Database

- Database System Architecture
- System Catalog
- Query Processing and Optimization,
- Transaction Management
- Concurrency Control
- Recovery
- Security Control
- Enhanced ER-diagram
- Spatial and temporal Databases
- Indexes
- Search engine in web browser and data stream

Final: Dec. 18, 2024, 1:30 pm – 4:30 pm
3C27, book closed

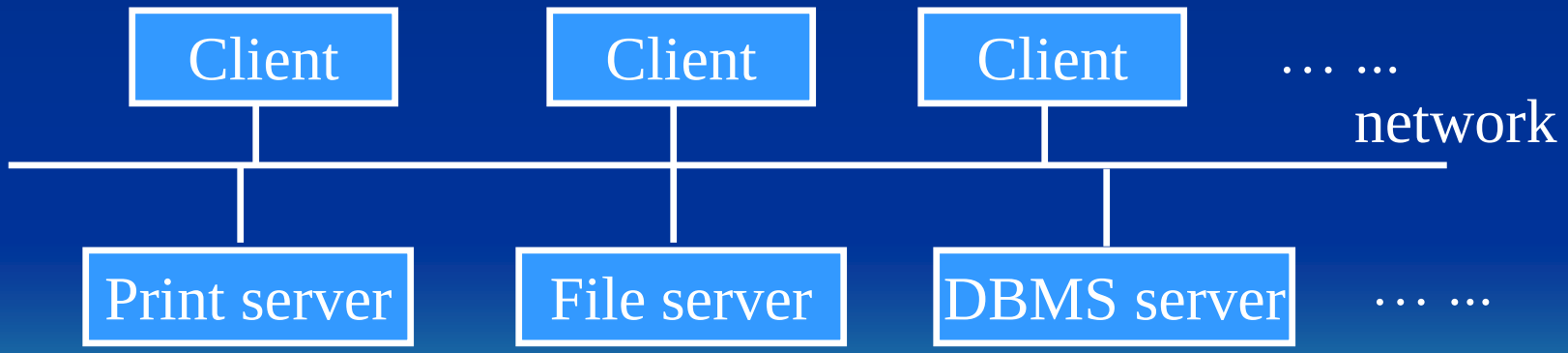
Database System Architecture

- Main frame computer
- Client-Server Computer Architecture
- Client-Server Database Architecture

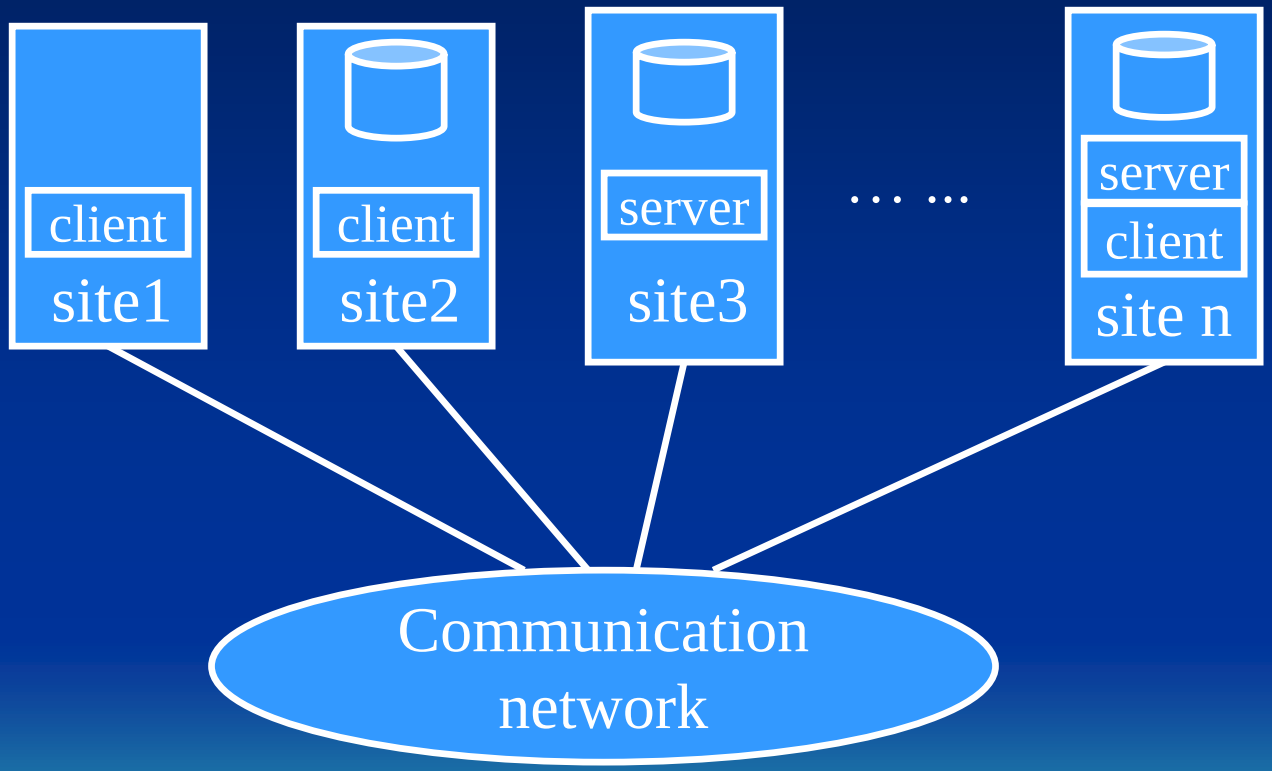
Client-Server Computer Architecture

- Terminals are replaced with PCs and workstations
- Mainframe computer is replaced with specialized servers (with specific functionalities).

  File server, DBMS server, mail server, print server, …



Database System Architectures



Client-Server Architecture in DBMSs

- database client

  user interface, data dictionary functions, DBMS interaction with programming language compiler, global query optimization, structuring of complex objects from the data in the buffers, ...

- database server

  data storage on disk, local concurrency control and recovery, buffering and caching of disk storage, ...

- database connection
  ODBC - open database connectivity
  API - application programming interface

System Catalog

- mata data for a relational schema
- relation names, attribute names, attribute domains (data types)
- description of key constraints
- views, storage structure, indexes
- security, authorization, owner of each relation

Catalog for Relational DBMSs

- Catalog is stored as relations.

  (It can then be queried, updated and managed using DBMS software - SQL.)

REL_AND_ATTR_CATALOG

REL_NAME	ATTR_NAME	ATTR_TYPE	MEMBER_OF_PK	MEMBER_OF_FK	FK_RELATION
EMPLOYEE	FNAME	VSTR15	no	no	
…	…				
EMPLOYEE	SUPERSSN	STR9	no	yes	EMPLOYEE
EMPLOYEE	DNO	INTEGER	no	yes	DEPARTMENT
…	…				

Catalog for Relational DBMSs

- Catalog is stored as relations.

  (It can then be queried, updated and managed using DBMS software - SQL.)

RELATION_KEYS

REL_NAME	KEY_NUM	MEMBER_ATTR

RELATION_INDEXES

REL_NAME	INDEX_NAME	MEMBER_ATTR	INDEX_TYPE	ATTR_NO	ASC_DESC

VIEW_QUERIES

VIEW_NAME	QUERY

VIEW_ATTRIBUTES

VIEW_NAME	ATTR_NAME	ATTR_NUM

RELATION_INDEXES

REL_NAME	INDEX_NAME	MEMBER_ATTR	INDEX_TYPE	ATTR_NO	ASC_DESC
Works_on	I1	SSN	Primary	1	ASC
Works_on	I1	Pno	Primary	2	ASC
Works_on	I2	SSN	Clustering	1	ASC

Primary index:

Index file: I1
(<k(i), p(i)> entries)

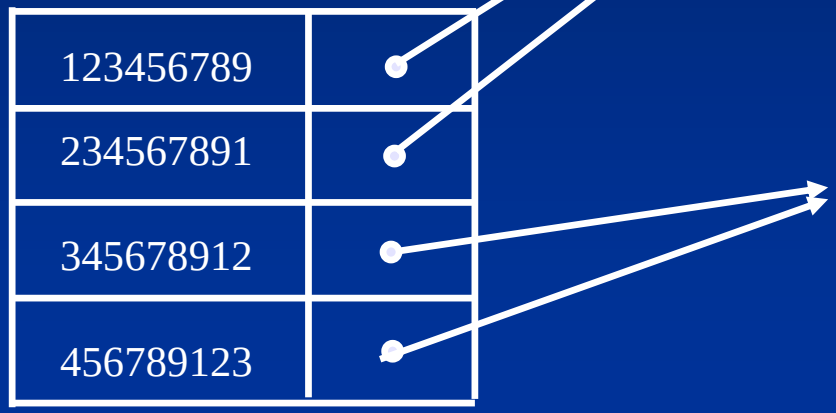
Data file: Works_on

SSN	Pno	hours
123456789	1	…
123456789	2	
123456789	3	
234567891	1	
234567891	2	
345678912	2	
345678912	3	
456789123	1	

… …

Clustering index:

Index file: I2
(<k(i), p(i)> entries)



Data file: Works_on

SSN	Pno	hours
123456789	1	…
123456789	2	
123456789	3	
234567891	1	
234567891	2	
345678912	2	
345678912	3	
456789123	1	

… …

Create View Works_on1

AS Select FNAME, LNAME, PNAME, hours

From EMPLOYEE, PROJECT, WORKS_ON

Where ssn = essn and

Pno. = PNUMBER

VIEW_QUERIES

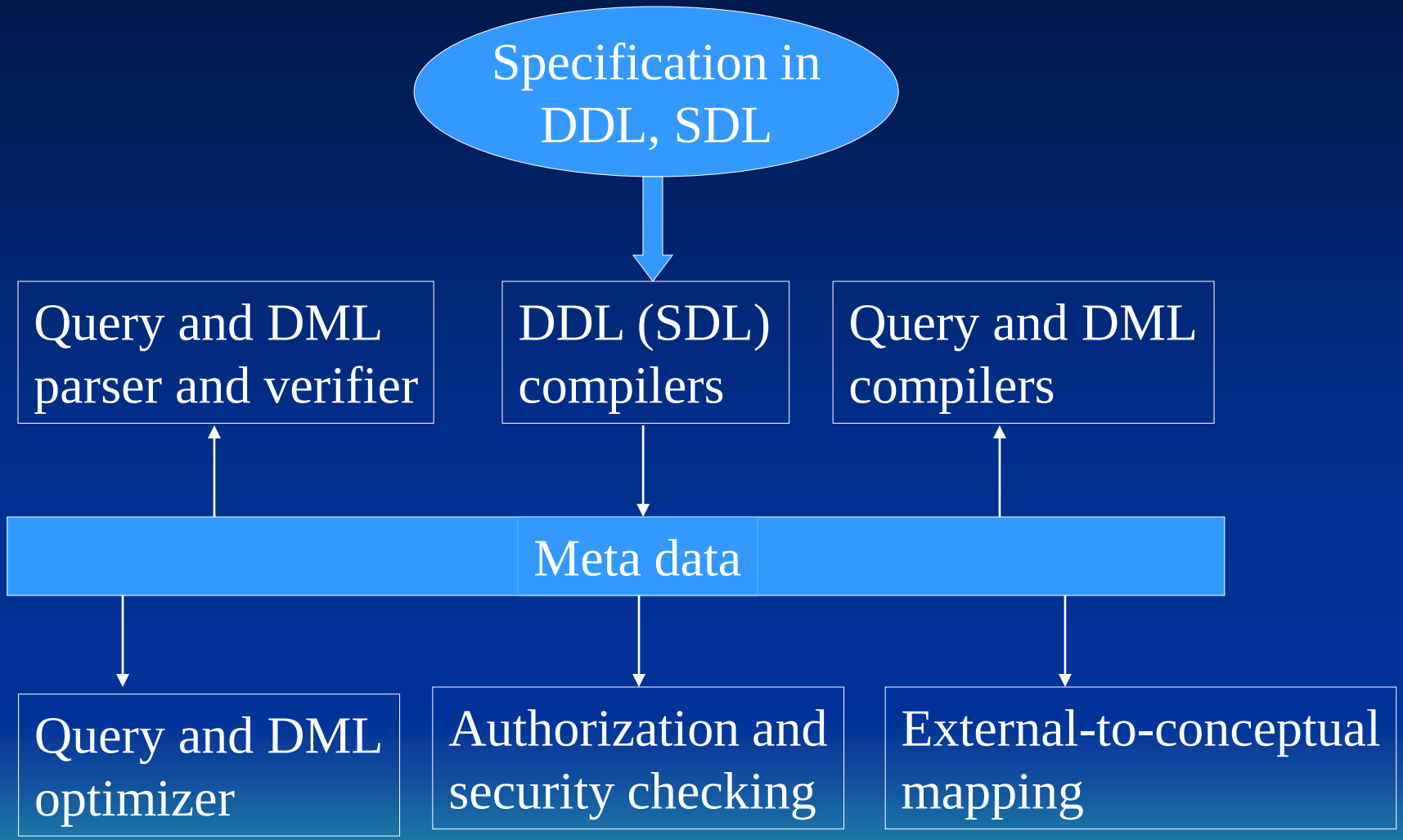
VIEW_NAME	QUERY

Works_on1 Select FNAME, LNAME, PNAME, hour

… …

VIEW_ATTRIBUTES

VIEW_NAME	ATTR_NAME	ATTR_NUM
Works_on1	FNAME	1
Works_on1	LNAME	2
Works_on1	PNAME	3
Works_on1	hours	4


Specification in DDL, SDL
Query and DML parser and verifier
DDL (SDL) compilers
Query and DML compilers
Meta data
Query and DML optimizer
Authorization and security checking
External-to-conceptual mapping

Query Processing and Optimization,

Processing a high-level query

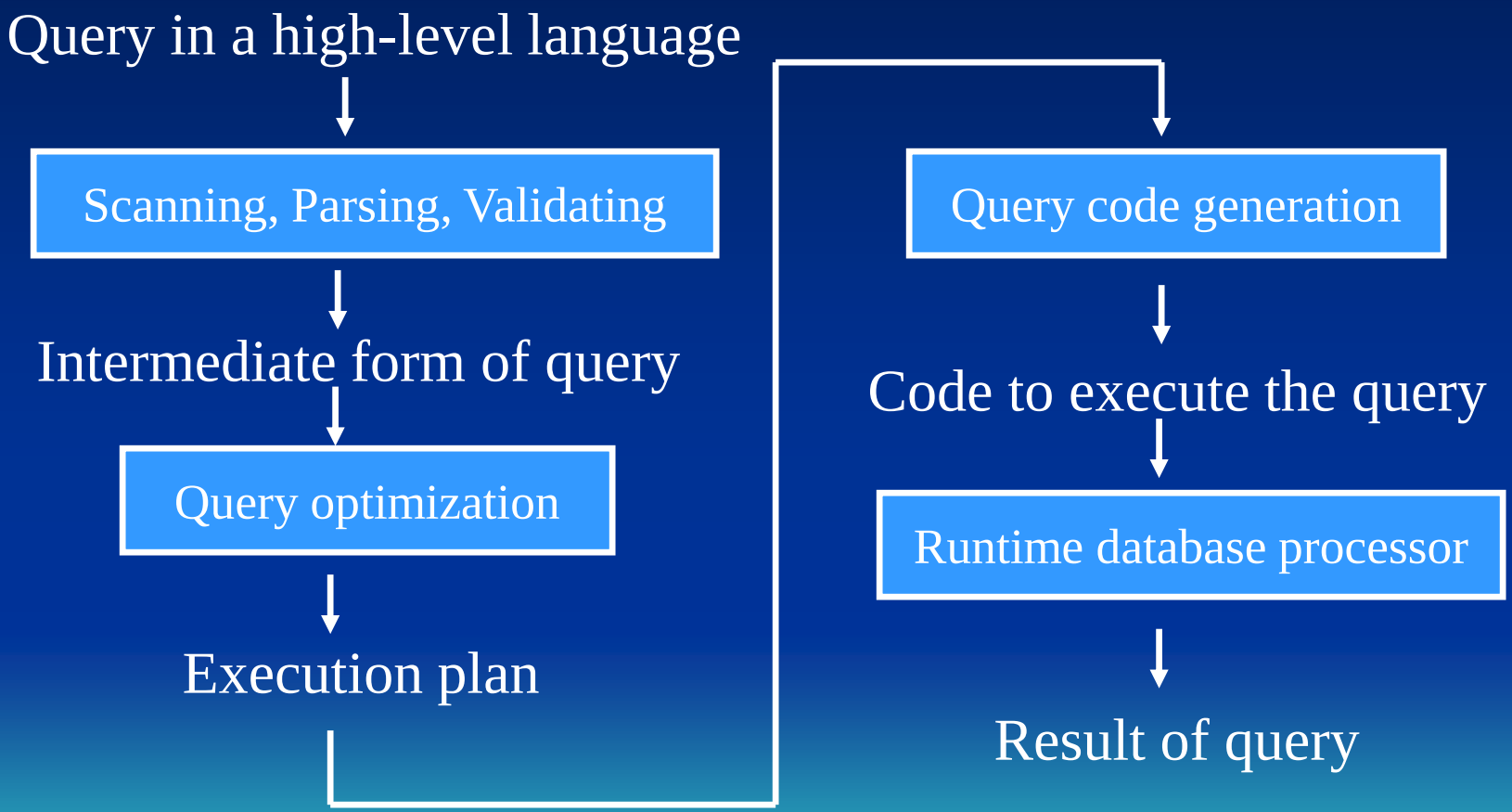
Translating SQL queries into relational algebra expressions

Basic algorithms

- Sorting: internal sorting and external sorting
- Implementing the SELECT operation
- Implementing the JOIN operation
- Other operations

Heuristics for query optimization

- **Steps of processing a high-level query**



- **Translating SQL queries into relational algebra**
  - decompose an SQL query into query blocks
    query block - SELECT-FROM-WHERE clause

Example:
```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   SALARY > (SELECT    MAX(SALARY)
                  FROM      EMPLOEE
                  WHERE     DNO = 5);
```

```
SELECT   MAX(SALARY)
FROM     EMPLOYEE
WHERE    DNO = 5
```

inner block

```
SELECT   LNAME, FNAME
FROM     EMPLOYEE
WHERE    SALARY > c
```

outer block

- **Translating SQL queries into relational algebra**
  - translate query blocks into relational algebra expressions

```
SELECT   MAX(SALARY)
FROM     EMPLOYEE
WHERE    DNO = 5
```

⇨ $\mathscr{F}_{\text{MAX SALARY}}(\sigma_{\text{DNO=5}}(\text{EMPLOYEE}))$

```
SELECT   LNAME, FNAME
FROM     EMPLOYEE
WHERE    SALARY > c
```

⇨ $\pi_{\text{LNAME FNAME}}(\sigma_{\text{SALARY>C}}(\text{EMPLOYEE}))$

- **Basic algorithms**
  - sorting: internal sorting and external sorting
  - algorithm for SELECT operation
  - algorithm for JOIN operation
  - algorithm for PROJECT operation
  - algorithm for SET operations
  - implementing AGGREGATE operation
  - implementing OUTER JOIN

- **Sorting algorithms**
  - internal sorting - sorting in main memory:
    sort a series of integers,
    sort a series of keys
    sort a series of records
  - different sorting methods:
    simple sorting
    heap sorting
  - external sorting – sorting a file which cannot be accommodated completely in main memory

- A simple sorting algorithm

Algorithm

Input: an array *A* containing *n* integers.
Output: sorted array.

1. *i* := 2;
2. Find the least element *a* from *A*(*i*) to *A*(*n*);
3. If *a* is less than *A*(*i* - 1), exchange *A*(*i* - 1) and *a*;
4. *i* := *i* + 1; goto step (2).

Not included in the final

- Merge-sorting

Algorithm *Merge-sorting*(s)

Input: a sequences s = < x_1, ..., x_m>

Output: a sorted sequence.

1. If |s| = 1, then return s;
2. $k := \lceil m/2 \rceil$;
3. s1 := Merge-sorting(x_1, ..., x_k);
4. s2 := Merge-sorting(x_{k+1}, ..., x_m);
5. return(Merge(s1, s2));

Not included in the final

- Merging algorithm

Algorithm(s1, s2)

Input: two sequences: s1 - $x1 \le x2 \ldots \le x_m$ and s2 - $y1 \le y2 \ldots \le y_n$

Output: a sorted sequence: $z1 \le z2 \ldots \le z_{m+n}$.

1.[initialize] $i := 1, j := 1, k := 1$;

2.[find smaller] if $x_i \le y_j$ goto step 3, otherwise goto step 5;

3.[output x_i] $z_k := x_i$, $k := k+1$, $i := i+1$. If $i \le m$, goto step 2;

4.[transmit $y_j \le \ldots \le y_n$] $z_k, \ldots, z_{m+n} := y_j, \ldots, y_n$. Terminate the algorithm;

5.[output y_j] $z_k := y_j$, $k := k+1$, $j := j+1$. If $j \le n$, goto step 2;

6.[transmit $x_i \le \ldots \le x_m$] $z_k, \ldots, z_{m+n} := x_i, \ldots, x_m$. Terminate the algorithm;

- **Basic algorithms** Not included in the final

  -quick sorting

  main idea:

  Algorithm *quick_sort*(from, center, to)

  Input: from - pointer to the starting position of array *A*

  center - pointer to the middle position of array *A*

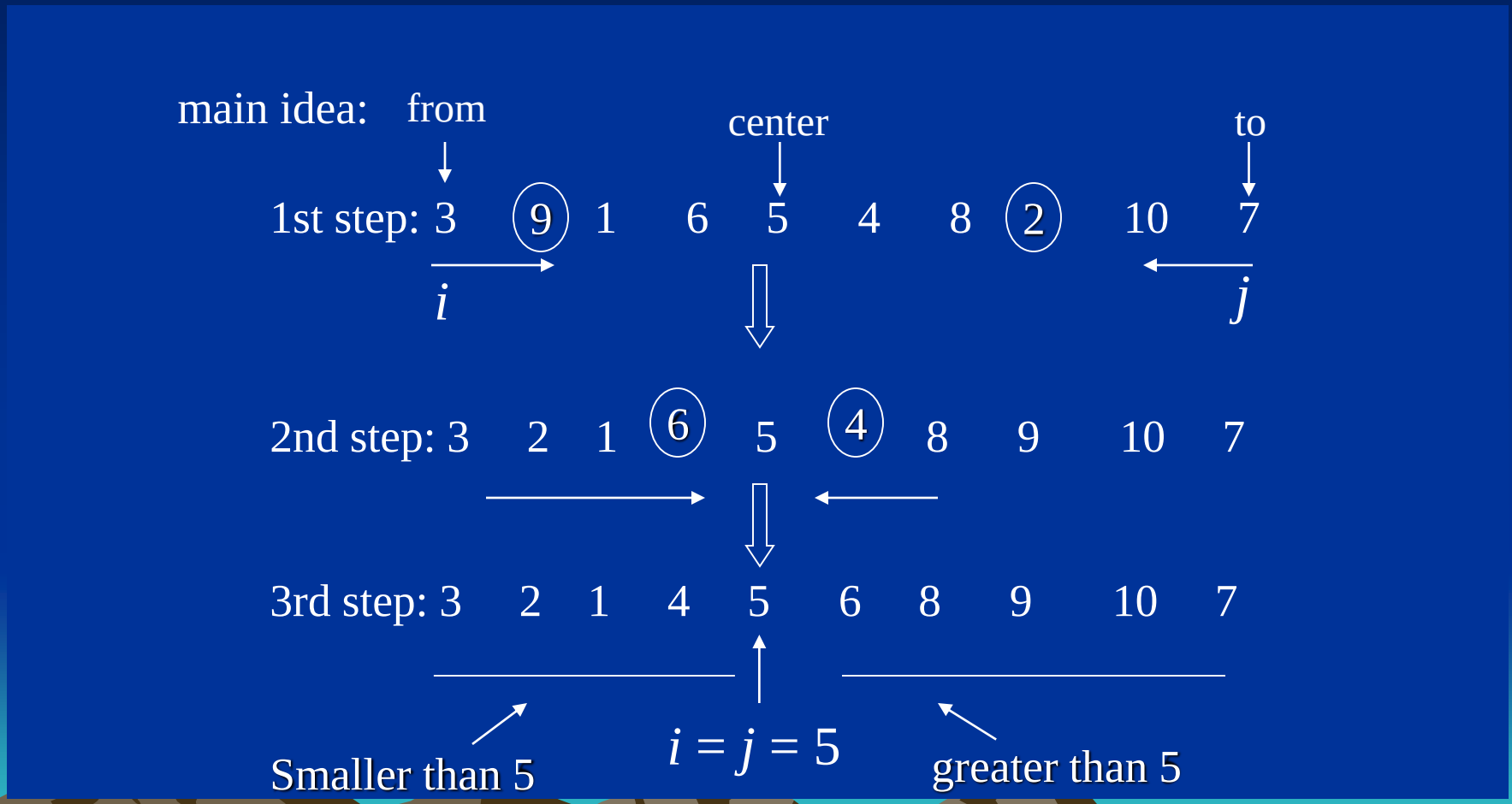
  to - pointer to the end position of array *A*

  Output: sorted array: *A*'

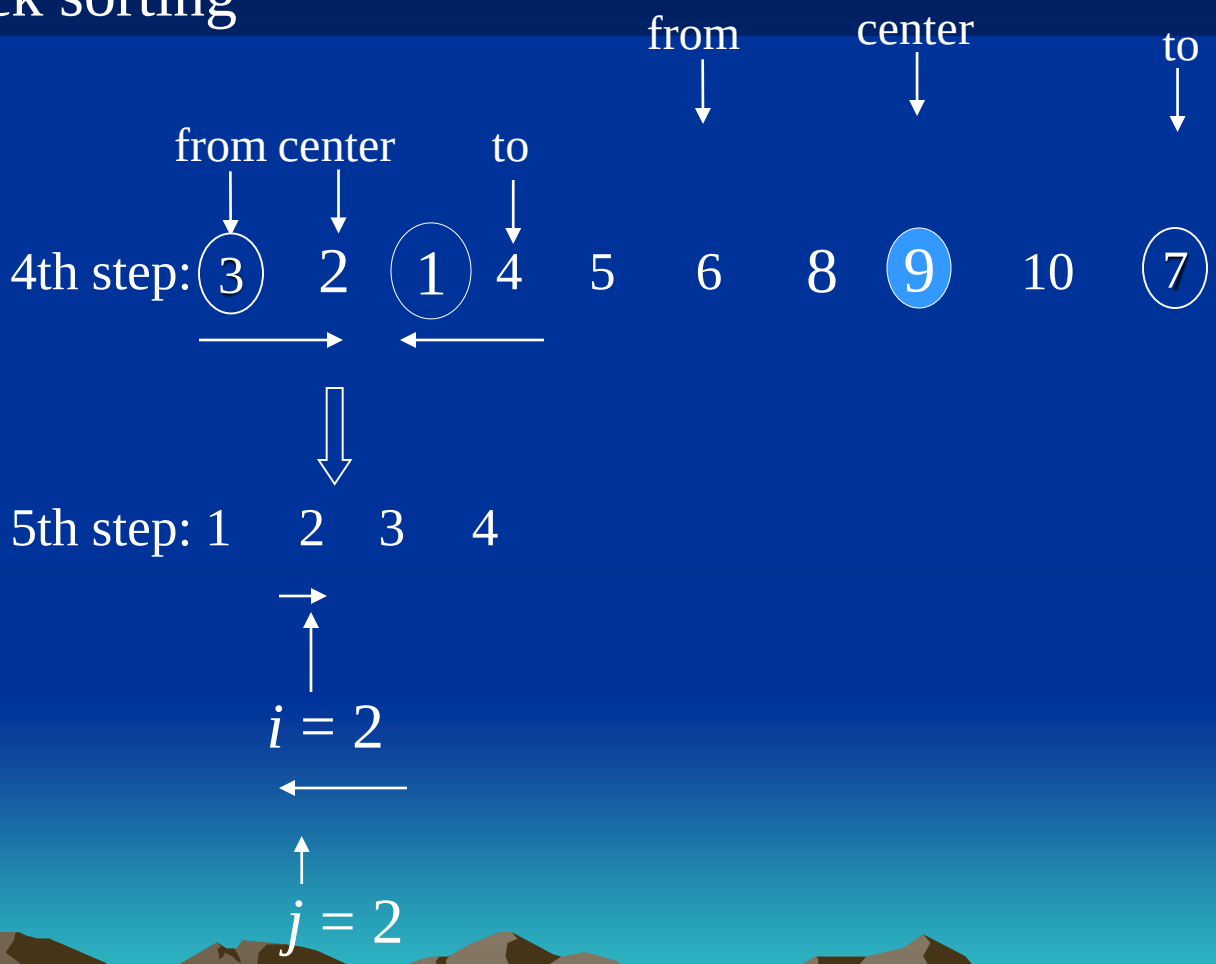
  0. i := 1; j := n;
  1. Find the first element *a* = *A*(i) larger than or equal to *A*(center) from *A*(from) to *A*(to); (i is used to scan from left to right.)
  2. Find the first element *b* = *A*(j) smaller than or equal to *A*(center) from *A*(to) to *A*(from); (j is used to scan from right to left.)
  3. If i < j then exchange *a* and *b;*
  4. Repeat step from 1 to 3 until j <= i;
  5. If from < j then recursive call *quick_sort*(from,(from + j)/2, j);
  6. If i < to then recursive call *quick_sort*(i, (i+ to)/2, to);

- **Basic algorithms**
  - quick sorting

The center element is 5.

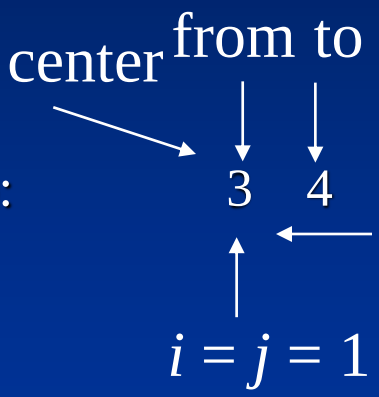


- **Basic algorithms**
  - quick sorting



6th step: 1

The sequence contains only one element, no sorting.



7th step:

The center element is 4.

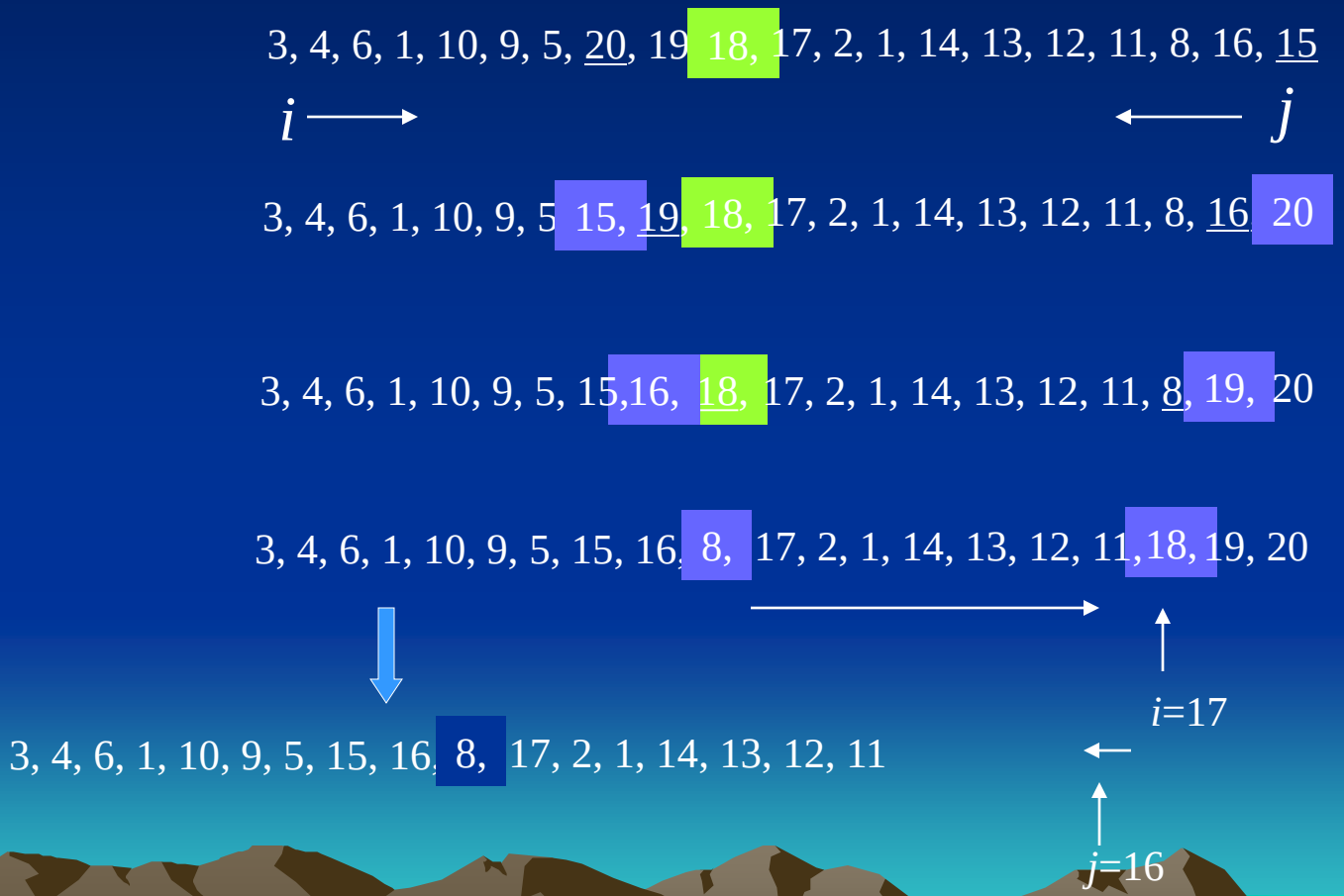
8th step: 4

The sequence contains only one element, no sorting.

1 2 3 4 5

- **Basic algorithms**
  - quick sorting

The center element is 18.

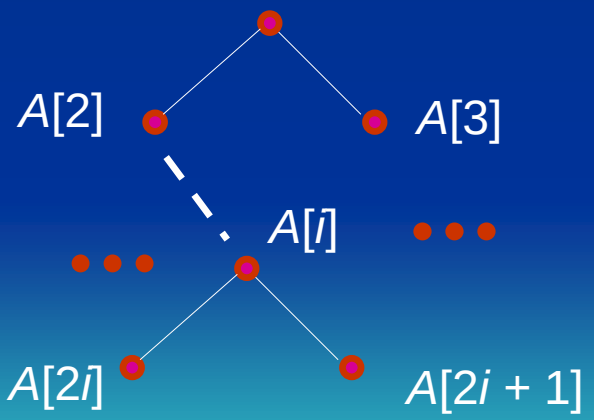


Heapsort

- Combines the better attributes of merge sort and insertion sort.
  - Like merge sort, but unlike insertion sort, running time is $O(n \lg n)$.
  - Like insertion sort, but unlike merge sort, sorts in place.
- Introduces an algorithm design technique
  - Create data structure (*heap*) to manage information during the execution of an algorithm.
- The *heap* has other applications beside sorting.
  - Priority Queues

Data Structure Binary Heap

- Array viewed as a nearly complete binary tree.
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest.)
- Map from array elements to tree nodes and vice versa
  - Root – $A[1]$, Left[Root] – $A[2]$, Right[Root] – $A[3]$
  - Left[i] – $A[2i]$
  - Right[i] – $A[2i+1]$
  - Parent[i] – $A[\lfloor i/2 \rfloor]$

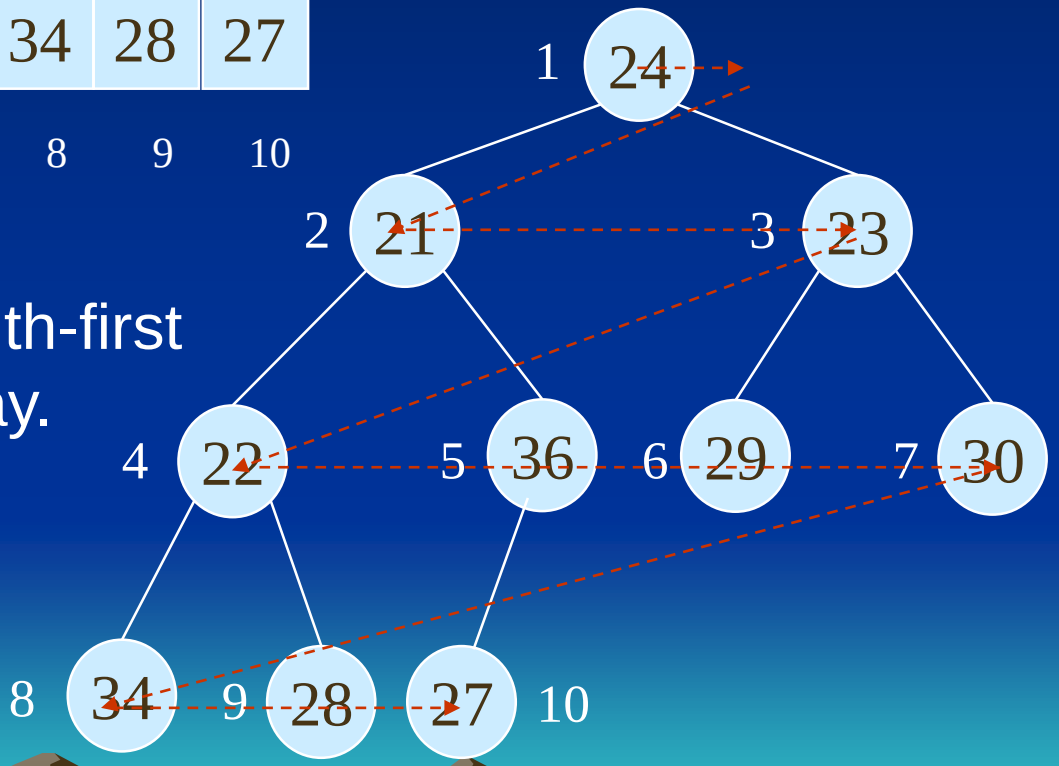


Data Structure Binary Heap

- length[*A*] – number of elements in array *A.*
- heap-size[*A*] – number of elements in heap stored in *A.*
  - heap-size[*A*] ≤ length[*A*]

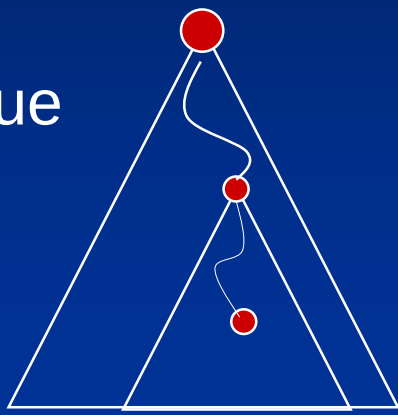
24	21	23	22	36	29	30	34	28	27
1	2	3	4	5	6	7	8	9	10

Searching the tree in breadth-first fashion, we will get the array.



Heap Property (Max and Min)

- Max-Heap
  - For every node excluding the root, the value stored in that node is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at the subtree's root.



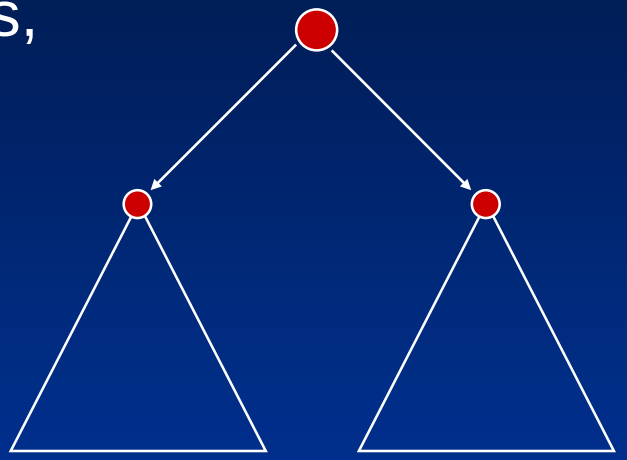
- Min-Heap
  - For every node excluding the root, the value stored in that node is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at the subtree's root

Heaps in Sorting

- Use max-heaps for sorting.
- The array representation of a max-heap is not sorted.
- Steps in sorting
  - (i) Convert the given array of size *n* to a max-heap (*BuildMaxHeap*)
  - (ii) Swap the first and last elements of the array.
    - Now, the largest element is in the last position – where it belongs.
    - That leaves *n* – 1 elements to be placed in their appropriate locations.
    - However, the array of first *n* – 1 elements is no longer a max-heap.
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (*MaxHeapify*)
    - Repeat step (ii) until the array is sorted.

Maintaining the heap property

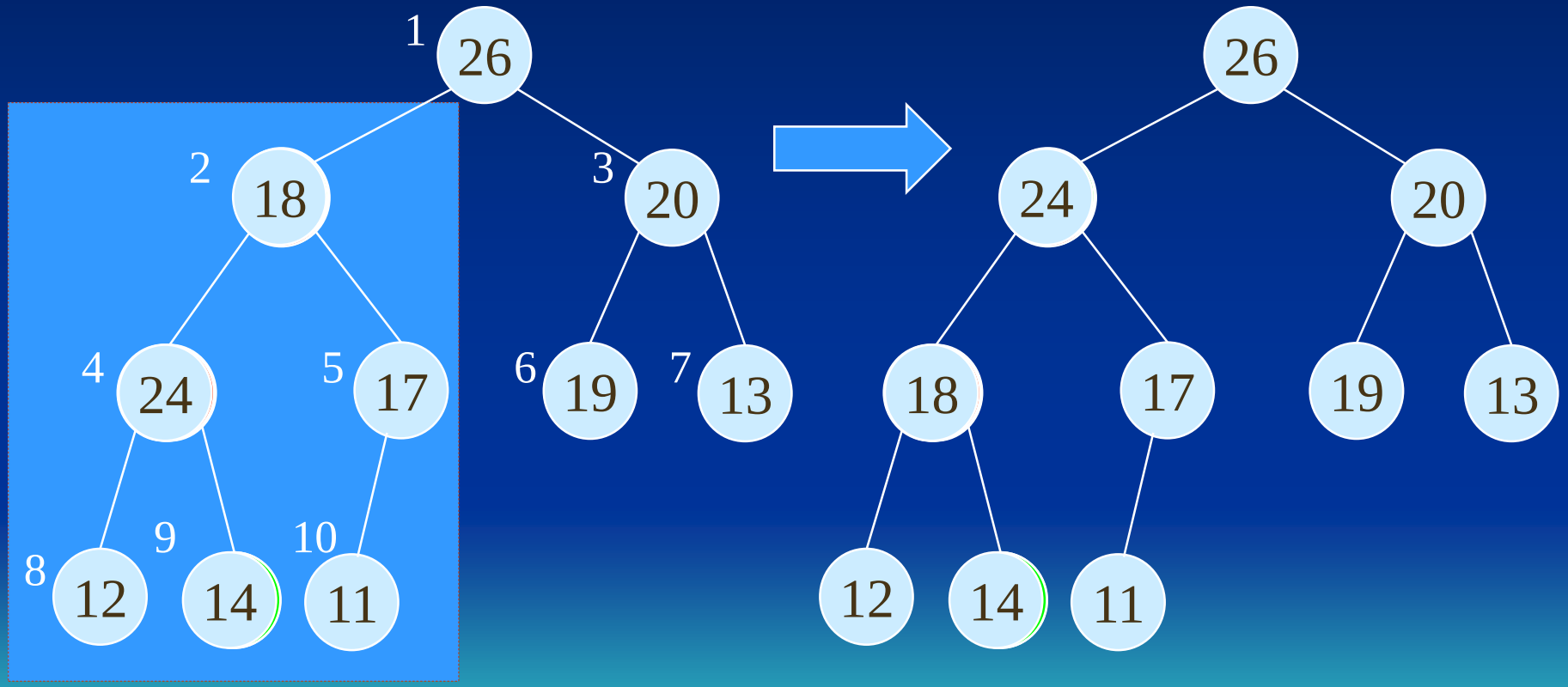
- Suppose two subtrees are max-heaps, but the root violates the max-heap property.



- Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  - May lead to the subtree at the child not being a max heap.
- Recursively fix the children until all of them satisfy the max-heap property.

MaxHeapify – Example

MaxHeapify(*A*, 2)



Procedure MaxHeapify

MaxHeapify(*A*, *i*)

1. $l \leftarrow$ left(i) (* $A[l]$ is the left child of $A[i]$.*)
2. $r \leftarrow$ right(i)
3. **if** $l \leq$ *heap-size*[A] and $A[l] > A[i]$
4. **then** *largest* $\leftarrow l$
5. **else** *largest* $\leftarrow i$
6. **if** $r \leq$ *heap-size*[A] **and** $A[r] > A[$*largest*$]$
7. **then** *largest* $\leftarrow r$
8. **if** *largest* $\neq i$
9. **then** exchange $A[i] \leftrightarrow A[$*largest*$]$
10. *MaxHeapify*(*A*, *largest*)

Assumption:
Left(*i*) and Right(*i*) are max-heaps.

$A[$*largest*$]$ must be the largest among $A[i]$, $A[l]$ and $A[r]$.

Building a heap

- Use *MaxHeapify* to convert an array *A* into a max-heap.
- How?
- Call MaxHeapify on each element in a bottom-up manner.

BuildMaxHeap(*A*)

1. *heap-size*[*A*] ← *length*[*A*]
2. **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 (**A*[*length*[*A*]/2⌋ +1],
3. **do** *MaxHeapify*(*A*, *i*) *A*[*length*[*A*]/2⌋ +2],
   … are leaf nodes.*)

Heapsort(*A*)

HeapSort(*A*)

1. *BuildMaxHeap*(*A*)
2. **for** *i* ← *length*[*A*] **downto** 2
3. **do** exchange *A*[1] ↔ *A*[*i*]
4. *heap-size*[*A*] ← *heap-size*[*A*] – 1
5. *MaxHeapify*(*A*, 1)

Time complexity: O(*n*·log*n*)

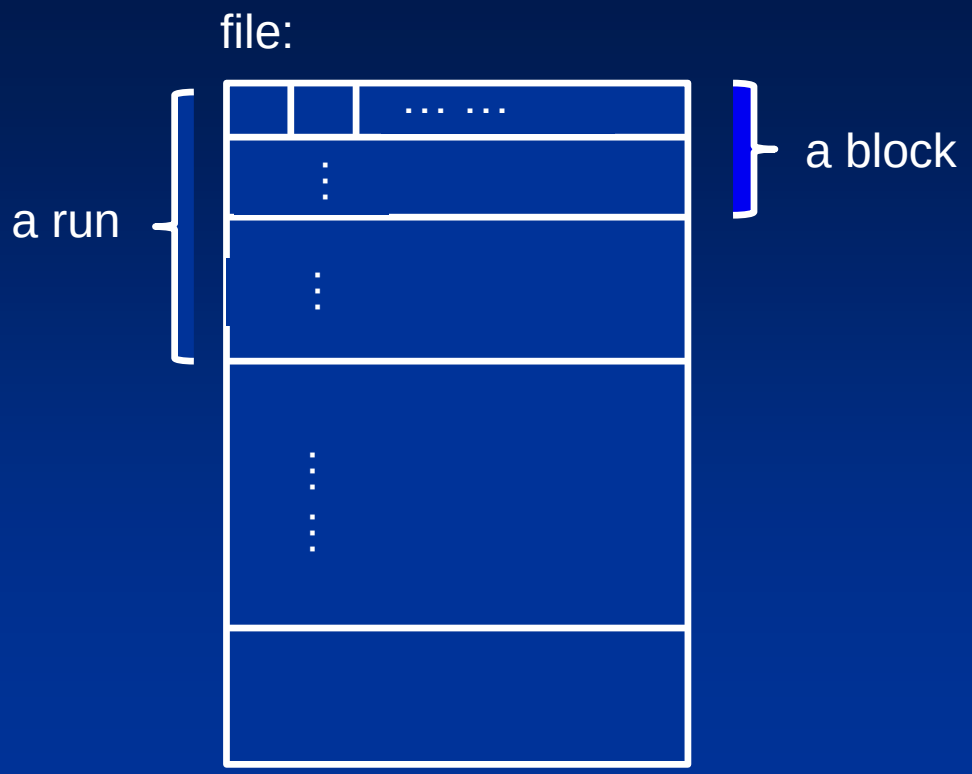
- **Basic algorithms**
  - External sorting method:

    Several parameters:

    b - number of file blocks

    n_R - number of initial runs

    n_B - available buffer space



$$n_R = \lceil b / n_B \rceil$$

Example: n_B = 5 blocks, b = 80 blocks,

n_R = 16 initial runs

d_M - number of runs that can be merged together in each pass

- **Basic algorithms**
  - External sorting method:

```
set     i ← 1;
        j ← b;          /*size of the file in blocks*/
        k ← nB;         /*size of buffer in blocks*/
        m ← ⌈j/k⌉;      /*number of runs*/
/*sort phase*/
while (i ≤ m)
        do {            read next k blocks of the file into the buffer or if there are less tha
        blocks remaining then read in the remaining blocks;
            sort the records in the buffer and write as a temporary subfile;
            i ← i +1;
            }
```

- **Basic algorithms**
  - External sorting method:

/*merge phase: merge subfiles until only 1 remains*/

set $i \leftarrow 1$;

$p \leftarrow \lceil \log_{k-1} m \rceil$; /*$p$ is the number of passes for the merging phase*/

$j \leftarrow m$; /*number of runs*/

while ($i \leq p$)

do {$n \leftarrow 1$;

$q \leftarrow \lceil j/k\text{-}1 \rceil$; /*$q$ is the number of subfiles to write in this pass*/

while ($n \leq q$) do

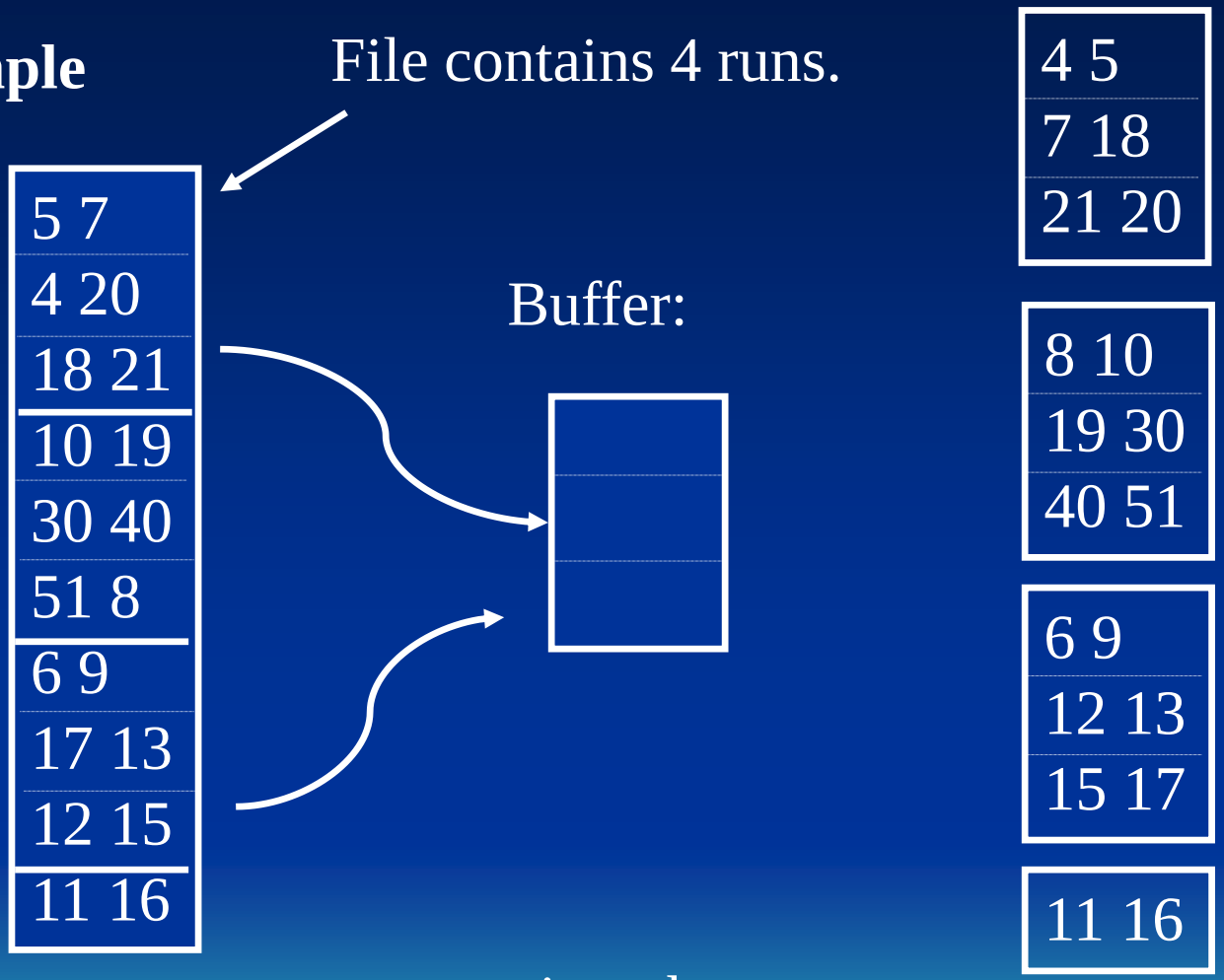
{read next k-1 subfiles or remaining subfiles (from previous pass) one block at a time;

merge and write as new subfile;

$n \leftarrow n+1$;}

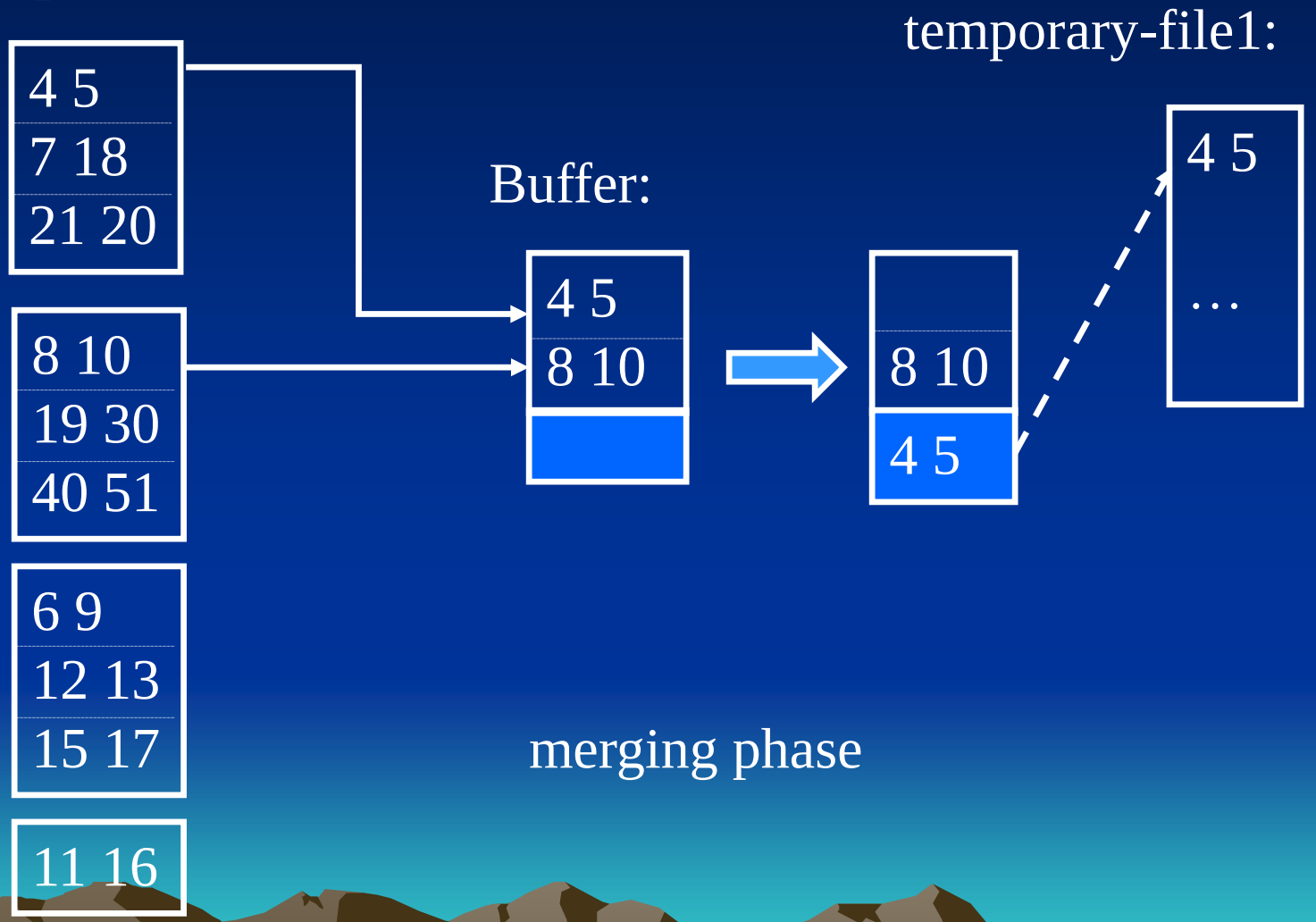
$j \leftarrow q$; $i \leftarrow i + 1$;}

- **Example**

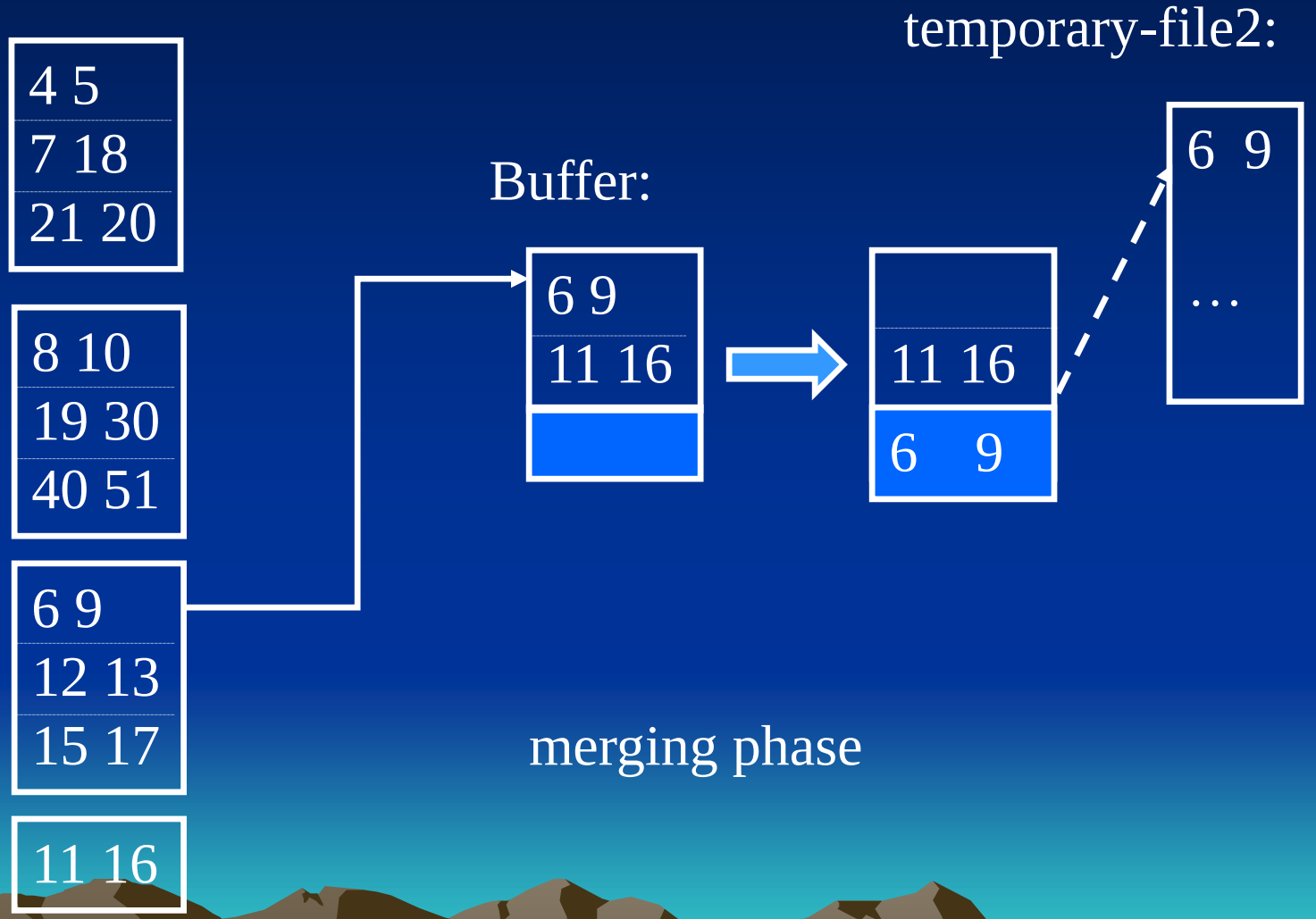


sorting phase

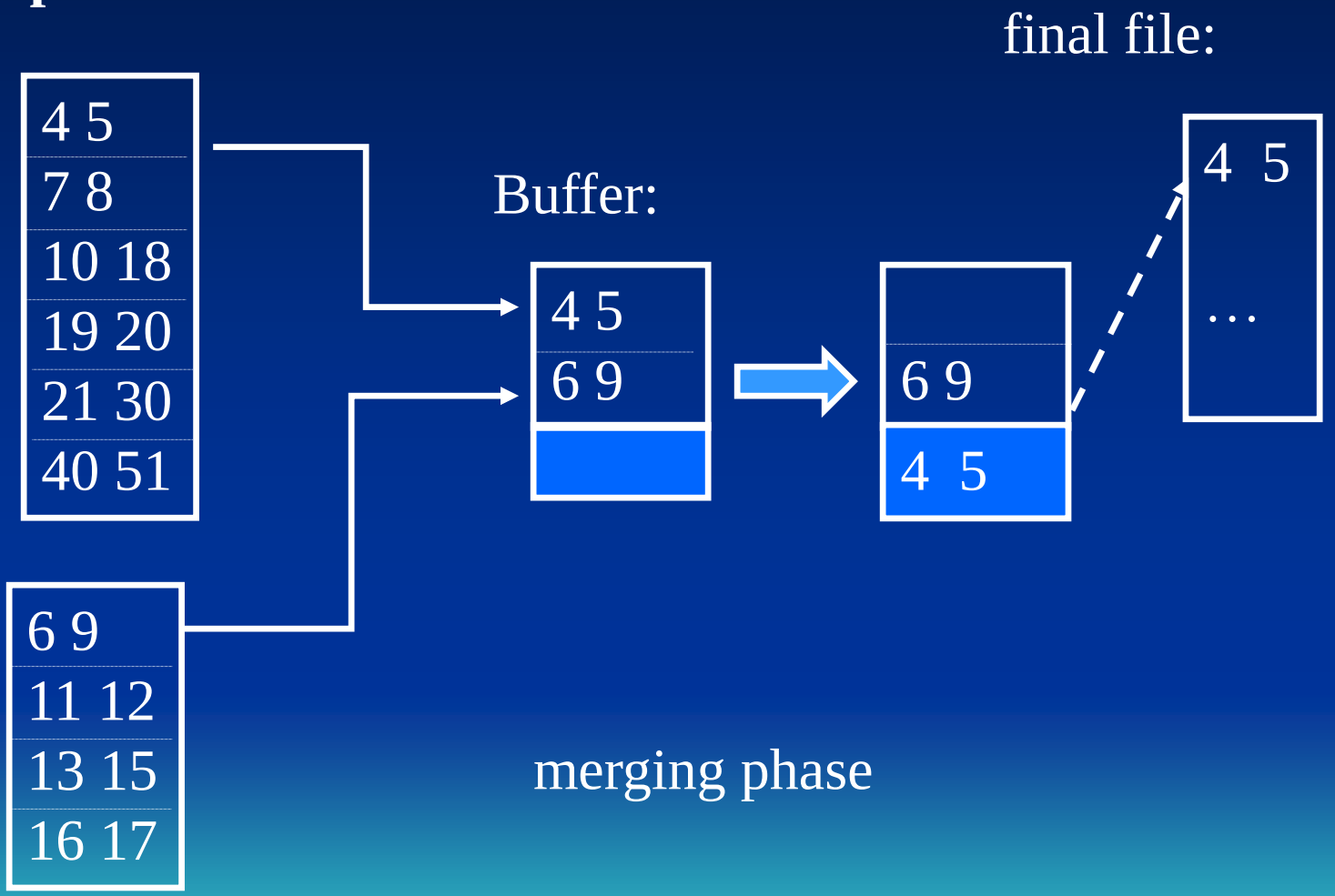
- **Example**


temporary-file1:
4 5
7 18
21 20
8 10
19 30
40 51
6 9
12 13
15 17
11 16
Buffer:
4 5
8 10
8 10
4 5
4 5
…
merging phase

- **Example**


temporary-file2:
4 5
7 18
21 20
8 10
19 30
40 51
6 9
12 13
15 17
11 16
Buffer:
6 9
11 16
11 16
6 9
6 9
…
merging phase

- **Example**


final file:
4 5
7 8
10 18
19 20
21 30
40 51
Buffer:
4 5
6 9
6 9
4 5
4 5
…
6 9
11 12
13 15
16 17
merging phase

- **Basic algorithms**
  - SELECT operation

    Example:

    (op1): $\sigma_{ssn='123456789'}(EMPLOYEE)$

    (op2): $\sigma_{DNUMBER>5}(DEPARTMENT)$

    (op3): $\sigma_{DNO=5}(EMPLOYEE)$

    (op4): $\sigma_{DNO=5 \wedge SALARY>30000 \wedge SEX='F'}(EMPLOYEE)$

    (op5): $\sigma_{ESSN='123456789' \wedge PNO=10}(WORKS\_ON)$

- **Basic algorithms**
  - Search method for simple selection
    - file scan

      linear search (brute force)

      binary search
    - index scan

      using a primary index (or hash key)

      using a primary index to retrieve multiple records

      using a clustering index to retrieve multiple records

- **Basic algorithms**
  - JOIN operation (two-way join)

$$R \underset{A=B}{\bowtie} S$$

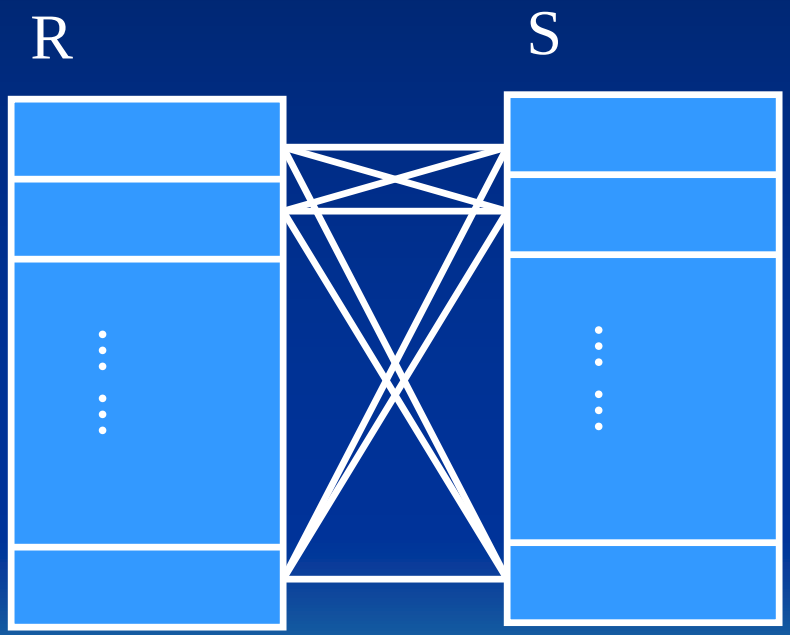
Example:

$$EMPLOYEE \underset{DNO=DNUMBER}{\bowtie} DEPARTMENT$$

$$DEPARTMENT \underset{MGRSSN=SSN}{\bowtie} EMPLOYEE$$

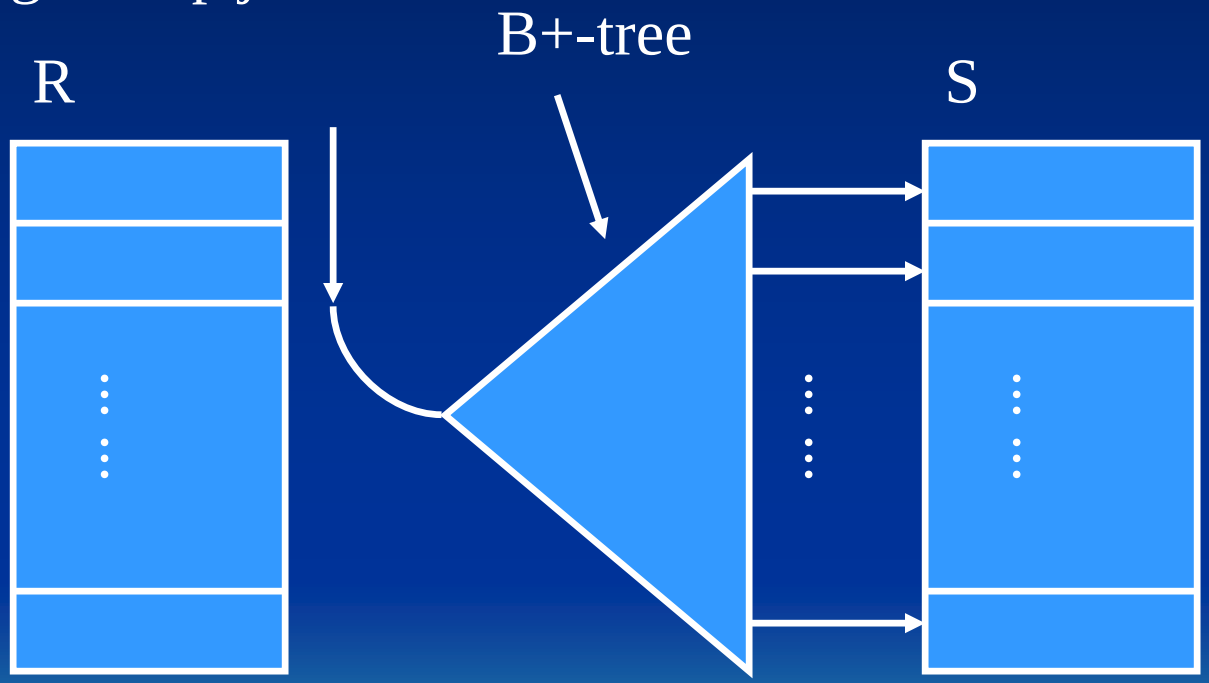
- **Basic algorithms**
  - Methods for implementing JOINs

    Nested-loop join:



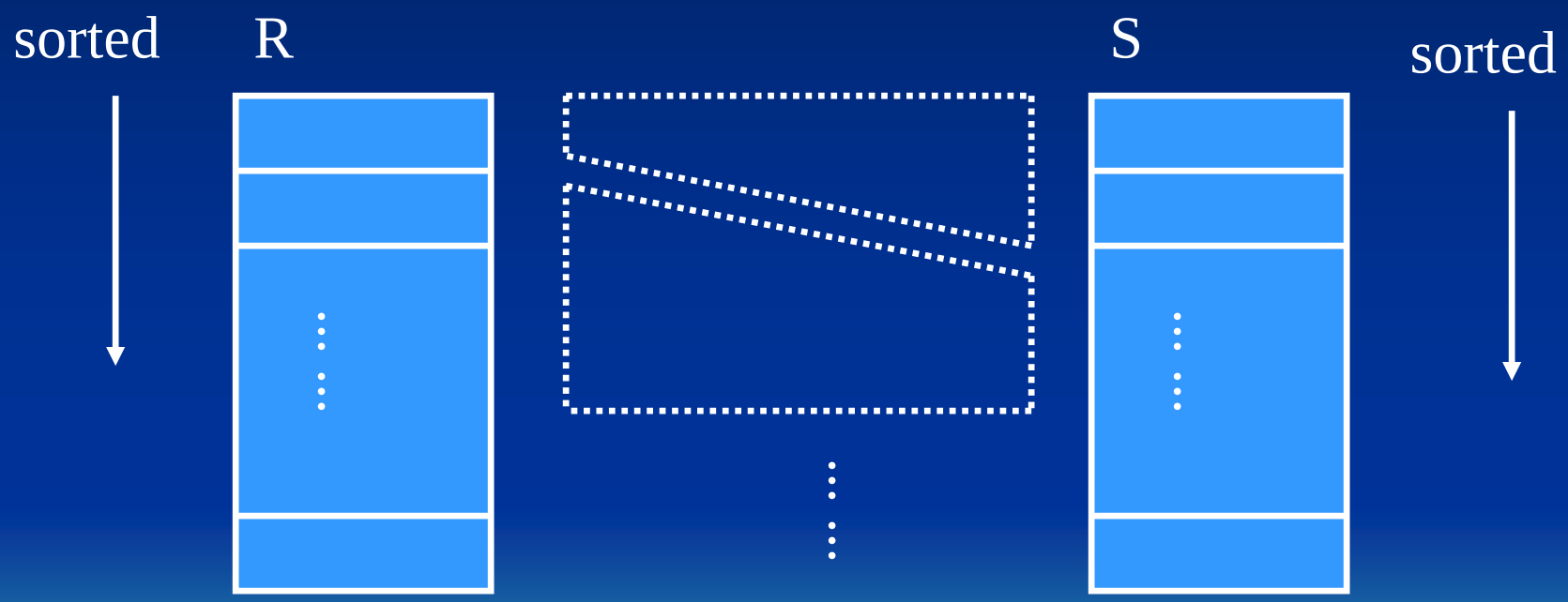
- **Basic algorithms**
  - Methods for implementing JOINs

    Single-loop join:



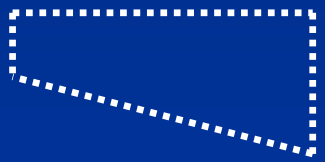
- **Basic algorithms**
  - Methods for implementing JOINs

    Sort-merge join:



```
set i ← 1; j ← 1;
while (i ≤ n) and (j ≤ m)
do {if R(i)[A] > S(j)[B] then set j ← j +1
      else R(i)[A] < S(j)[B] then set i ← i +1
      else {/* R(i)[A] = S(j)[B], so we output a matched tuple*/
              set k ← i;
              while (k  ≤ n) and (R(k)[A] = S(j)[B])
                  do {set l ← j;
                        while (l  ≤ m) and (R(i)[A] = S(l)[B])
                        do {output a combined tuple; l ← l + 1;}
                        set k ← k +1;}
              set i ← k, j ← l;}}
```

R(*i*)[A] S(*j*)[B]



- **Basic algorithms**
  - PROJECT operation

$$\pi_{<\text{Attribute list}>}(R)$$

Example:

$$\pi_{\text{FNAME, LNAME, SEX}}(\text{EMPLOYEE})$$

Algorithm:

1. Construct a table according to <Attribute list> of R.
2. Do the duplication elimination.

- **Heuristics for query optimization**

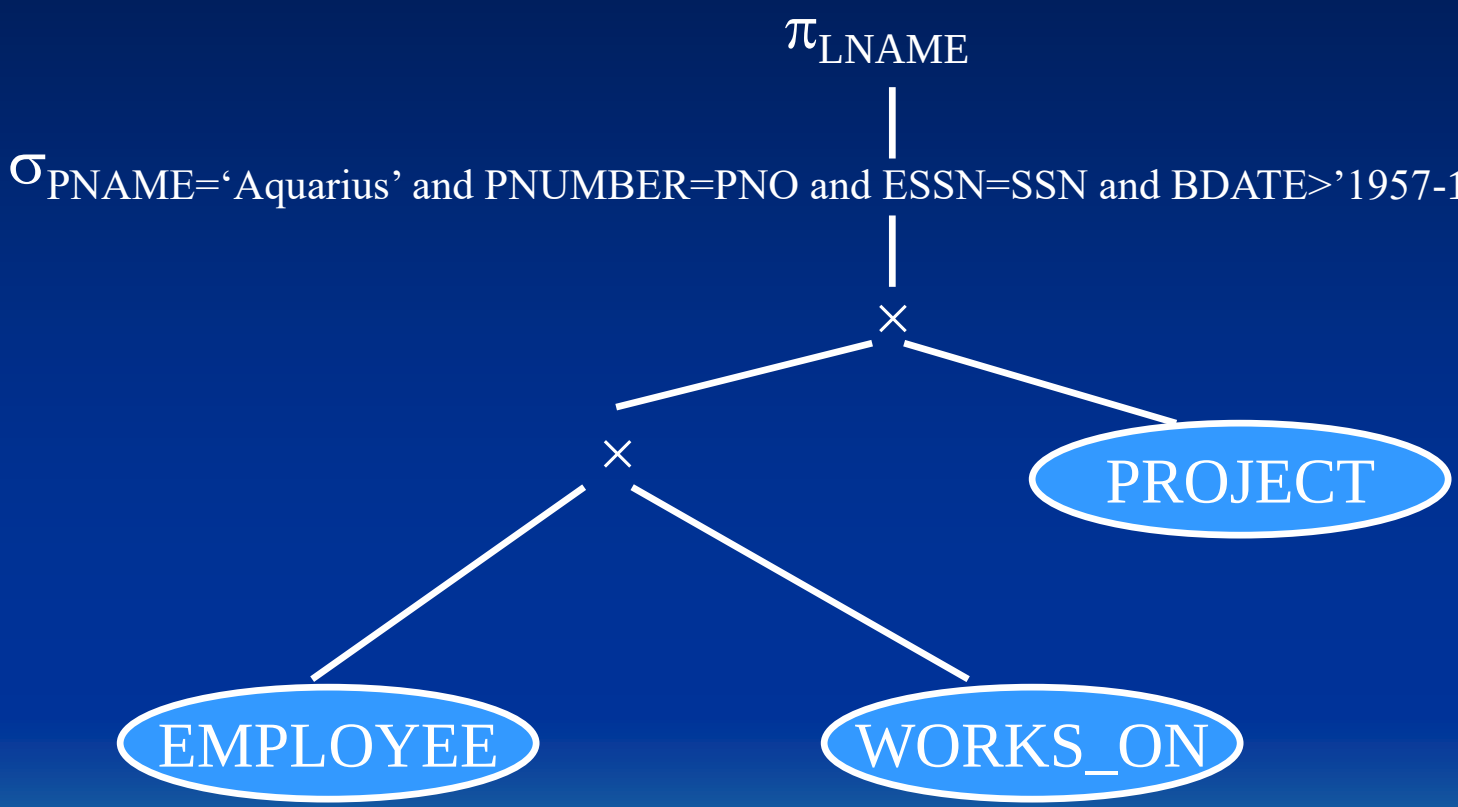
  - Query trees and query graphs
  - Heuristic optimization of query trees
  - General transformation rules for relational algebra operations
  - Outline of a heuristic algebraic optimization algorithm

- **Heuristic optimization of query trees**
  - Generate an initial query tree for a query
  - Using the rules for equivalence to transform the query tree in such a way that a transformed tree is more efficient than the previous one.

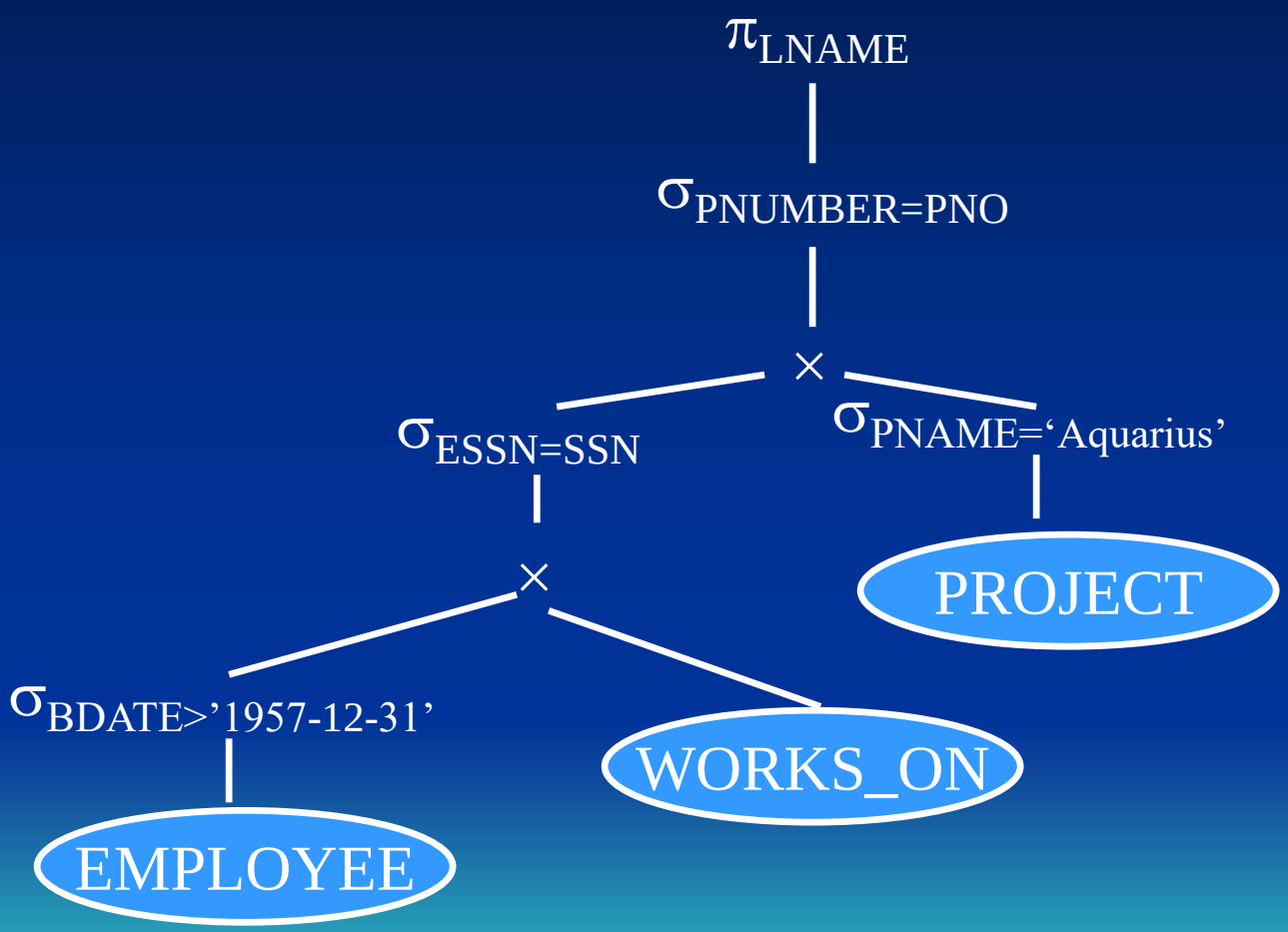
Example:

```
Q:  SELECT   LNAME
    FROM     EMPLOYEE, WORKS_ON, PROJECT
    WHERE    PNAME='Aquarius' and PNUMBER=PNO
             and    ESSN =SSN
             and    BDATE>'1957-12-31'
```

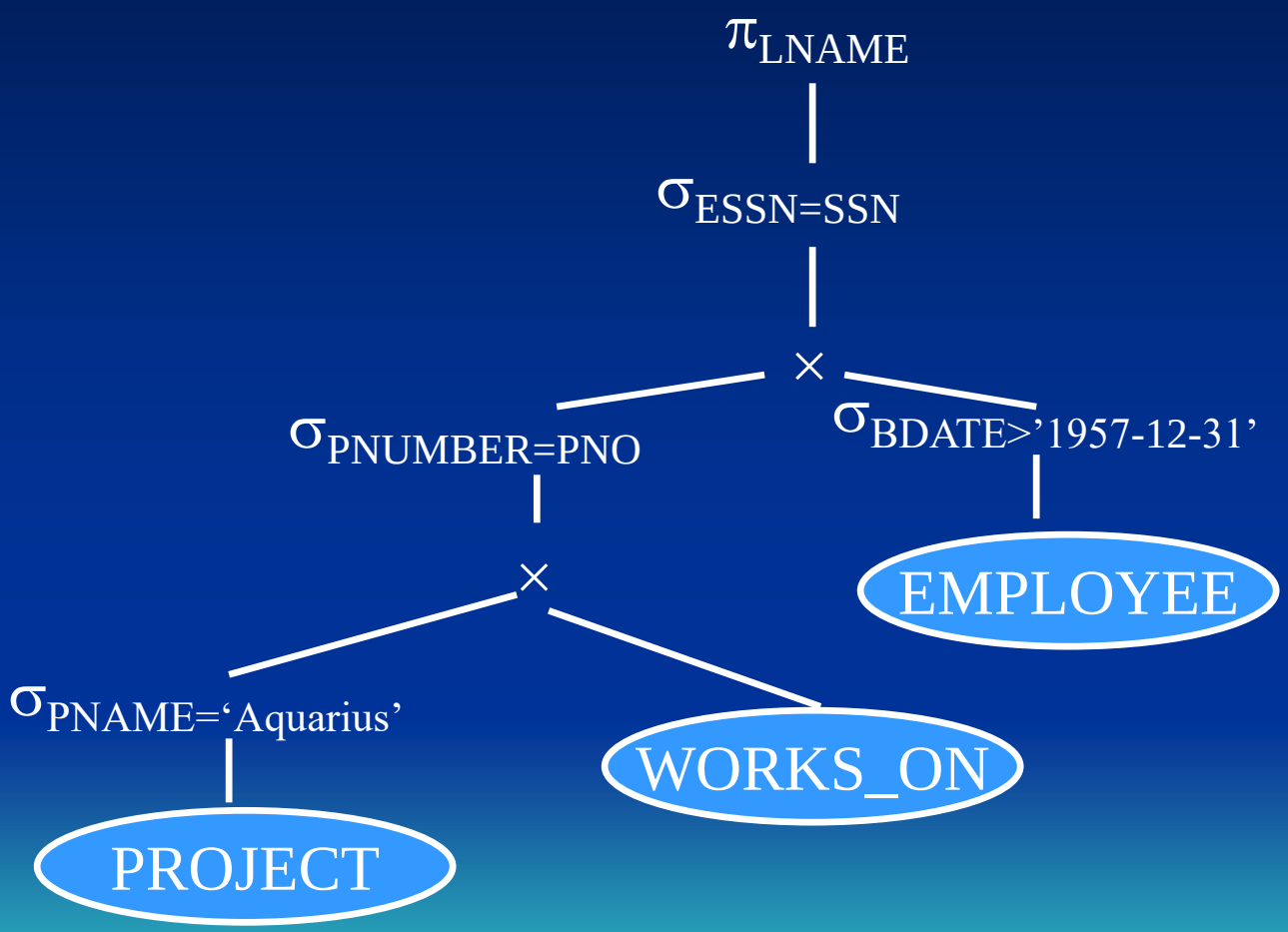
Initial query tree:



