <xs: field xpath = path description >
            more field specification
</xs: key>
```

```
Create table EMPLOYEE
            (…,
             DNO INT NOT NULL DEFAULT 1,
   CONSTRAINT EMPPK
            PRIMARY KEY(SSN),
   CONSTRAINT EMPSUPERFK
   FOREIGN KEY(SUPERSSN)
   REFERENCES
                        EMPLOYEE(SSN)
            ON DELETE SET NULL  ON
   UPDATE
                        CASCADE,
   CONSTRAINT EMPDEPTFK
   FOREIGN KEY(DNO) REFERENCES

            DEPARTMENT(DNUMBER)
             ON DELETE SET DEFAULT
            ON UPDATE CASCADE);
```

```
<? Xml version = “1.0” encoding = “utf-8” ?>
<xs: schema xmlns: xs = “http://www.w3.org/2001/XMLSchema”>

  <xs: simpleType name = “genretype” >
    <xs: restriction base = “xs: string” >
      <xs: enumeration value = “comedy” />
      <xs: enumeration value = “drama” />
      <xs: enumeration value = “sciFi” />
      <xs: enumeration value = “teen” />
    </xs: restriction>
  </xs: simpleType>

  <xs: complexType name = “movieType”>
    <xs: attribute name = “title” type = “xs: string” />
    <xs: attribute name = “year” type = “xs: integer” />
    <xs: attribute name = “Genre” type = “genreType”
      minOccurs = “0” maxOccurs = “1” />
  </xs: complexType>
```

```
<xs: element name = “Movies”>
  <xs: complexTyp>
    <xs: sequence>
      <xs: element name = “Movie” type = “movieType”
      minOccurs = “0” maxOcurs = “unbounded” />
    </xs: sequence>
  </xs: complexTyp>
  <xs: key name = “movieKey”>
    <xs: selector xpath = “Movie” />
    <xs: field xpath = “@Title” />
    <xs: field xpath = “@Year” />
  </xs: key>
</xs: element>
</xs: schema>
```

/Movies/Movie

/Movies/Movie@Title

/Movies/Movie@Year

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
  <Movies>
      … …
    <Movie Title = “Star Wars” Year = 1977 Genre = “comedy” />
      … …
  </Movies>
```

Foreign Keys in XML Schema

We can declare that an element has, perhaps deeply nested within it, a field or fields that serve as a reference to the key for some other element. It is similar to what we get with ID's and IDREF's in DTD.

In DTD: untyped references
In XML schema: typed references

```
<xs: keyref name = foreign-key name
      refer = key name>
      <xs: selector xpath = path description >
      <xs: field xpath = path description >
            more field specification
</xs: keyref>
```

```
Create table EMPLOYEE
            (…,
             DNO INT NOT NULL DEFAULT 1,
    CONSTRAINT EMPPK
            PRIMARY KEY(SSN),
    CONSTRAINT EMPSUPERFK
    FOREIGN KEY(SUPERSSN)
    REFERENCES
                        EMPLOYEE(SSN)
            ON DELETE SET NULL  ON
    UPDATE
                        CASCADE,
    CONSTRAINT EMPDEPTFK
    FOREIGN KEY(DNO) REFERENCES
            DEPARTMENT(DNUMBER)
             ON DELETE SET DEFAULT
            ON UPDATE CASCADE);
```

```
<? Xml version = “1.0” encoding = “utf-8” ?>
<xs: schema xmlns: xs = “http://www.w3.org/2001/XMLSchema”>
<xs: element name = “Stars”>
  <xs: complxType>
    <xs: sequence>
      <xs: element name = “Star” minOccurs = “1” maxOccurs = “unbounded”>
        <xs: complexType>
          <xs: sequence>
            <xs: element name = “Name” type = “xs: string” />
            <xs: element name = “Address” type = “xs: string” />
            <xs: element name = “StarredIn” minOccurs = “0” maxOccurs = “1”>
              <xs: complexType>
                <xs: attribute name = “title” type = “xs: string” />
                <xs: attribute name = “year” type = “xs: integer” />
              </xs: complexType>
            </xs: element>
          </xs: sequence>
        </xs: complexType>
      </xs: element>
    </xs: sequence>
  </xs: complexType>
```

```
<xs: keyref name = "movieRef" refers = "movieKey">
    <xs: selector xpath = "Star/StarredIn" />
    <xs: field xpath = "@title" />
    <xs: field xpath = "@year" />
</xs: keyref>
</xs: element>
</xs: schema>
```

```
<? Xml version = "1.0" encoding = "utf-8" standalone =
"yes" ?>
  <Stars>
    <Star>
        <Name>Mark Hamill</Name>
        <Address>456 Oak Rd. Brentwood</Address>
        <StarredIn title = "star war" year = "1977"/>
    </Star>
    … …

  </Stars>
```

About usage of XML schema

```
<?xml version="1.0"?>
<note xmlns: xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi: schemaLocation = "https://www.w3schools.com/xml note.xsd">

  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

The following example is an XML Schema file called "note.xsd" that defines the elements of the above XML document ("note.xml"):

```
<?xml version="1.0"?>
<xs: schema xmlns: xs = "http://www.w3.org/2001/XMLSchema">
        <xs: element name = "note">
        <xs:complexType>
        <xs:sequence>
        <xs:element name = "to" type = "xs:string"/>
        <xs:element name = "from" type = "xs:string"/>
        <xs:element name = "heading" type = "xs:string"/>
        <xs:element name = "body" type = "xs:string"/>
        </xs:sequence>
         </xs:complexType>
        </xs:element>
</xs:schema>
```

Programming Languages for XML

- XPath
- XQuery
- Extensible StyleSheets Language (XSLT)

XPath

XPath is a simple language for describing sets of similar paths in a graph of semistrucured data.

The XPath Data Model

Sequence of items corresponds to a set of tuples in the relational algebra.

An item is either:

1. A value of primitive type: integer, real, boolean, or string.
2. A node (three kinds of nodes)

Three kinds of nodes:

(a) Documents. These are files containing an XML document, perhaps denoted by their local path name or URL.
(b) Elements. These are XML elements, including their opening tags, their matching closing tags if there is one, and everything in between (i.e., below them in the tree of semistructured data that an XML document represents).
(c) Attributes. These are found inside opening tags.

The items in a sequence needn't be all of the same type although often they will be.

A sequence of five items:

10
“ten”
10.0
<Number base = “8”>
<Digit>1</Digit>
<Digit>2</Digit>
</Number>
@val=“10”

Document Nodes

It is common to apply XPath to documents that are files. We can make a document node from a file by applying the function:

doc(*file name*)

The named file should be an XML document. We can name a file either by giving its local name or a URL if it is remote.

doc(*"movie.xml"*)
doc(*"/usr/slly/data/movies.xml"*)
doc(*"infolab.stanford.edu/~hector/movies.xml"*)

Path Expressions

An XPath expression starts at the root of a document and gives a sequence of tags and slashes (/).

doc(*file name*)/T_1/T_2/.../T_n

doc("*movie.xml*")/StarMoviedata/Star/Name

Evaluation of XPath expressions:

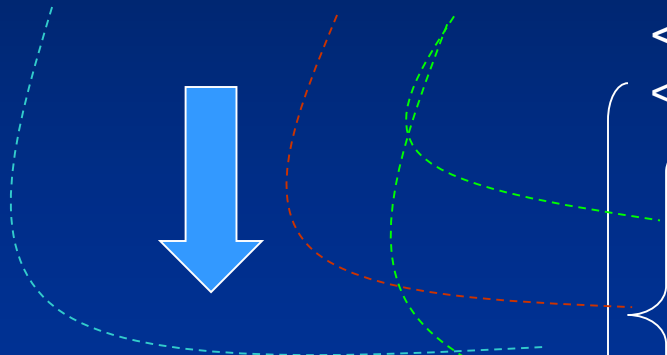
1. Start with a sequence of items consisting of one node: the document node.
2. Then, process each of T_1, T_2, …, T_n in turn.
3. To process T_i, consider the sequence of items that results from processing the previous tag, if any. Examine those items, in order, and find each of all its subelements whose tag is T_i.

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<StarMovieData>
    <Star starID = “cf” starredIn = “sw”>
        <Name>Carrie Fishes</Name>
        <Address>
            <Street>123 Maple St.</Street><City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street><City>Malibu</City>
        <Address>
    </Star>
    <Star starID = “mh” starredIn = “sw”>
        <Name>Mark Hamill</Name><Street>456 Oak Rd.</Street>
        <City>Brentwood</City>
    </Star>
    <Movie movieID = “sw” starOf = “cf mh”>
        <Title>Star Wars</title><Year>1977</Year>
    </Movie>
</StarMovieData>
```

doc(*“movie.xml”*)/StarMoviedata/Star/Name

In the following discussion, the document node is not included in an XPath for simplicity.

/StarMoviedata/Star/Name



```
<Name>Carrie Fisher</Name>
<Name>Mark Hamill</Name>
```

```
<? Xml version = "1.0" …  ?>
<StarMovieData>
    <Star starID = "cf" starredIn = "sw">
        <Name>Carrie Fishes</Name>
        … …
    </Star>
    <Star starID = "mh" starredIn = "sw">
        <Name>Mark Hamill</Name>
    </Star>
    <Movie>
        … ..
    </Movie>
</StarMovieData>
```

Relative Path Expressions

In several contexts, we shall use XPath expressions that are relative to the *current node* or sequence of nodes.

```
<xs: element name = "Movies">
   <xs: complexTyp>
      <xs: sequence>
         <xs: element name = "Movie" type = "movieType"
                      minOccurs = "0" maxOcurs = "unbouned" />
      </xs: sequence>
   </xs: complexTyp>
   <xs: key name = "movieKey">
      <xs: selector xpath = "Movie" />
      <xs: field xpath = "@Title" />
      <xs: field xpath = "@Year" />
   </xs: key>
</xs: element>
</xs: schema>
```

a current node

/StarMovieData/Movies

a relative path, equal to /StartMovieData/Movies/Movie

/StartMovieData/Movies/Movie/@Title

/StartMovieData/Movies/Movie/@Year

Attribute in Path Expressions

- Path expressions allow us to find all the elements within a document that are reached from the root along a particular path.

$$/T_1/T_2/.../T_n$$

- We can also end a path by an attribute name preceded by an *at-sign*.

$$/T_1/T_2/.../T_n/@A$$

```
/StarMovieData/Star/@starID
```

Axes

So far, we have only navigated though semistructured-data graphs in two ways: from a node to its children or to an attribute. In fact, XPath provides several axes to navigate a graph in different ways. Two of these axes are *child* (the default axis) and *attribute*, for which @ is really a shorthand.

Axes used in Xpath expressions: /axis::

Self	/self::	/next-sibling::
Parent	/parent::	/following::
descendant	/descendant::	/preceding::
Ancestor	/ancestor::	/child::
Next-sibling		/attribute::
Following		
Preceding		

/child::StarMovieData/descentend::Star/attribute::starID

self	Selects the current node
parent	Selects the parent of the current node
descendant	Selects all descendants (children, grandchildren, etc.) of the current node
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
next-sibling	Select the next sibling
following	Selects everything in the document after the closing tag of the current node
preceding	Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes
child	Selects all children of the current node
attribute	Selects all attributes of the current node

- All the children of the current node are referred to as siblings.
- All those nodes visited after the current node during a DFS search are referred as the following nodes.
- All those nodes visited before the current node during a DFS search are referred as the preceding nodes.

Abbreviated axes

/ - stands for *child*
@ – stands for *attribute*

. - stands for *self*
.. – stands for *parent*
// - stands for *descendant*

/child::StarMovieData/descentend::Star/attribute::starID

/StarMovieData//Star/@starID

/descendant::City

//City

/StarMovieData//Star//City
produces the same results as //City.

Context of Expression

- By “context”, we mean an element in a document, working as a reference point (current node).
- So it makes sense to apply axes like *parent*, *ancestor*, or *next-sibling* to a current node.

- Two functions: text( ), node( )
  - /child::text( ) – select all those children of the current node, which are text nodes
  - /child::node( ) – select all the children of the current node, whatever their node type
  - /self::node( ) – select the current node

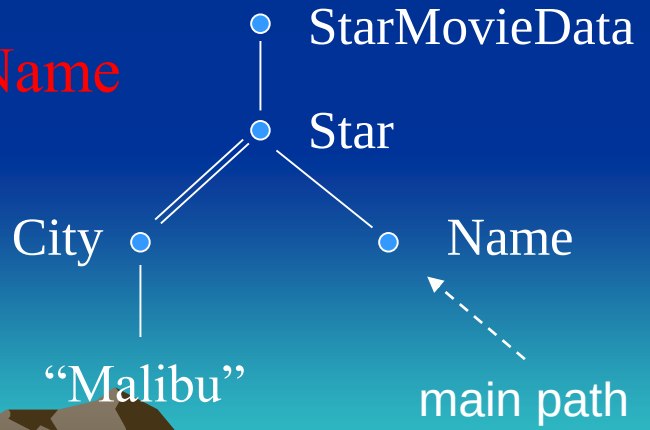
/StarMovieData//Star/self::node( )

/StarMovieData//Star

Conditions in Path Expressions

As we evaluate a path expression, we can restrict ourselves to follow only a subset of the paths whose tags match the tags in the expression. To do so, we follow a tag by a condition, surrounded by square brackets. Such a condition can be anything that has a boolean value. Values can be compared by comparison operators: = , >=, !=. A compound condition can be constructed by connecting comparisons with logic operations: ∨, ∧.