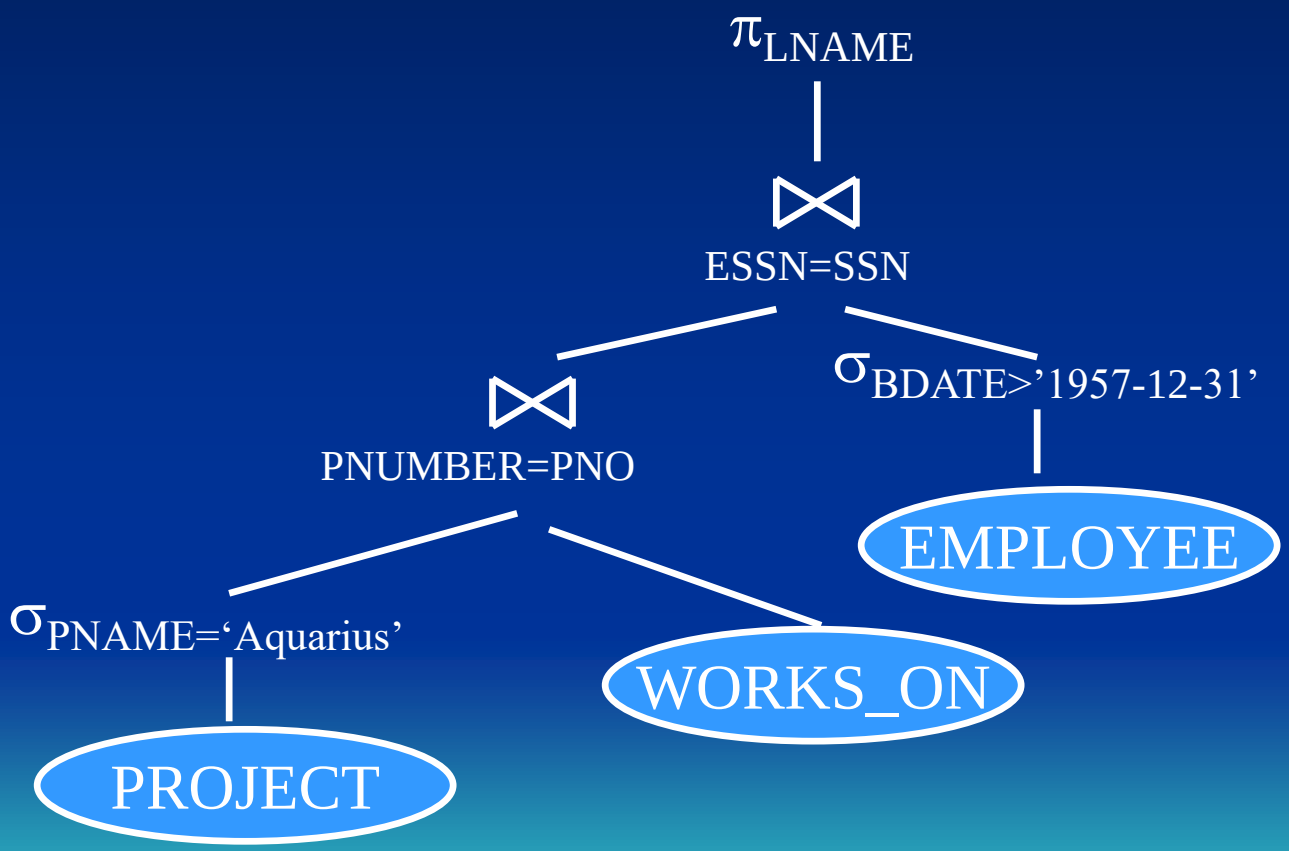
First transformation:



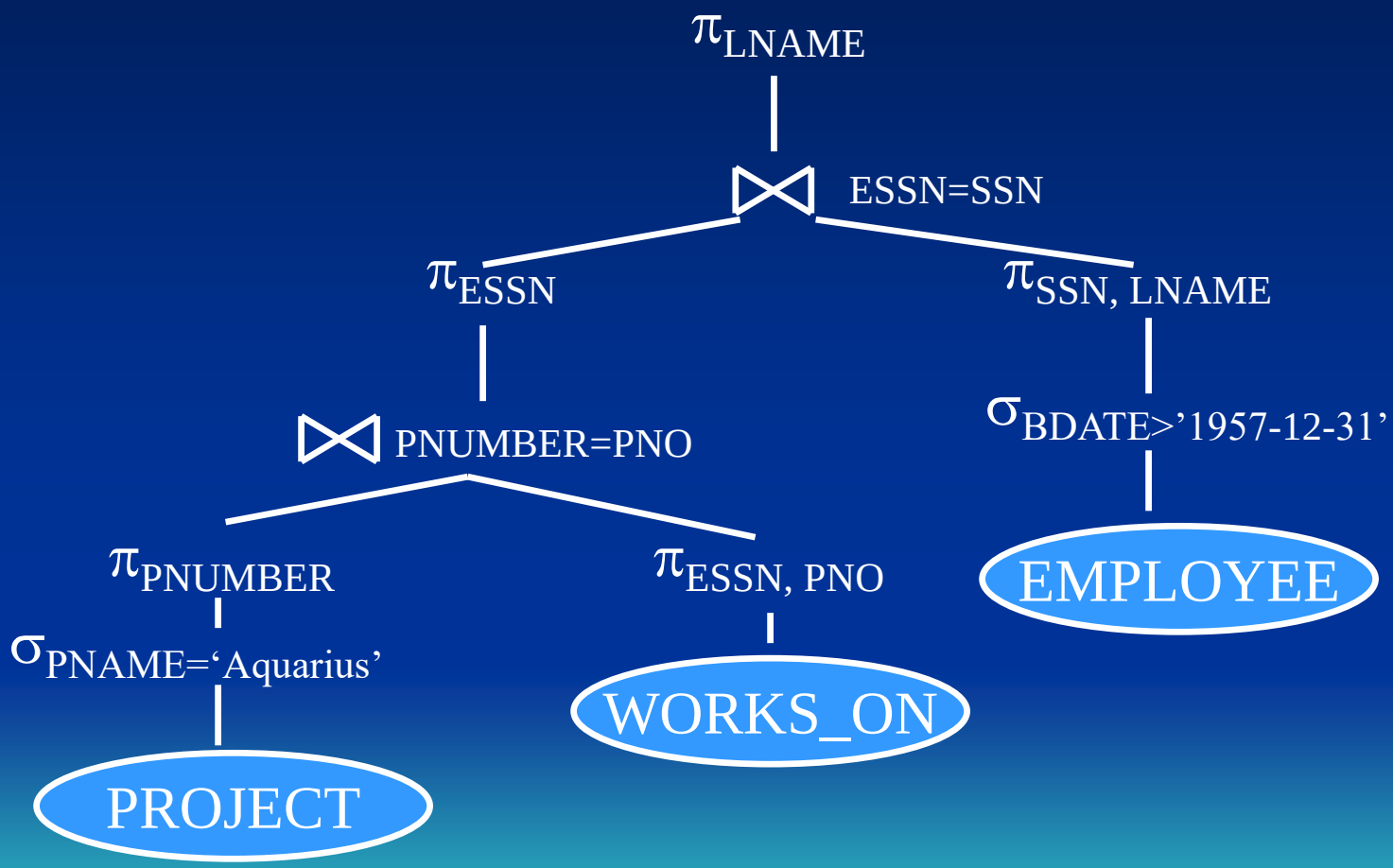
Second transformation:



Third transformation:



Fourth transformation:



- **General transformation rules for relational algebra operations** (altogether 12 rules)
  1. Cascade of σ: A conjunctive selection condition can be broken into a cascade (i.e., a sequence) of individual σ operations:

     $\sigma_{c1 \text{ and } c2 \text{ and } \ldots \text{ And } cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$

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

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

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

     $\pi_{list1}(\pi_{list2}(\ldots(\pi_{listn}(R))\ldots)) \equiv \pi_{list1}(R)$,
     where $list1 \subseteq list2 \subseteq \ldots \subseteq listn$.

- **General transformation rules for relational algebra operations** (altogether 12 rules)

4. Commuting σ with π: If the selection condition c involves only those attributes A1, …, A*n* in the projection list, the two operations can be commuted:

$$\pi_{A1, \ldots, An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1, \ldots, An}(R))$$

5. Commutativity of $\bowtie$ (and $\times$): The $\bowtie$ operation is commutative, as is the $\times$ operation:

$$R \bowtie_c S \equiv S \bowtie_c R$$

$$R \times S \equiv S \times R$$

- **General transformation rules for relational algebra operations** (altogether 12 rules)

  6. Commuting σ with $\bowtie$ (or $\times$): If all the attributes in the selection condition c involves only the attributes of one of the relations being joined - say, R - the two operations can be commuted as follows:

$$\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$$

If c is of the form: c1 and c2, and c1 involves only the attributes of R and c2 involves only the attributes of S, then:

$$\sigma_c(R \bowtie S) \equiv (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$$

- **General transformation rules for relational algebra operations** (altogether 12 rules)

7. Commuting π with $\bowtie$ (or $\times$): Suppose that the projection list is L = {A1, …, A*n*, B1, …, B*m*}, where A1, …, A*n* in R and B1, …, B*m* in S. If the join condition c involves L, we have

$$\pi_{L}(R \bowtie_{C} S) \equiv (\pi_{A1, \ldots, An}(R)) \bowtie_{C} (\pi_{B1, \ldots, Bm}(S))$$

8. Commutativity of set operations: The set operation "$\cup$" and "$\cap$" are commutative, but "-" is not.

9. Associativity of $\bowtie$, $\times$, $\cup$ and $\cap$: These four operations are individually associative; i.e., if θ stands for any one of these four operations, we have:

$$(R \theta S) \theta T \equiv R \theta (S \theta T)$$

- **General transformation rules for relational algebra operations** (altogether 12 rules)

10. Commuting σ with set operations: The σ operation commutes with "∪", "∩" and "-". If θ stands for any one of these three operations, we have:

$$\sigma_c(R\ \theta\ S) \equiv \sigma_c(R)\ \theta\ \sigma_c(S)$$

11. The π operation commutes with ∪:

$$\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$$

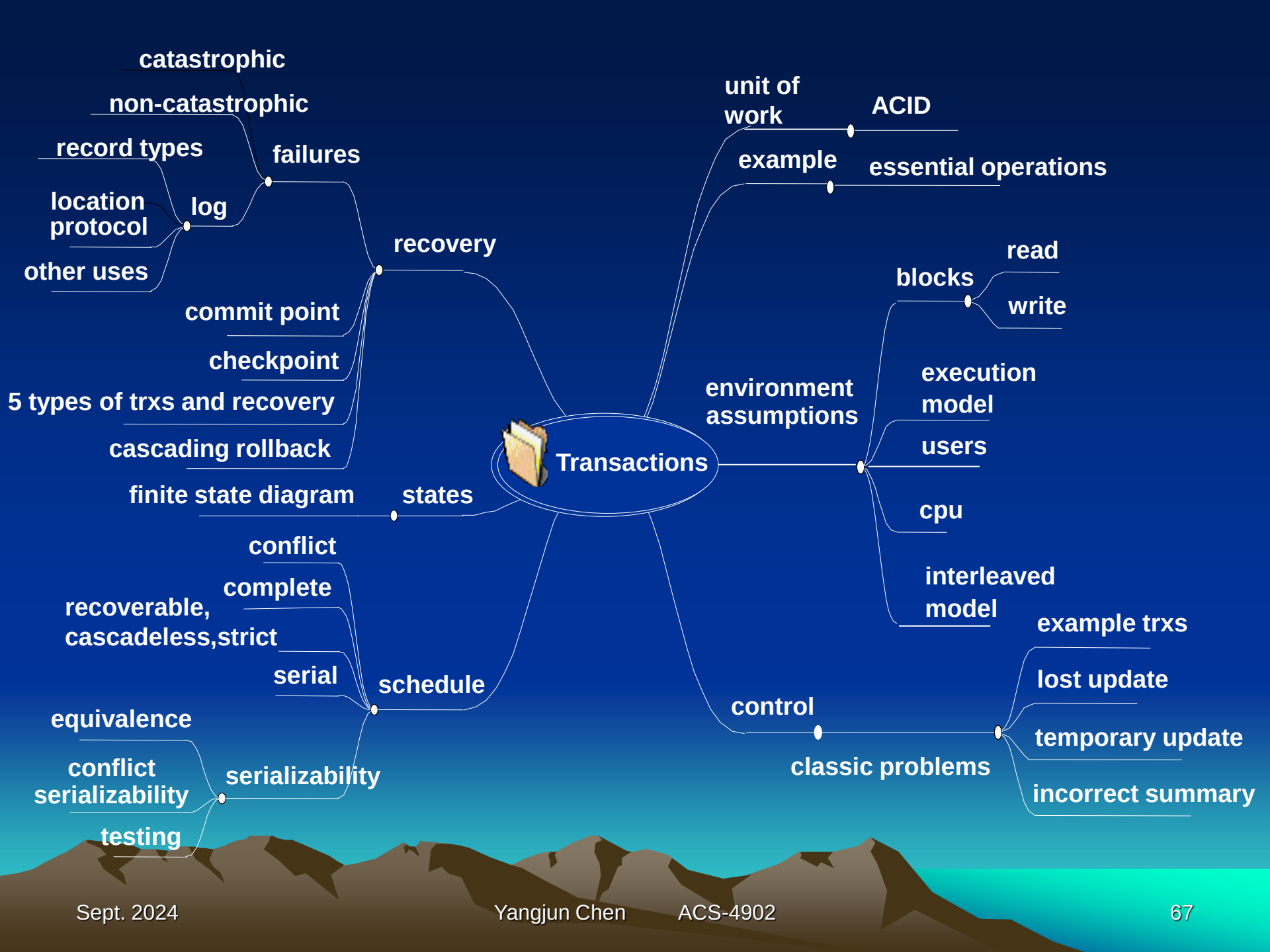
12. Converting a (σ, ×) sequence into ⋈: If the condition c of a σ that follows a × corresponds to a join condition, convert then (σ, ×) sequence into ⋈ as follows:

$$\sigma_c(R \times S) \equiv R \bowtie_c S$$

- **General transformation rules for relational algebra operations** (other rules for transformation)

  DeMorgan's rule:

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


Transactions
recovery
failures
catastrophic
non-catastrophic
log
record types
location
protocol
other uses
commit point
checkpoint
5 types of trxs and recovery
cascading rollback
unit of
work
ACID
example
essential operations
environment
assumptions
blocks
read
write
execution
model
users
cpu
interleaved
model
states
finite state diagram
schedule
conflict
complete
recoverable,
cascadeless,strict
serial
equivalence
serializability
conflict
serializability
testing
control
classic problems
example trxs
lost update
temporary update
incorrect summary

ACID principles:

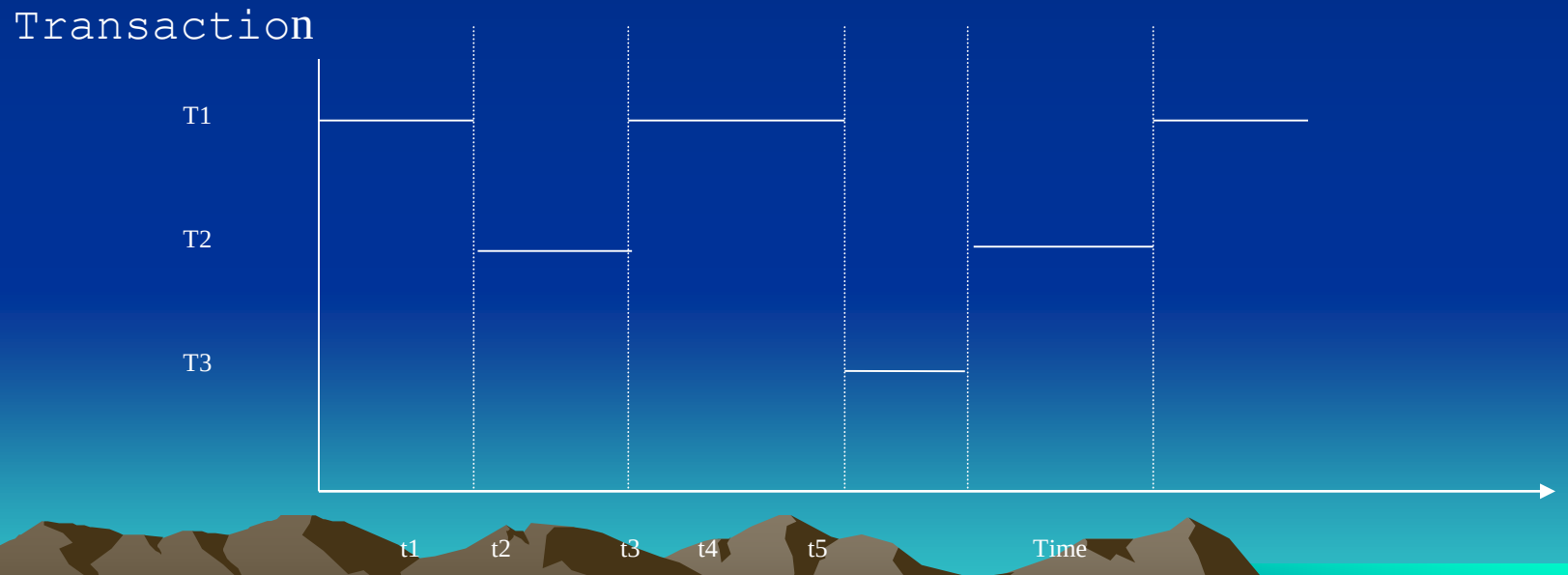
To generate faith in the computing system, a transaction will have the **ACID** properties:

- Atomic – a transaction is done in its entirety, or not at all
- Consistent – a transaction leaves the database in a correct state. This is generally up to the programmer to guarantee.
- Isolation – a transaction is isolated from other transactions so that there is not adverse inter-transaction interference
- Durable – once completed (committed) the result of the transaction is not lost.

Environment

Interleaved model of transaction execution

Several transactions, initiated by any number of users, are concurrently executing. Over a long enough time interval, several transactions may have executed without any of them completing.



Lost Update Problem

We have Transactions 1 and 2 concurrently executing in the system. They happen to interleave in the following way, which results in an incorrect value stored for flight X (try this for X=10, Y=12, N=5 and M=8).

Time	Transaction1	Transaction2
1	READ(X)	
2	X:=X-N	
3		READ(X)
4		X:=X+M
5	WRITE(X)	
6	READ(Y)	
7		WRITE(X)
8	Y:=Y+N	
9	WRITE(Y)	

Temporary Update Problem

We have transactions 1 and 2 running again. This time Transaction 1 terminates before it completes – it just stops, perhaps it tried to execute an illegal instruction or accessed memory outside its allocation. The important point is that it doesn't complete its unit of work; Transaction 2 reads 'dirty data' using a value derived from an inconsistent database state.

Time	Transaction1	Transaction2
1	READ(X)	
2	X:=X-N	
3	WRITE(X)	
4		READ(X)
5		X:=X+M
6		WRITE(X)
7	READ(Y)	
8	terminates!	

Transaction2 reads a 'dirty' value – one that Transaction1 has not committed to the database

X should be rolled back to what it was at Time2

Incorrect Summary Problem

Transactions 1 and 3 are executing and interleaved in such a way that the total number of seats calculated by transaction 3 is incorrect.

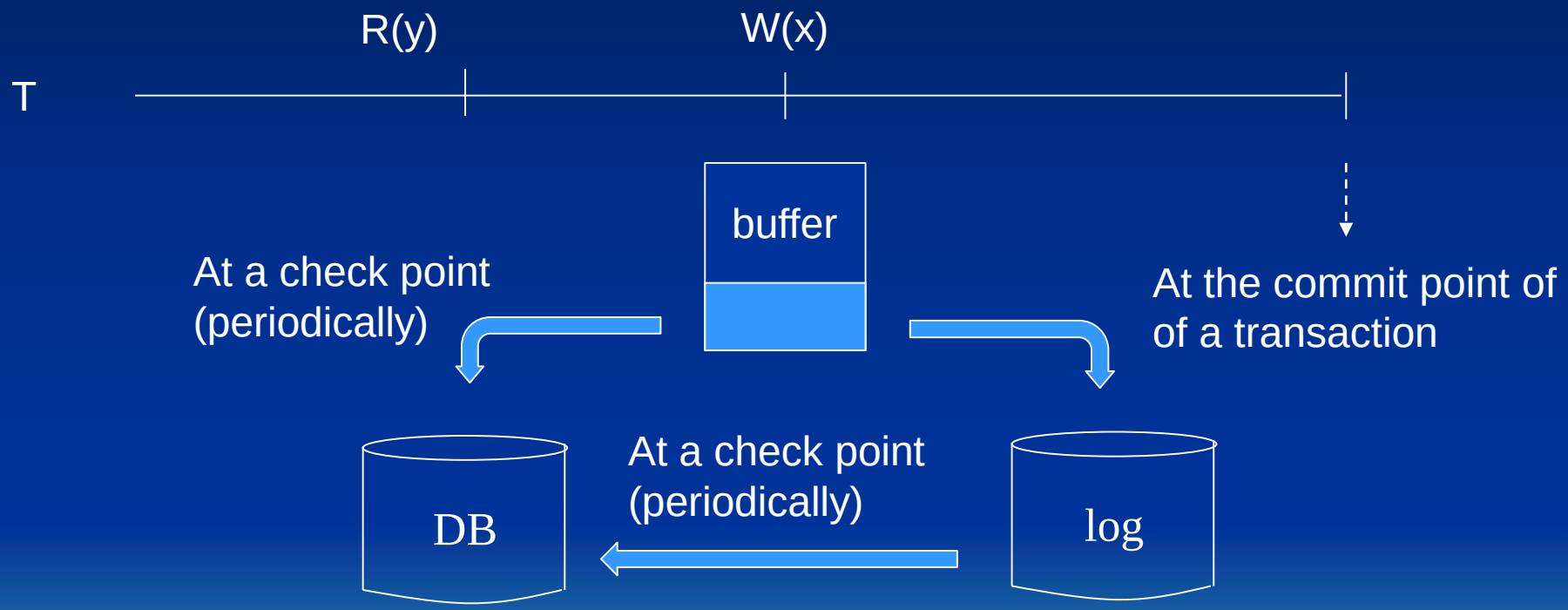
Time	Transaction1	Transaction3
1		SUM:=0
2	READ(X)	
3	X:=X-N	
4	WRITE(X)	
5		READ(X)
6		SUM:=SUM+X
7		READ(Y)
8		SUM:=SUM+Y
9	READ(Y)	
10	Y:=Y+N	
11	WRITE(Y)	
12		READ(Z)
13		SUM:=SUM+Z

Values obtained for X and Y will not be consistent

To allow for recovery we use a **Log**

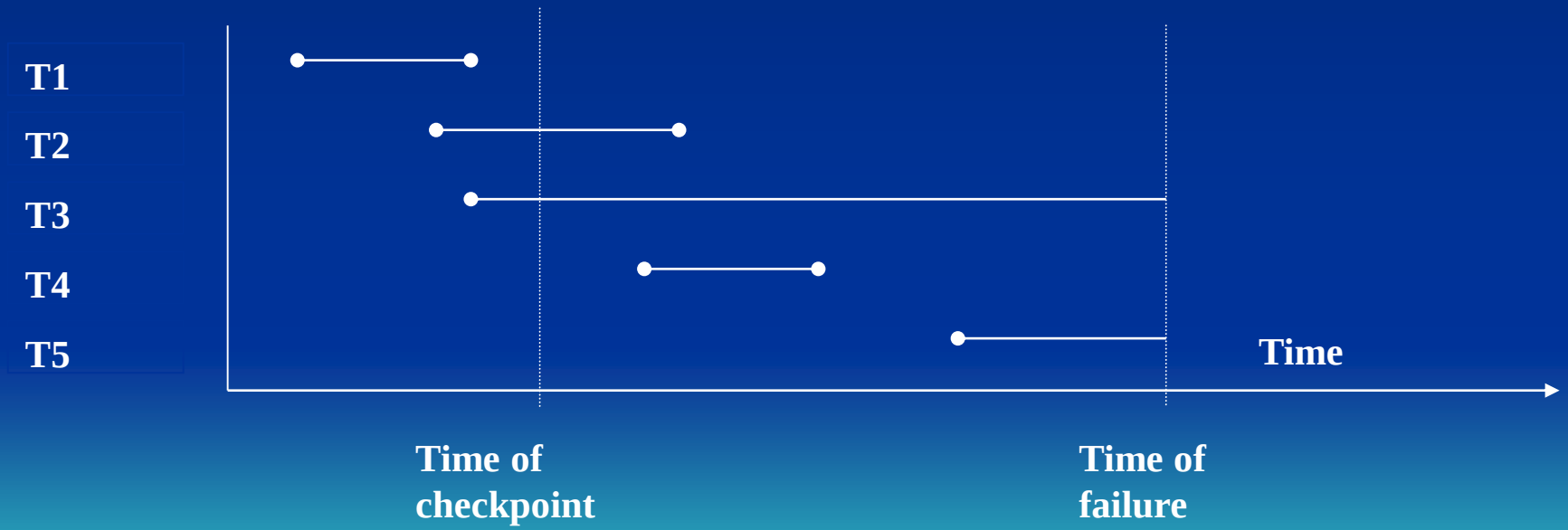
- The log contains several records for each transaction

1. [start_transaction, T] Indicates that transaction T has started execution.
2. [write_item, T, X, old_value, new_value] Indicates that transaction T has changed the value of database item X from old_value to new_value.
3. [Read_item, T, X] Indicates that transaction T has read the value of database item X.
4. [commit, T] Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort, T] Indicates that transaction T has been aborted.
6. [Checkpoint]: A checkpoint record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified.


R(y)
W(x)
T
buffer
At a check point
(periodically)
At the commit point of
of a transaction
DB
At a check point
(periodically)
log

Transaction types at recovery time

After a system crash some transactions will need to redone or undone.
Consider the five types below. Which need to be redone/undone after the crash?



Comparison of the three schedules

Recoverable

A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

Cascadeless

Every transaction in the schedule reads only items that were written by committed transaction.

Strict

a transaction can neither read nor write an item X until the last transaction that wrote X has committed or aborted.

increasing concurrency

decreasing difficulty of recovery

Serializability

- A schedule is said to be serializable if it is *equivalent* to a serial schedule
- What do we mean by equivalent?

  Text mentions *result* equivalence and *conflict* equivalence

Conflict equivalence

- Two schedules are said to be *conflict equivalent* if the ordering of any two conflicting operations is the same in both schedules
- Recall

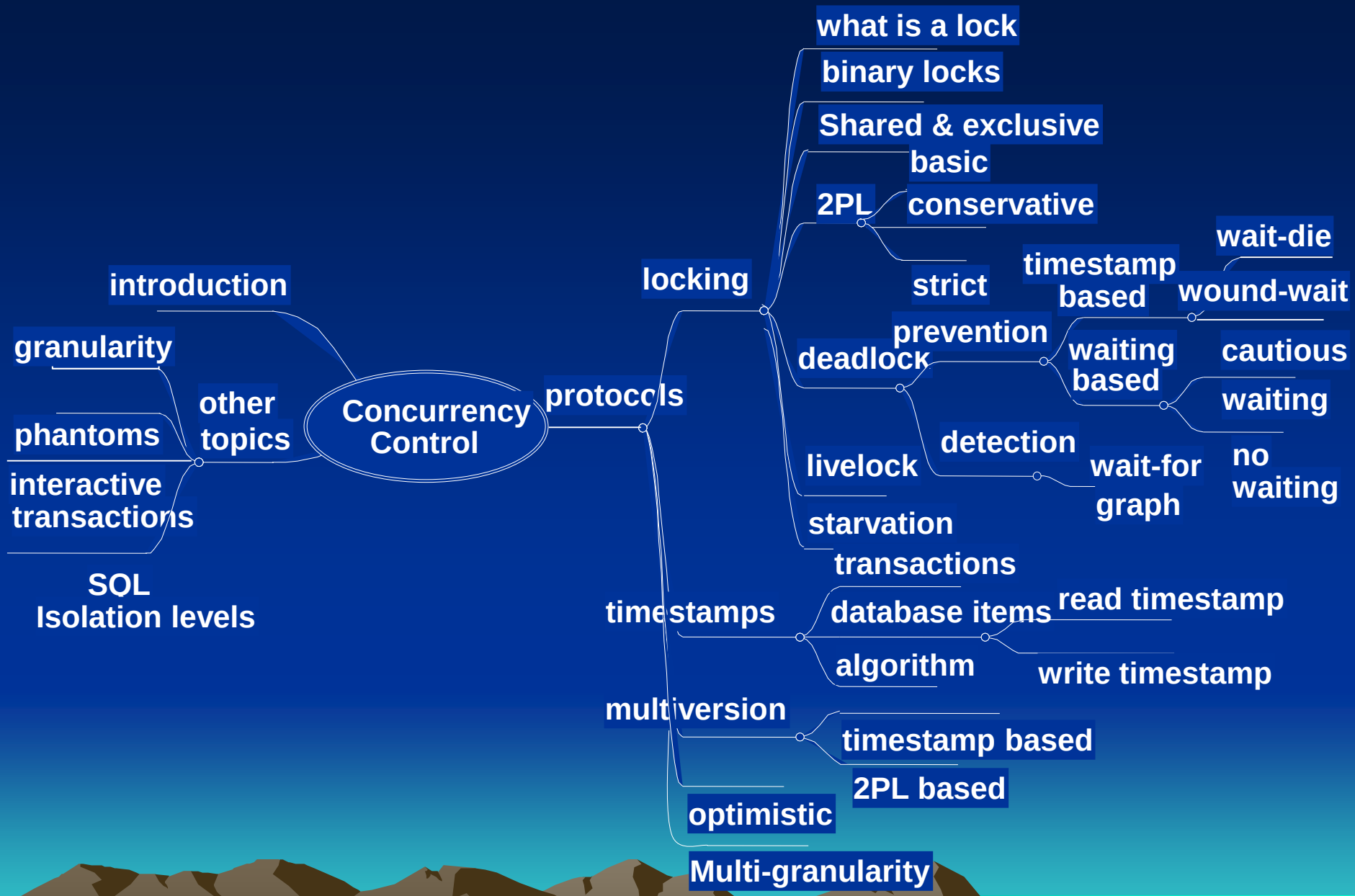
  Two operations *conflict* if they belong to two different transactions, are accessing the same data item X and one of the operations is a WRITE

Conflict Serializability

A schedule S is conflict serializable if it is conflict equivalent to some serial schedule S'

Time	T1	T2
1	READ(X)	
2	X:=X-N	
3		READ(X)
4		X:=X+M
5	WRITE(X)	
6		
7	READ(Y)	
8		WRITE(X)
9		
10	Y:=Y+N	
11	WRITE(Y)	

The graph has a cycle!


Concurrency Control
introduction
granularity
other topics
phantoms
interactive transactions
SQL Isolation levels
protocols
locking
what is a lock
binary locks
Shared & exclusive
basic
2PL
conservative
strict
deadlock
prevention
wait-die
timestamp based
wound-wait
waiting based
cautious
waiting
detection
wait-for graph
no waiting
livelock
starvation
transactions
timestamps
database items
read timestamp
write timestamp
algorithm
multiversion
timestamp based
2PL based
optimistic
Multi-granularity

Locking

What is a lock?

A lock is a variable associated with a database item that describes the status of the database item with respect to database operations that can be applied to the database item.

Locks are managed by the Lock Manager within the DBMS

Database items that could be locked vary from a field value up to a whole database:

- field value in a row
- row
- block
- table
- database

See section on granularity

Binary Locks: operations

lock_item(X)

- used to gain exclusive access to item X

- if a transaction executes lock_item(X) then
  if lock(X)=0 then
    the lock is granted {lock(X) is set to 1} and the
    transaction can carry on
    {the transaction is said to hold a lock on X}
  otherwise
    the transaction is placed in a wait queue until
    lock_item(X) can be granted
    {i.e. until some other transaction unlocks X}

Binary Locks: operations

unlock_item(X)

- used to relinquish exclusive access to item X
- if a transaction executes unlock_item(X) then
  lock(X) is set to 0
  {note that this may enable some other blocked transaction to resume execution}

Shared and Exclusive Locks

Three operations:

- read_lock(X)
- write_lock(X)
- unlock(X)

'read' locks allow more than one transaction to access a data item

Each is an indivisible operation

Use a multiple-mode lock with three possible states

- read-locked — *also called share-locked*
- write-locked — *also called exclusive-locked*
- unlocked

Shared and Exclusive Locks: operations

read_lock(X)

- used to gain shared access to item X
- if a transaction executes read_lock(X) then
  if lock(X) is not "write_locked" then
    the lock is granted
    {lock(X) is set to "read_locked",
    the "no_of_readers" is incremented by 1},
    and the transaction can carry on
    {the transaction is said to hold a share lock on X}
  otherwise
    the transaction is placed in a wait queue until
    read_lock(X) can be granted
    {i.e. until some transaction relinquishes exclusive
    access to X}

Shared and Exclusive Locks: operations

write_lock(X)

- used to gain exclusive access to item X
- if a transaction executes write_lock(X) then
  if lock(X) is “unlocked” then
    the lock is granted {lock(X) is set to “write_locked”},
    and the transaction can carry on
    {the transaction is said to hold an exclusive lock on X}
  otherwise
    the transaction is placed in a wait queue until
    write_lock(X) can be granted
    {i.e. until all other transactions have relinquished their
    access rights to X - that could be a single “writer” or
    several “readers”}

Shared and Exclusive Locks: operations

unlock(X)

- used to relinquish access to item X
- if a transaction executes unlock(X) then

  if lock(X) is “read_locked” then

  decrement no_of_readers by 1

  if no_of_readers=0 then set lock(X) to “unlocked”

  otherwise

  set lock(X) to “unlocked”

  {note that setting lock(X) to “unlocked” may enable a blocked transaction to resume execution}

Shared and Exclusive Locks

locking protocol (rules); a transaction T

- must issue read_lock(X) before read-item(X)
- must issue write_lock(X) before write-item(X)
- must issue unlock(X) after all read_item(X) and write_item(X) operations are completed
- will not issue a read_lock(X) if it already holds a read or write lock on X *(can be relaxed, to be discussed)*
- will not issue a write_lock(X) if it already holds a read or write lock on X *(can be relaxed, to be discussed)*
- will not issue an unlock unless it already holds a read lock or write lock on X

Shared and Exclusive Locks (2PL)

Conversion of Locks

Recall a transaction T

- will not issue a read_lock(X) if it already holds a read or write lock on X

  Can permit a transaction to ***downgrade*** a lock from a write to a read lock

- will not issue a write lock(X) if it already holds a read or write lock on X

  Can permit a transaction to ***upgrade*** a lock on X from a read to a write lock if no other transaction holds a read lock on X

Shared and Exclusive Locks (2PL)

Two-phase locking: A transaction is said to follow the two-phase locking protocol if all locking operations (read-lock, write-lock) precede the first unlock operations in the transaction.

- previous protocols do not guarantee serializability
- Serializability is guaranteed if we enforce the two-phase locking protocol:

  all locks must be acquired before any locks are relinquished

- transactions will have a *growing* and a *shrinking* phase
- any downgrading of locks must occur in the shrinking phase
- any upgrading of locks must occur in the growing phase

Variations on 2PL

Basic 2PL

- previous protocol

Conservative 2PL

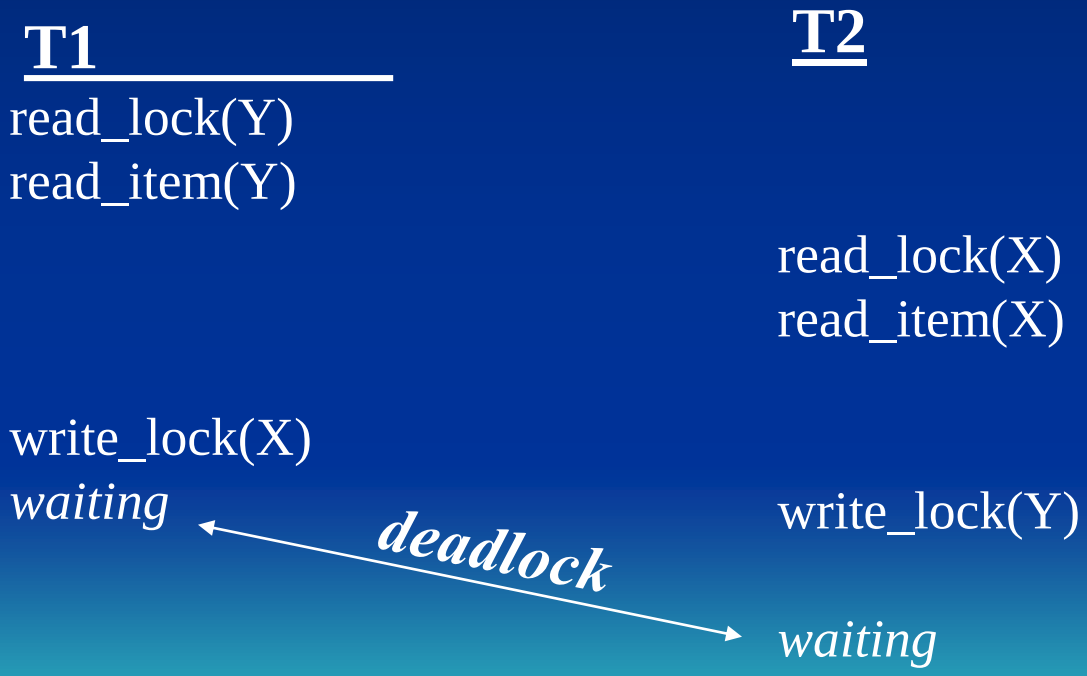
- transactions must lock all items prior to the transaction executing
- if any lock is not available then none are acquired - all must be available before execution can start
- free of deadlocks

Strict 2PL

- a transaction does not release any write-locks until after it commits or aborts
- **most popular** of these schemes
- recall strict schedule avoids cascading rollback

Deadlock

Deadlock occurs when two or more transactions are in a simultaneous wait state, each one waiting for one of the others to release a lock.



Deadlock Prevention

1. Conservative 2PL
2. Always locking in a predefined sequence
3. **Timestamp based**
4. **Waiting based**
5. Timeout based

Deadlock Prevention - Timestamp based

- Each transaction is assigned a timestamp (TS)

  If a transaction T1 starts before transaction T2,

  then TS(T1) < TS(T2); T1 is *older* than T2
- Two schemes:

  Wait-die

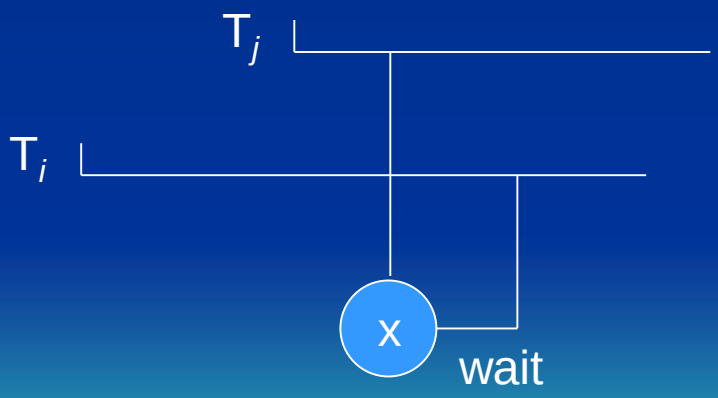
  Wound-wait
- Both schemes will cause aborts even though deadlock would not have occurred

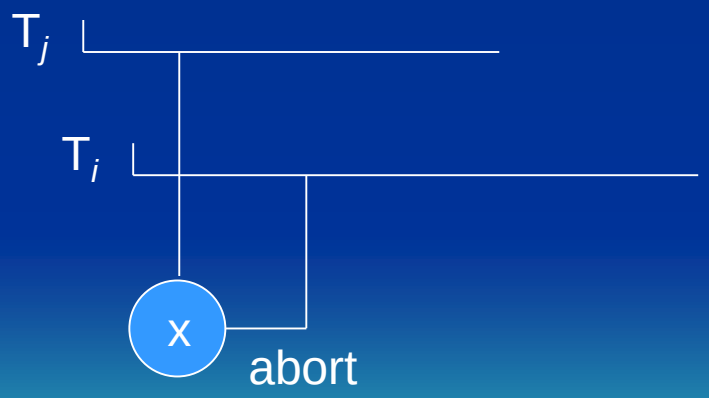
Deadlock Prevention: Wait-die

Suppose Ti tries to lock an item locked by Tj.

If Ti is the older transaction then Ti will wait.

Otherwise, Ti is aborted and restarts later with the same timestamp.





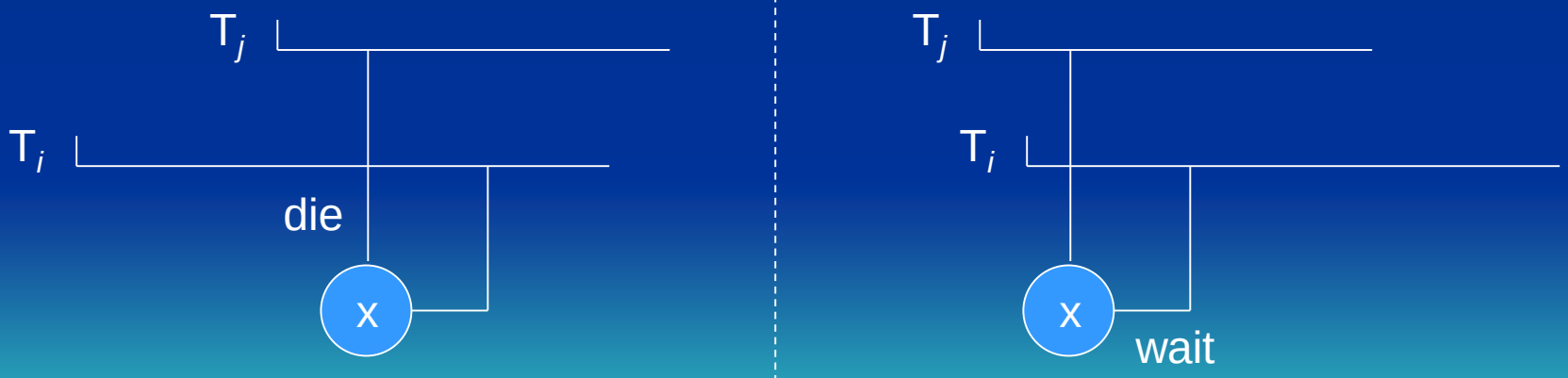
Deadlock Prevention: Wound-wait

Suppose Ti tries to lock an item locked by Tj.

If Ti is the older transaction

then Tj is aborted and restarts later with the same timestamp;

otherwise Ti is allowed to wait.