/StarMovieData/Star[.//City = “Malibu”]/Name



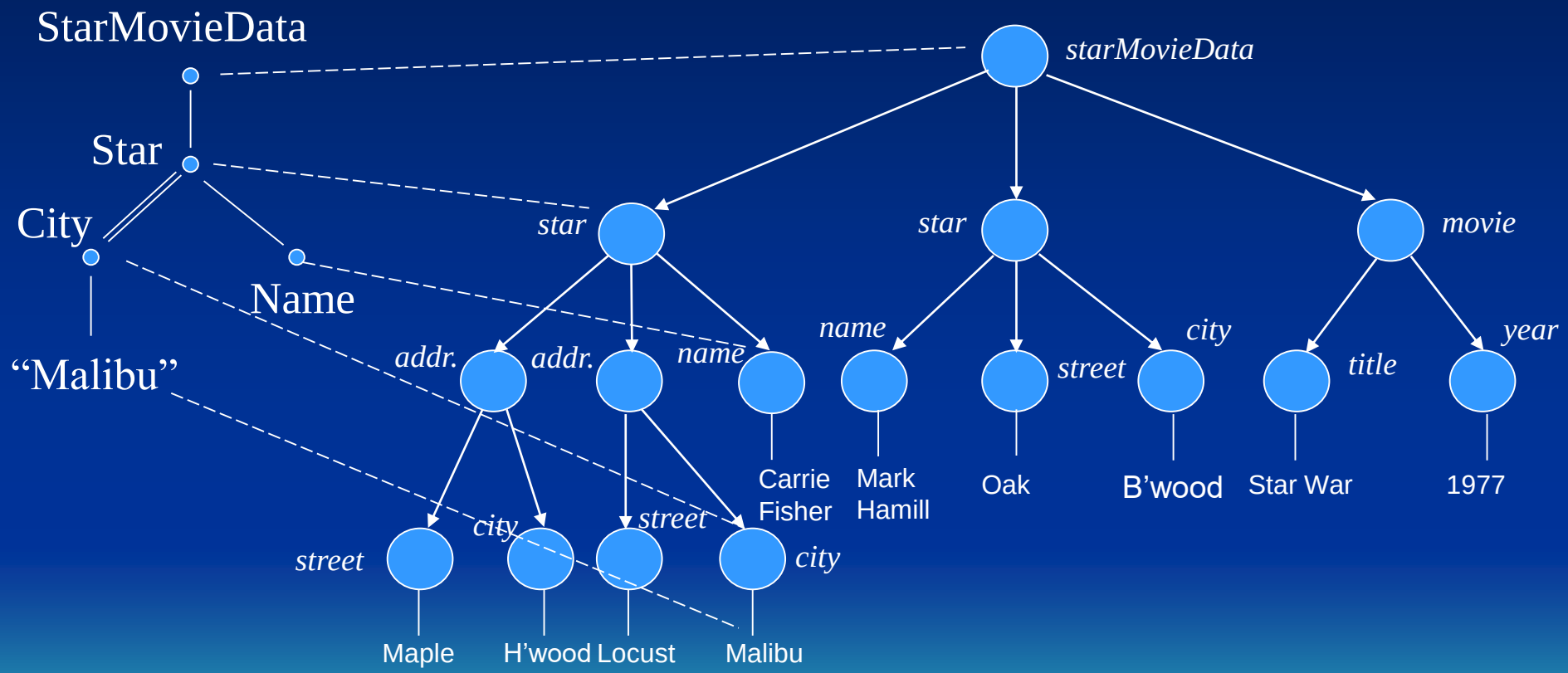

```
<Name>Carrie Fisher</Name>
```

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<StarMovieData>
    <Star starID = “cf” starredIn = “sw”>
        <Name>Carrie Fishes</Name>
        <Address>
            <Street>123 Maple St.</Street><City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street><City>Malibu</City>
        <Address>
    </Star>
    <Star starID = “mh” starredIn = “sw”>
        <Name>Mark Hamill</Name><Street>456 Oak Rd.</Street>
        <City>Brentwood</City>
    </Star>
    <Movie movieID = “sw” starOf = “cf mh”>
        <Title>Star Wars</title><Year>1977</Year>
    </Movie>
</StarMovieData>
```

```
<Name>Carrie Fisher</Name>
```

/StarMovieData/Star[.//City = “Malibu”]/Name

/StarMovieData/Star[.//City = “Malibu”]/Name

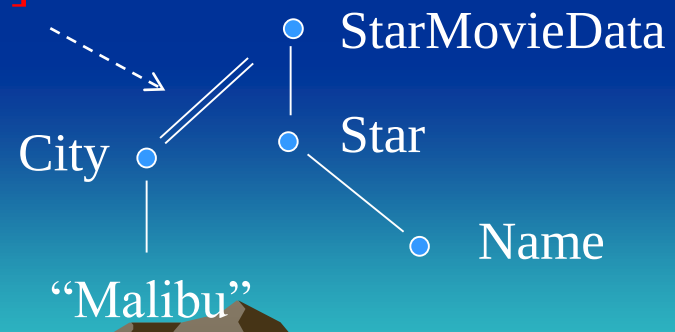


Name = “Carrie Fisher”

Conditions in Path Expressions

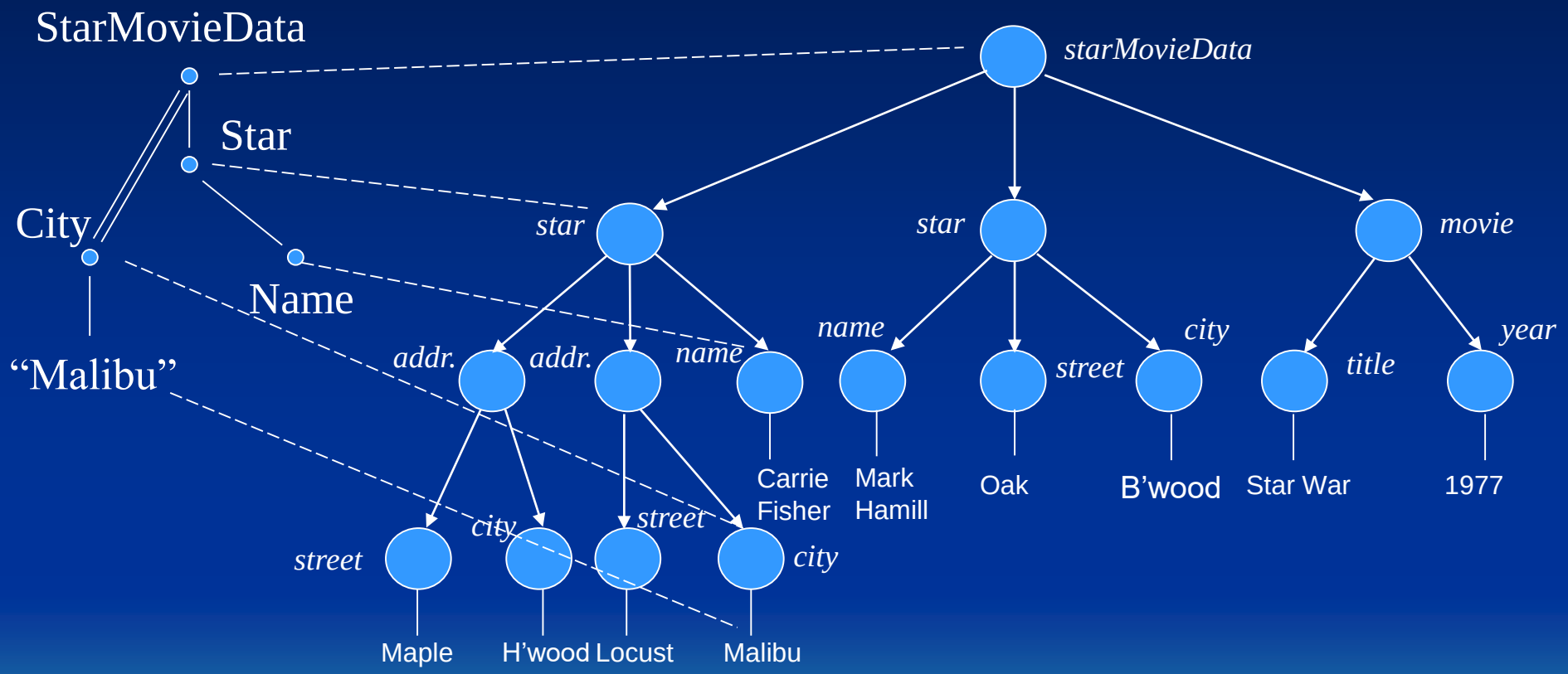
As we evaluate a path expression, we can restrict ourselves to follow only a subset of the paths whose tags match the tags in the expression. To do so, we follow a tag by a condition, surrounded by square brackets. Such a condition can be anything that has a boolean value. Values can be compared by comparison operators: = , >=, !=. A compound condition can be constructed by connecting comparisons with operations: $\vee$, $\wedge$.

/StarMovieData/Star[..//City = “Malibu”]/Name



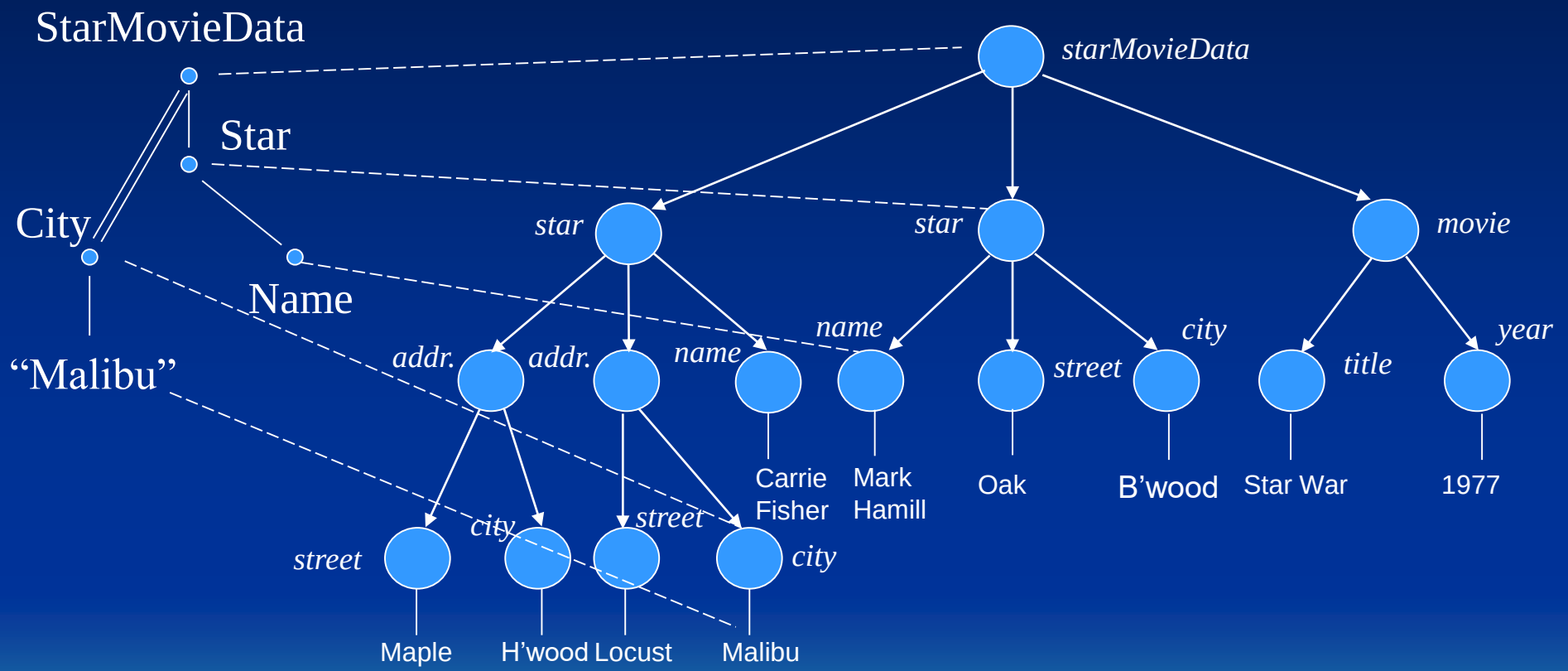
/StarMovieData/Star[.//City = “Malibu”]/Name

/StarMovieData/Star[..//City = “Malibu”]/Name



Name = “Carrie Fisher”

/StarMovieData/Star[..//City = “Malibu”]/Name



Name = “Mark Hamil”

Conditions in Path Expressions

Several other useful forms of condition are:

- An integer [*i*] by itself is true only when applied the *i*th child of its parent.

/StarMovieData/Stars/Star[2]

- A tag [*T*] by itself is true only for elements that have one or more subelements with tag *T*.

/StarMovieData/Stars/Star[Address]

- An attribute [*A*] by itself is true only for elements that have an attribute *A*.

/StarMovieData/Stars/Star[@startID]

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<Movies>
    <Movie title = “King Kong” >
        <Version year = “1933”>
            <Star>Fay Wray</Star>
        </Version>
        <Version year = “1976”>
            <Star>Jeff Bridegs</Star>
            <Star>Jessica Lange</Star>
        </Version>
    </Movie>
    <Movie title = “Footloose”>
        <Version year = “1984”>
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parkr</Star>
        </Version>
    </Movie>
</Movies>
```