Deadlock Prevention - Waiting based

- No timestamps
- Two schemes:

  no waiting

  cautious waiting
- Both schemes will cause aborts even though deadlock would not have occurred

Deadlock Prevention: No waiting

Suppose Ti tries to lock an item locked by Tj

If Ti is unable to get the lock
then Ti is aborted and restarted after some time delay

Transactions may be aborted and restarted needlessly

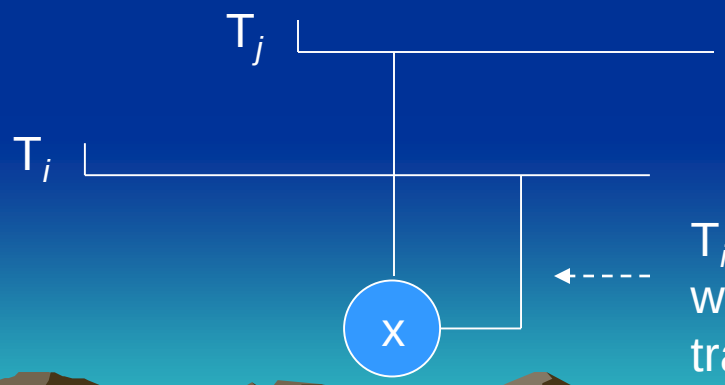
Deadlock Prevention: Cautious waiting

Suppose Ti tries to lock an item locked by Tj.

If Tj is not waiting on another transaction,

then Ti is allowed to wait;

otherwise Ti is aborted.



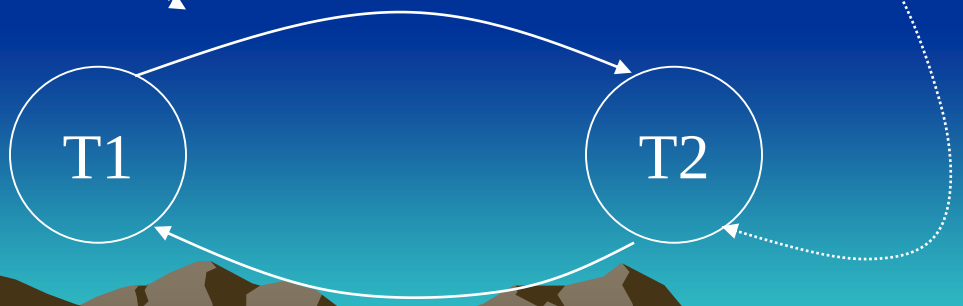
T_i waits or aborts depending on whether T_j is waiting for some other transaction or not.

Example: Deadlock Detection

T1	T2
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
waiting	
	write_lock(Y)
	waiting

Wait-for graph:

has a cycle!



Concurrency Control - Timestamps

- Each transaction is assigned a timestamp (TS)

  If a transaction T1 starts before transaction T2,

  then TS(T1) < TS(T2); T1 is *older* than T2

- whereas locking synchronizes transaction execution so that the interleaved execution is equivalent to *some* serial schedule, timestamping synchronizes transaction execution so that the interleaved execution is equivalent to a *specific* serial execution - namely, that defined by the chronological order of the transaction timestamps.

Consider four transactions: T1, T2, T3, T4.

Assume that TS(T1) < TS(T2) < TS(T3) < TS(T4).

We may have 4! = 24 different serial execution of these transactions. Each of them is considered correct:

T1 ⇒ T2 ⇒ T3 ⇒ T4
T2 ⇒ T1 ⇒ T3 ⇒ T4
… …

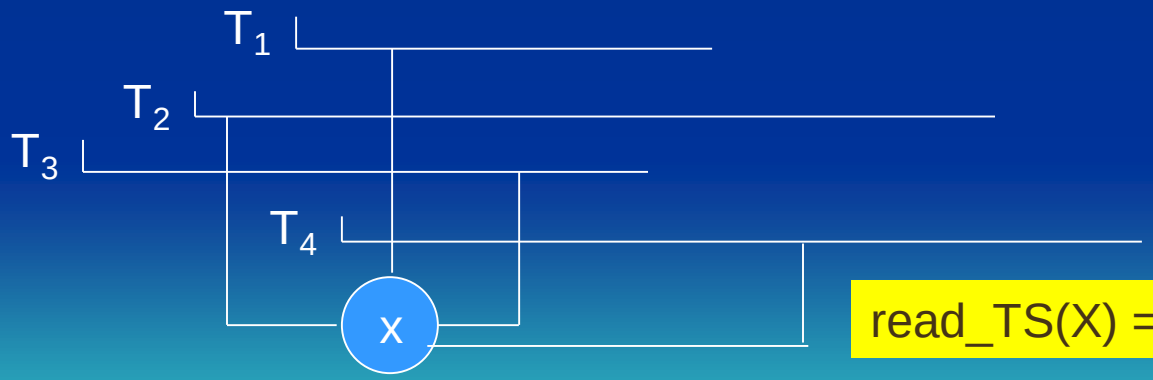
T4 ⇒ T3 ⇒ T2 ⇒ T1

But the method based on ‘timestamps’ synchronizes the interleaved execution of transactions so that it is equivalent to a specific serial execution:

T1 ⇒ T2 ⇒ T3 ⇒ T4

Database Item Timestamps

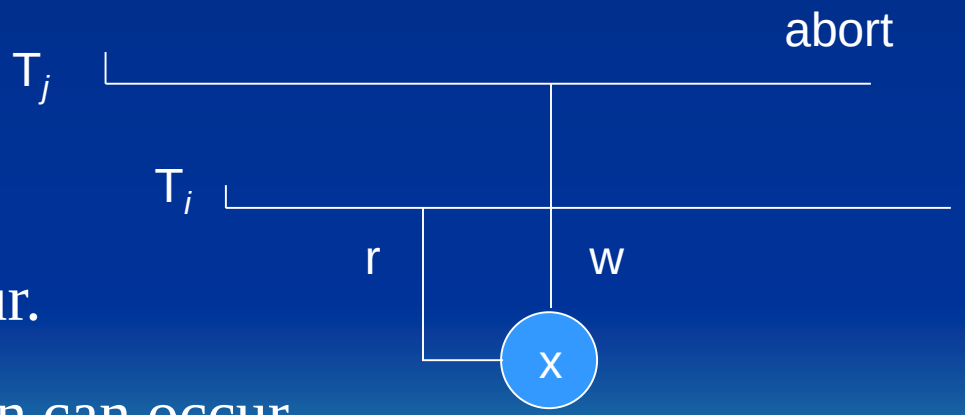
- Each database item X has 2 timestamps:
  - the **read timestamp** of X, read_TS(X), is the largest timestamp among all transaction timestamps that have successfully read X.
  - the **write timestamp** of X, write_TS(X), is the largest timestamp among all transaction timestamps that have successfully written X.



read_TS(X) = timestamp of T_4

Timestamp Ordering (TO) Algorithm

- When a transaction T tries to read or write an item X, the timestamp of T is compared to the read and write timestamps of X to ensure the timestamp order of execution is not violated.
- If the timestamp order of execution is violated, then T is aborted and resubmitted later with a new timestamp.



- Deadlock will not occur.
- Cascading rollback can occur.
- Cyclic restart of a transaction can occur.

read_TS(X) = timestamp of T_i

Timestamp Ordering (TO) Algorithm - in detail

- If T issues **write_item**(X) then

  if {read_TS(X) > TS(T) or write_TS(X) > TS(T)} then abort T

  otherwise (*TS(T) ≥ read_TS(X) and TS(T) ≥ write_TS(X)*)

  execute write_item(X)

  set write_TS(X) to TS(T)

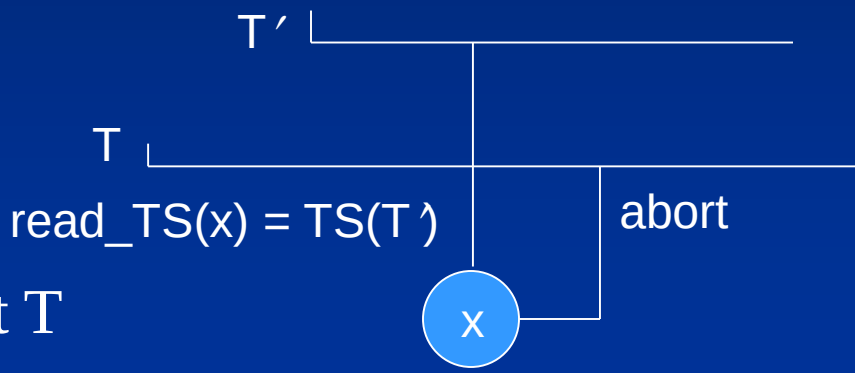
- if T issues **read_item**(X) then

  if write_TS(X) > TS(T) then abort T

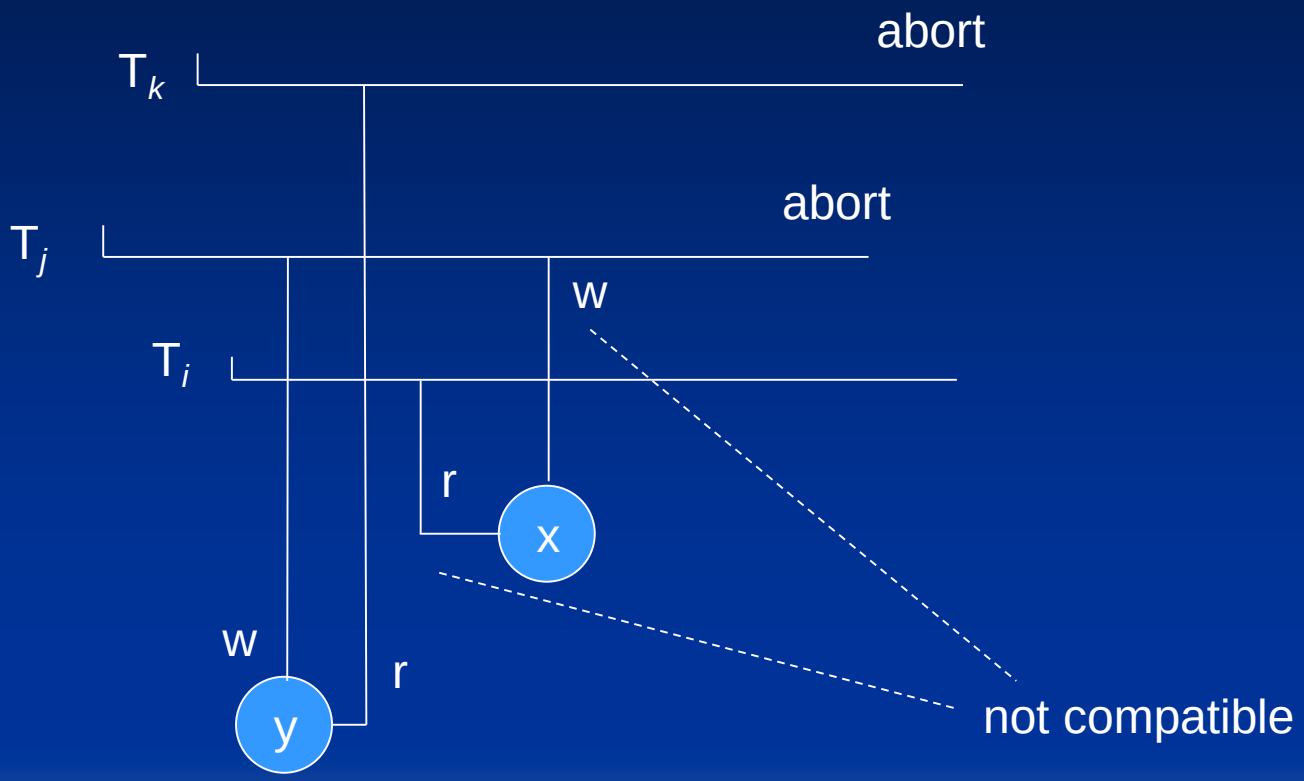
- otherwise (*TS(T) ≥ write_TS(X)*)

  execute read_item(X)

  set read_TS(X) to max{TS(T), read_TS(X)}



Why does the cascading rollback can occur?



The abortion of T_j leads to the abortion of T_k.

Concurrency Control - Multiversion 2PL

- Basic idea is to keep older version of data items around.
- When a transaction requires access to an item, an appropriate version is chosen to maintain serializability, if possible.
- Some read operations that would be rejected by other techniques can still be accepted by reading an older version of an item.

- Particularly adaptable to temporal databases.
- No cascading rollback.
- Deadlock can occur.
- In general, requires more storage.

Concurrency Control - Multiversion 2PL

- Three locking modes: read, write, certify
- Two versions of data items
- Certify lock is issued before a transaction's commit on all those data items which are currently write-locked by itself.
- Avoids cascading aborts

Lock compatibility table:

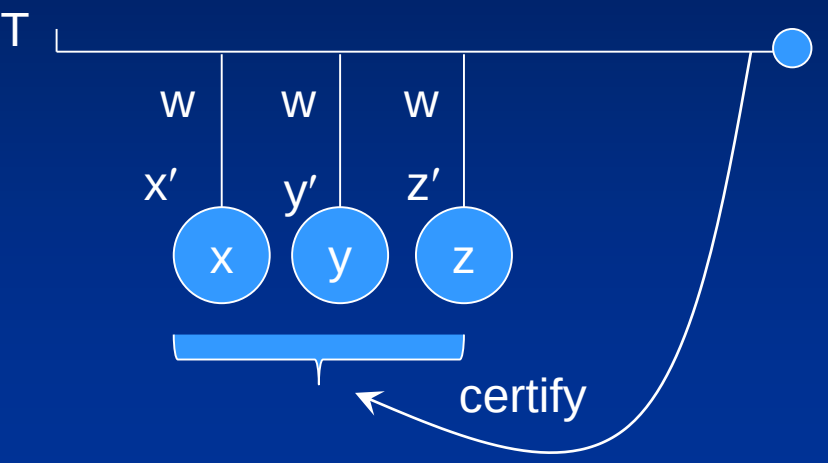
	read	*write*	*certify*
read	yes	yes	no
write	yes	no	no
certify	no	no	no

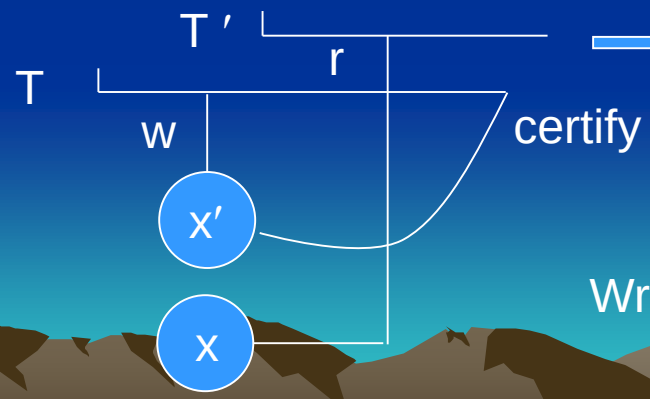
Concurrency Control - Multiversion 2PL

held by another transaction

	read	*write*	*certify*
read	yes	yes	no
write	yes	no	no
certify	no	no	no

Not allowed under standard 2PL, but is allowed here …





Wrong if no certify

Granularity of Data Items and Multiple Granularity Locking

- Database is formed of a number of named data items.

- Data item:
  - a database record
  - a field value of a database record
  - a disk block
  - a whole table
  - a whole file
  - a whole database

- The size of data item is often called the data item granularity.
  fine granularity - small data size
  coarse granularity - large data size

Granularity of Data Items and Multiple Granularity Locking

- The larger the data item size is, the lower the degree of concurrency.
- The smaller the data size is, the more the number of items in the database.

  A larger number of active locks will be handled by the lock manager.

  More lock and unlock operations will be performed, causing a higher overhead.

  More storage space will be required for the lock table.

What is the best item size?

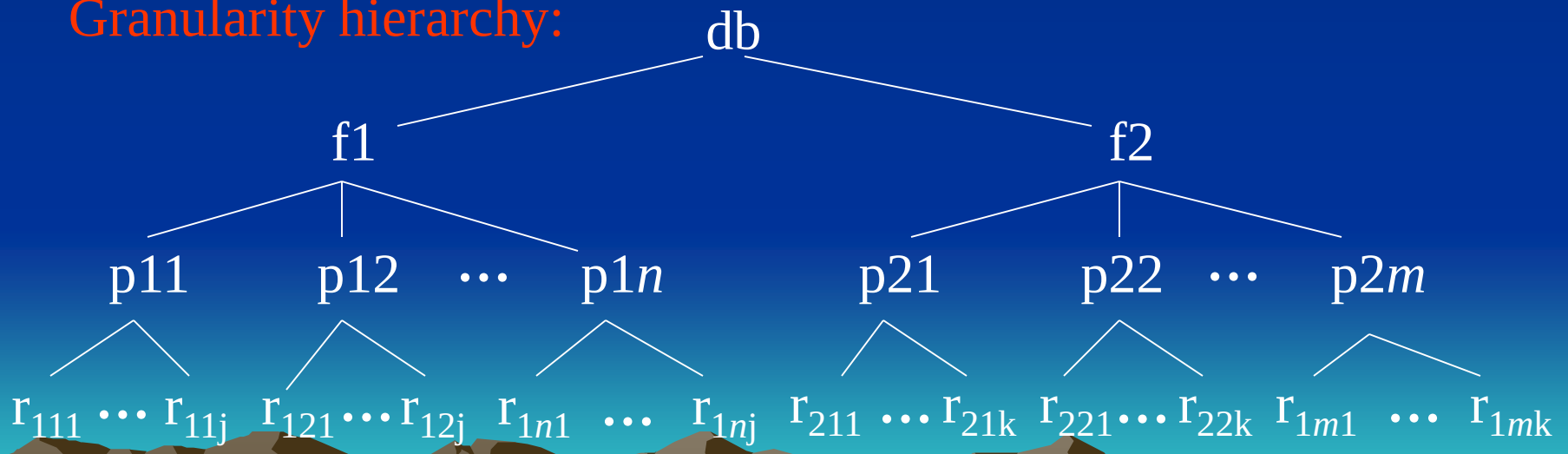
Answer: it depends on the types of transactions involved.

Granularity of Data Items and Multiple Granularity Locking

- Multiple granularity level locking

  Since the best granularity size depends on the given transaction, it seems appropriate that a database system supports multiple levels of granularity, where the granularity level can be different for various mixes of transactions.

  Granularity hierarchy:



Granularity of Data Items and Multiple Granularity Locking

- Solution: intention locks.

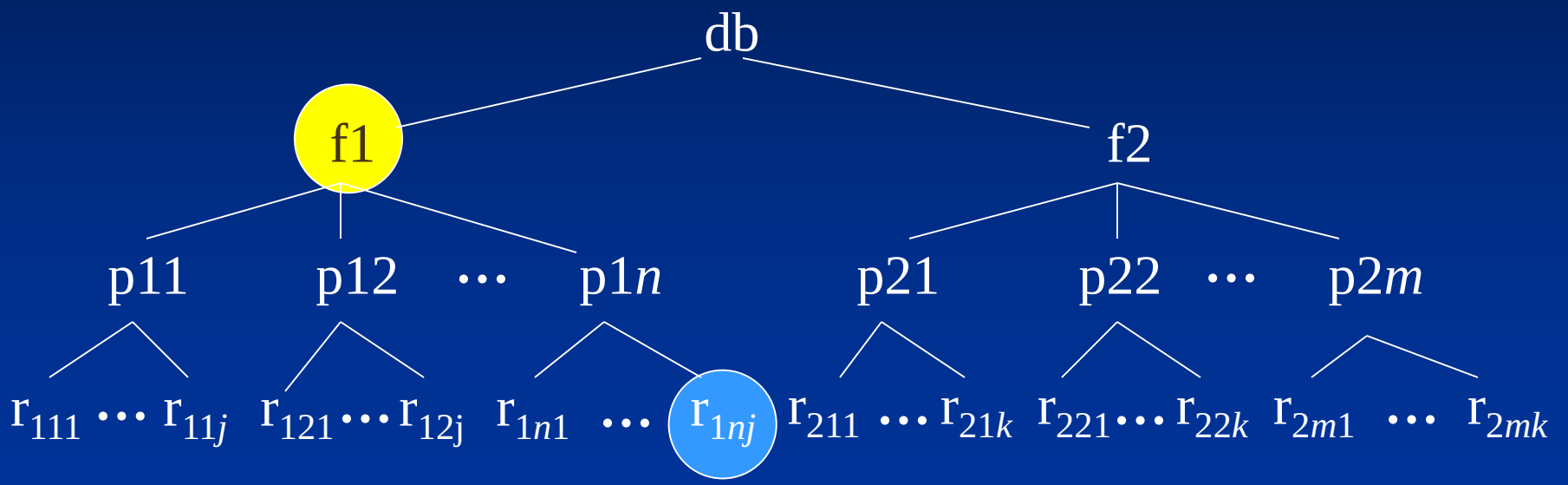
Three types of intention locks:

1. Intention-shared (IS) indicates that a shared lock(s) will be requested on some descendant node(s).
2. Intention-exclusive (IX) indicates that an exclusive lock(s) will be requested on some descendant node(s).
3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendant node(s).

Granularity of Data Items and Multiple Granularity Locking

- Lock compatibility matrix for multiple granularity locking

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no


db
f1
f2
p11
p12
p1n
p21
p22
p2m
r111 r11j
r121 r12j
r1n1 r1nj
r211 r21k
r221 r22k
r2m1 r2mk

Granularity of Data Items and Multiple Granularity Locking

- Multiple granularity locking (MGL) protocol:

1. The lock compatibility must be adhere to.
2. The root of the granularity hierarchy must be locked first, in any mode.
3. A node N can be locked by a transaction T in S or IS mode only if the parent of node N is already locked by transaction T in either IS or IX mode.
4. A node N can be locked by a transaction T in X, IX, or SIX mode only if the parent of node N is already locked by transaction T in either IX or SIX mode.
5. A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
6. A transaction T can unlock a node N only if none of the children of node N are currently locked by T.

Granularity of Data Items and Multiple Granularity Locking

- Example:

T1: updates all the records in file f1.
T2: read record r_{1nj}.

T1:	T2:
IX(db)	IS(db)
X(f1)	IS(f1)
write-item(f1)	IS(p1*n*)
unlock(f1)	S(r_{1nj})
unlock(db)	read-item(r_{1nj})
	unlock(r_{1nj})
	unlock(p1*n*)
	unlock(f1)
	unlock(db)

Granularity of Data Items and Multiple Granularity Locking

T2:	T1:
IS(db)	
IS(f1)	
IS(p1*n*)	
S(r_{1nj})	
	IX(db)
	X(f1)
read-item(r_{1nj})	
unlock(r_{1nj})	
unlock(p1*n*)	
unlock(f1)	
unlock(db)	
	write-item(f1)
	unlock(f1)
	unlock(db)

Concurrency - other topics

- Phantoms

  a phantom with respect to transaction T1 is a new record that comes into existence, created by a concurrent transaction T2, that satisfies a search condition used by T1.

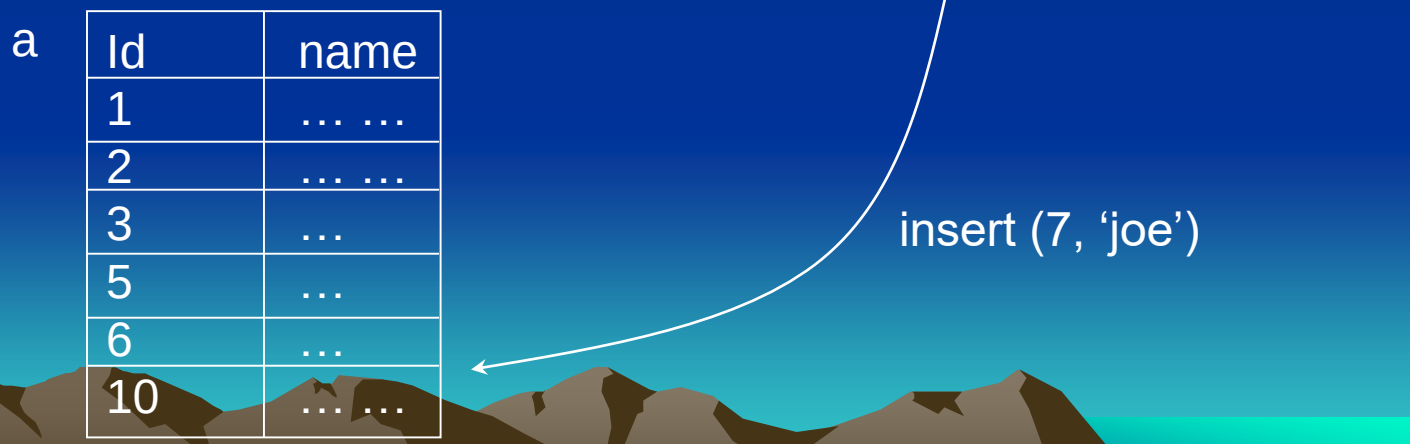
- consider transactions that include the following operations:

T1

```
SELECT * FROM a
WHERE id BETWEEN 5 AND 10
```

T2

```
INSERT INTO a
VALUES (id, name) (7, 'joe')
```



Concurrency - other topics

- Interactive transactions

  values written to a user terminal prior to commit could be used as input to other transactions

  this inter-transaction dependency is outside the scope of any DBMS concurrency controls

Concurrency - in SQL databases

- SQL isolation levels

If write lock is kept till T is committed, and a read lock on X cannot be released until all read operations on X have been conducted.

SET TRANSACTION

< SERIALIZABLE |

REPEATABLE READ |

READ COMMITTED |

READ UNCOMMITTED >

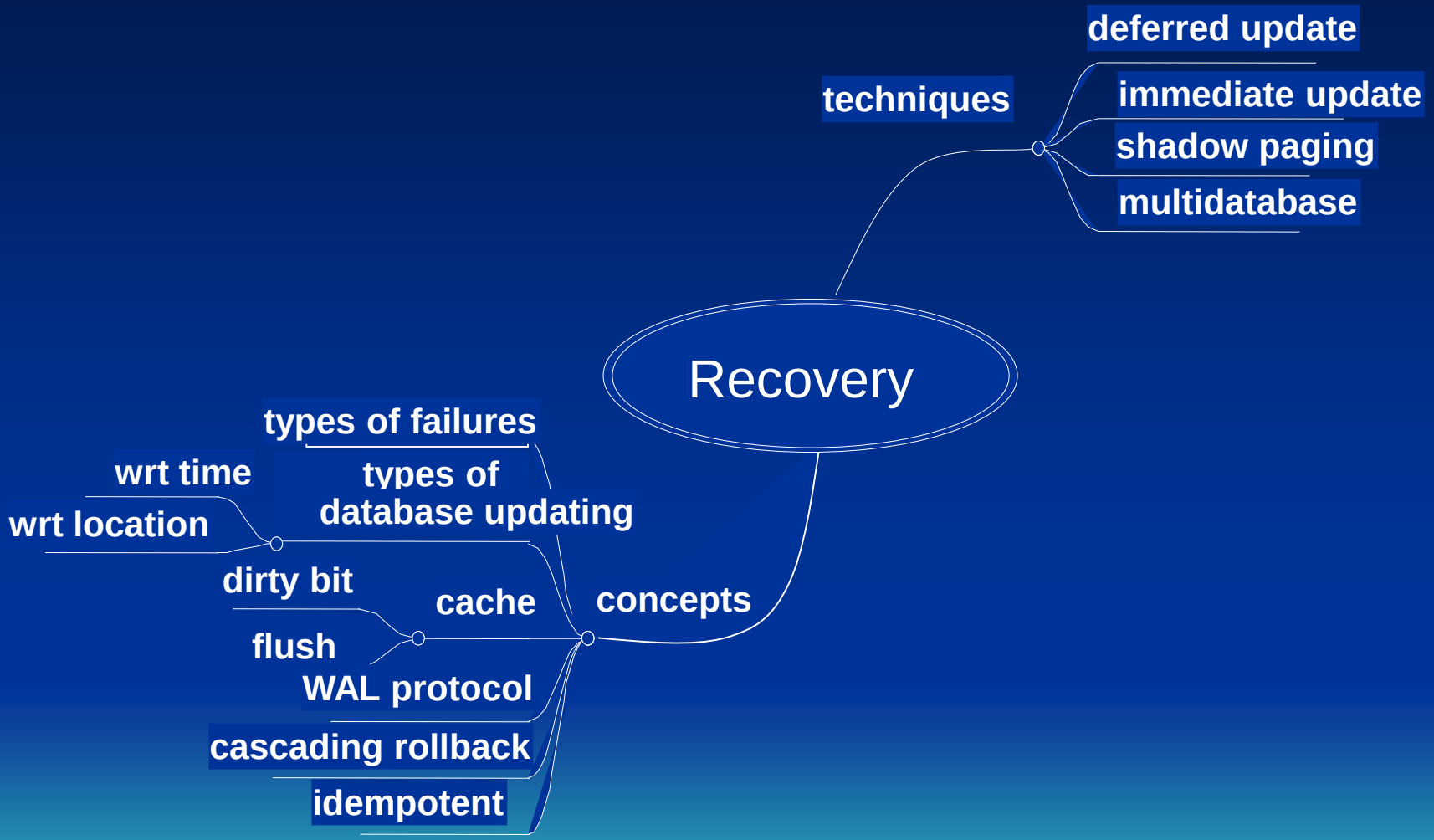
If write lock is kept till T is committed, but read lock can be released earlier.

Concurrency - SQL

Phenomena	description
P1	dirty read (transaction can read data that is not committed)
P2	nonrepeatable read (transaction can read the same row twice, and it could be different)
P3	phantom

Concurrency - SQL

	Phenomena occurs?		
	P1	P2	P3
serializable	no	no	no
repeatable read	no	no	yes
read committed	no	yes	yes
read uncommitted	yes	yes	yes


Recovery
techniques
deferred update
immediate update
shadow paging
multidatabase
concepts
types of failures
types of database updating
wrt time
wrt location
cache
dirty bit
flush
WAL protocol
cascading rollback
idempotent

Concepts

Recovery … “database is restored to some state from the past so that a correct state - close to the time of failure - can be *reconstructed* from that past state”

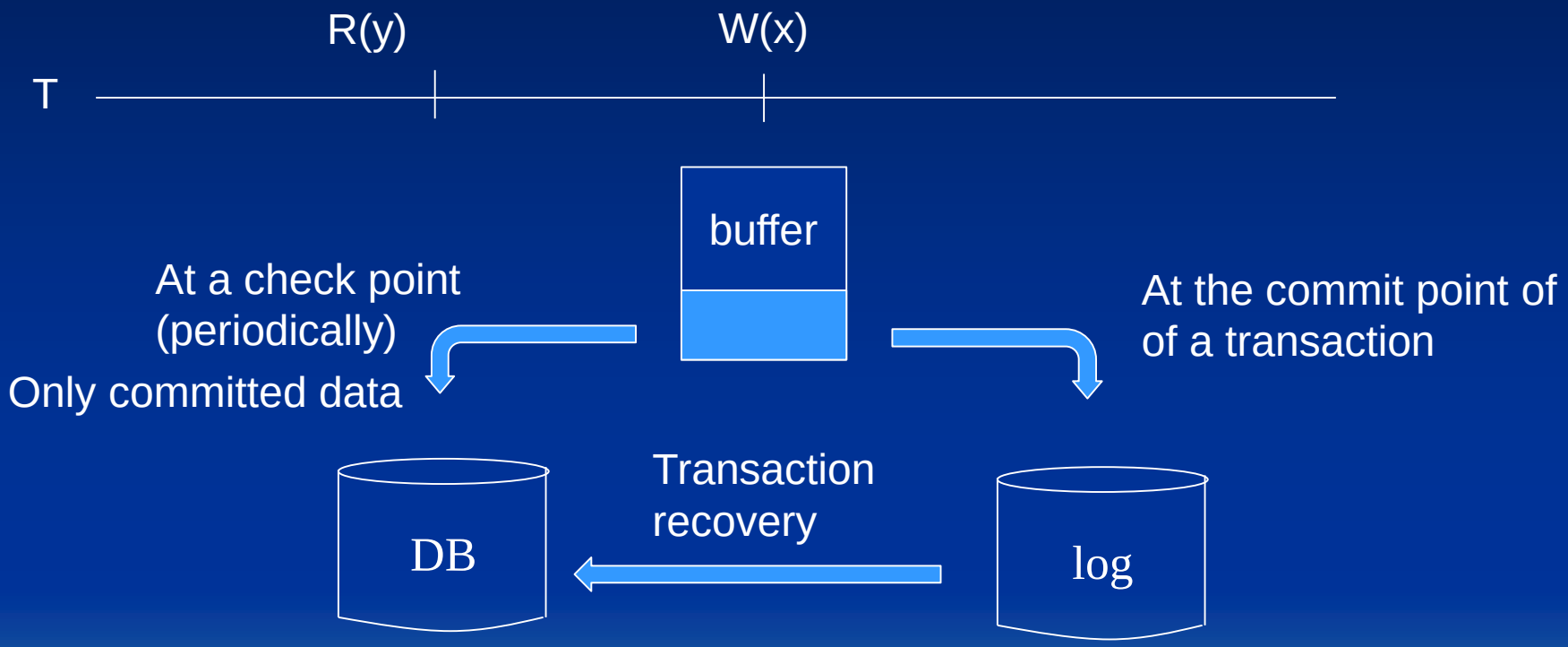
Recovery is needed to ensure the atomicity of transactions , and their durability (**A**CI**D** properties)

- How is recovery implemented? …. typically a log plays an important part
  - *BFIM* - before image - an *undo* entry
  - *AFIM* - after image - a *redo* entry

Concepts

An update to the database is called a:

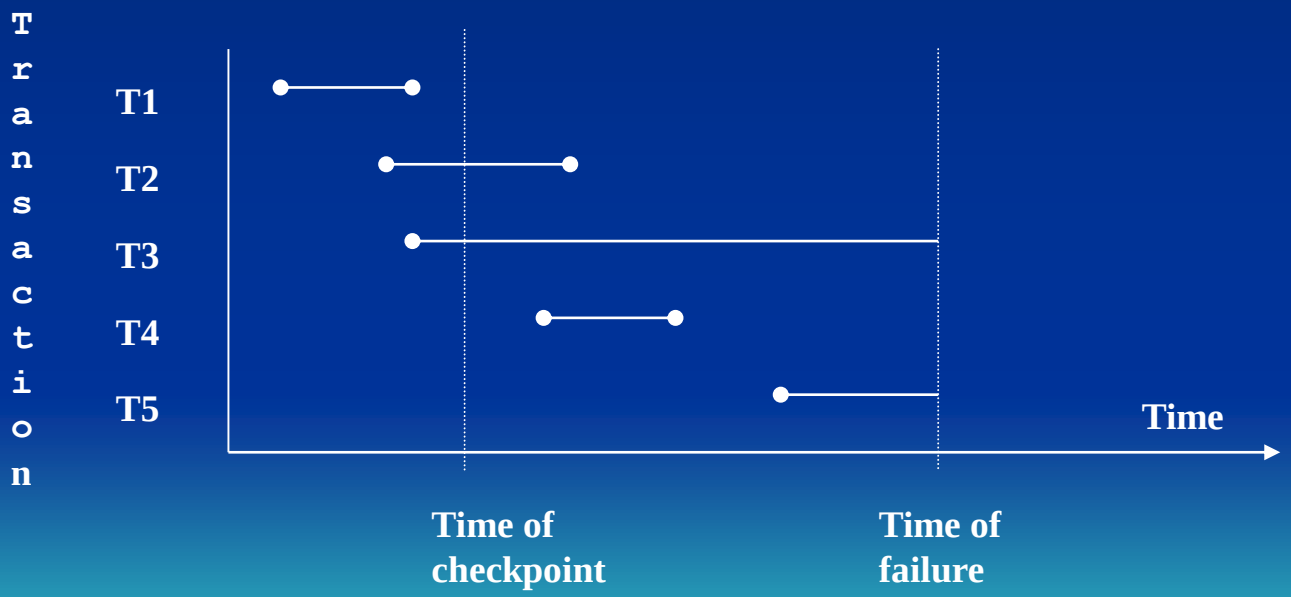
- *deferred update* if the database update does not actually occur until after a transaction reaches its commit point
  - recall that when a transaction reaches its commit point all changes have been recorded (persistently) in the log
  - what are the implications for recovery?
    - is undo needed?
    - is redo needed?


R(y)
W(x)
T
buffer
At a check point
(periodically)
Only committed data
At the commit point of
of a transaction
DB
Transaction
recovery
log

Recovery Technique for Deferred Update

Transaction types at recovery time

Consider the five types below. Which need to be redone after the crash?



No redo, no undo

redo

No redo, no undo

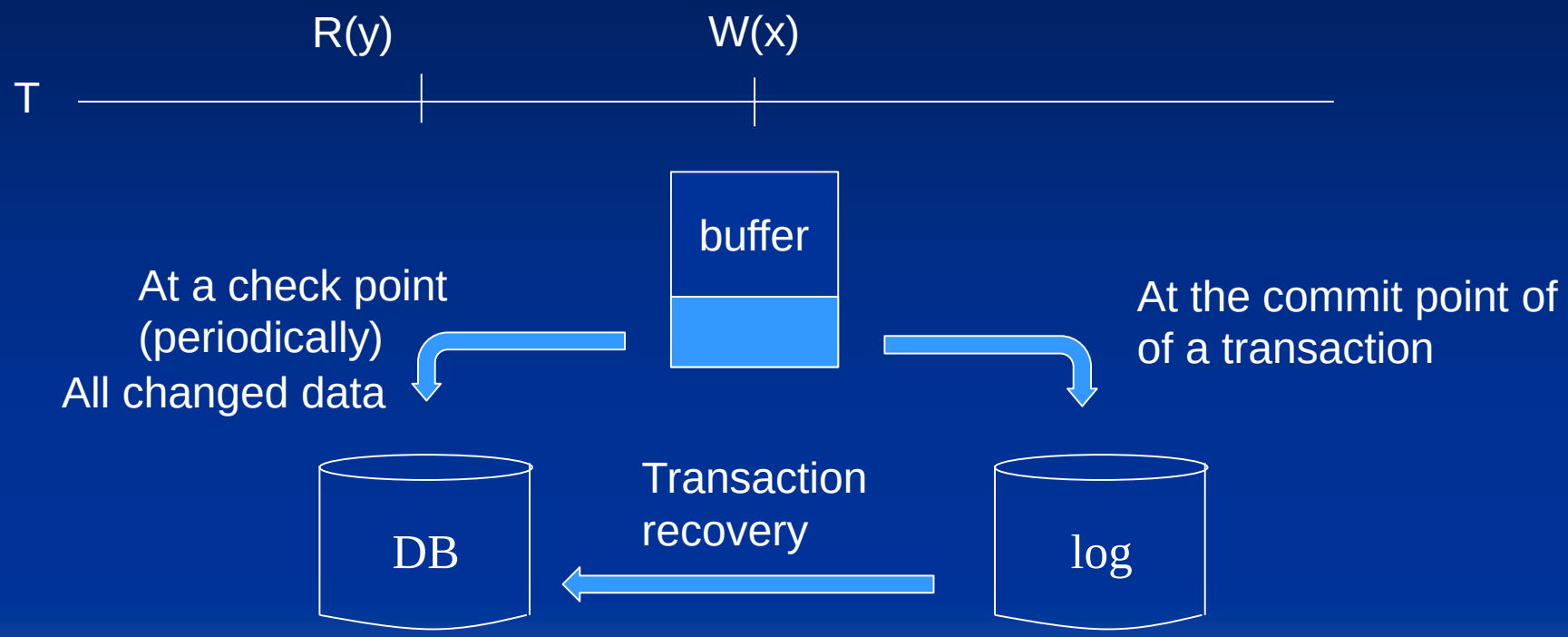
redo

ignored

Concepts

An update to the database is called an:

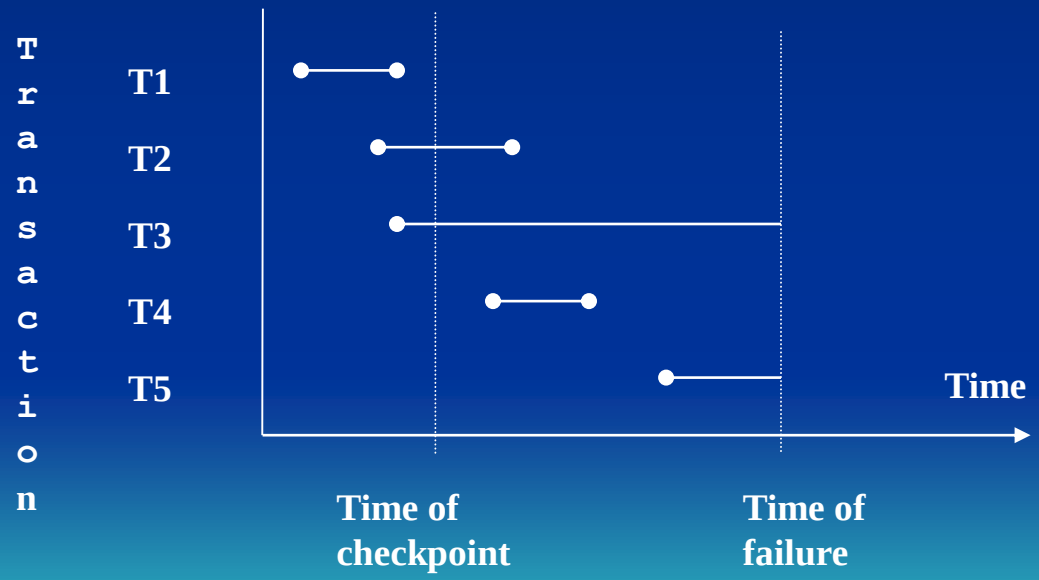
- *immediate update* if the update can occur before the transaction reaches its commit point
  - a very typical situation in practice
  - what are the implications for recovery?
    - is undo needed?
    - is redo needed?


R(y)
W(x)
T
buffer
At a check point
(periodically)
All changed data
At the commit point of
of a transaction
DB
Transaction
recovery
log

Recovery Technique for Immediate Update

Transaction types at recovery time

Consider the five types below. Which need to be undone / redone after the crash?



No redo, no undo

redo

undo

redo

ignored

Concepts

- when data is written to disk it may be written:
  - as a *shadow,* or
  - *in-place* which requires the *write-ahead logging (WAL)* protocol:
    - a log file is needed, which keeps undo records (BFIM) and redo records (AFIM)
    - data records cannot be overwritten until the undo records have been force-written to the log on disk
    - redo records and the undo records must be force-written to the log on disk before the commit can be considered completed

Recovery Technique for Shadow Paging

What is shadow paging?