/Movies/Movie/Version[1]/@year

/Movies/Movie/Version/Star?

/Movies/Movie/Version[Star] ?

Wildcards

In an XPath expression, we can use * to say “any tag”. Likewise, @* says “any attribute.”

```
/StarMovieData/*/@*
```

Results: “cf”, “sw”, “mh”, “sw”, “sw”, “cf mh”

```
<StarMovieData>
    <Star starID = “cf” starredIn = “sw”>
        … …
    </Star>
    <Star starID = “mh” starredIn = “sw”>
        … …
    </Star>
    <Movie movieID = “sw” starOf = “cf mh”>
        … …
    </Movie>
</StarMovieData>
```

The XPath expressions are mainly used in HTML, XQuery and XSLT languages.

Example:

```
<ul>
     {doc(starMovie.xml)/StarMovieData/*/@*}
</ul>
```

- Cf
- Sw
- Mh
- Sw
- Sw
- cf mh

XQuery

- XQuery is an extension of XPath that has become a standard for high-level querying of databases containing XML data.

- XQuery is designed to take data from multiple databases, from XML files, from remote Web documents, even from CGI (common gate interface) scripts, and to produce XML results that you can process with XSLT.

XQuery Basics

All values produced by XQuery expressions are sequences of items.

Items:

primitive values

nodes: document, element, attribute nodes

XQuery is a *functional language*, which implies that any XQuery expression can be used in any place that an expression is expected.

FLWR Expressions

FLWR (pronounced “flower”) expressions are in some sense analogous to SQL select-from-where expressions.
An XQuery expression may involve clauses of four types, called for-, let-, where-, and return-clauses (FLWR).

1. The query begins with zero or more for- and let-clauses. There can be more than one of each kind, and they can be interlaced in any order, e.g., for, for, let, for, let.
2. Then comes an optional where-clause.
3. Finally, there is exactly one return-clause.

Return <Greeting>“Hello World”</Greeting>

Let Clause

let *variable* := *expression*

- The intent of this clause is that the expression is evaluated and assigned to the variable for the remainder of the FLWR expression.
- Variables in XQuery must begin with a dollar-sign.
- More generally, a comma-separated list of assignments to variables can appear.

```
let $stars := doc("stars.xml")
```

```
let   $movies := doc("movies.xml")
      $stars := doc("stars.xml")
```

for Clause

for *variable* in *expression*

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
```

Stars.xml

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<Stars>
    <Star>
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maples St.</street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ave.</Street>
            <City>Malibu</City>
        </Address>
    </Star>
            … more stars
</Stars>
```

Movies.xml

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<Movies>
    <Movie title = “King Kong”>
        <Version year = “1993”>
            <Star>Fay Wray</Star>
        </Version>
        <Version year = “1976”>
            <Star>Jeff Brideges</Star>
            <Star>Jessica Lange</Star>
        </version>
    </Movie>
    <Movie title = “Footloose”>
        <Version year = “1984”>
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parkr</Star>
        </Version>
    </Movie>
</Movies>
```

Where Clause

where *condition*

where $s/Address/Street = “123 Maple St.” and
$s/Address/City = “Malibu”

This clause is applied to an item, and the condition, which is an expression, evaluates to true or false.

return Clause

return *expression*

This clause returns the values obtained by evaluating *expression*.

```
let $movies := doc(“movies.xml”)
for $m in $movies/Movies/Movie
return $m/Version/Star
```

```
<Star>Fay Wray</Star>
<Star>Jeff Brideges</Star>
<Star>Jessica Lange</Star>
<Star>Kevin Bacon</Star>
<Star>John Lithgow</Star>
<Star>Sarah Jessica Parker</Star>
```

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return $m/Version/Star
```

```
<Star>Fay Wray</Star>
<Star>Jeff Brideges</Star>
<Star>Jessica Lange</Star>
<Star>Kevin Bacon</Star>
<Star>John Lithgow</Star>
<Star>Sarah Jessica Parker</Star>
```

```
<? Xml version = "1.0" encoding = "utf-8" … ?>
<Movies>
    <Movie title = "King Kong">
        <Version year = "1993">
            <Star>Fay Wray</Star>
        </Version>
        <Version year = "1976">
            <Star>Jeff Brideges</Star>
            <Star>Jessica Lange</Star>
        </version>
    </Movie>
    <Movie title = "Footloose">
        <Version year = "1984">
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parkr</Star>
        </Version>
    </Movie>
</Movies>
```

Replacement of variables by their Values

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return <Movie title = $m/@title>$m/Version/Star</Movie>
```

Not correct! The variable will not be replaced by its values.

```
<Movie title = $m/@title>$m/Version/Star</Movie>
<Movie title = $m/@title>$m/Version/Star</Movie>
<Movie title = $m/@title>$m/Version/Star</Movie>
   … …
```

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return <Movie title = {$m/@title}>{$m/Version/Star}</Movie>
```

```
<Movie title = "King Kong"><Star>Fay Wray</Star></Movie>
<Movie title = "King Kong"><Star>Jeff Brideges</Star></Movie>
<Movie title = "King Kong"><Star>Jessica Lange</Star></Movie>
<Movie title = "Footloose"><Star>Kevin Bacon</Star></Movie>
<Movie title = "Footloose"><Star>John Lithgow</Star></Movie>
<Movie title = "Footloose"><Star>Sarah Jessica Parker</Star></Movie>
        … …
```

Joins in XQuery

We can join two or more documents in XQuery in much the same way as in SQL. In each case, we need variables, each of which ranges over elements of one of the documents or tuples of one of the relations, respectively.

1. In SQL, we use a from-clause to introduce the needed tuple variables
2. In XQuery, we use a for-clause.

```
let   $movies := doc("movies.xml")
      $stars := doc("stars.xml")
for   $s1 in $movies/Movies/Movie/Version/Star
      $s2 in $Stars/Stars/Star
where data($s1) = data($s2/Name)
return $s2/Address/City
```

Select ssn, lname, Dname
From employees s1, departments s2
Where s1.dno = s2. Dnumber

```
let   $movies := doc("movies.xml")
      $stars := doc("stars.xml")
for   $s1 in $movies/Movies/Movie/Version/Star
      $s2 in $Stars/Stars/Star
where data($s1) = data($s2/Name)
return $s2/Address/City
```

```
<? Xml version = "1.0" …. … ?>
<Movies>
    <Movie title = "King Kong">
        <Version year = "1993">
            <Star>Fay Wray</Star>
        </Version>
        <Version year = "1976">
            <Star>Jeff Brideges</Star>
            <Star>Jessica Lange</Star>
        </version>
    </Movie>
    <Movie title = "Footloose">
        <Version year = "1984">
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parkr</Star>
        </Version>
    </Movie>
</Movies>
```

```
<? Xml version = "1.0" encoding = "utf-8" … ?>
<Stars>
    <Star>
        <Name>Fay Wray</Name>
        <Address>
            <Street>123 Maples St.</street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Mallibu</City>
        </Address>
    </Star>
            … more stars
</Stars>
```

XQuery Comparison Operators

A query: find all the stars that live at 123 Maple St., Malibu.

The following FLWR seems correct. But it does not work.

```
let     $stars := doc("stars.xml")
for     $s in $stars/Stars/Star
where $s/Address/Street = "123 Maple St
        and $s/Address/City = "Malibu"
return $s/Name
```

Correct query:

```
let     $stars := doc("stars.xml")
for     $s in $stars/Stars/Star,
        $s1 in $s/Address
where $s1/Street = "123 Maple St." and
        $s1//City = "Malibu"
return $s/Name
```

```
<? Xml version = "1.0" encoding = "utf-8" …
?>
<Stars>
    <Star>
        <Name>Fay Wray</Name>
        <Address>
            <Street>123 Maples St.</street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ave.</Street>
            <City>Mallibu</City>
        </Address>
    </Star>
            … more stars
</Stars>
```

Elimination of Duplicates

XQuery allows us to eliminate duplicates in sequences of any kind, by applying the built-in distinct values.

Example. The result obtained by executing the following first query may contain duplicates. But the second not.

```
let     $starsSeq := (
        let $movies := doc("movies.xml")
        for $m in $movies/Movies/Movie
        return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>
```

```
let     $starsSeq := distinct-values(
        let $movies := doc("movies.xml")
        for $m in $movies/Movies/Movie
        return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>
```

Select average(distinct *salary*) from *employee*;

Quantification in XQuery

There are expressions that say, in effect, *for all* (∀), and *there exists* (∃):

every *variable* in *expression1* satisfies *expression2*
some *variable* in *expression1* satisfies *expression2*

```
let     $stars := doc("stars.xml")
for     $s in $stars/Stars/Star
where     every $c in $s/Address/City
          satisfies $c = "Hollywood"
return $s/Name
```

Find the stars who have houses only in Hollywood.

```
let     $stars := doc("stars.xml")
for     $s in $stars/Stars/Star
where     $c in $s/Address/City satisfies
          $c = "Hollywood"
return $s/Name
```

Find the stars with a home in Hollywood. (Key word *some* is not used.)

Select *ssn, fname, salary* **from** *employee* ***where*** *salary*

> **all** *(**select** salary **from** employee **where** dno = 4);*

```
Select fname, lname
from employee
where
        exists (select *
                from dependent
                where essn = ssn);
```

Aggregation

XQuery provides built-in functions to compute the usual aggregations such as count, average, sum, min, or max. They take any sequence as argument. That is, they can be applied to the result of any XPath expression.

```
let     $movies := doc("movies.xml")
for     $m in $movies/Movies/Movie
where count($m/Version) > 1
return $m
```

Find the movies with multiple versions.

Select *s.ssn, s.lname, count(r.lname)*

from *employee s, employee r*

where *s.ssn = r.superssn*

group by *s.ssn, s.lname;*

having *count(s.name)* < 3;

Branching in XQuery Expressions

There is an *if-then* expression in Xquery of the form:

if (*expression1*) **then** (*expression2*)

```
let     $kk := doc("movies.xml")/Movies/Movie/Movie[@title = "King Kong"]
for     $v in $kk/Version
return    if ($v/@year = max($kk/Version/@year))
          then <Latest>{$v}</Latest>
          else <Old>{$v}</Old>
```

Tag the version of *King Kong*.

```
<? Xml version = "1.0" …. … ?>
<Movies>
      <Movie title = "King Kong">
            <Version year = "1993">
                  <Star>Fay Wray</Star>
            </Version>
            <Version year = "1976">
                  <Star>Jeff Brideges</Star>
                  <Star>Jessica Lange</Star>
            </version>
      </Movie>
      <Movie title = "Footloose">
            <Version year = "1984">
                  <Star>Kevin Bacon</Star>
                  <Star>John Lithgow</Star>
                  <Star>Sarah Jessica Parkr</Star>
            </Version>
      </Movie>
</Movies>
```

Movies.xml

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<Movies>
    <Movie title = “King Kong”>
        <Version year = “1993”>
            <Star>Fay Wray</Star>
        </Version>
        <Version year = “1976”>
            <Star>Jeff Brideges</Star>
            <Star>Jessica Lange</Star>
        </version>
    </Movie>
    <Movie title = “Footloose”>
        <Version year = “1984”>
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parkr</Star>
        </Version>
    </Movie>
</Movies>
```

```
Let $kk :=
    doc(“movies.xml”)/Movies/Movie/Movie
    [@title = “King Kong”]
For $v in $kk/Version
Return if ($v/@year =
    max($kk/Version/@year))
        then <Latest>{$v}</Latest>
        else <Old>{$v}</Old>
```

```
<Latest><Version year = “1993”> … </Latest>
<Old><Version year = “1976”> … </Old>
```

Ordering the Result of a Query

It is possible to sort the result as part of a FLWR query

order *list of expressions*