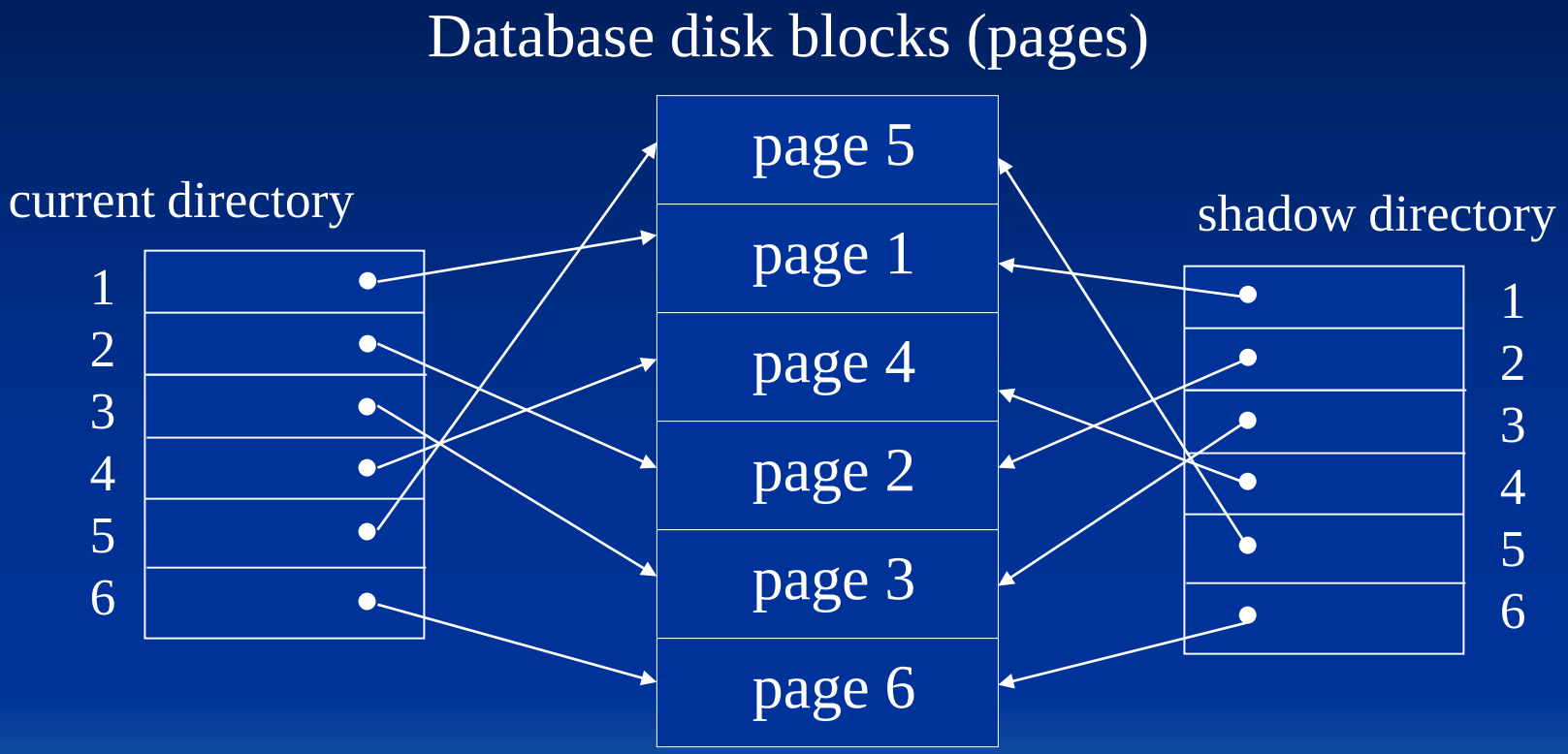
It is a technique pioneered in System R where changes are made to a copy of a page (block). When a transaction commits, the copy becomes the current page and the original is discarded

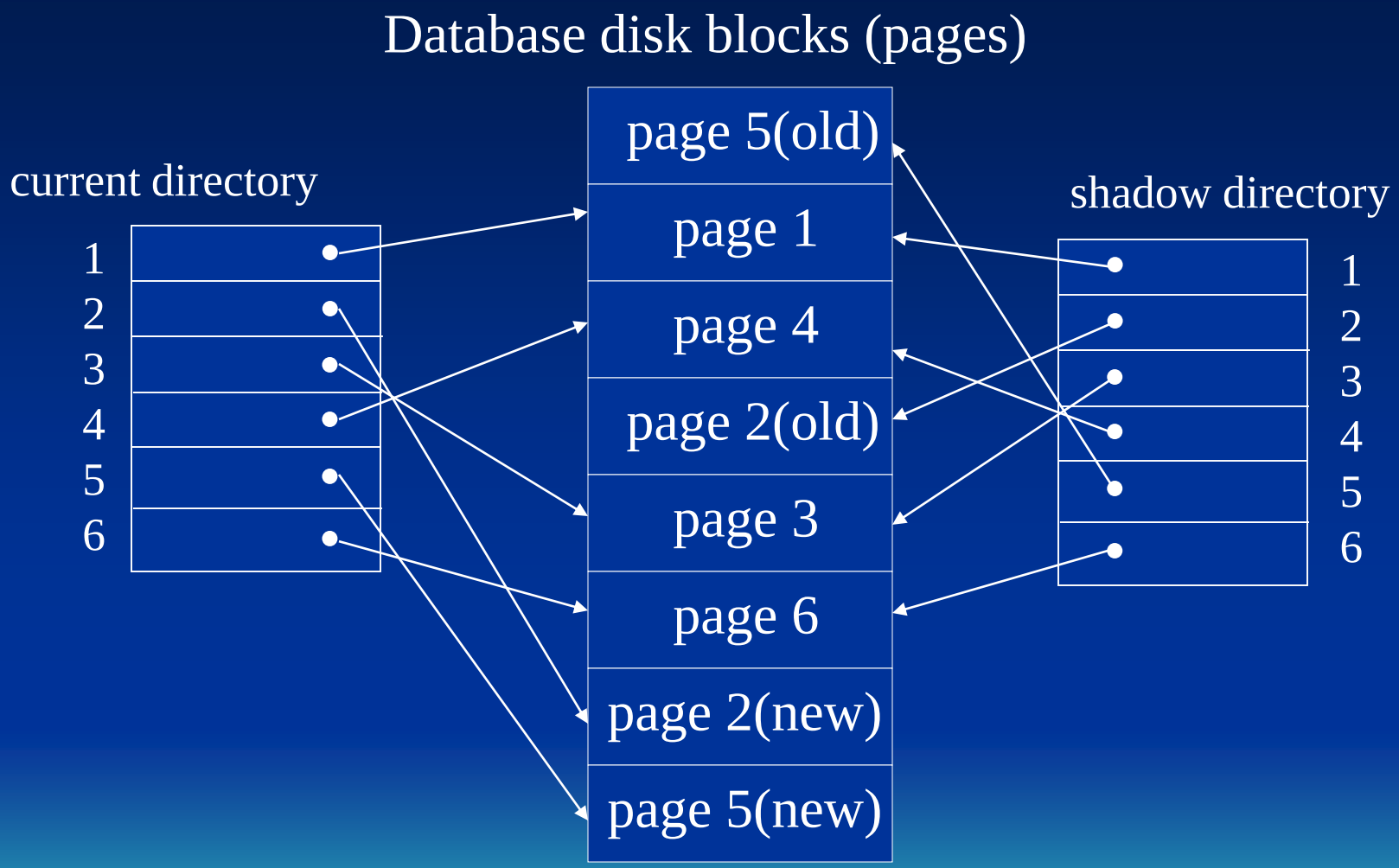
Recovery Technique for Shadow Paging

How a single transaction would be handled:

Suppose transaction A starts up:

- the current page table (directory) is copied to the shadow page table (shadow directory)
- if the transaction updates a page, the original page is not altered, rather a copy is created and that is modified
- the copy is pointed to by the current page table - the shadow page table is never modified


Database disk blocks (pages)
current directory
1
2
3
4
5
6
page 5
page 1
page 4
page 2
page 3
page 6
shadow directory
1
2
3
4
5
6


Database disk blocks (pages)
current directory
1
2
3
4
5
6
page 5(old)
page 1
page 4
page 2(old)
page 3
page 6
page 2(new)
page 5(new)
shadow directory
1
2
3
4
5
6

Security control

- Discretionary access control
- Mandatory access control
- SQL injection
- Asymmetric encryption

How do we protect the database from unauthorized access?

Who should be able to see employee salaries, student grades, … ?

Who should be able to update … ?

Techniques include/involve:

- passwords
- log-in process - new entry for the log:
  {who, where, when}
- privileges
- encryption
- accounts - system (DBA), user

We study two mechanisms:

- Discretionary access control (DAC)
  - privileges such as read, write, update are granted to users
  - a certain amount of discretion is given to the owner or anyone else with appropriate authority
- Mandatory access control (MAC)
  - multilevel security is applied to data and users
  - controlled by a central authority, not by owners of an object
  - the owner/creator of an object does not decide who has clearance to see the object

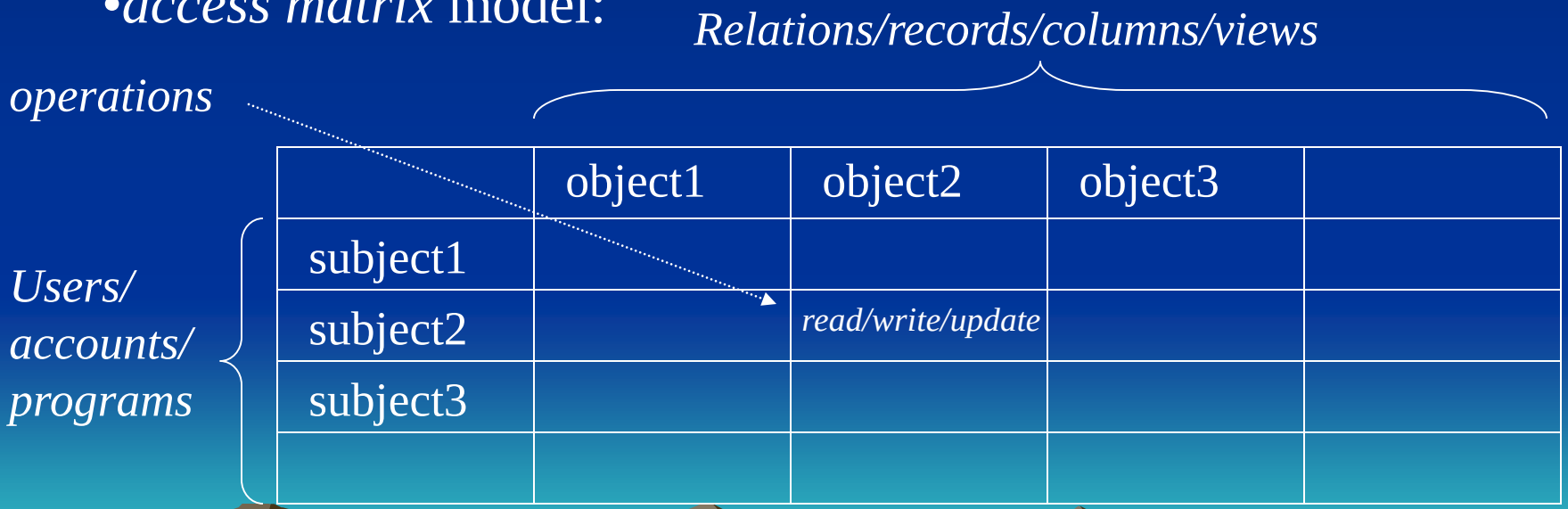
Discretionary access control

- account level
  - create - schema, table, view
  - alter - indexes, table (attributes, indexes)
  - drop - table, index, view
- example
  - grant *createtab* to A1 ;

  if A1 now creates a table X, then A1 *owns* X, and has all privileges on table X, and A1 can grant privileges to others

Discretionary access control

- relation level
  - type of access: read, write, update
  - privileges on relations and columns
  - *access matrix* model:



Discretionary access control: suppose A1 executes:

- create table employee (…);
- create table department (…);
- grant insert, delete on employee, department to A2;
- grant select on employee, department to A3;
- grant update on employee(salary) to A4;

	employee	department	employee.salary	
A1	*all*	*all*		
A2	*insert, delete*	*insert, delete*		
A3	*select*	*select*		
A4			*update*	

Mandatory access control for multilevel security

Bell LaPadula model:

- specifies the allowable paths of information flow:

  information with high secret - information with low secret

- set of subjects S, and a set of objects O

S:
- user
- account
- programs

O:
- relation
- tuple
- column
- view
- operations

•each s in S and o in O has a fixed security class

class(s) a clearance of s

class(o) a classification level of o

•security classes are ordered by <=

U (Unclassified) <= C (confidential) <= S (Secret) <= TS (Top Secret)

(public <= sensitive <= top secret)

Mandatory access control for multilevel security

Two properties of the *Bell LaPadula* model:

- Simple Security Property

  a subject s is not allowed read access to an object o unless

  class(s) >= class(o)

  To see something, your clearance must be at least that of what you want

In the military model, the security clearance of someone receiving a piece of information must be at least as high as the classification of the information

Mandatory access control for multilevel security

Second property of the *Bell LaPadula* model:

- Star Property

  a subject s is not allowed write access to an object o unless

  class(s) <= class(o)

  To create/update something, your clearance must be no greater than the object you are creating/updating

In the military model, a person writting some information at one level may pass that information along only to people at levels no lower than the level of the person

Mandatory access control for multilevel security

Implementation of the *Bell LaPadula* model:

- for each original attribute in a relation, add a classification attribute
- add a classification attribute for the tuple (row) - value is maximum of all classifications within the tuple
- these classification attributes are transparent to the user

Mandatory access control for multilevel security

Implementation example: suppose U <= C <= S

Employee relation

Name	Salary	JobPerformance
Smith	40,000	Fair
Brown	80,000	Good

The user view without MAC

Name	C_1	Salary	C_2	JobPerformance	C_3	TC
Smith	U	40,000	C	Fair	S	S
Brown	C	80,000	S	Good	C	S

system view with MAC

SQL Injection

SQL injection is a web security vulnerability that allows an attacker

•to interfere with the queries that an application makes to its database.

•to view data that they are not normally able to access. This might include data belonging to other users, or any other data that the application itself is not able to access.

•In many cases, an attacker can modify or delete some data, causing persistent changes to the application's content or behavior.

•In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure, or perform a denial-of-service attack.

Retrieving hidden data

Consider a shopping application that displays products in different categories. When the user clicks on the Gifts category, their browser requests the URL:

```
https://insecure-website.com/products?category='Gifts'
```

This causes the application to make an SQL query to retrieve details of the relevant products from the database:

```
SELECT * FROM products
WHERE category = 'Gifts'
AND released = 1
```

The application doesn't implement any defenses against SQL injection attacks, so an attacker can construct an attack like:

```
https://insecure-website.com/products?category='Gifts'--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts'-- AND released = 1
```

The key thing here is that the double-dash sequence -- is a comment indicator in SQL and means that the rest of the query is interpreted as a comment. This effectively removes the remainder of the query, so it no longer includes `AND released = 1`. This means that all products are displayed, including unreleased products.

Subverting application logic

Consider an application that lets users log in with a username and password. If a user submits the username WIENER and the password BLUECHEESE, the application checks the credentials by performing the following SQL query:

```
SELECT * FROM users WHERE username = 'wiener' AND password = 'bluecheese'
```

- If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.

- Here, an attacker can log in as any user without a password simply by using the SQL comment sequence -- to remove the password check from the WHERE clause of the query. For example, submitting the username administrator' -- and a blank password results in the following query:

```
SELECT * FROM users WHERE username =
'administrator'--AND
password='bluecheese'
```

This query returns the user whose username is administrator and successfully logs the attacker in as that user.

Protection against SQL Injection

Protection against SQL injection attacks can be achieved by applying certain programming rules to all Web accessible procedures and functions.

- **Bind Variables (using parameterized statements)**

  - The use of bind-variables (also known as parameters) protects against injection attacks and also improves performance.

Protection against SQL Injection

Consider the following example using Java and JDBC:

PreparedStatement stmt = conn.preparedStatement(“SELECT * FROM EMPLOYEEE WHERE EMPLOYEE_ID = ? AND PASSWARD = ?”);

stmt.setString(1, employee_id);

stmt.setString(2, passward);

ResultSet resultSet = stmt.executeQuery();

Encryption and Decryption

Symmetric encryption is a type of encryption where only one key (a secret key) is used to both encrypt and decrypt electronic information.

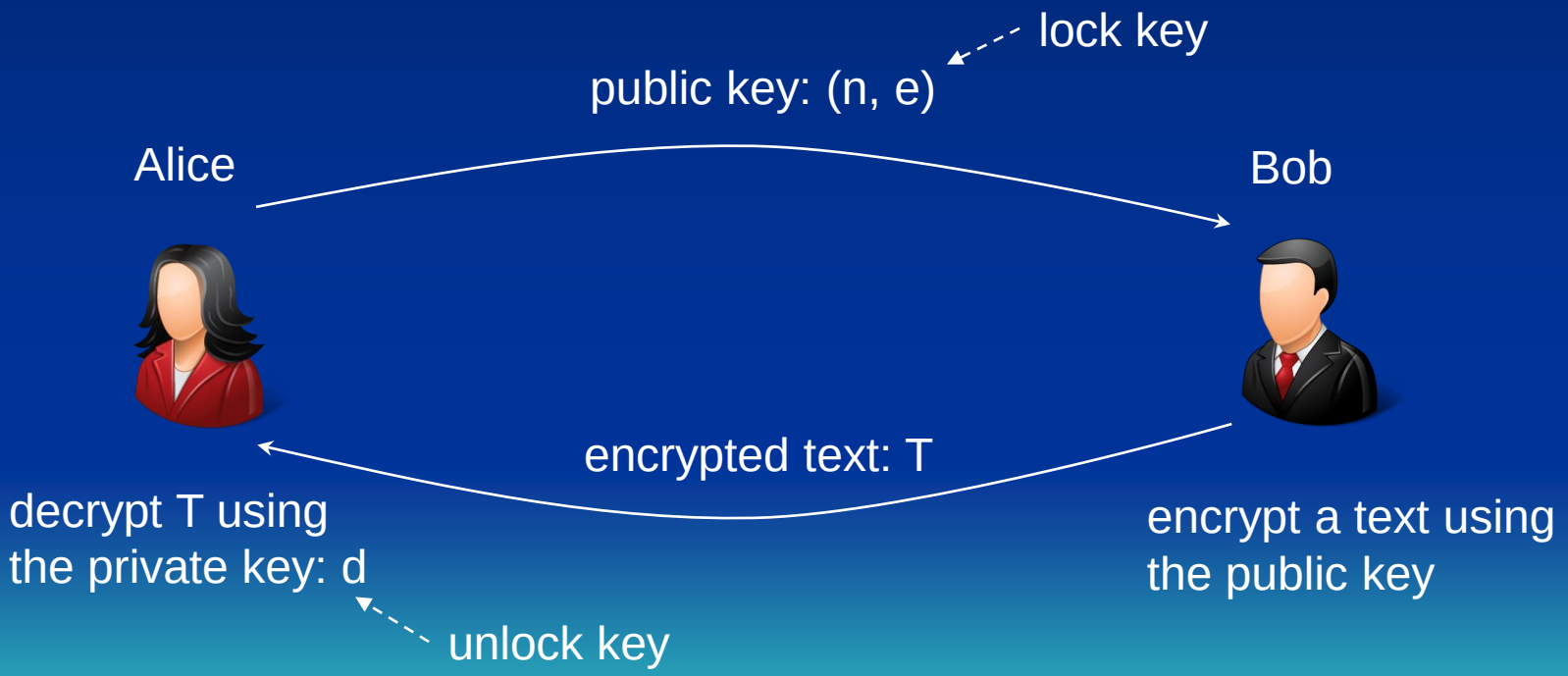
- The entities communicating via symmetric encryption must exchange the key so that it can be used in the decryption process.
- This encryption method differs from asymmetric encryption where a pair of keys, one public and one private, is used to encrypt and decrypt messages.

Encryption and Decryption

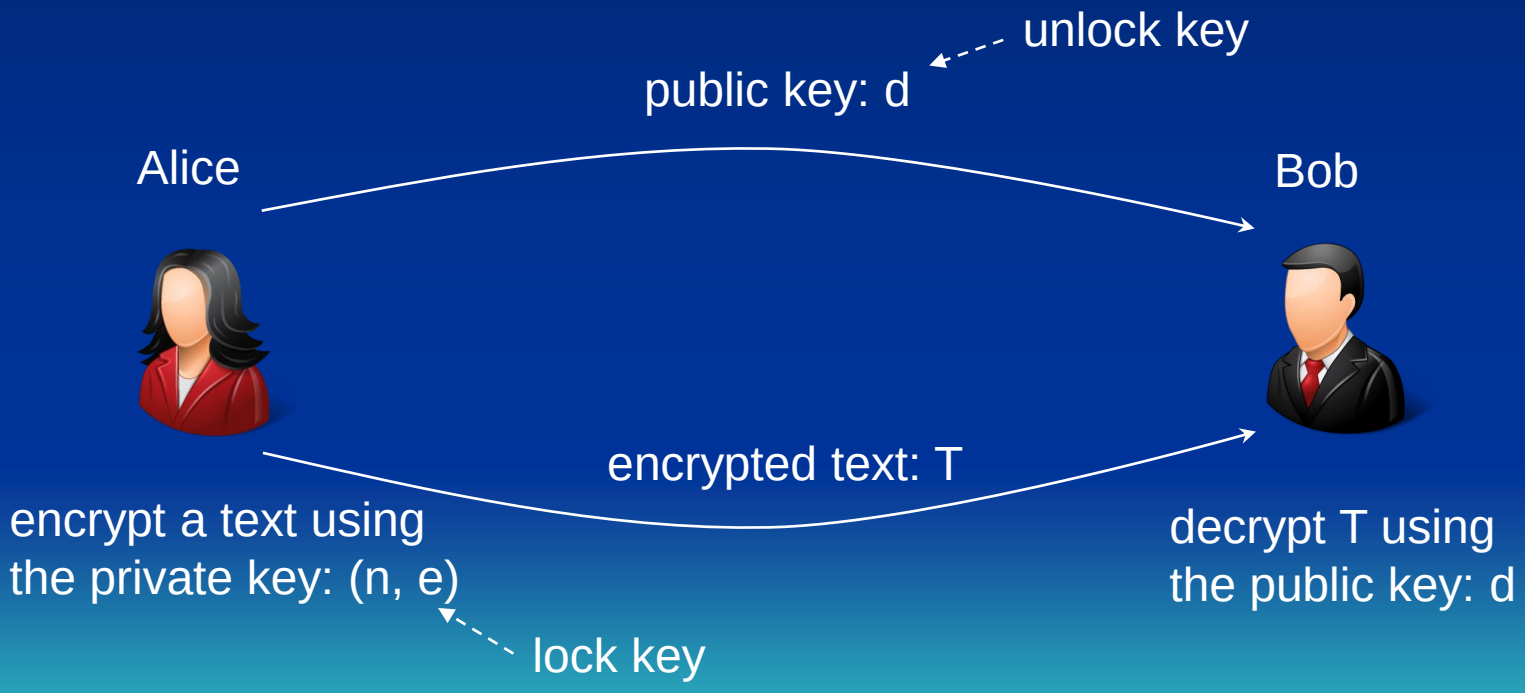
Asymmetric keys are the foundation of Public Key Infrastructure (PKI) - a cryptographic scheme

- Requiring two different keys, one to lock or encrypt the plaintext, and one to unlock or decrypt the cyphertext. Neither key will do both functions.
- One key is published (public key) and the other is kept private (private key).
- This system also is called asymmetric key cryptography.
- The asymmetric encryption can be used in two ways.

- If the lock/encryption key is the one published, the system enables private communication from the public to the unlocking key's owner.



- If the unlock/decryption key is the one published, then the system serves as a signature verifier of documents locked by the owner of the private key.



Encryption and Decryption

Key distribution

- Suppose that Bob wants to send information to Alice. If they decide to use *RSA* (an encryption algorithm, proposed by Rivest, Shamir, Adleman), Bob must know Alice's public key to encrypt the message and Alice must use her private key to decrypt the message.
- To enable Bob to send his encrypted messages, Alice transmits her public key (*n, e*) to Bob via a reliable, but not necessarily secret, route. Alice's private key (*d*) is never distributed.

Encryption and Decryption

Encryption

- After Bob obtains Alice's public key (n, e), he can send a message M to Alice.
- To do it, he first turns M (strictly speaking, the un-padded plaintext) into an integer m (strictly speaking, the padded plaintext), such that $0 \leq m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext c, using Alice's public key e, corresponding to

$$m^e \equiv c \pmod{n}$$

This can be done reasonably quickly, even for very large numbers, using modular exponentiation.

- Bob then transmits c to Alice.

Encryption and Decryption

Decryption

•Alice can recover *m* from *c* by using her private key exponent *d* by computing

$$c^d \equiv (m^e)^d \pmod{n}$$

Given *m*, she can recover the original message *M* by reversing the padding scheme.

Encryption and Decryption

Example

1. Choose two distinct prime numbers, such as

$p = 61$, $q = 53$

lock key: (n, e)

2. Compute $n = p \times q$ giving

$n = 61 \times 53 = 3233$

3. Compute the Carmichael's totient function of the product as $\lambda(n) = lcm(p - 1, q - 1)$, giving

lcm – least common multiple

$\lambda(3233) = lcm(60, 52) = 780$

4. Choose any number $1 < e < 780$ that is *coprime* to 780.

Choosing a prime number for e leaves us only to check that e is not a divisor of 780.

coprime – two numbers have no common factors other than 1

Encryption and Decryption

Example

4. Choose any number $1 < e < 780$ that is coprime to 780.

Choosing a prime number for e leaves us only to check that e is not a divisor of 780.

Let $e = 17$.

5. Compute d, the modular multiplicative inverse *of* e (i.e., e is a number satisfying $1 = (e \times d) \bmod 780$), yielding

$d = 413$,

as $1 = (17 \times 413) \bmod 780$.

$(17 \times 413 = 7021 = 9 \times 780 + 1)$

unlock key: d

Encryption and Decryption

Example

The public key is (n = 3233, e = 17). For a padded plaintext message m, the ciphertext is

$$c = m^e \bmod n = m^{17} \bmod 3233.$$

The private key is d = 417. By using d, we can get

$$m = c^d \bmod n = c^{417} \bmod 3233.$$

For instance, in order to encrypt m = 65, we calculate

$$c = 65^{17} \bmod 3233 = 2790$$

To decrypt c = 2790, we calculate

$$m = 2790^{413} \bmod 3233 = 65.$$

Modular exponentiation:

$(a \times b) \bmod m$

$= (a \bmod m) \times (b \bmod m) \bmod m$

$2790^{413} \bmod 3233$

$= (2790^{2} \bmod 3233) \times 2790^{411} \bmod 3233$

$= (7784100 \bmod 3233) \times (2790^{411} \bmod 3233)$

$= (249 \bmod 3233) \times (2790^{411} \bmod 3233)$

$= (249 \times 2790 \bmod 3233) \times (2790^{410} \bmod 3233)$

$= \ldots \ldots = 65$

Encryption and Decryption

Comments:

- Both of these calculations can be computed efficiently using the square-and-multiply algorithm for molular exponentiation.
- In real-life situations the primes selected would be much larger; in our example it would be trivial to factor *n*, 3233 (obtained from the freely available public key) back to the primes *p* and *q*. *e*, also from the public key, is then inverted to get *d* (according to $1 = (e \times d) \bmod 780$), thus acquiring the private key.

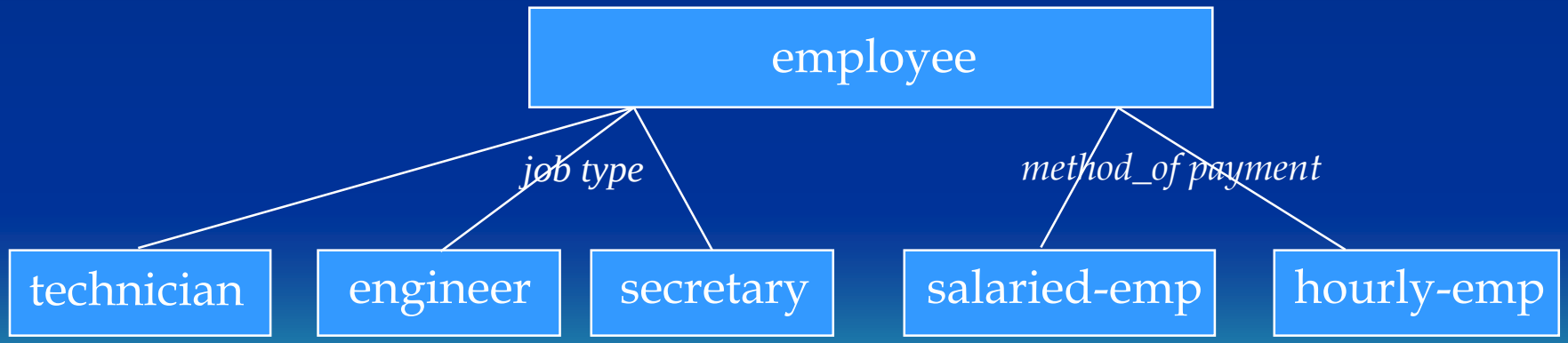
Enhanced ER-Diagram

- Specialization
- Generalization
- Shared subclasses
- Category

Specialization

Specialization is the process of defining a set of sub-entities of some entity type.

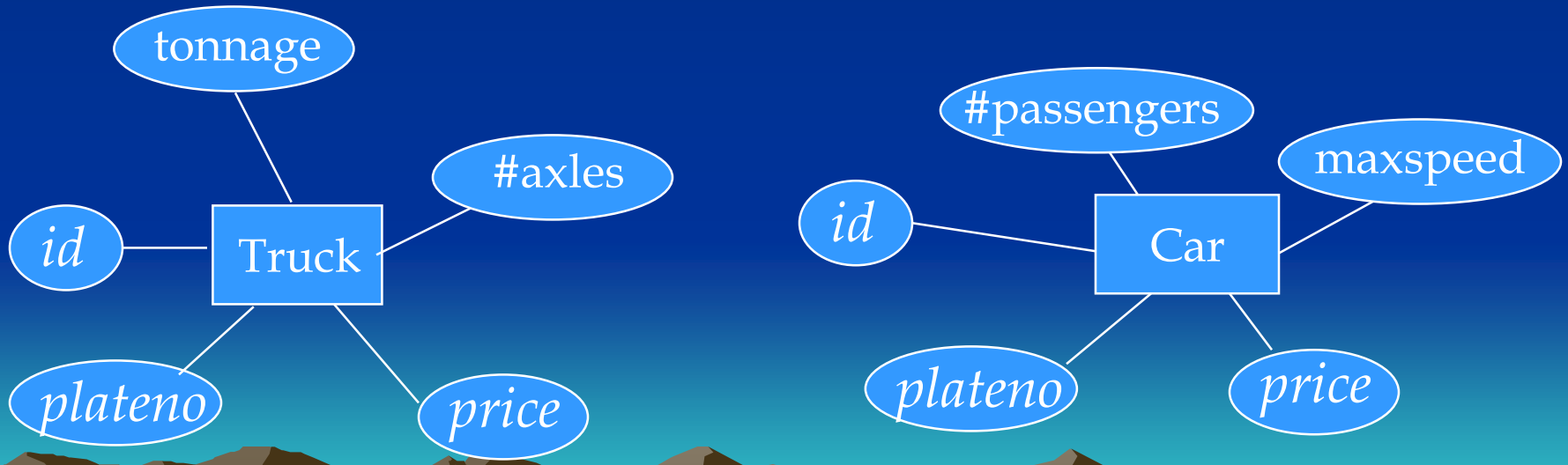
- Starting with Employee
- Consider the *Job Type* and *Method of payment* attributes
- We can specialize to create:



Generalization

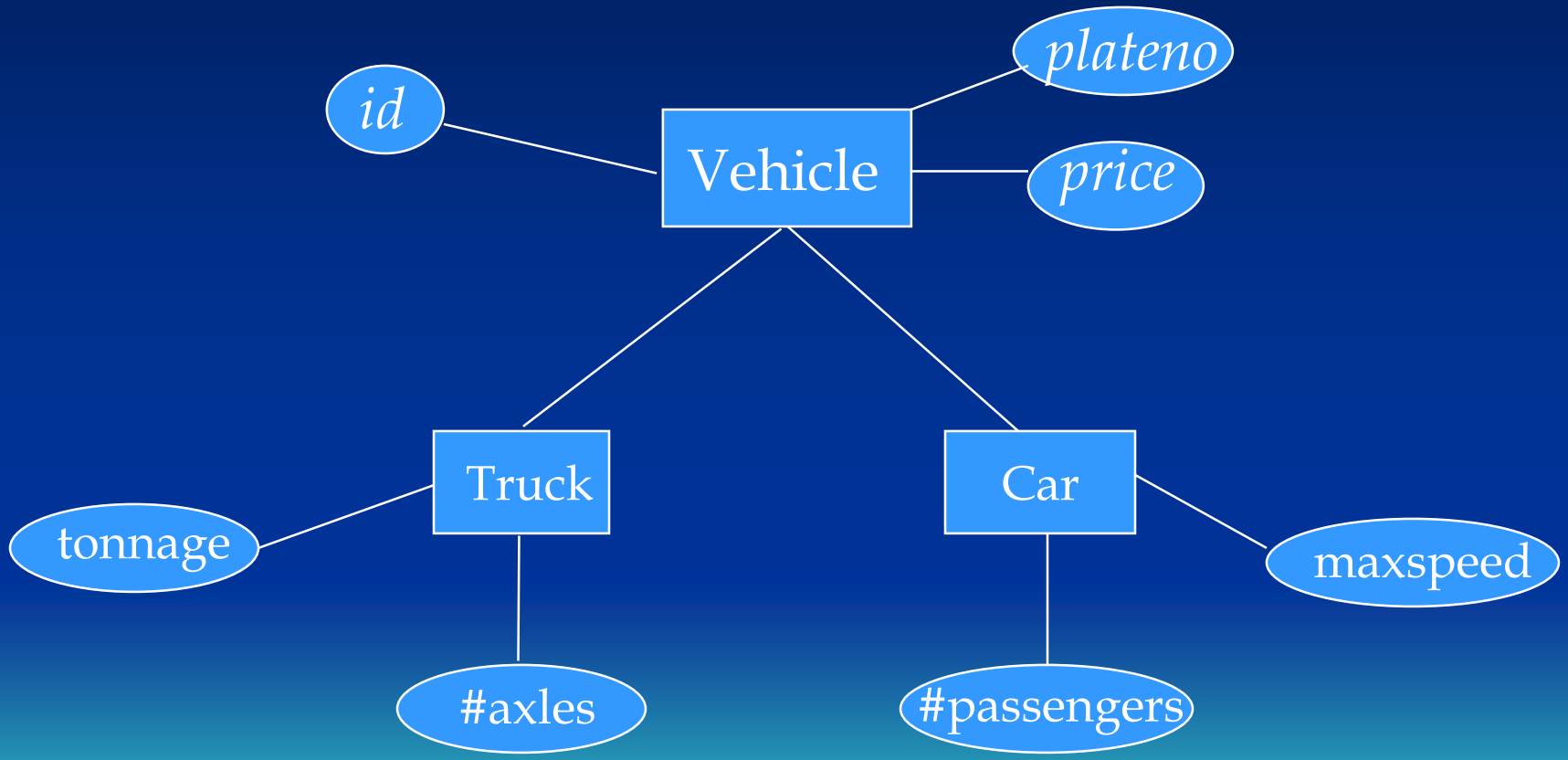
Generalization is the opposite approach/process of determining a supertype based on certain entities having common characteristics.

- reverse process of defining subclasses
- bottom up approach
- bring together common attributes from similar entity types, and suppress the differences (to form a superclass)
- example: suppose we begin with Cars and Trucks

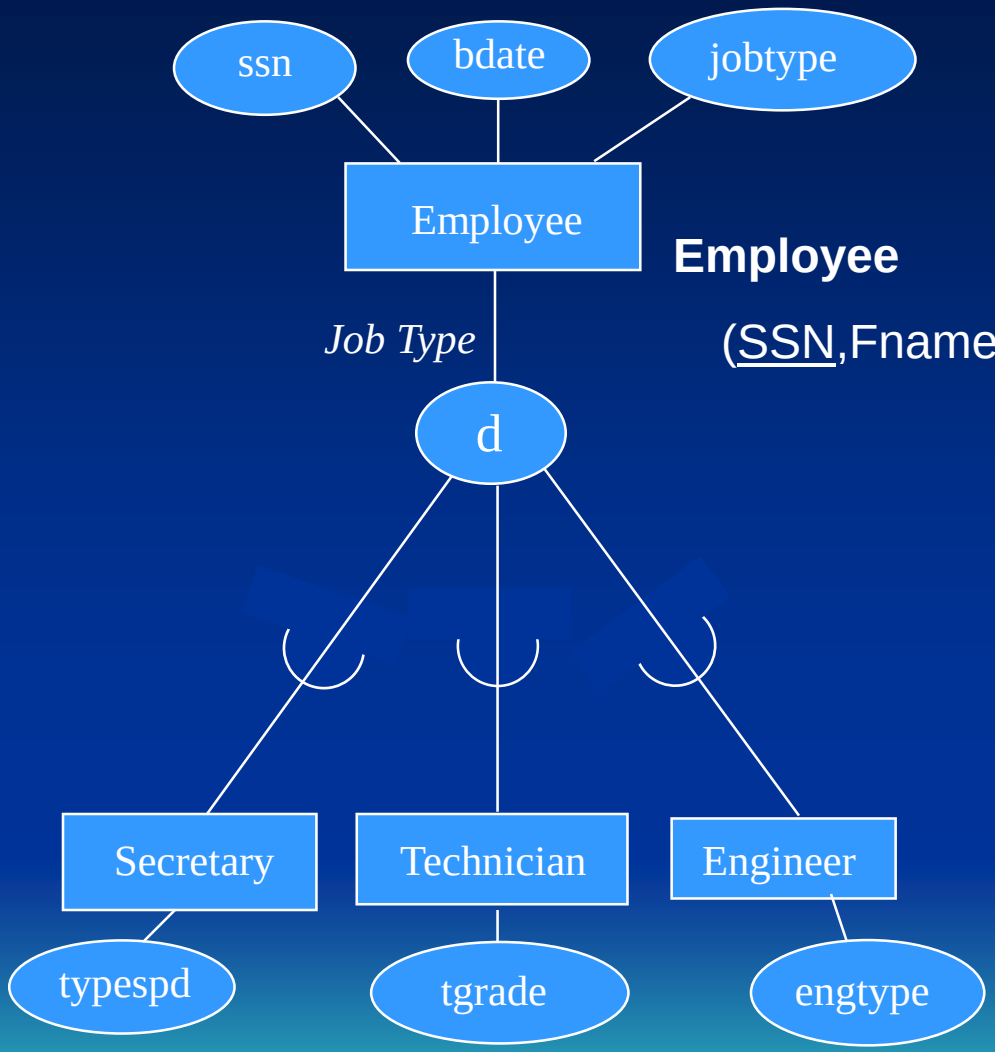


Generalization

- we generalize to get Vehicle



EER to relational



Option A

Works for any kind of constraint: disjoint, overlapping, partial or total

Employee

(SSN,Fname,Minit,Lname,Bdate,Address,JobType)

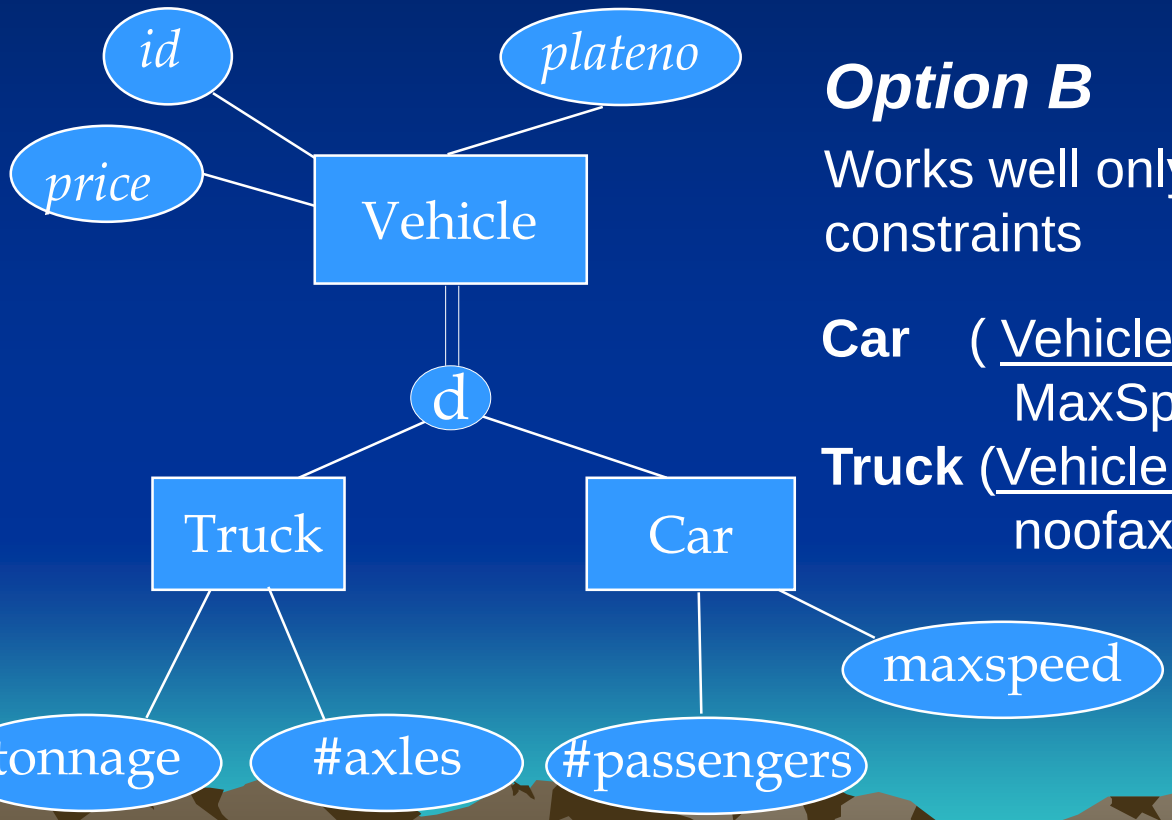
Secretary (SSN, typing Speed)

Technician (SSN, Tgrade)

Engineer (SSN, Engtype)

EER to relational

- Create tables for each subclass, but not for the superclass
- Move all the attributes of the superclass and include them as attributes of each subclass