```
let     $movies := doc("movies.xml")
for     $m in $movies/Movies/Movie,
        $v in $m/Version
order   $v/@year
return  <Movie title = "{$m/@title}" year = "{$v/@year}" />
```

Select *
From employees
order by ssn

Construct the sequence of *title-year* pairs, ordered by *year*.

Movies.xml

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<Movies>
    <Movie title = “King Kong”>
        <Version year = “1993”>
            <Star>Fay Wray</Star>
        </Version>
        <Version year = “1976”>
            <Star>Jeff Brideges</Star>
            <Star>Jessica Lange</Star>
        </version>
    </Movie>
    <Movie title = “Footloose”>
        <Version year = “1984”>
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parkr</Star>
        </Version>
    </Movie>
</Movies>
```

```
let     $movies := doc(“movies.xml”)
for     $m in $movies/Movies/Movie,
        $v in $m/Version
order   $v/@year
return <Movie title = “{$m/@title}”
year = “{$v/@year}” />
```

```
<Movie title = “King Kong” year = “1976” />
<Movie title = “Footloose” year = “1984” />
<Movie title = “King Kong” year = “1993” />
```

```
let     $movies := doc("movies.xml")
for     $m in $movies/Movies/Movie,
        $v in $m/Version
order $m/@title, $v/@year
return   <Movie title = "{$m/@title}" year = "{$v/@year}" />
```

```
<Movie title = "Footloose" year = "1984" />
<Movie title = "King Kong" year = "1976" />
<Movie title = "King Kong" year = "1993" />
```

About usage of XQuery

An XQuery expression can be embedded in an HTML file.

```
...
<ul>
        {
          for $x in doc("books.xml")/bookstore/book/title
          order by $x
          return <li>{$x}</li>
        }
</ul>
...
```

Extensible Stylesheet Language

XSLT (Extensible Stylesheet Language for Transformation) is a standard of the World-Wide-Web Consortium.

- Its original purpose was to allow XML documents to be transformed into HTML or similar forms that allowed the document to be viewed or printed.
- In practice, XSLT is another query language for XML to extract data from documents or turn one document form into another form.

XSLT Basics

Like XML schema, XSLT specifications are XML documents, called *stylsheet*. The tag used in XSLT are found in a name-space: http://www.w3.org/1999/XSL/Transform.

At the highest level, a stylesheet looks like:

```
<? Xml version = ‘1.0” encoding = “utf-8” ?>
<xsl:stylesheet xmlns:xsl =
        http://www.w3.org/1999/XSL/Transform>
… …
</xsl:stylesheet>
```

Templates

A stylesheet will have one or more templates. To apply a stylesheet to an XML document, we go down the list of templates until we find one that matches the root.

<xsl:template match = “*XPath expression*”>

Templates

<xsl:template match = "*XPath expression*">

XPath expression can be either rooted (beginning with a slash) or relative. It describes the elements of XML documents to which this template is applied.

Rooted expression – the template is applied to every element of the document that matches the path (absolute path).

Relative expression – part of an Xpath, evaluated relative to a reference point (the current node).

```
<? Xml version = “1.0” encoding = “utf-8” ?>
<xsl:stylesheet xmlns:xsl =
                http://www.w3.org/1999/XSL/Transform>
        <xsl:template match = “/”>
            <HTML>
                <BODY>
                    <B>This is a document</B>
                </BODY>
            </HTML>
        </xsl:template >
</xsl:stylesheet>
```

Applying the template, an XML document is transformed to a HTML file:

```
<HTML>
    <BODY>
        <B>This is a document</B>
    </BODY>
</HTML>
```

Obtaining Values from XML Data

<xsl:value-of select = "*expression*" />

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<Movies>
    <Movie title = "King Kong">
        <Version year = "1993">
            <Star>Fay Wray</Star>
        </Version>
        <Version year = "1976">
            <Star>Jeff Brideges</Star>
            <Star>Jessica Lange</Star>
        <version year = "2005" />
    </Movie>
    <Movie title = "Footloose">
        <Version year = "1984">
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parkr</Star>
        </Version>
    </Movie>
</Movies>
```

```
<? Xml version = "1.0" encoding = "utf-8" ?>
<xsl:stylesheet xmlns:xsl =
        http://www.w3.org/1999/XSL/Transform>
        <xsl:template match = "/Movies/Movie">
            <xsl:value-of select = "@title" />
            <BR/>
        </xsl:template >
</xsl:stylesheet>
```

"King Kong"

"Footloose"

This ability makes XSTL a query language.

Recursive Use of Templates

Powerful transformations require recursive application of templates at various elements of the input.

<xsl:apply-template select = “*expression*” />

```
<? Xml version = “1.0” encoding = “utf-8” ?>
<xsl:stylesheet xmlns:xsl =
 http://www.w3.org/1999/XSL/Transform>

    <xsl:template match = “/Movies”>
       <Movies>
            <xsl:apply-templates />
       </Movies>
    </xsl:template >

    <xsl:template match = “Movie”>
       <Movie title = “<xsl:value-of select = “@title” />
            <xsl:apply-templates />
       </Movie>
    </xsl:template>

    <xsl:template match = “Version”>
         <xsl:apply-template />
    </xsl:template>

    <xsl:template match = “Star”>
         <Star name = “<xsl:value-of select = “.” />”/>
    </xsl:template>
</xsl:stylesheet>
```

use this template

use this template

use this template

```
<? Xml version = “1.0” encoding = “utf-8”
    standalone = “yes” ?>
<Movies>
    <Movie title = “King Kong”>
        <Version year = “1993”>
            <Star>Fay Wray</Star>
        </Version>
        <Version year = “1976”>
            <Star>Jeff Brideges</Star>
            <Star>Jessica Lange</Star>
        </version>
    </Movie>
    <Movie title = “Footloose”>
        <Version year = “1984”>
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parkr
            </Star>
        </Version>
    </Movie>
</Movies>
```

```
<? Xml version = “1.0” encoding = “utf-8”
    standalone = “yes” ?>
<Movies>
    <Movie title = “King Kong”>
        <Version year = “1993”>
            <Star>Fay Wray</Star>
        </Version>
        <Version year = “1976”>
            <Star>Jeff Brideges</Star>
            <Star>Jessica Lange</Star>
        </version>
    </Movie>
    <Movie title = “Footloose”>
        <Version year = “1984”>
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica
Parkr</Star>
        </Version>
    </Movie>
</Movies>
```

```
<? Xml version = “1.0” encoding = “utf-8”
                    standalone = “yes” ?>
<Movies>
    <Movie title = “King Kong”>
        <Star name = “Fay Wray” />
        <Star name = “Jeff Brideges” />
        <Star name = “Jessica Lange” />
    </Movie>
    <Movie title = “Footloose”>
        <Star name = “Kevin Bacon” />
        <Star name = “John Lithgow” />
        <Star name = “Sarah Jessica Parkr” />
    </Movie>
</Movies>
```

Iteration in XSLT

We can put a loop within a template that gives us freedom over the order in which we visit certain subelements of the element to which the template is being applied.

<xsl:for-each select = "*expression*" >

The expression is an XPath expression whose value is a sequence of items. Whatever is between the opening <for-each> tag and its matching closing tag is executed for each item, in turn.