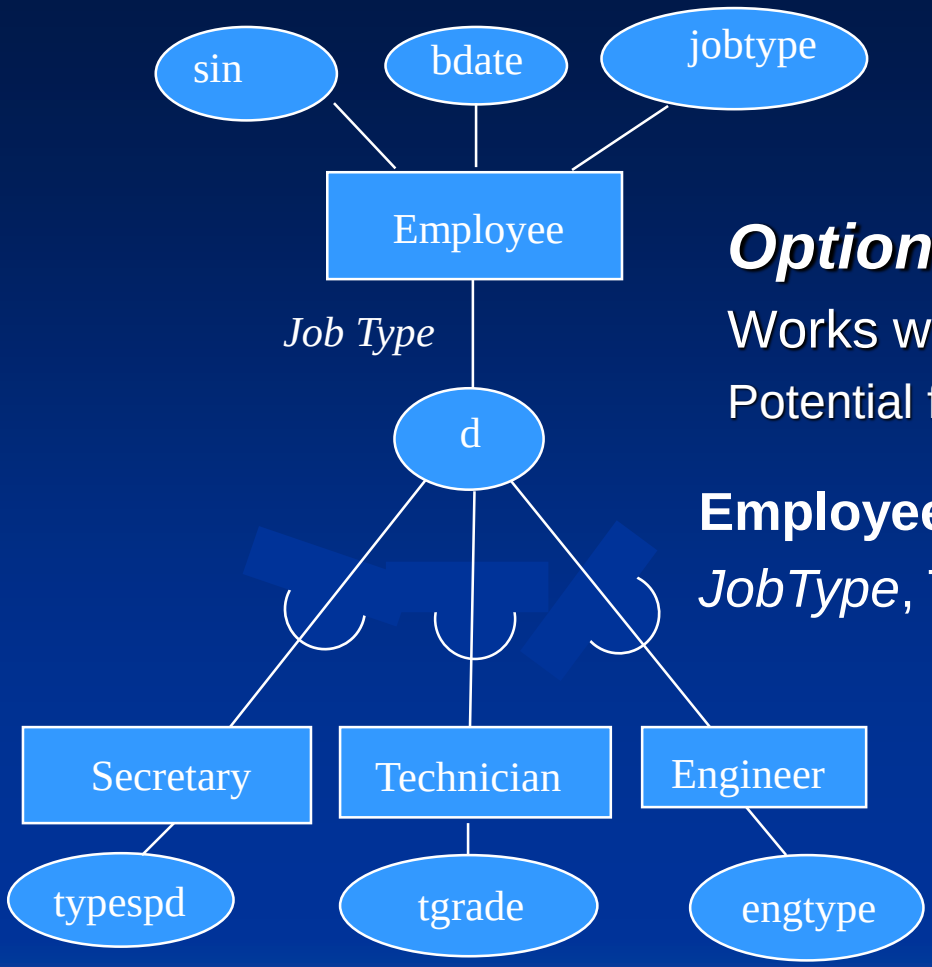
Option B

Works well only for disjoint and total constraints

Car (VehicleID, LicensePlateNo, Price, MaxSpeed, NoOfpassengers)

Truck (VehicleID, LicensePlateNo, Price, noofaxles, tonnage)

EER to relational



Option C

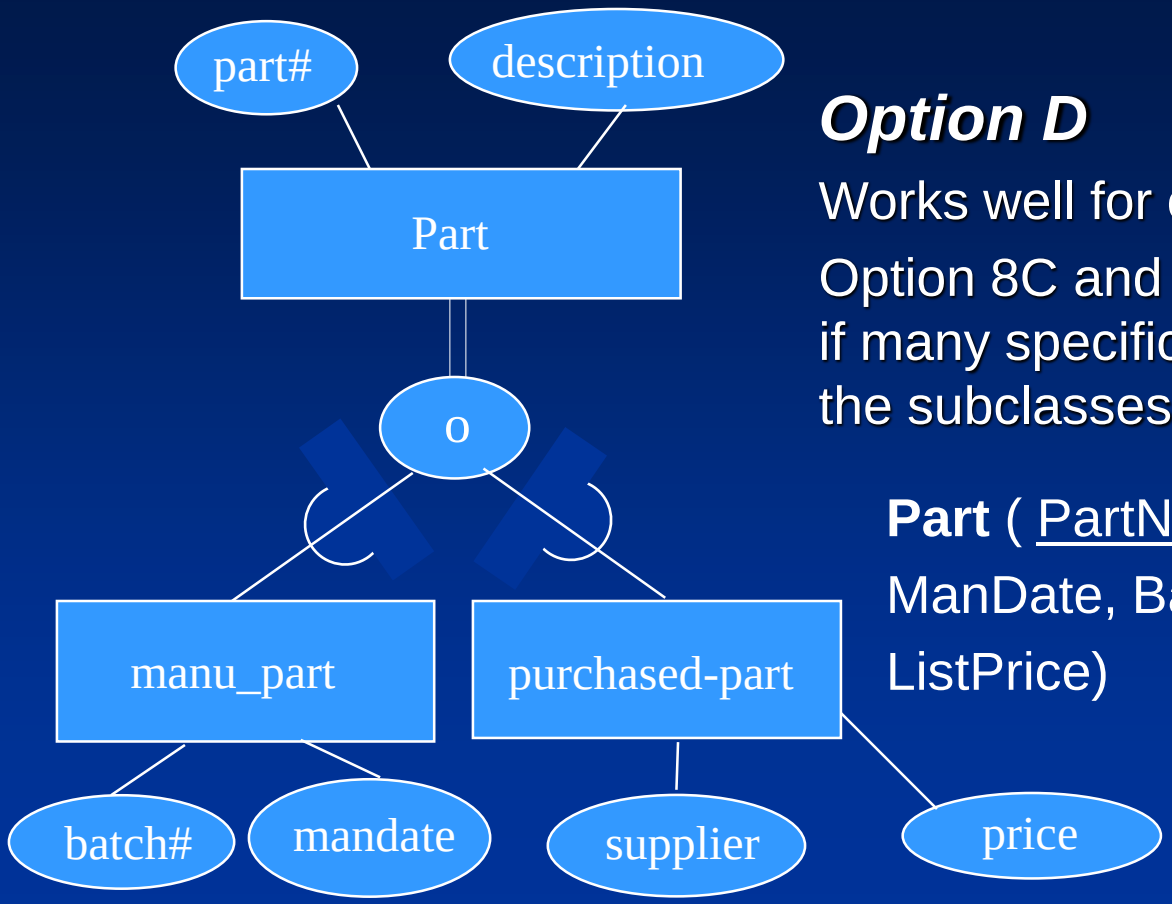
Works well for disjoint constraints

Potential for generating large number of nulls

Employee (SSN, bdate, Address, *JobType*, Typing Speed, Tgrade, EngType)

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

EER to relational



Option D

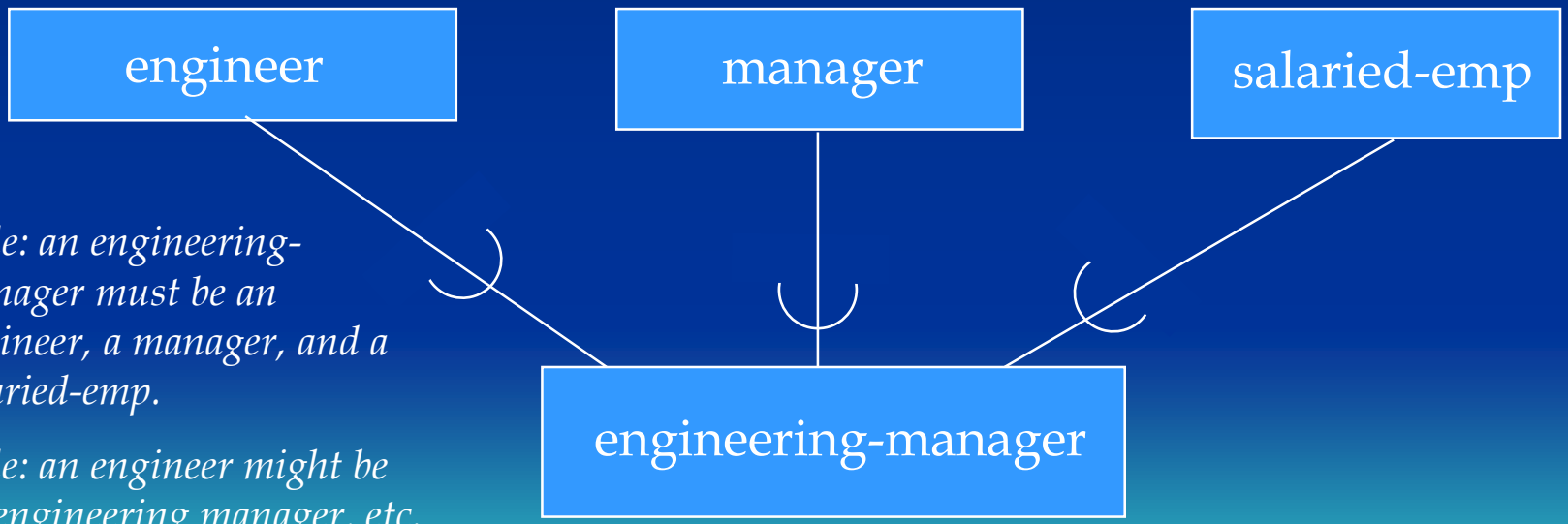
Works well for overlapping constraints

Option 8C and 8D are not recommended if many specific attributes are defined for the subclasses

Part (PartNo, Descr, *Mflag*, DrawingNo, ManDate, BatchNo, *Pflag*, SupName, ListPrice)

1		**screw**		**1**		**…**		**…**				**…**		**…**	
2		bolt				…		…		1		…		…	
3		nail		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

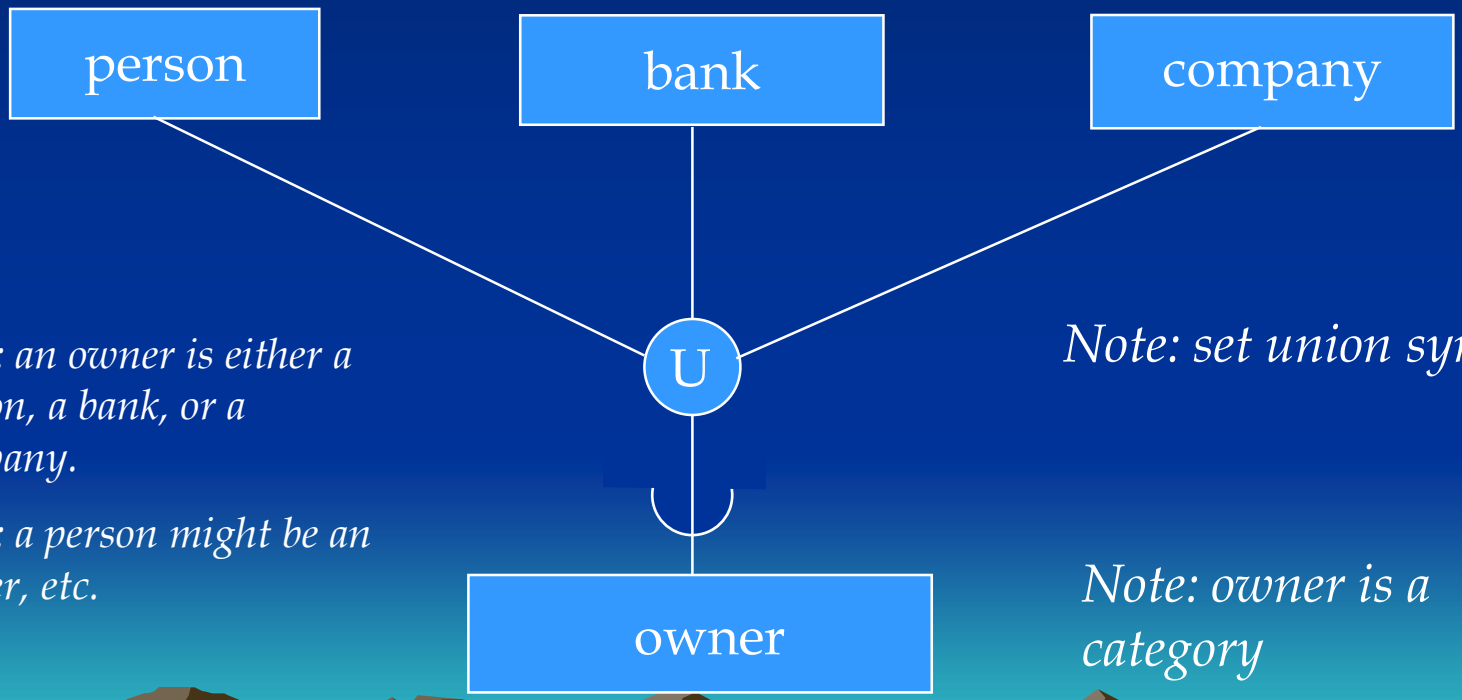


Rule: an engineering-manager must be an engineer, a manager, and a salaried-emp.

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

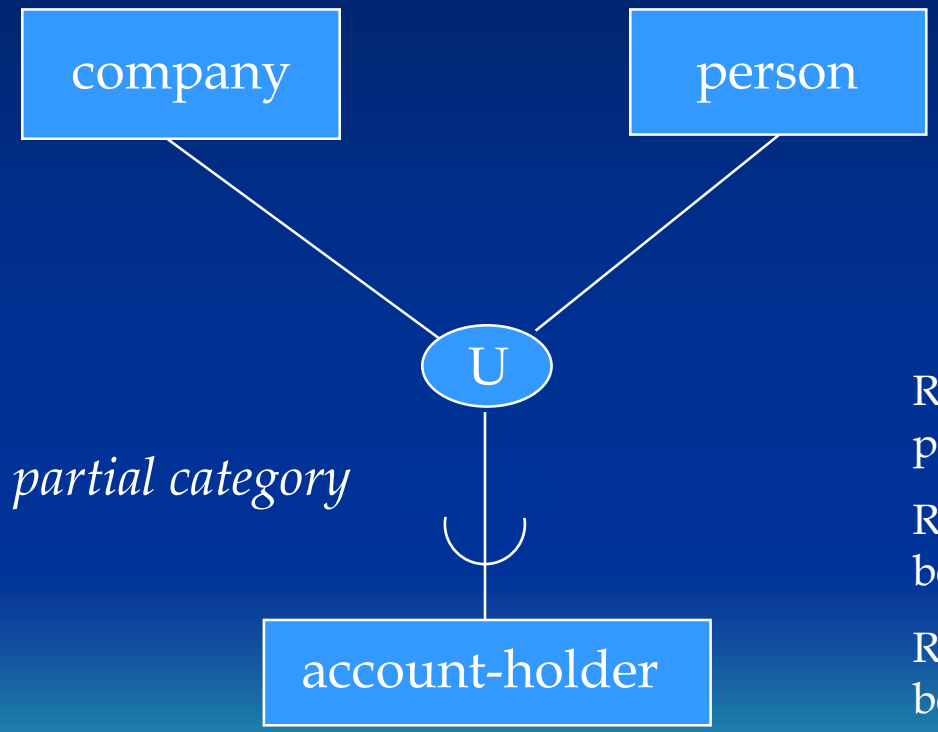
Categories

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



Categories

- A category can be either total or partial



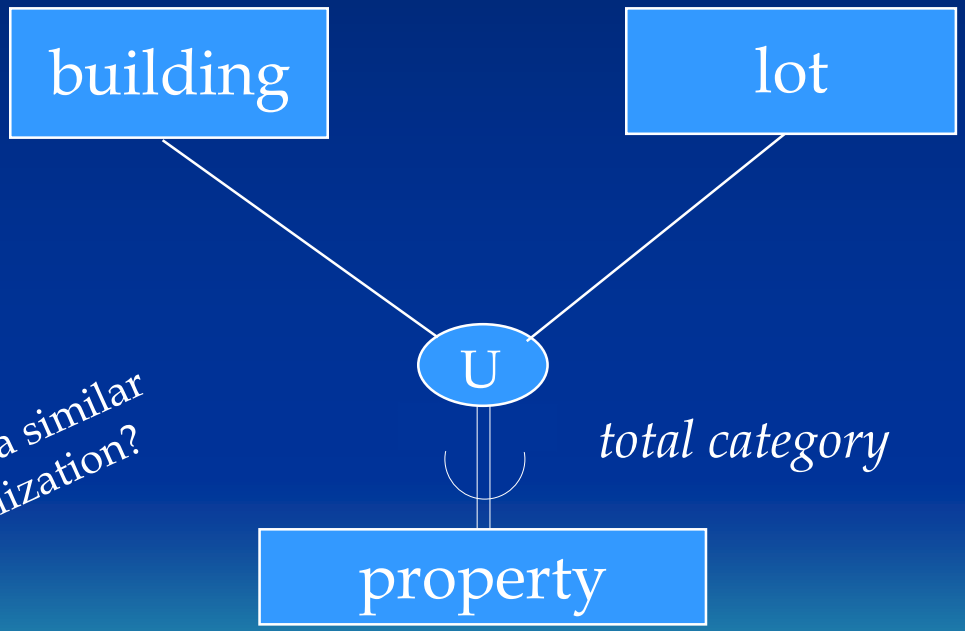
Rule: an account holder is either a person or a company.

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



Is there a similar generalization?

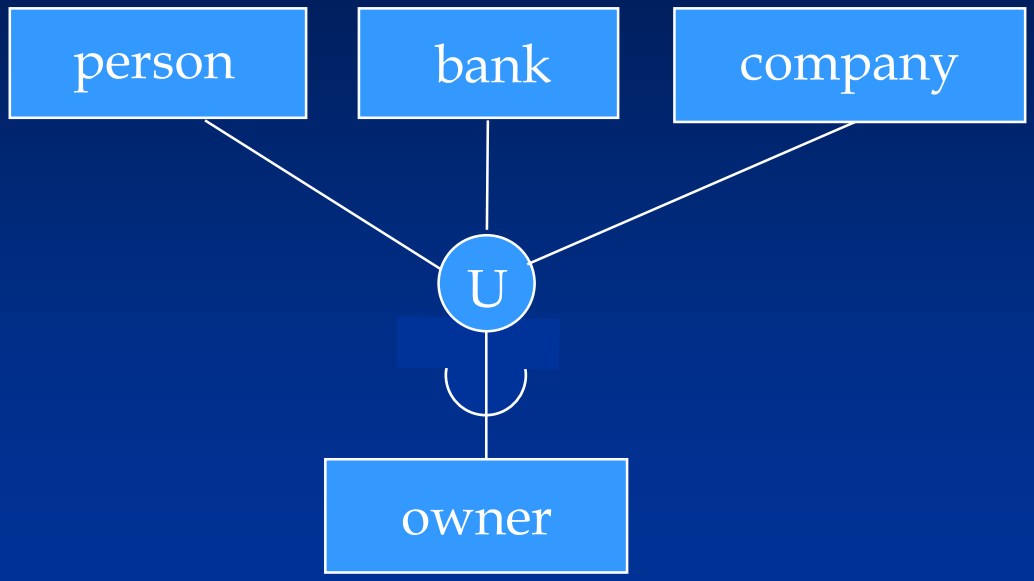
total category

Rule: a property is either a building or a lot

Rule: a building is a property

Rule: a lot is a property

- Categories - Superclasses with different keys



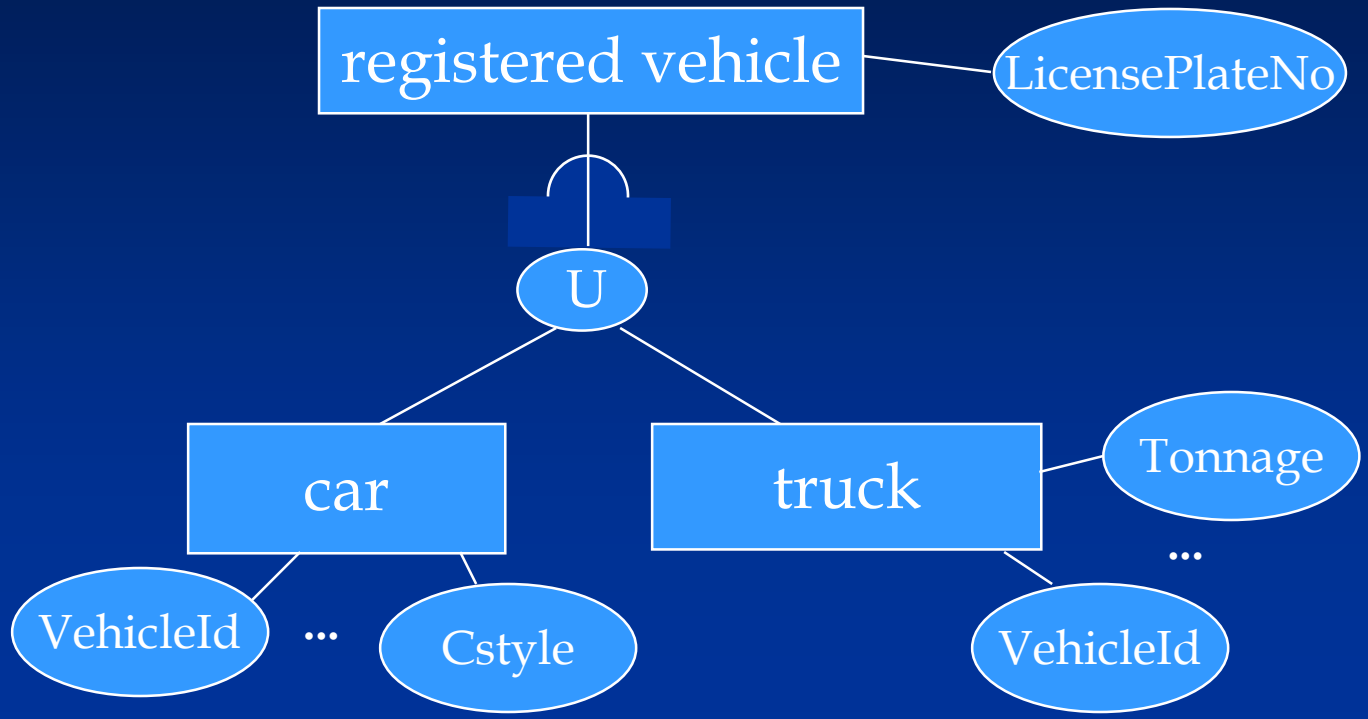
Person (SSN, DrLicNo, Name, Address, *Ownerid*)
Bank (Bname, BAddress, *Ownerid*)
Company (CName, CAddress, *Ownerid*)
Owner (*Ownerid*)

Note the Foreign Keys

Surrogate key

EER to relational

- 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

Spatial databases

- Theme
- Map
- Geographic objects

- **Spatial data management**
  - A spatial database is a data management system for the collection, storage, manipulation and output of spatially referenced information.
  - Theme: refers to data describing a particular topic (e.g., scenic lookouts, rivers, cities) and is the spatial counterpart of an entity type.

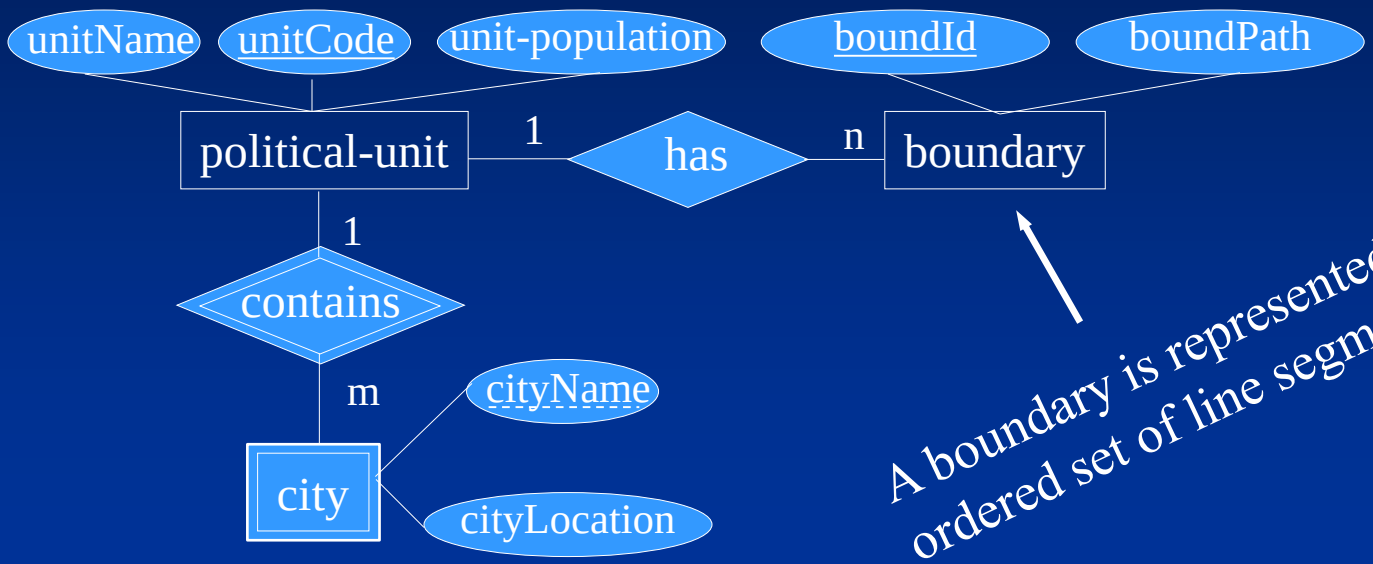
    When a theme is presented on a screen or paper, it is commonly seen in conjunction with a *map*. Colour may be used to indicate different themes (e.g., blue for rivers and black for roads).
  - Map: A map will usually have a scale, legend, and possibly some explanatory text.

- **Spatial data management**
  - Geographic objects: A geographic object is an instance of a theme (e.g., a river).
    - attributes
    - spatial components: geometry and topology

    Geometry refers to the location-based data: shape, length

    Topology refers to spatial relationships among objects: adjacency

A database for political units:



Example: Canada has a boundary for the continental portion, and each of its sovereign islands, such as Prince Edward Island.

Map a geographic data model to tables:

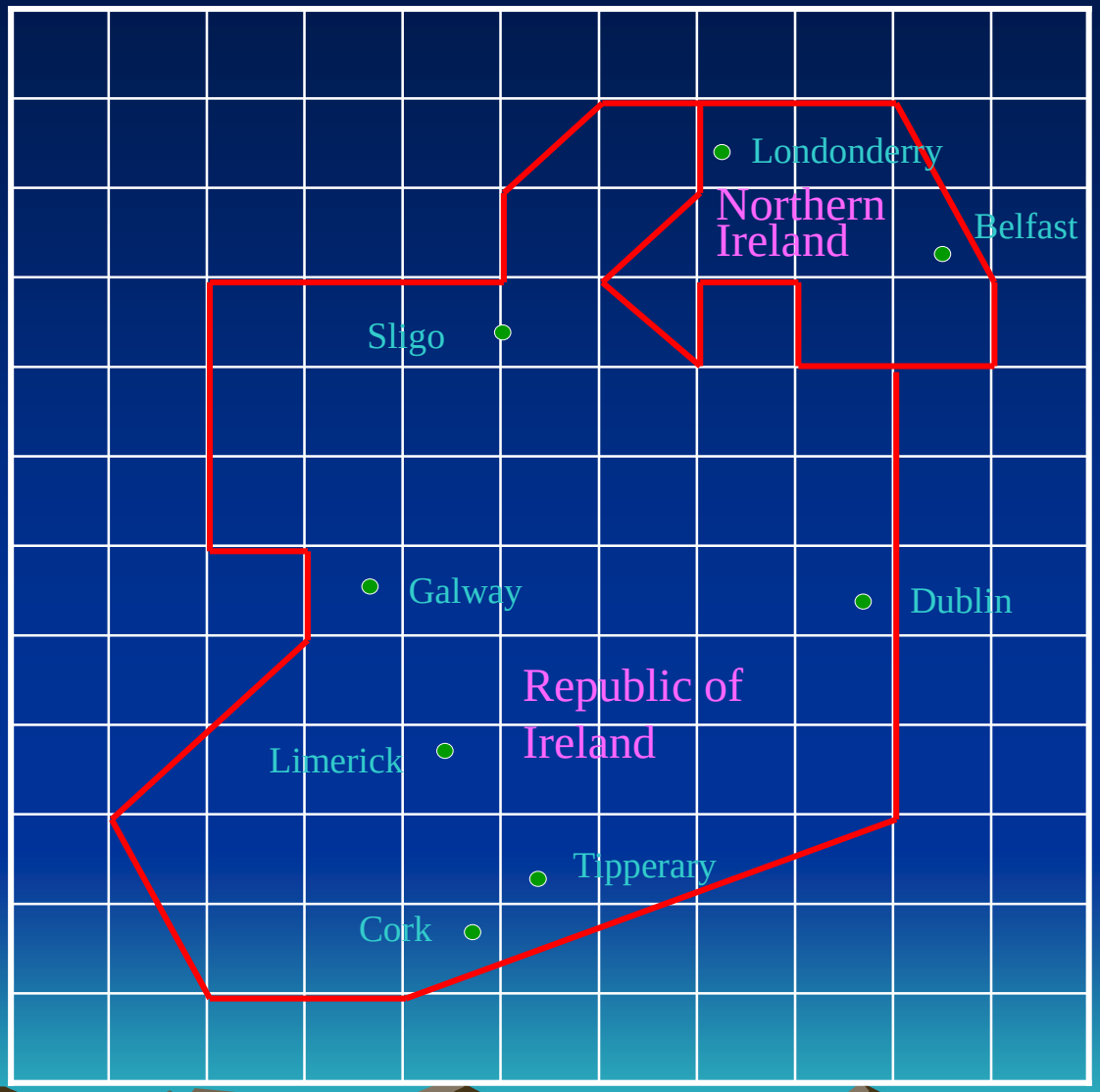
Geometric data type in PostgreSQL:

Geometric type	Representation	Description
BOX	((x1, y1), (x2, y2))	Rectangular box
CIRCLE	<(x, y), r>	Circle (center and radius)
LINE	((x1, y1), (x2, y2))	Infinite line
LSEG	[(x1, y1), (x2, y2)]	Finite line segment
PATH	((x1, y1), …)	Closed path (similar to polygon)
PATH	[(x1, y1), …]	Open path
POINT	(x, y)	Point in space
POLYGON	((x1, y1), …)	Polygon (similar to closed path)

Map a geographic data model to tables:

Table definition for Political unit data model:

```
CREATE TABLE political_unit (
        unitname        VARCHAR(30)        NOT NULL,
        unitcode        CHAR(2),
        unitpop         DECIMAL(6, 2),
        PRIMARY KEY (unitcode));
CREATE TABLE boundary (
        boundid         INTEGER,
        boundpath       PATH               NOT NULL,
        unitcode        CHAR(2),
        PRIMARY KEY (boundid),
        CONSTRAINT fk_boundary_polunit FOREIGN KEY (unitcode) REFERENCES political_unit);
CREATE TABLE city (
        cityname        VARCHAR(30),
        cityloc         POINT              NOT NULL,
        unitcode        CHAR(2),
        PRIMARY KEY (unitcode, cityname),
        CONSTRAINT fk_city_polunit FOREIGN KEY (unitcode) REFERENCES political_unit);
```


Londonderry
Northern Ireland
Belfast
Sligo
Galway
Dublin
Republic of Ireland
Limerick
Tipperary
Cork

Insert statements for populating database

```
INSERT INTO political_unit VALUES ('Republic of Ireland', 'ie', 3.9);
INSERT INTO political_unit VALUES ('Northern Ireland', 'ni', 1.7);
INSERT INTO boundary VALUES
        (1, '[(9, 8), (9, 3), (4, 1), (2, 2), (1, 3), (3, 5), (3, 6), (2, 6),
        (2, 9), (5, 9), (5, 10), (6, 11), (7, 11), (7, 10), (6, 9), (7, 8),
        (7, 9), (8, 9), (8, 8), (9, 8)]', 'ie');
INSERT INTO boundary VALUES
        (2, '[(7, 11), (9, 11), (10, 9), (10, 8), (8, 8), (8, 9), (7, 9),
        (7, 8), (6, 9), (7, 10), (7, 11)]', 'ni');
INSERT INTO city VALUES ('Dublin', '(9, 6)', 'ie');
INSERT INTO city VALUES ('Cork', '(5, 2)', 'ie');
INSERT INTO city VALUES ('Limerick', '(4, 4)', 'ie');
INSERT INTO city VALUES ('Galway', '(4, 6)', 'ie');
INSERT INTO city VALUES ('Sligo', '(9, 6)', 'ie');
INSERT INTO city VALUES ('Tipperary', '(5, 3)', 'ie');
INSERT INTO city VALUES ('Belfast', '(9, 9)', 'ni');
INSERT INTO city VALUES ('Londonderry', '(7, 10)', 'ni');
```

Geometric functions and operators in PostgreSQL for processing spatial data

Functions:

Function	Returns	Description
LENGTH(OBJECT)	double precision	length of item
NPOINTS(PATH)	integer	Number of points

Operators:

Operator	Description
<->	Distance between
<<	Is left of?
<^	Is below?
>>	Is right of?
>^	Is above?

Queries:

1. What is the length of the Republic of Ireland border?

```
SELECT SUM(LENGTH((boundpath))) * 37.5
     AS “Border (kms)” FROM Political_unit, boundary
WHERE unitname = “Republic of Ireland”
AND political_unit.unitcode = boundary.unitcode;
```

Border (kms)
1353.99

Queries:

2. How far, as the crow flies, is it from Sligo to Dublin?

```
SELECT (orig.cityloc<->dest.cityloc) * 37.5
      AS “Distance (kms)”
FROM city orig, city dest
WHERE orig.cityname = ‘Sligo’
AND dest.cityname = ‘Dublin’;
```

Distance (kms)
167.71

Queries:

3. What is the closest city to Limerick?

```
SELECT dest.cityname FROM city orig, city dest
WHERE orig.cityname = 'Limerick'
AND orig.cityloc <-> dest.cityloc =
(SELECT MIN(orig.cityloc<->dest.cityloc)
FROM city orig, city dest
WHERE orig.cityname = 'Limerick' AND
                dest.cityname <> 'Limerick');
```

cityname
Tipperary

Queries:

4. What is the westernmost city?

```
SELECT west.cityname FROM city west
WHERE NOT EXISTS
    (SELECT * FROM city other
    WHERE other.cityloc << west.cityloc);
```

cityname
Limerick
Galway

Queries:

5. What is the westernmost city in the Republic of Ireland?

```
SELECT west.cityname FROM city west, political_unit p
WHERE west.unitcode = p.unitcode
      and p.unitname = 'Republic of Ireland'
      and NOT EXISTS
            (SELECT * FROM city other
            WHERE other.cityloc << west.cityloc
                  and other.unitcode = p.unitcode
                  and other.unitname = 'Republic of Ireland')
```

cityname
Limerick
Galway

- **Managing temporal data**
  - With a temporal database, stored data have an associated time period indicating when the item was valid or stored in the database.
  - Transaction time: the timestamp applied by the system when data are entered and cannot be changed by an application. It can be applied to a particular item or row.

    For example, the old and new price of a product would automatically have separate timestamps when they are entered into the database.

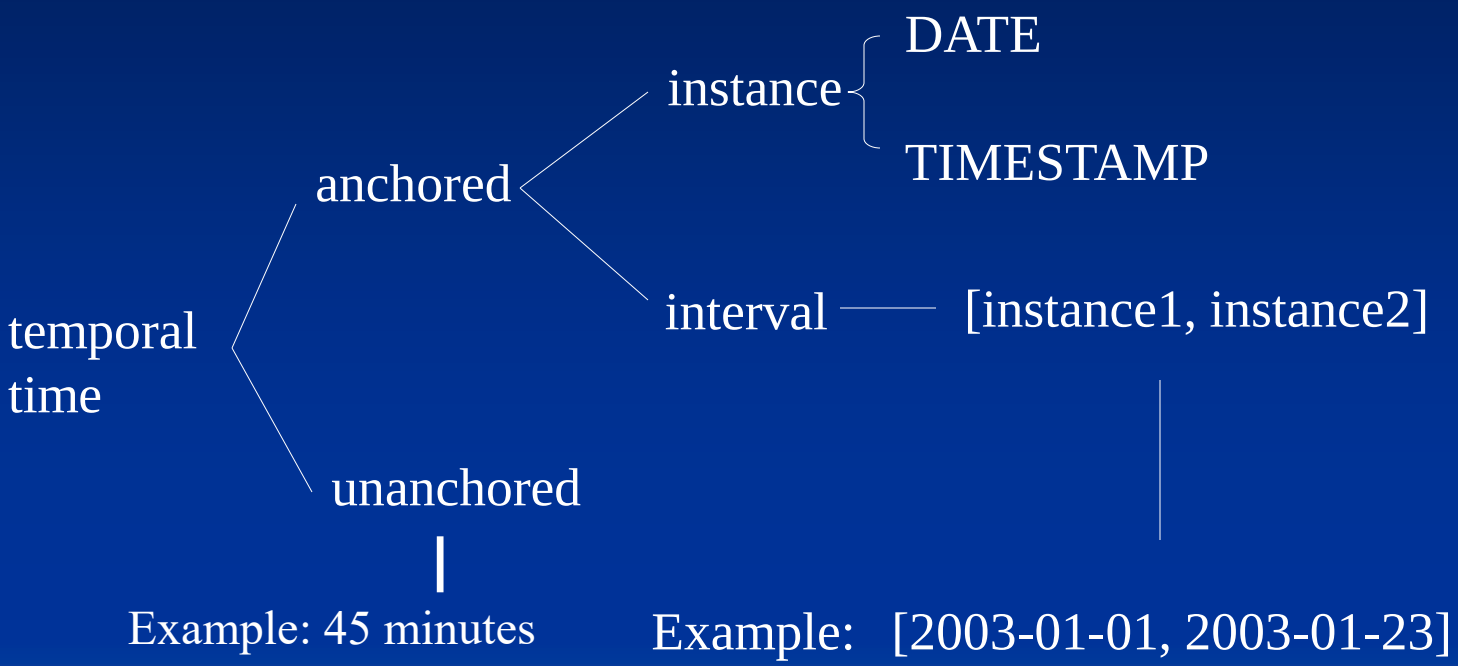
- Valid time: the actual time at which an item was a valid or true value. It can be changed by an application.
  For example, consider the case where a firm plans to increase its prices on a specific date. It might post new prices some time before their effective date.

Difference between transaction time and valid time:

Valid time records when the change takes effect, and transaction time records when the change was entered.

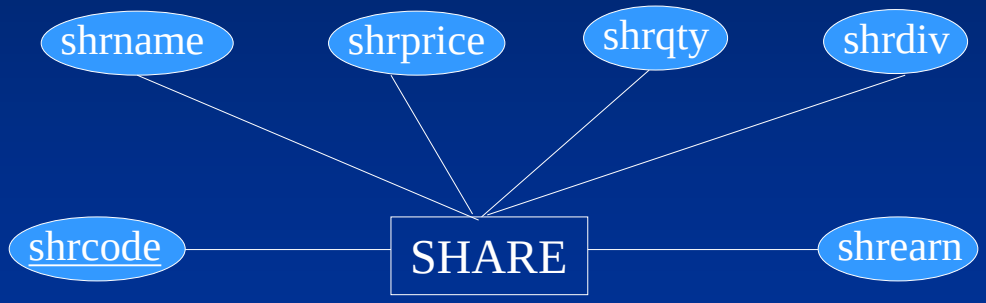
- Storing transaction time is essential for database recovery because the DBMS can roll back the database to a previous state.
- Valid time provides a historic record of the state of the database.

- Anchored time: a time having a defined starting point (e,g., October 15, 2003)



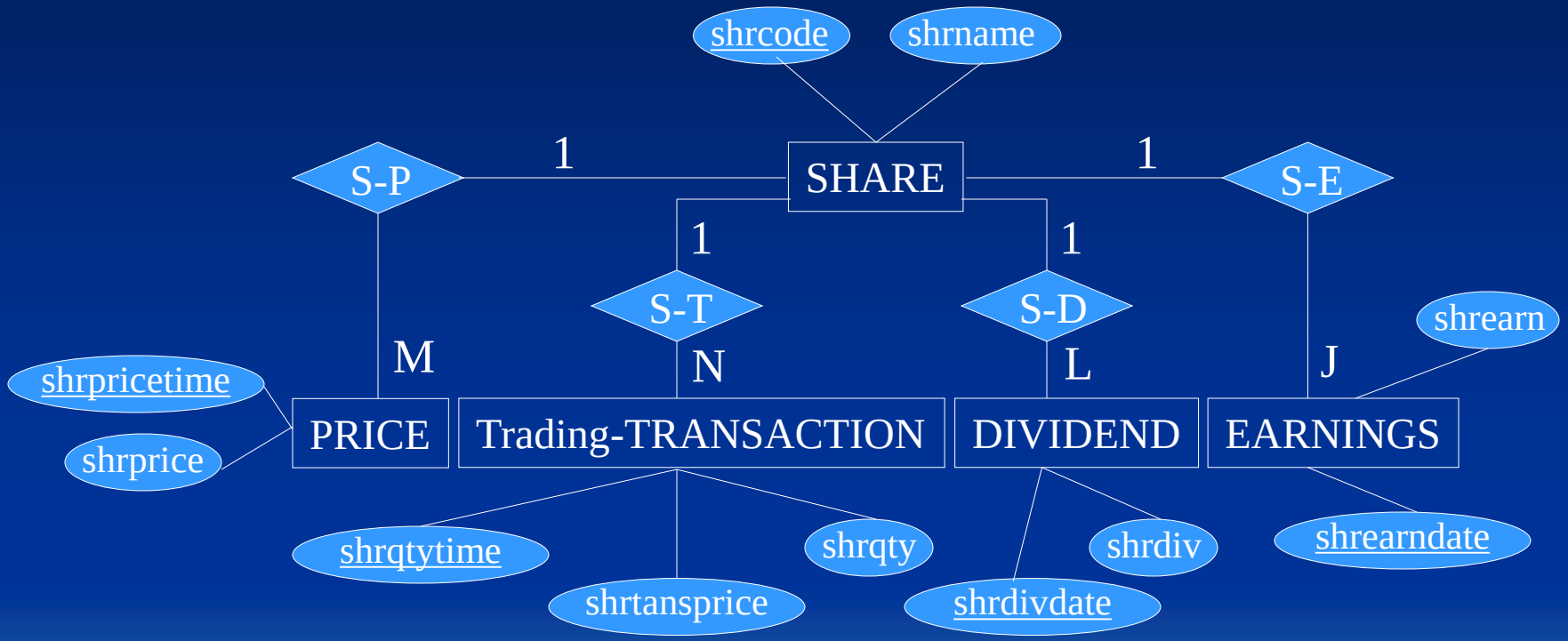
Modeling temporal data

Consider an application for managing information on Shares:



However, *share price*, *quantity owned*, *dividend* and *price-to-earning* ratio are all time-varying. The above data model is not able to capture this feature. So temporal information should be added.

Modeling temporal data



Not included in the final

Signature files and signature trees

- Signatures
  - signature for attributes
  - signature for records
- Signature files
- Signature trees

- **Signature file**
  - A signature file is a set of bit strings, which are called *signatures*.
  - In a signature file, each signature is constructed for a record in a table, a block of text, or an image.
  - When a query arrives, a query signature will be constructed according to the key words involved in the query. Then, the signature file will be searched against the query signature to discard non-qualifying signatures, as well as the objects represented by those signatures.

- **Signature file**

  - Decompose an attribute value (or a key word) into a series of triplets
  - Using a hash function to map a triplet to an integer *p*, indicating that the *p*th bit in the signature will be set to 1.

Example: Consider the word “professor”. We will decompose it into 6 triplets:
“pro”, “rof”, “ofe”, “fes”, “ess”, “sor”.
Assume that hash(pro) = 2, hash(rof) = 4, hash(ofe) =8, and hash(fes) = 9.