```
<? Xml version = “1.0” encoding = “utf-8” standalone = “yes” ?>
<Stars>
    <Star>
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maples St.</stree
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Mallibu</City>
        </Address>
    </Star>
        … more stars
</Stars>
```

1. Carrie Fishes
2. Mark Hamill

… …

1. Hollywood
2. Malibu

… …

```
<? Xml version = “1.0” encoding = “utf-8” ?>
<xsl:stylesheet xmlns:xsl =
      http://www.w3.org/1999/XSL/Transform >
    <xsl:template match = “/”>
        <0L>
            <xsl:for-each select = “Stars/Star” >
                <LI>
                    <xsl:value-of select = “Name”>
                </LI>
            </xsl:for-each>
        </0L><P />
        <0L>
            <xsl:for-each select =
                    “Stars/Star/Address”>
                <LI>
                    <xsl:value-of select = “City”>
                </LI>
            </xsl:for-each>
        </0L>
    </xsl:template >
</xsl:stylesheet>
```

```
<Stars>
    <Star>
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maples St.</street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Mallibu</City>
        </Address>
    </Star>
        … more stars
</Stars>
```

```
<0L>
    <LI>
        Carrie Fisher
    </LI>
    <LI>
        Mark Hamil
    </LI>
    … more stars
</0L><P/>
<0L>
    <LI>
        Hollywood
    </LI>
    <LI>
        Malibu
    </LI>
    … more cities
</0L>
```

1. Carrie Fishes
2. Mark Hamill

… …

1. Hollywood
2. Malibu

… …

```
<? Xml version = “1.0” encoding = “utf-8” ?>
<xsl:stylesheet xmlns:xsl =
http://www.w3.org/1999/XSL/Transform>
    <xsl:template match = “/”>
        <0L>
            <xsl:for-each select =
                “Stars/Star” >
                <LI>
                    <xsl:value-of select =
                    “Name”>
                </LI>
            </xsl:for-each>
        </0L><P/>
        <0L>
            <xsl:for-each select =
                “Stars/Star/Address”>
                <LI>
                    <xsl:value-of select =
                    “City”>
                </LI>
            </xsl:for-each>
        </0L>
    </xsl:template >
</xsl:stylesheet>
```

Conditions in XSLT

We can introduce branching into our templates by using an if tag.

<xsl:if test = "*boolean expression*" >

Whatever appears between its tag and its matched closing tag is executed if and only if the boolean expression is *true*.

```
<? Xml version = "1.0" encoding = "utf-8" ?>
<xsl:stylesheet xmlns:xsl =
http://www.w3.org/1999/XSL/Transform>
        <xsl:template match = "/">
          <TABLE border = "5"><TR><TH>Stars</TH><TR>
            <xsl:for-each select = "Stars/Star" >
              <xsl:if test = "Address/City = 'Hollywood'">
                  <TR><TD><xsl:value-of select = "Name"</TD>
                  </TR>
              </xsl:if>
            </xsl:for-each>
          </TABLE>
        </xsl:template >
</xsl:stylesheet>
```

Stars
Carrie Fishes
⋮

```
<TABLE border = “5”><TR><TH>Stars</TH><TR>
        <TR>
                <TD>
                        Carrie Fishes
                </TD>
        </TR>
        <TR>
                <TD>
                        … …
                </TD>
        </TR>
                … …
</TABLE>
```

List all those stars who have a house in Hollywood.

Stars
Carrie Fishes
⋮

```
<html>
<body>
        <table border="1">
        <tr>
        <th>Month</th>
        <th>Savings</th>
         </tr>
        <tr>
        <td>January</td>
        <td>$100</td>
        </tr>
        </table>
</body>
</html>
```

Month	Savings
January	$100

How to use XSTL to make document transformation?

In this example, creating the XML file that contains the information about three students and displaying the XML file using XSLT.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<?xml-stylesheet type = "text/xsl "href = “transform.xsl" ?>
<Student>
 <s>
  <name> David John Agarwal</name><branch> CSE</branch>
  <age> 23</age><city> Manibu</city>
 </s>
 <s>
  <name> Mary Chen</name><branch> CSE</branch>
  <age> 17</age><city> New York</city>
 </s>
 <s>
  <name> Christ Henry</name><branch> IT</branch>
  <age> 25</age> <city> Washington</city>
 </s>
</student>
```

students.xml

transform.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
 <html> <body>
  <h1 align="center">Students' Basic Details</h1>
   <table border="3" align="center" >
            <tr>
                     <th>Name</th>
                     <th>Branch</th>
                     <th>Age</th>
                     <th>City</th>
            </tr>
            <xsl:for-each select="student/s">
            <tr>
                     <td><xsl:value-of select="name"/></td>
                     <td><xsl:value-of select="branch"/></td>
                      <td><xsl:value-of select="age"/></td>
                     <td><xsl:value-of select="city"/></td>
            </tr>
            </xsl:for-each>
   </table> </body> </html> </xsl:template> </xsl:stylesheet>
```

Name	Branch	Age	City
David John	CSE	23	Malibu
Mary Chen	CSE	21	New York
Christ Henry	CSE	22	Washington

How to use XSTL to make document transformation?

(in Java)

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class Main {
  public static void main(String args[]) throws Exception {

    StreamSource source = new StreamSource(args[0]);
    StreamSource stylesource = new StreamSource(args[1]);

    TransformerFactory factory = TransformerFactory.newInstance();
    Transformer transformer = factory.newTransformer(stylesource);

    StreamResult result = new StreamResult(System.out);
    transformer.transform(source, result);
  }
}
```

How to use XSTL to make document transformation?

```
XslTransform xslTran = new XslTransform();
xslTran.Load("transform.xsl");
XmlTextWriter writer = new XmlTextWriter("xslt_output.html",
System.Text.Encoding.UTF8);
xslTran.Transform(students.xml, null, writer);
```

an XSTL sheet

create a file to store the output

a file containing an XML document to be transformed