Signature: 010 100 011 000

- **Signature file**
  - Generate a signature for a record (or a block of text)

block: ... SGML ... databases ... information ...

word signature:

SGML		010 000 100 110
database		100 010 010 100
information	$\vee$	010 100 011 000
object signature (OS)		110 110 111 110

superimposing

- **Signature file**
  - Search a signature file

block: ... SGML ... databases ... information ...

object signature (OS): 110 110 111 110

queries:	query signatures:	matching results:
SGML	010 000 100 110	match with OS
XML	011 000 100 100	no match with OS
informatik	110 100 100 000	false drop

- **Signature file**
  - Generate a signature for a record (or a block of text)

relation:

name	sex	
John	male	… …
…	…	

signature file:

s_1	1011 0110
s_2	1011 1001
s_3	1010 0111
s_4	0111 0110
s_5	0111 0101
s_6	0101 1100
s_7	1110 0100
s_8	1010 1011

- **Signature file**
  - Search a signature file

s_1	1011 0110
s_2	1011 1001
s_3	1010 0111
s_4	0111 0110
s_5	0111 0101
s_6	0101 1100
s_7	1110 0100
s_8	1010 1011

query: John $\wedge$ male

$\Downarrow$

query signature: 1010 0101

- **Signature tree**
  - Signature tree construction

Consider a signature s_i of length m. We denote it as $s_i = s_i[1] .. s_i[m]$, where each $s_i[j] \in \{0, 1\}$ ($j = 1, ..., m$). We also use $s_i(j_1, ..., j_h)$ to denote a sequence of pairs with respect to s_i: $(j_1, s_i[j_1])(j_2, s_i[j_2]) ... (j_h, s_i[j_h])$, where $1 \leq j_k \leq m$ for $k \in \{1, ..., h\}$.

Definition (*signature identifier*) Let $S = s_1.s_2 ... .s_n$ denote a signature file. Consider s_i ($1 \leq i \leq n$). If there exists a sequence: $j_1, ..., j_h$ such that for any $k \neq i$ ($1 \leq k \leq n$) we have $\boldsymbol{s_i(j_1, ..., j_h)} \neq s_k(j_1, ..., j_h)$, then we say $s_i(j_1, ..., j_h)$ identifies the signature s_i or say $s_i(j_1, ..., j_h)$ is an identifier of s_i.

- **Signature tree**
  - Signature tree construction

s_1	1011 0110
s_2	1011 1001
s_3	1010 0111
s_4	0111 0110
s_5	0111 0101
s_6	0101 1100
s_7	1110 0100
s_8	1010 1011

Example:

$s_8(5, 1, 4) = (5, 1)(1, 1)(4, 0)$

For any $i \neq 8$ we have $s_i(5, 1, 4) \neq s_8(5, 1, 4)$. For instance, $s_5(5, 1, 4) = (5, 0)(1, 0)(4, 1) \neq s_8(5, 1, 4)$, $s_2(5, 1, 4) = (5, 1)(1, 1)(4, 1) \neq s_8(5, 1, 4)$, and so on.

$s_1(5, 4, 1) = (5, 0)(4, 1)(1, 1)$

For any $i \neq 1$ we have $s_i(5, 4, 1) \neq s_1(5, 4, 1)$.

- **Signature tree**

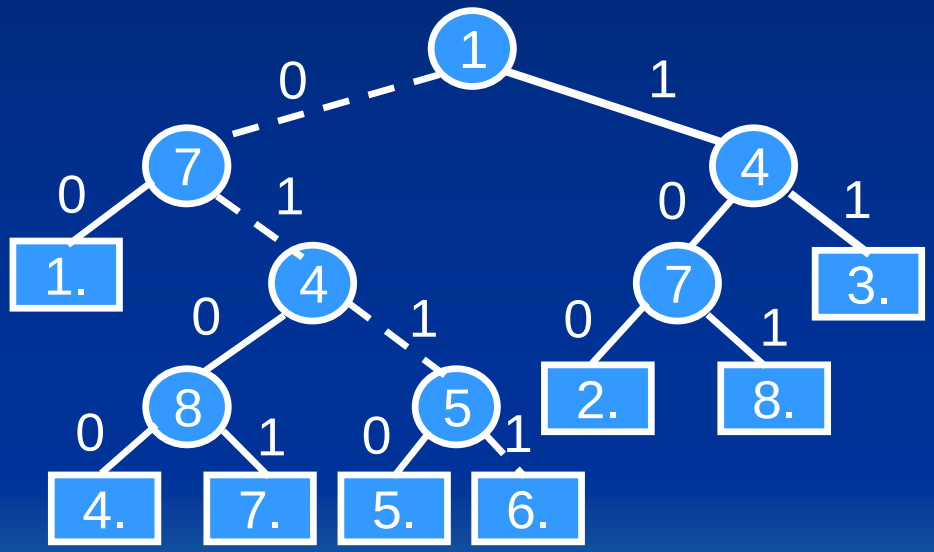
  - Signature tree construction

Definition (*signature tree*) A signature tree for a signature file $S = s_1.s_2 \ldots .s_n$, where $s_i \neq s_j$ for $i \neq j$ and $|s_k| = m$ for $k = 1, \ldots, n$, is a binary tree T such that

1. For each internal node of T, the left edge leaving it is always labeled with 0 and the right edge is always labeled with 1.
2. T has n leaves labeled 1, 2, ..., n, used as pointers to n different positions of s_1, s_2, ... and s_n in S. Let v be a leaf node. Denote $p(v)$ the pointer to the corresponding signature.
3. Each internal node v is associated with a number, denoted $sk(v)$, to tells which bit will be checked.
4. Let $i_1, \ldots, i_h$ be the numbers associated with the nodes on a path from the root to a leaf v labeled i (then, this leaf node is a pointer to the ith signature in S, *i.e.*, $p(v) = i$). Let $p_1, \ldots, p_h$ be the sequence of labels of edges on this path. Then, $(j_1, p_1) \ldots (j_h, p_h)$ makes up a signature identifier for s_i, $s_i(j_1, \ldots, j_h)$.

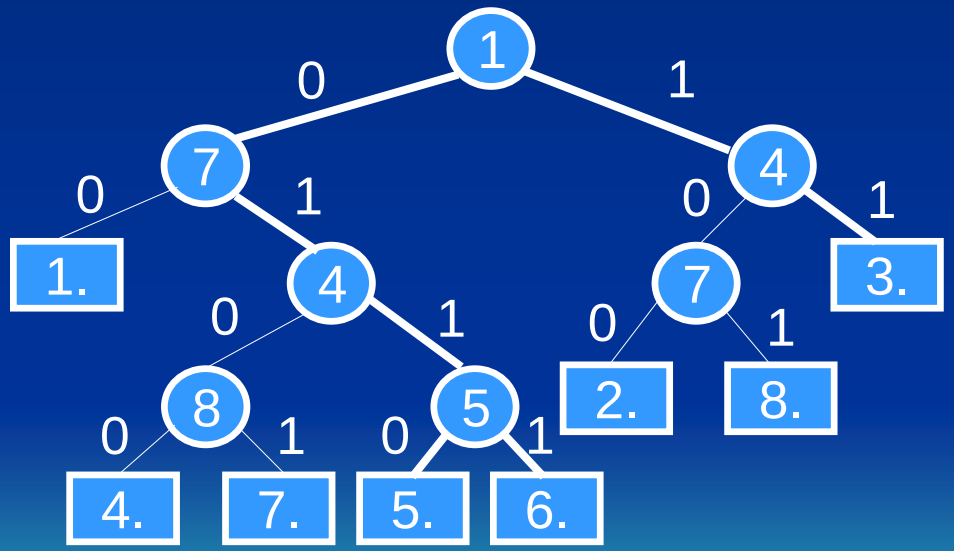
- **Signature tree**
  - Signature tree construction

s_1 011 001 000 101
s_2 111 011 001 111
s_3 111 101 010 111
s_4 011 001 101 111
s_5 011 101 110 101
s_6 011 111 110 101
s_7 011 001 111 111
s_8 111 011 111 111



- **Signature tree**
  - Searching of a signature tree

    query signature: s_q = 000 100 100 000.



- **Signature tree**
  - About balanced signature trees

    A signature tree can be quite skewed.

S1: 100 100 100 100
S2: 010 010 010 010
S3: 001 001 001 001
S4: 000 110 010 010
S5: 000 011 001 001
S6: 000 001 100 100
S7: 000 000 110 010
S8: 000 000 010 110

- **Signature tree**
  - About balanced signature trees

    Weight-based method:

    A signature file $S = s_1.s_2 \ ... \ .s_n$ can be considered as a boolean matrix. We use $S[i]$ to represent the ith column of S. We calculate the weight of each $S[i]$, *i.e.*, the number of 1s appearing in $S[i]$, denoted $w(S[i])$. Then, we choose an j such that $|w(S[i]) - n/2|$ is minimum. Here, the tie is resolved arbitrarily. Using this j, we divide S into two groups $g_1 = \{ s_{i_1}, ..., s_{i_k} \}$ with each $s_{i_p}[j] = 0$ ($p = 1, ..., k$) and $g_2 = \{ s_{i_{k+1}}, ..., S_{i_n} \}$ with each $s_{i_q}[j] = 1$ ($q = k + 1, ..., n$).

- **Signature tree**
  - About balanced signature trees

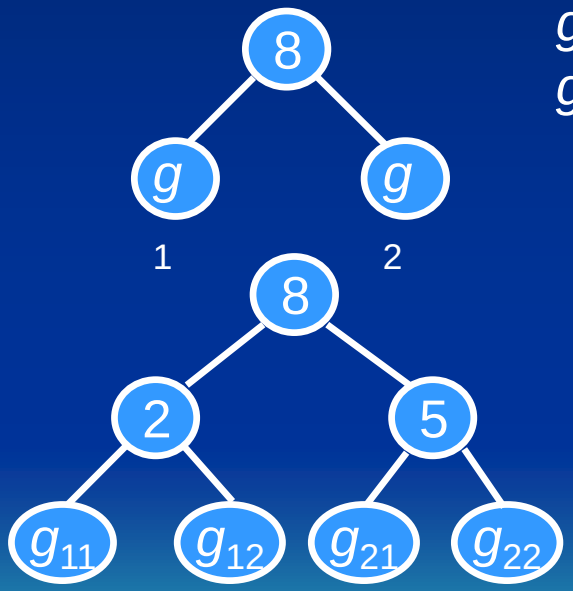
    Weight-based method (continued):

In a next step, we consider each g_i (i = 1, 2) as a single signature file and perform the same operations as above, leading to two trees generated for g_1 and g_2, respectively. Replacing g_1 and g_2 with the corresponding trees, we get another tree. We repeat this process until the leaf nodes of a generated tree cannot be divided any more.

- **Signature tree**
  - About balanced signature trees

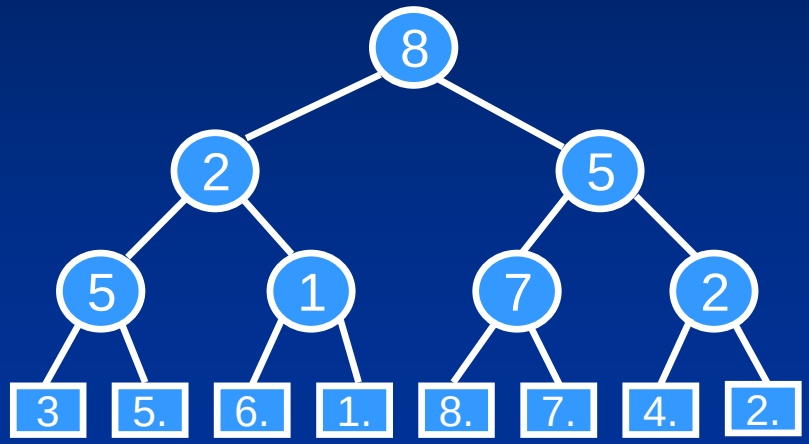
    Example:

S1: 100 100 100 100
S2: 010 010 010 010
S3: 001 001 001 001
S4: 000 110 010 010
S5: 000 011 001 001
S6: 000 001 100 100
S7: 000 000 110 010
S8: 000 000 010 110



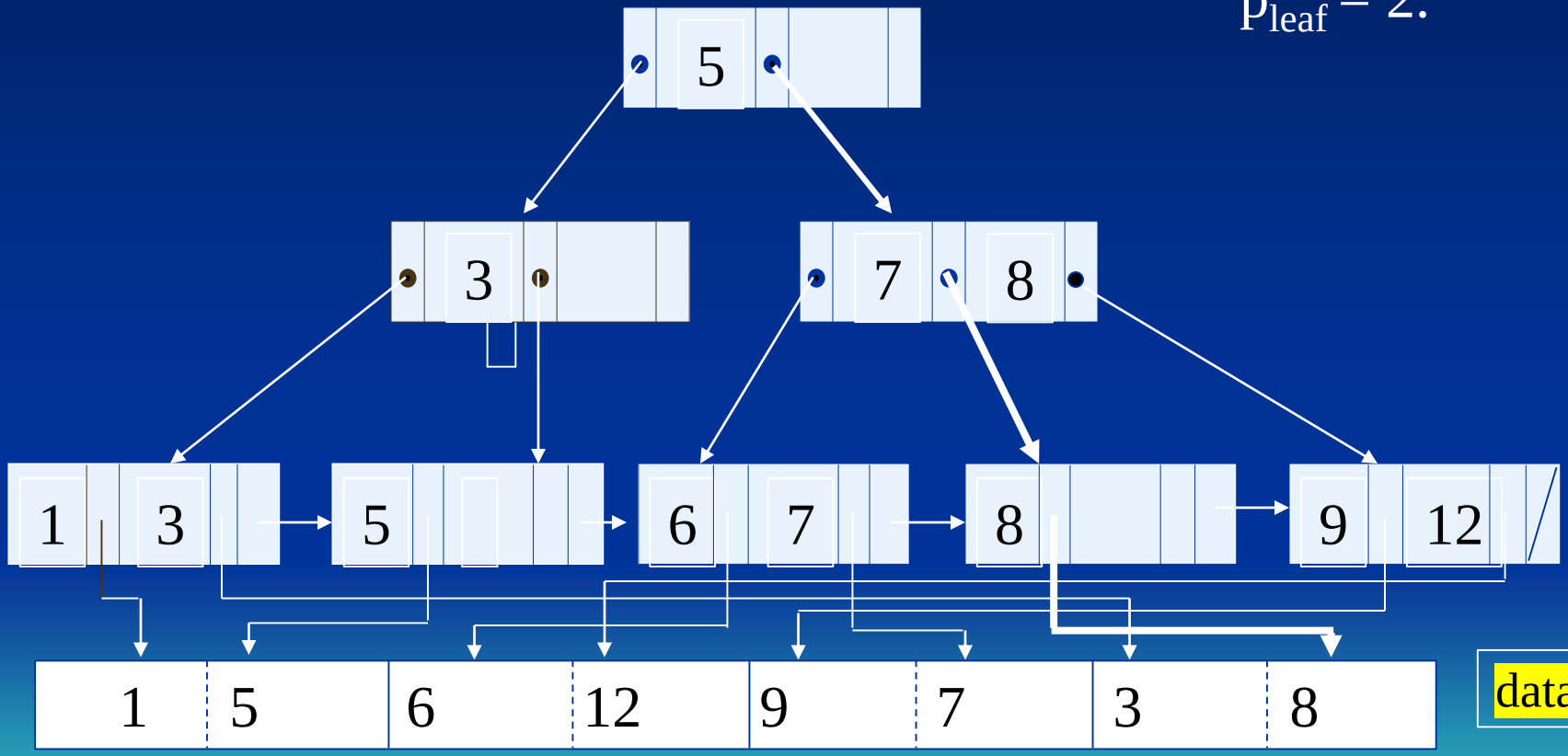
$g_1 = \{s_1, s_3, s_5, s_6\}$
$g_2 = \{s_2, s_4, s_7, s_8\}$

$g_{11} = \{s_3, s_5\}$
$g_{12} = \{s_6, s_1\}$
$g_{21} = \{s_8, s_7\}$
$g_{22} = \{s_4, s_2\}$


8
2
5
5
1
7
2
3
5.
6.
1.
8.
7.
4.
2.

A B+-tree

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



B^+-tree Maintenance

- Inserting a key into a B^+-tree

  (Same as discussed on B^+-tree construction)

- Deleting a key from a B^+-tree

  i) Find the leaf node containing the key to be removed and delete it from the leaf node.

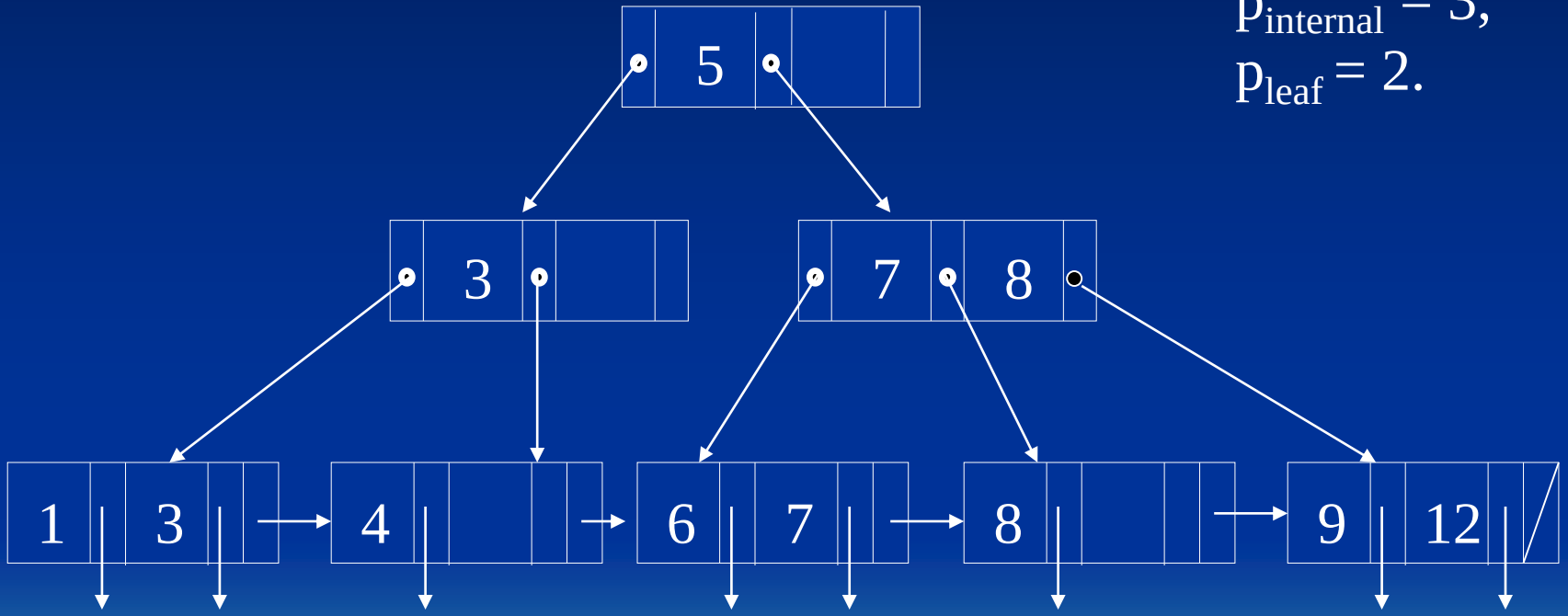
  ii) If *underflow*, redistribute the leaf node and one of its siblings (left or right) so that both are at least half full.

  iii) Otherwise, the node is merged with its siblings and the number of leaf nodes is reduced.

Entry deletion

- deletion sequence: 8, 12, 9, 7

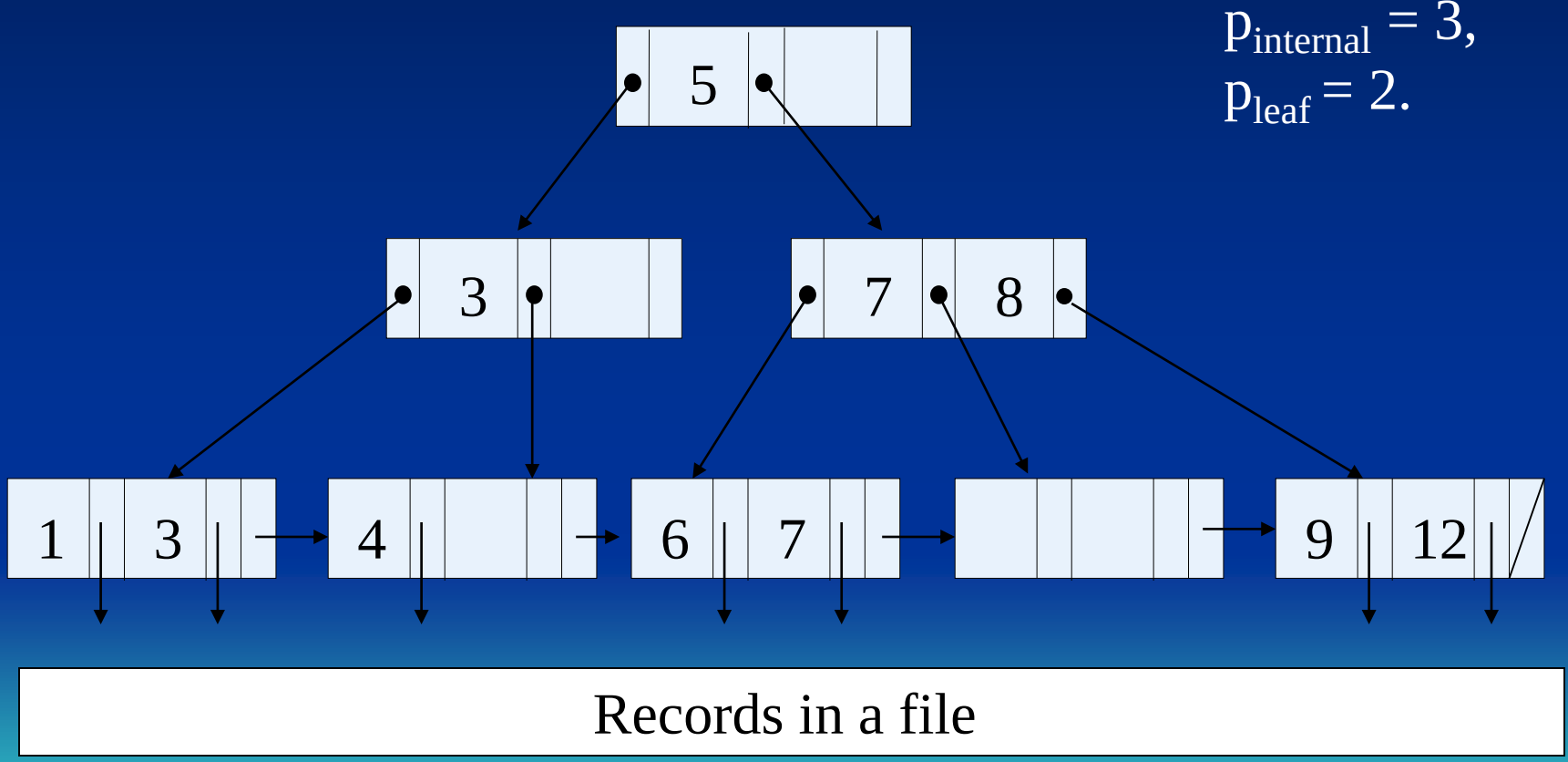
$p_{internal} = 3$,
$p_{leaf} = 2$.



Records in a file

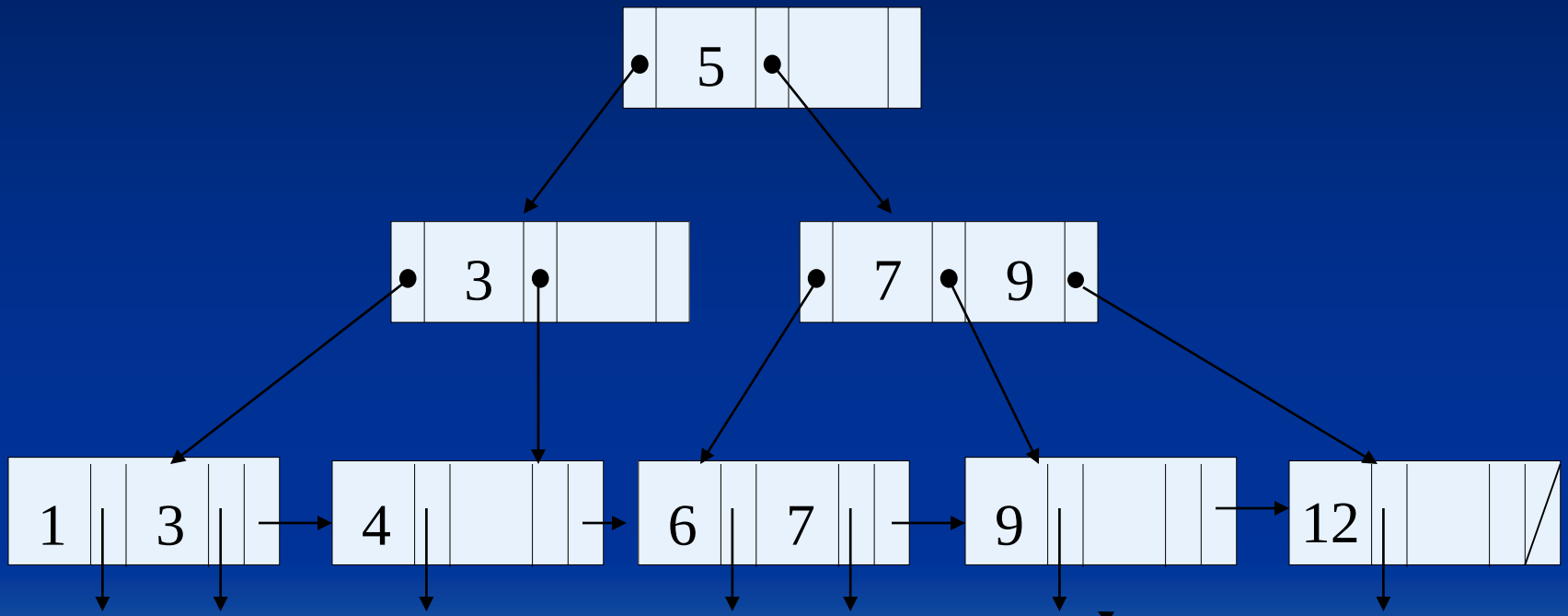
Entry deletion

- deletion sequence: 8, 12, 9, 7



Entry deletion

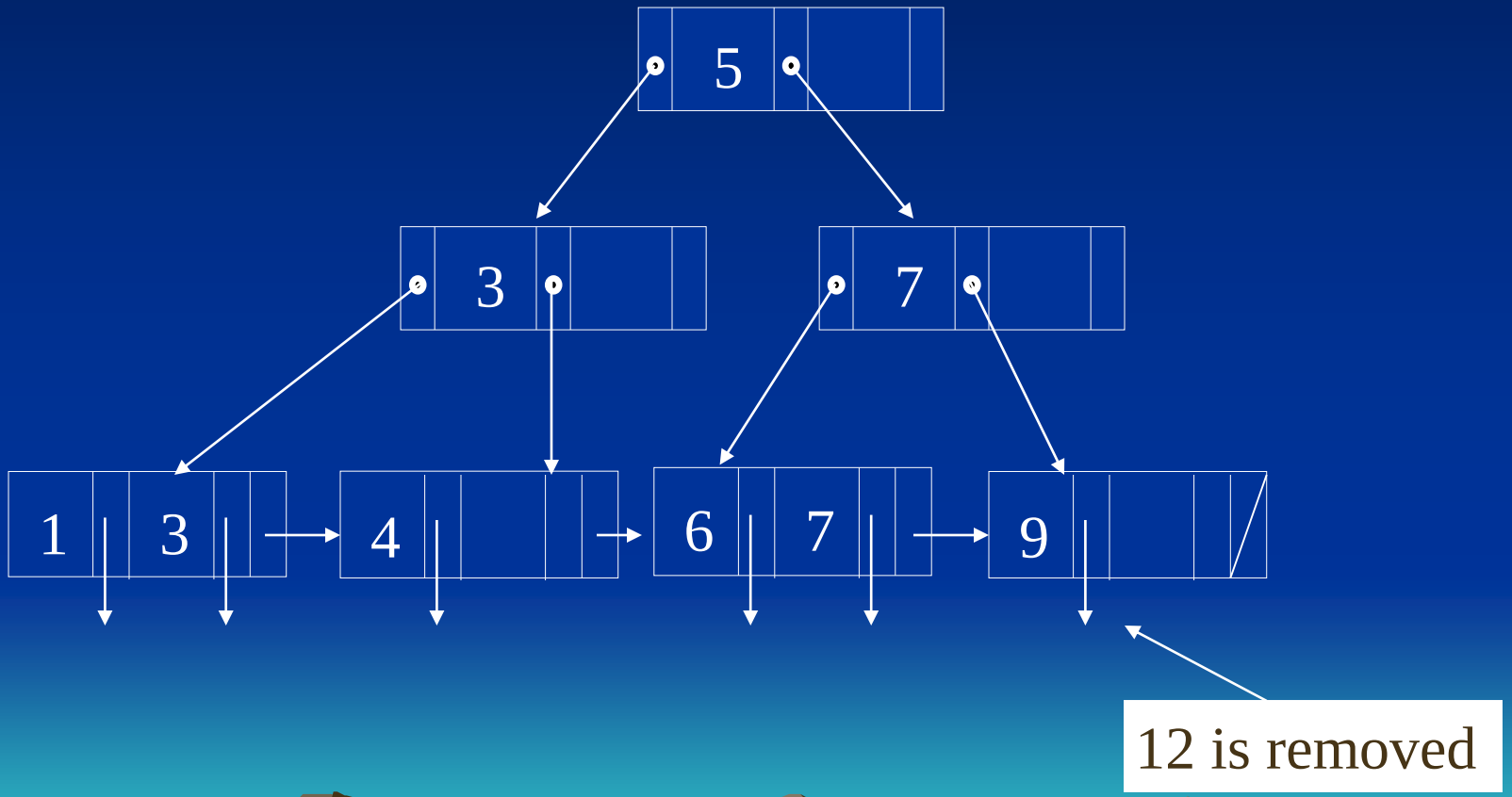
- deletion sequence: 8, 12, 9, 7



Deleting 8 causes the node redistribute.

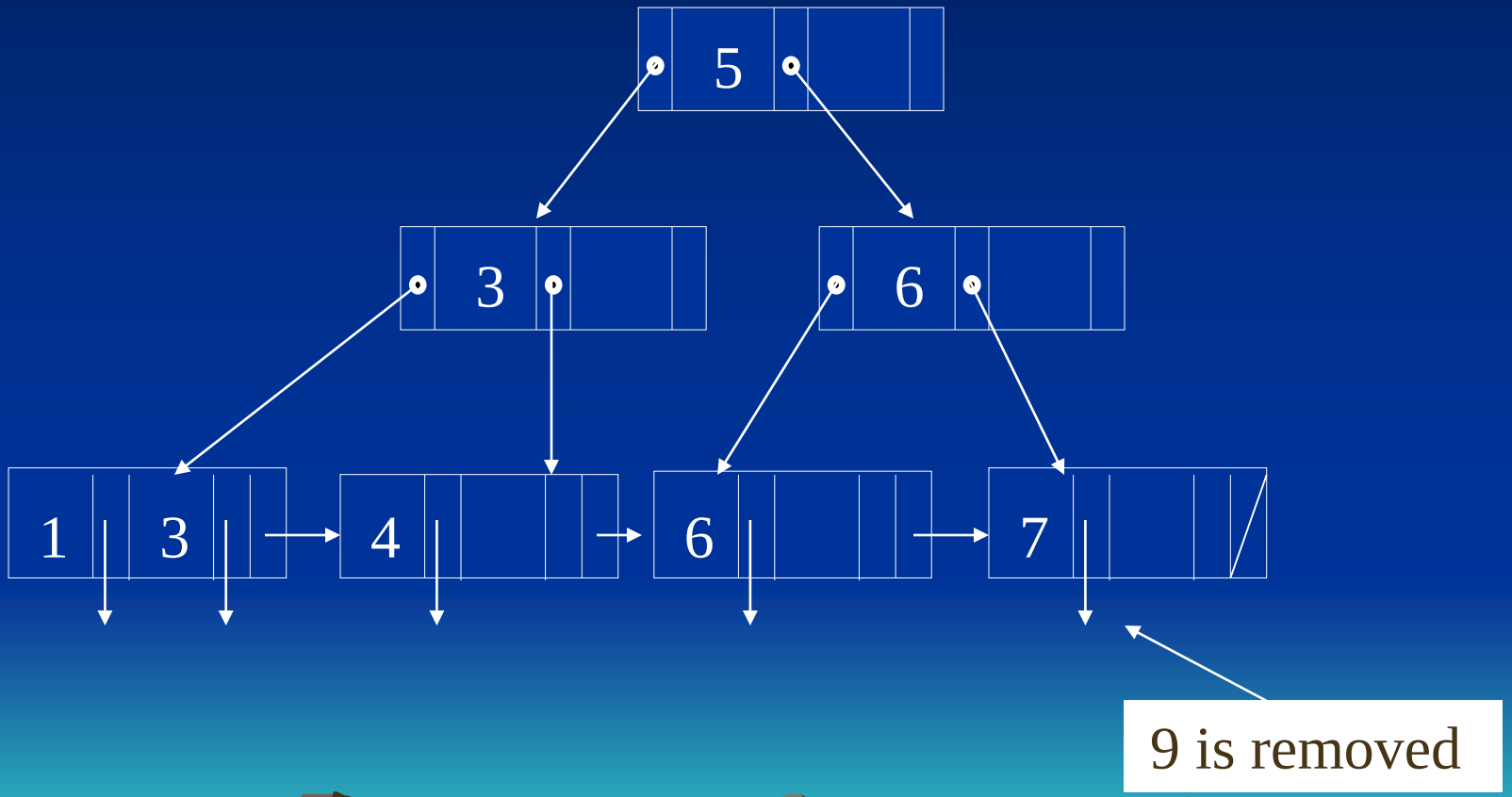
Entry deletion

- deletion sequence: 8, 12, 9, 7



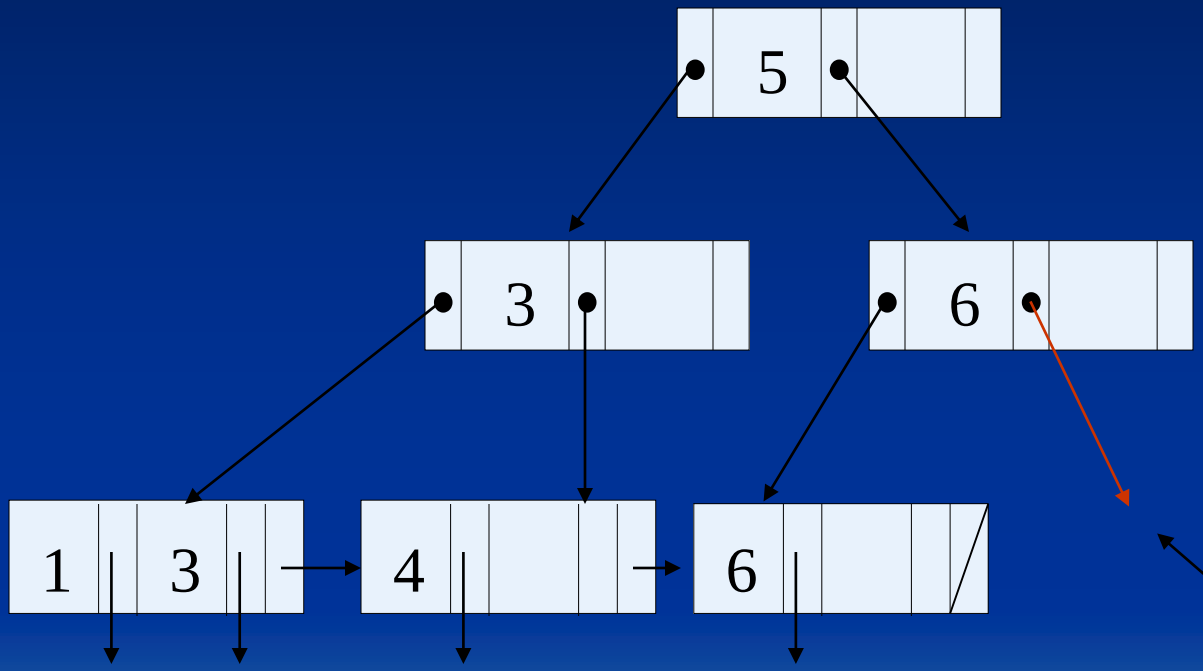
Entry deletion

- deletion sequence: 8, 12, 9, 7



Entry deletion

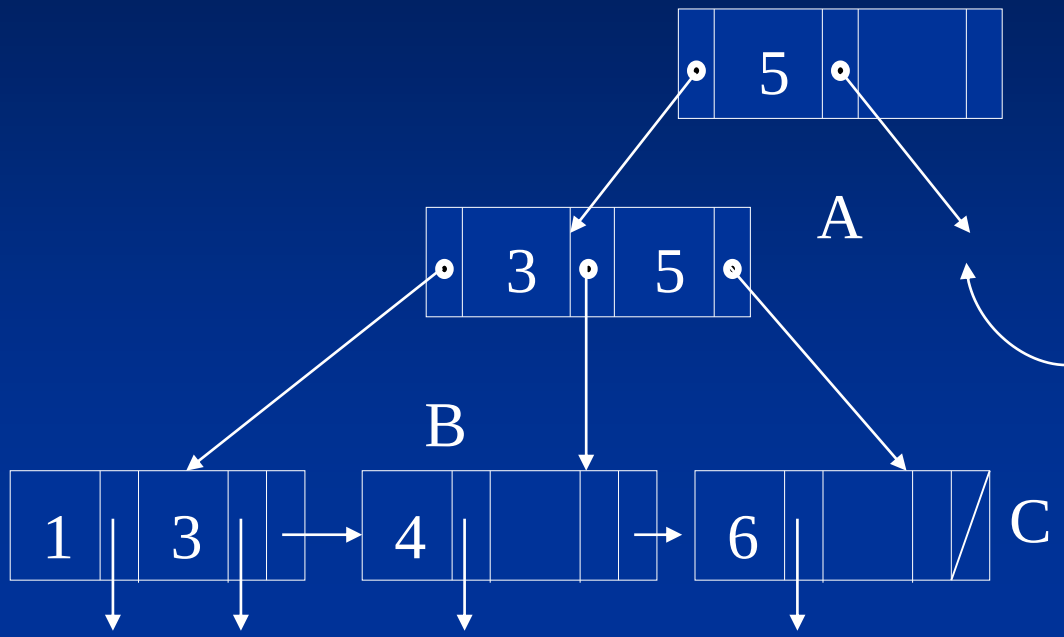
- deletion sequence: 8, 12, 9, 7



Deleting 7 makes this pointer no use. Therefore, a merge at the level above the leaf level occurs.

Entry deletion

- deletion sequence: 8, 12, 9, 7

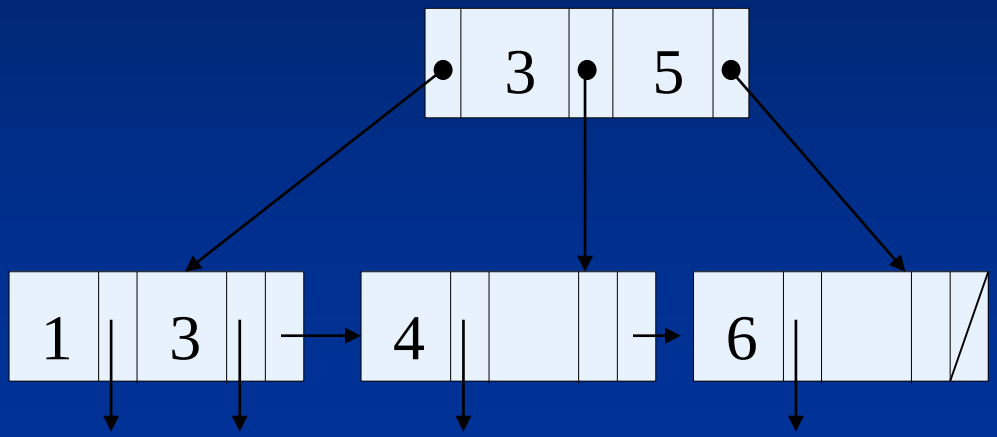


This point becomes useless. The corresponding node should also be removed.

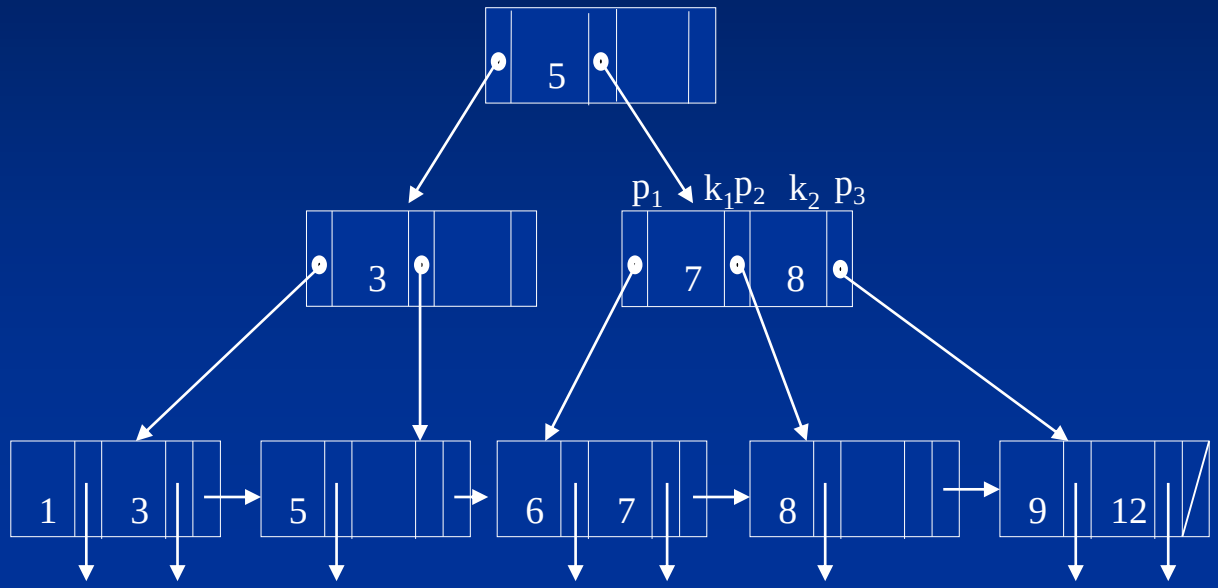
For this merge, 5 will be taken as a key value in A since any key value in B is less than or equal to 5 but any key value in C is larger than 5.

Entry deletion

- deletion sequence: 8, 12, 9, 7



Storing a B+-tree in a file on hard disk:



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

Index Structures for Multidimensional Data

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

Multiple-key indexes

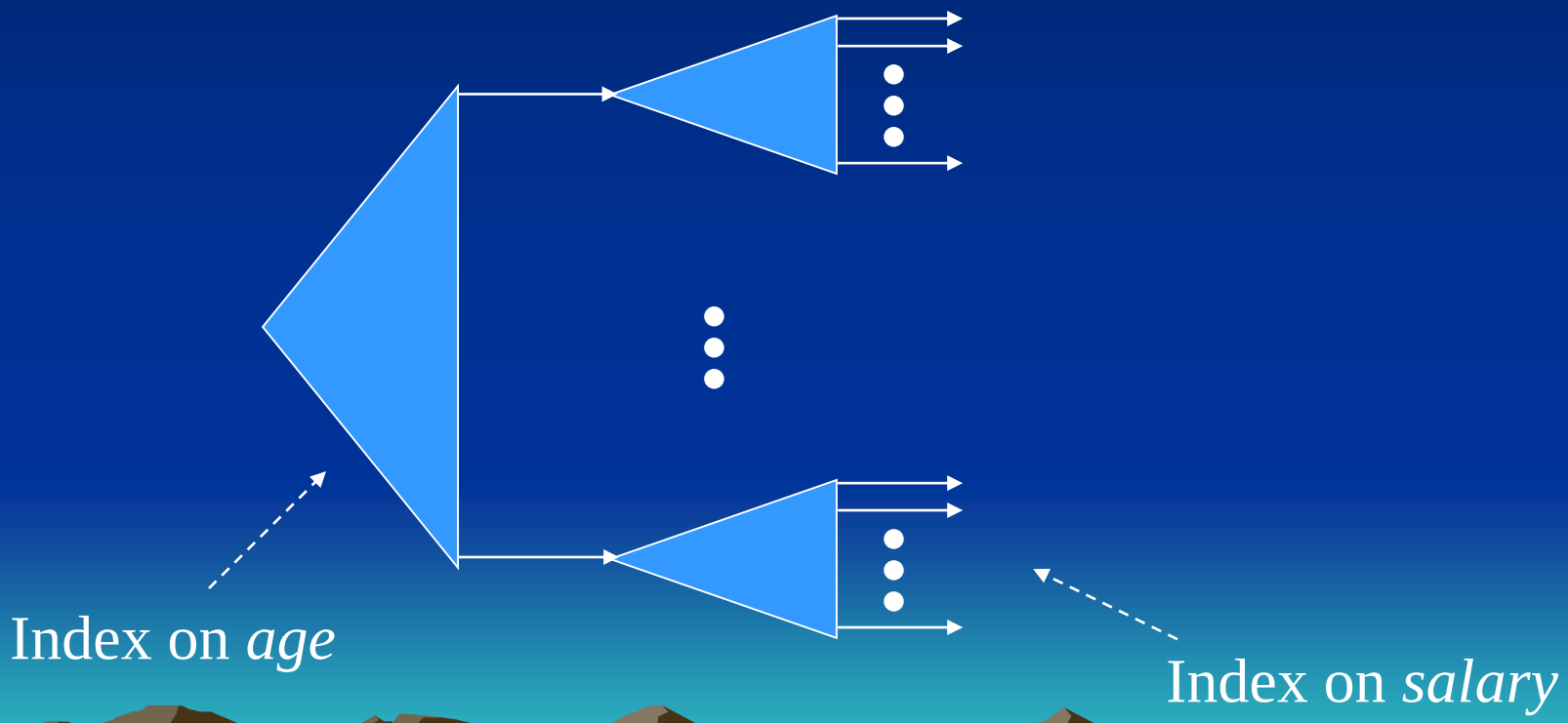
(Indexes over more than one attributes)

Employee

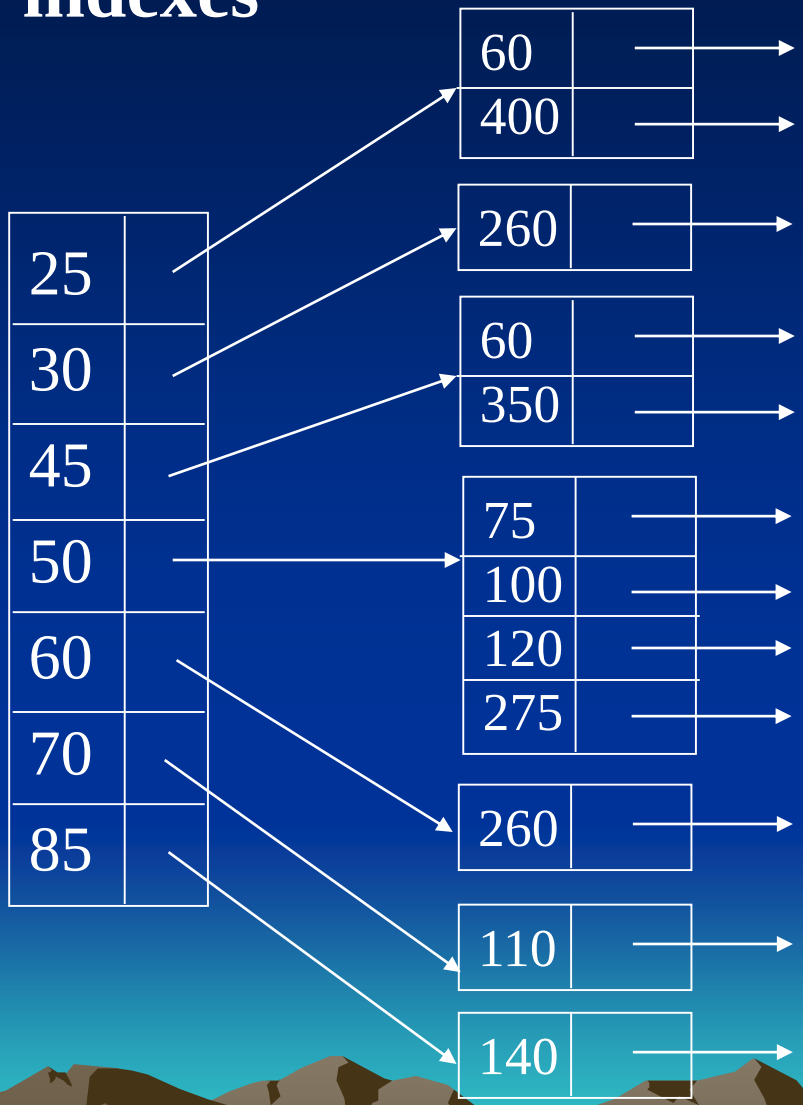
ename	ssn	age	salary	dnumber
Aaron, Ed				
Abbott, Diane				
Adams, John				
Adams, Robin				

Multiple-key indexes

(Indexes over more than one attributes)



Multiple-key indexes

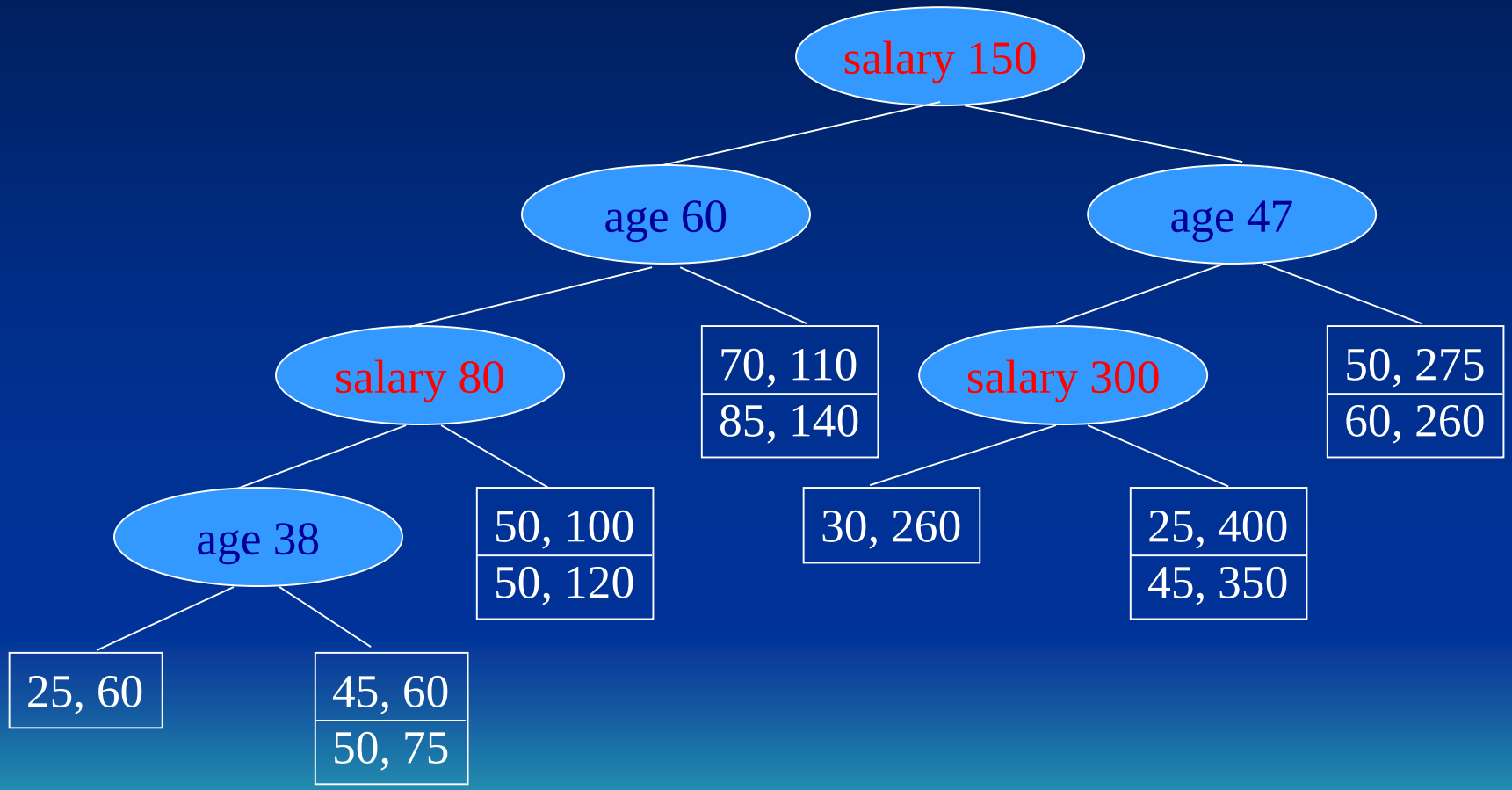


kd-Trees

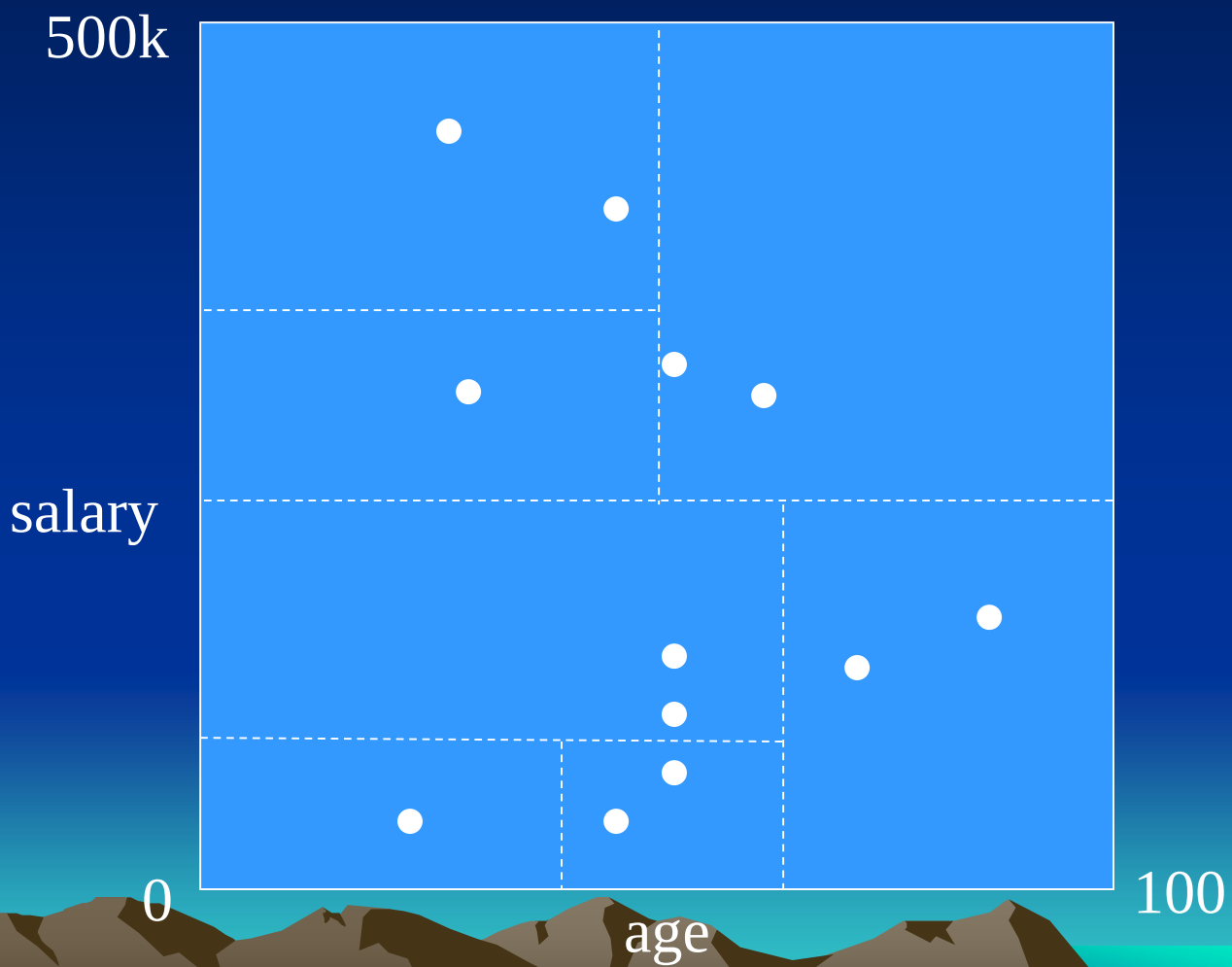
(A generalization of binary trees)

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 or larger than *v*.

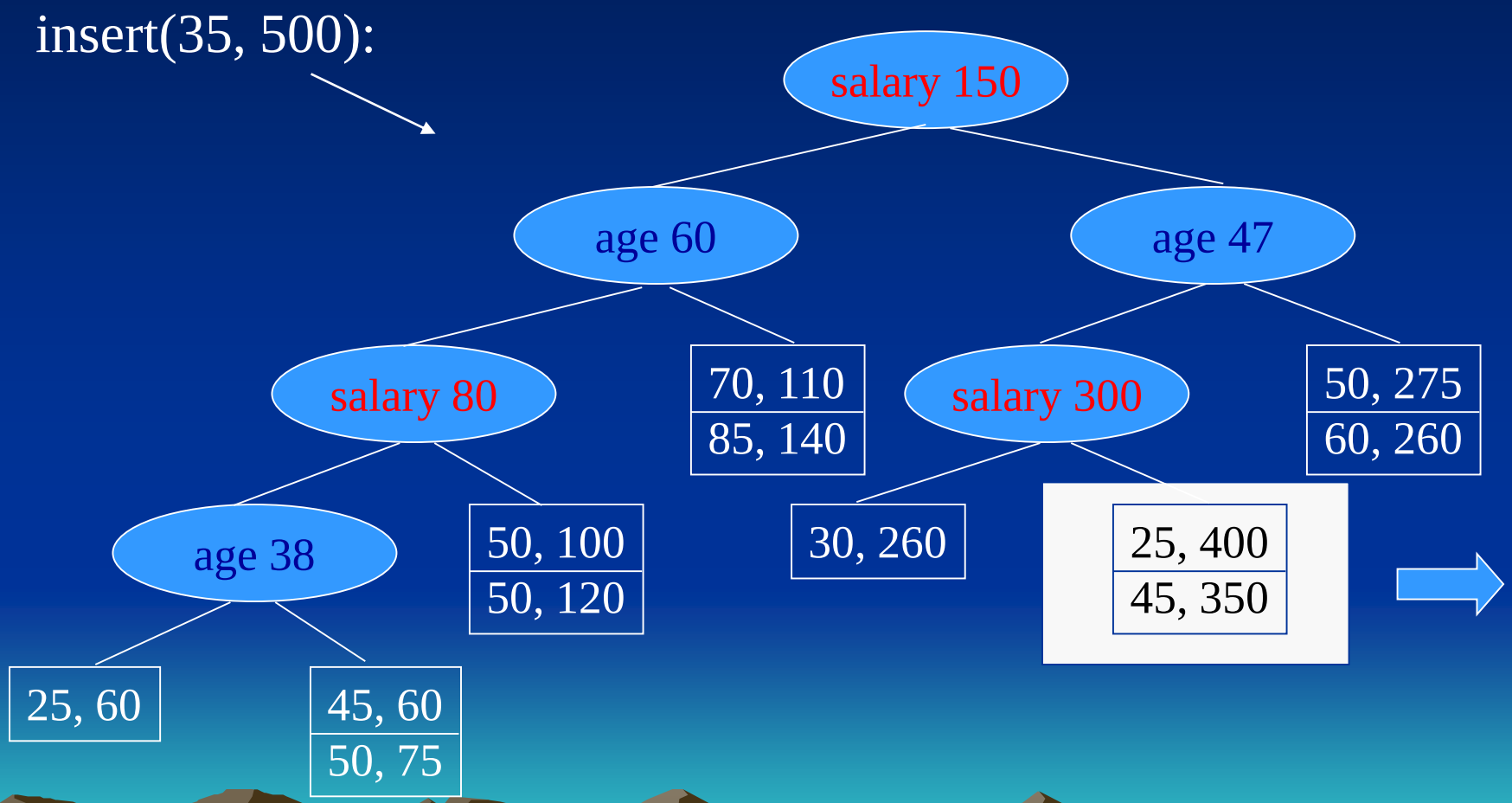
kd-Trees


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

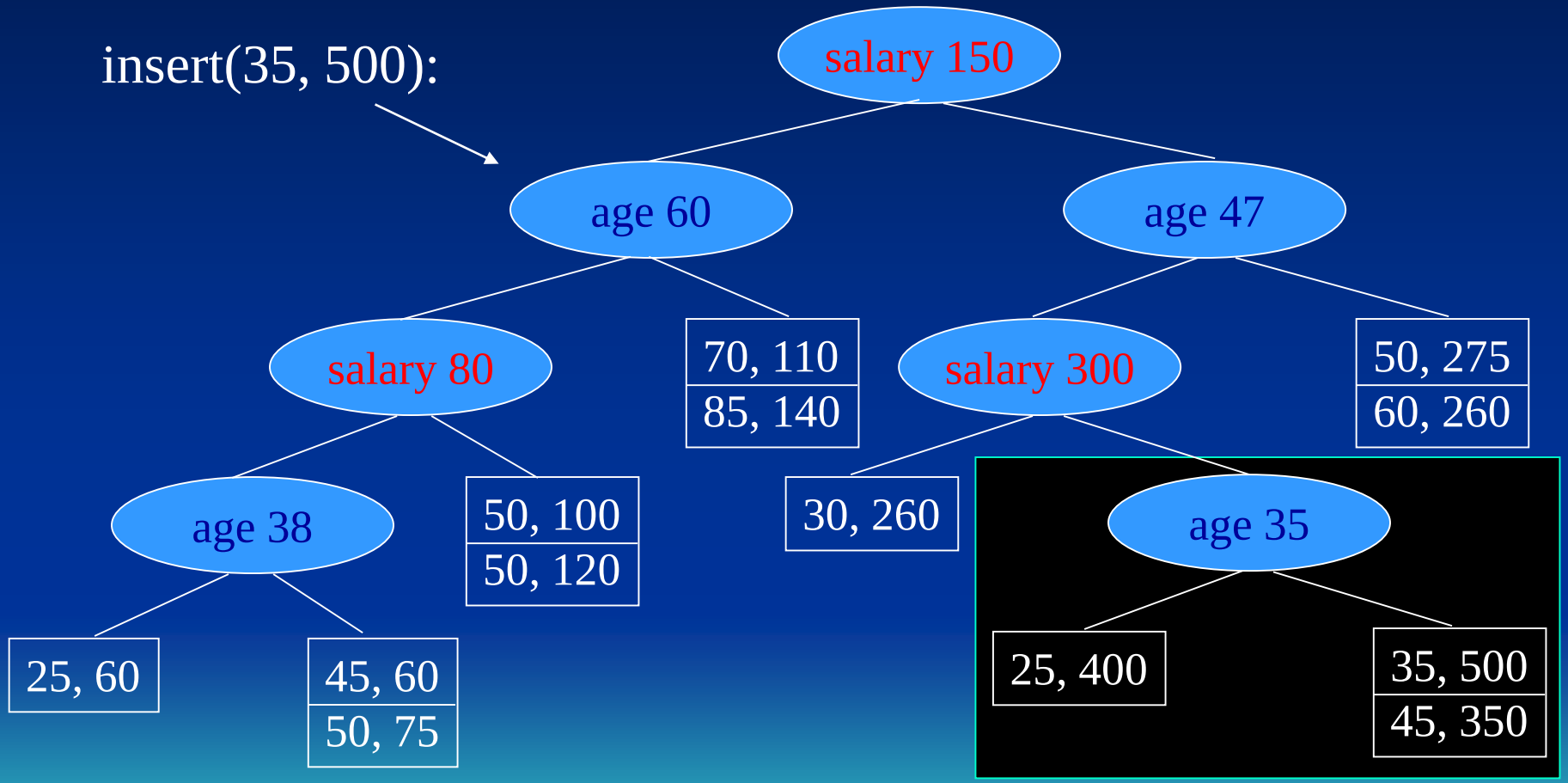
kd-trees



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



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



Quad-trees

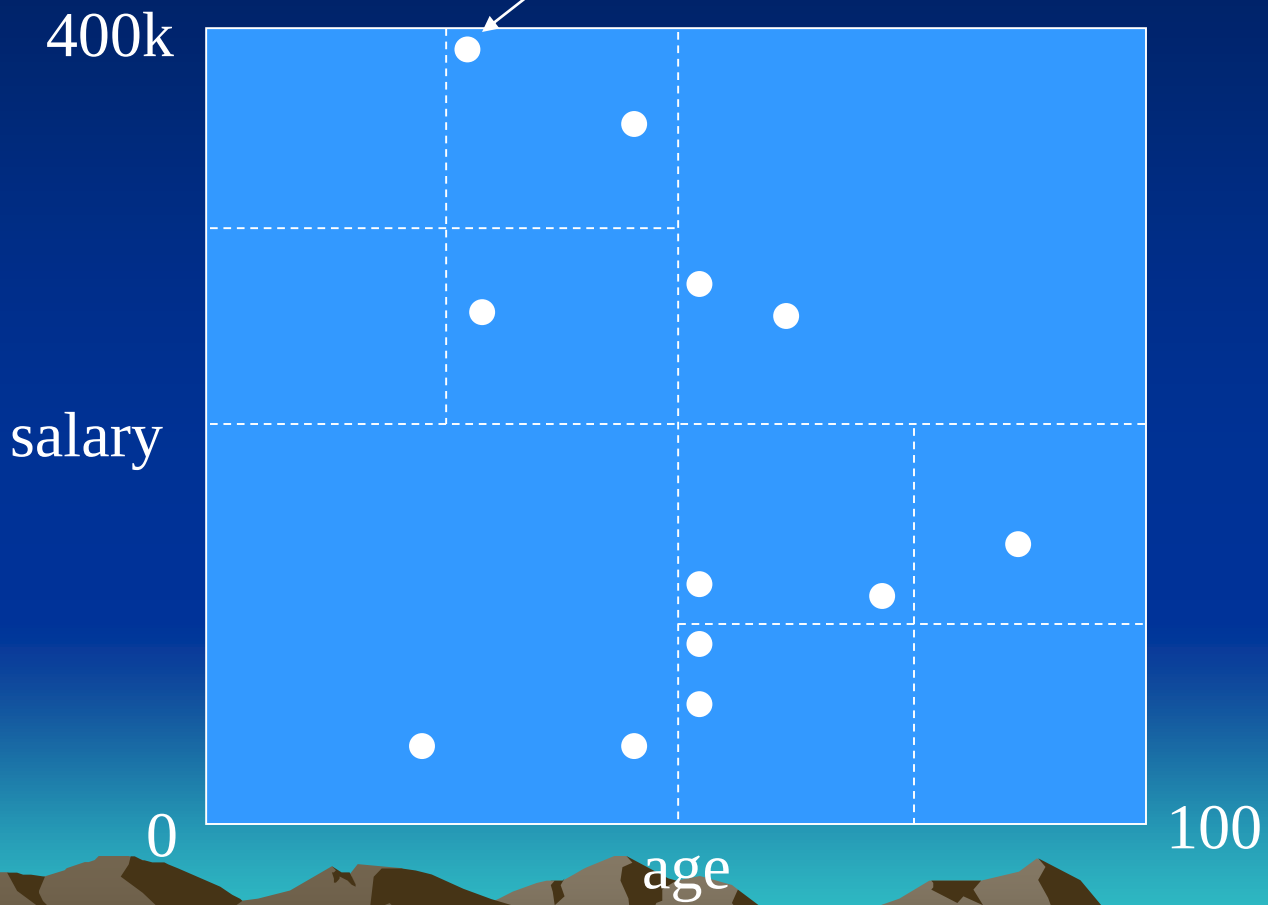
In a Quad-tree, each node corresponds to a square region in two dimensions, or to a *k*-dimensional cube in *k* dimensions.

- 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.

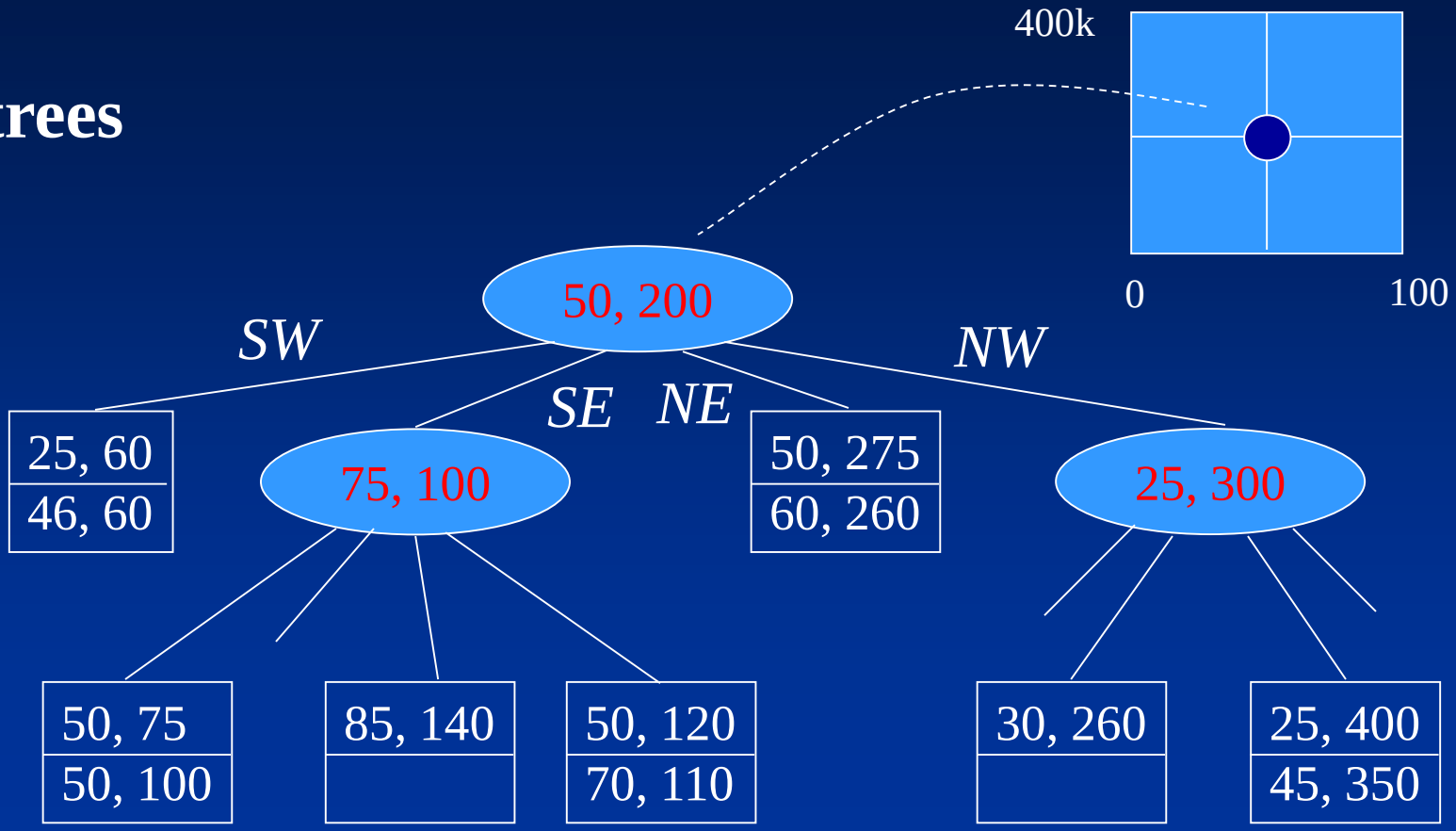
- If there are too many data entries to fit in one block, then we treat the square as an interior node, whose children correspond to its four quadrants.

Quad-trees

name	age	…	salary	…
…	25	…	400	…



Quad-trees



SW – south-west *NW – north-west*

SE – south-east *NE – north-east*

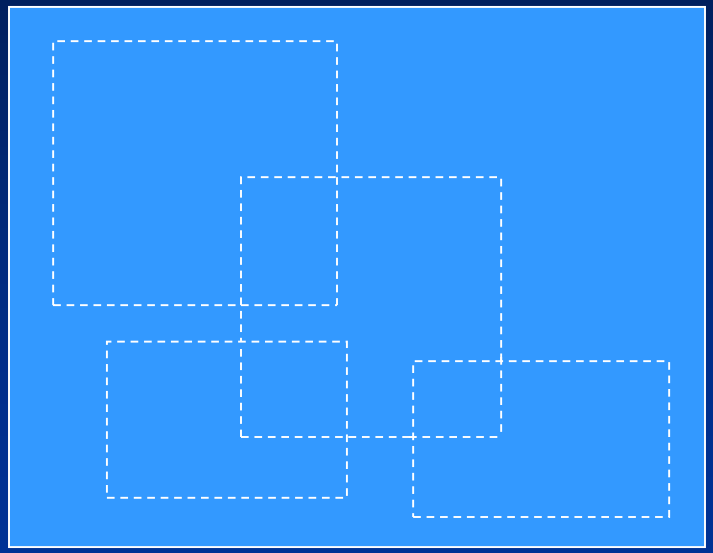
R-trees

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

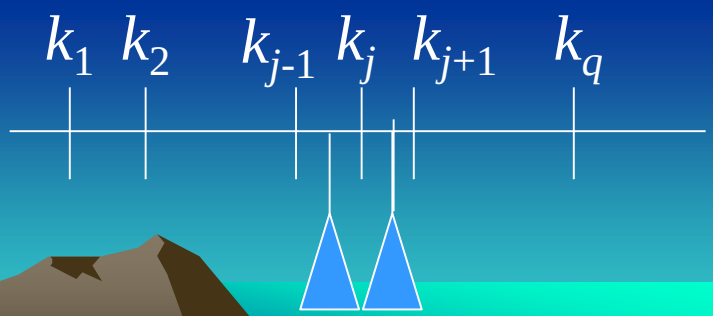
- An R-tree corresponds to a whole area (a rectangle for two-dimensional data.)
- In an R-tree, any interior node corresponds to some interior regions, or just regions, which are usually a rectangle
- Each region *x* in an interior node *n* is associated with a link to a child of *n*, which corresponds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

R-trees

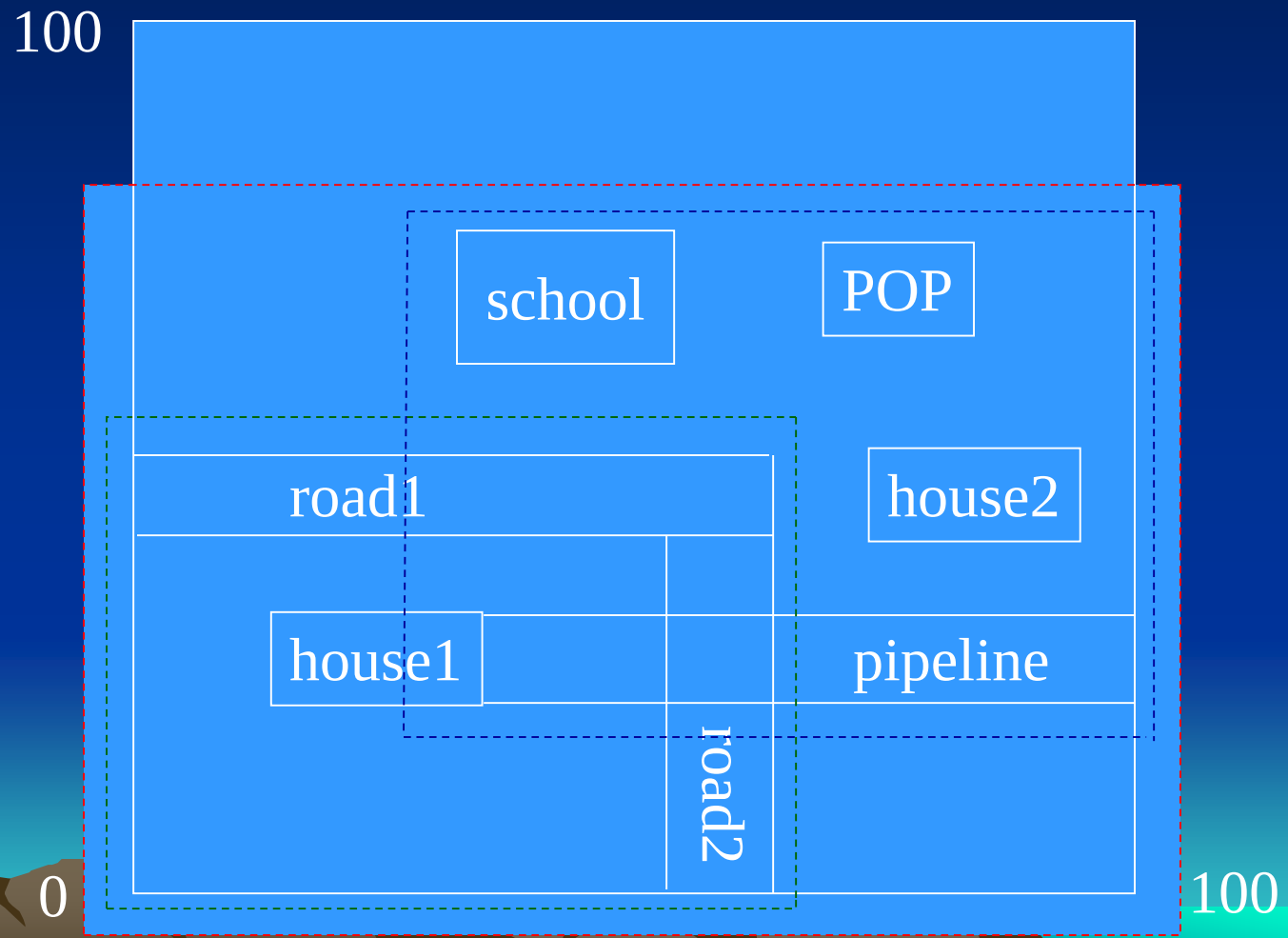
In an R-tree, each interior node contains several subregions.



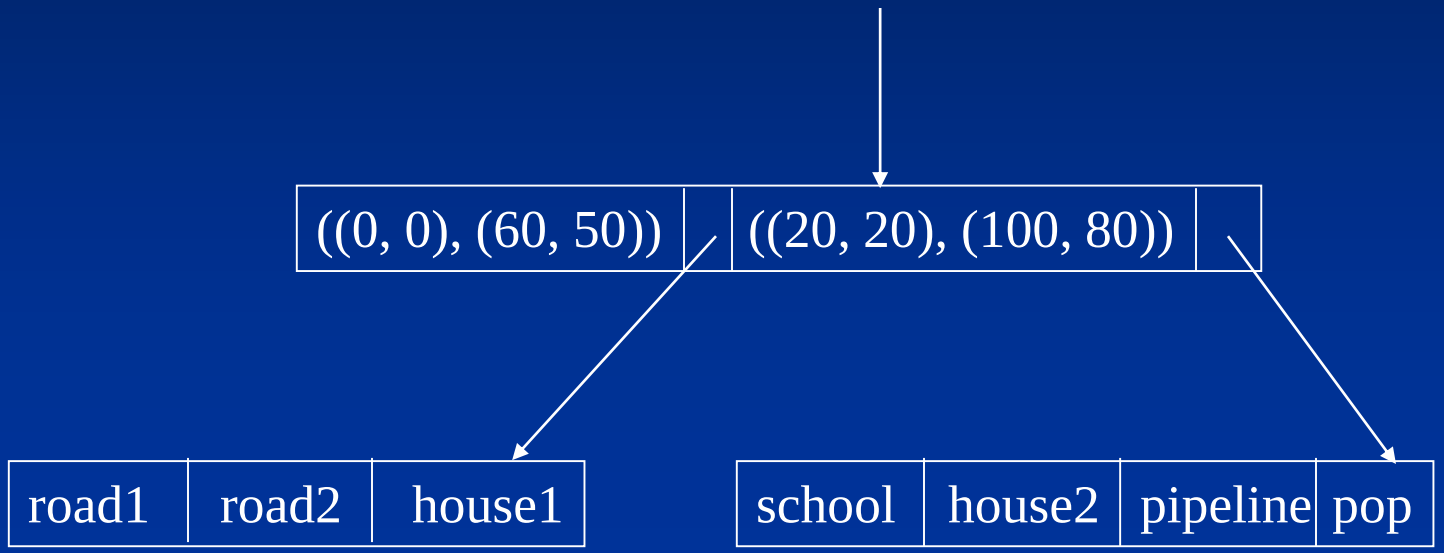
In a B+-tree, each interior node contains a set of keys that divides a line into segments.



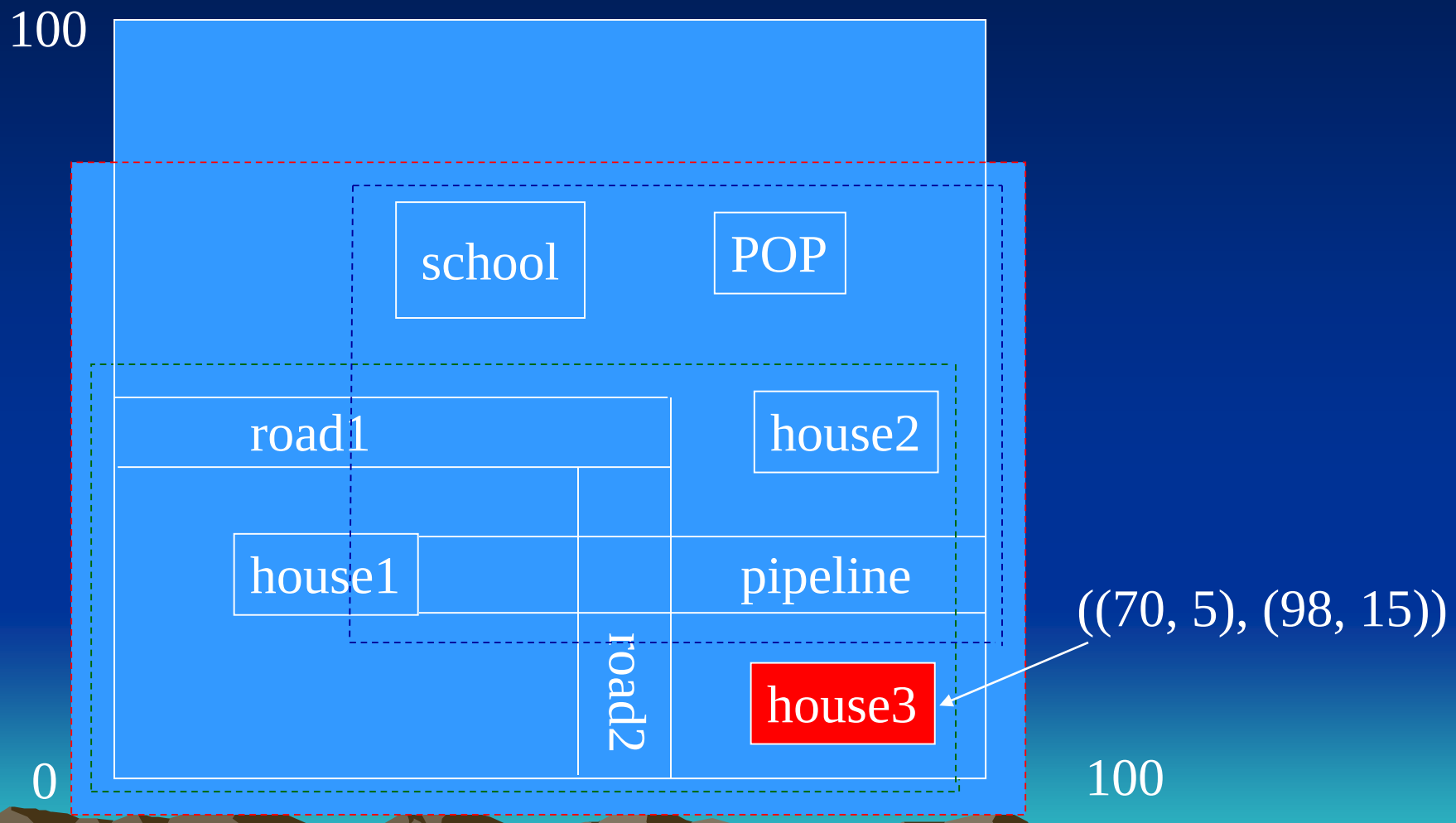
Suppose that the local cellular phone company adds a POP (point of presence, or base station) at the position shown below.



R-trees


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

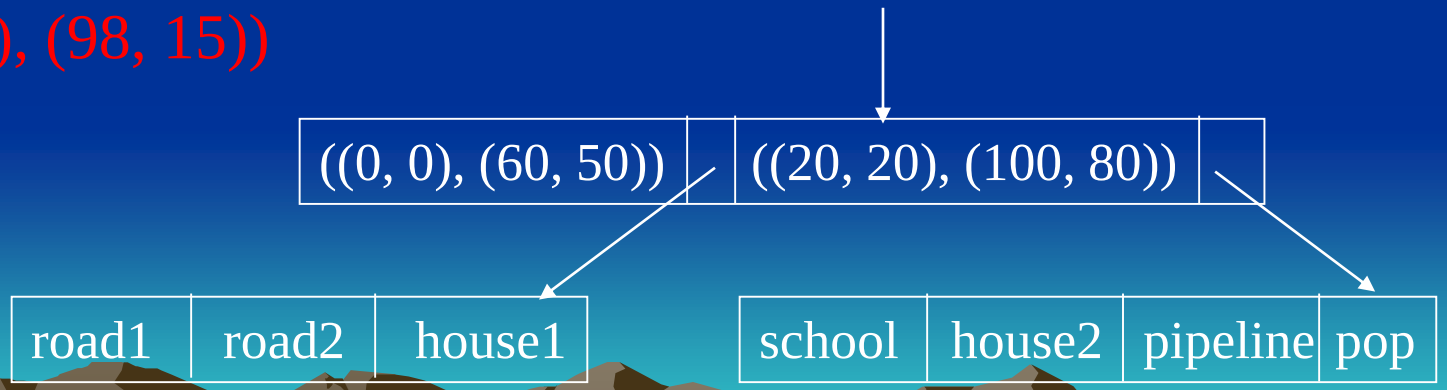
Insert a new region *r* into an R-tree.



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

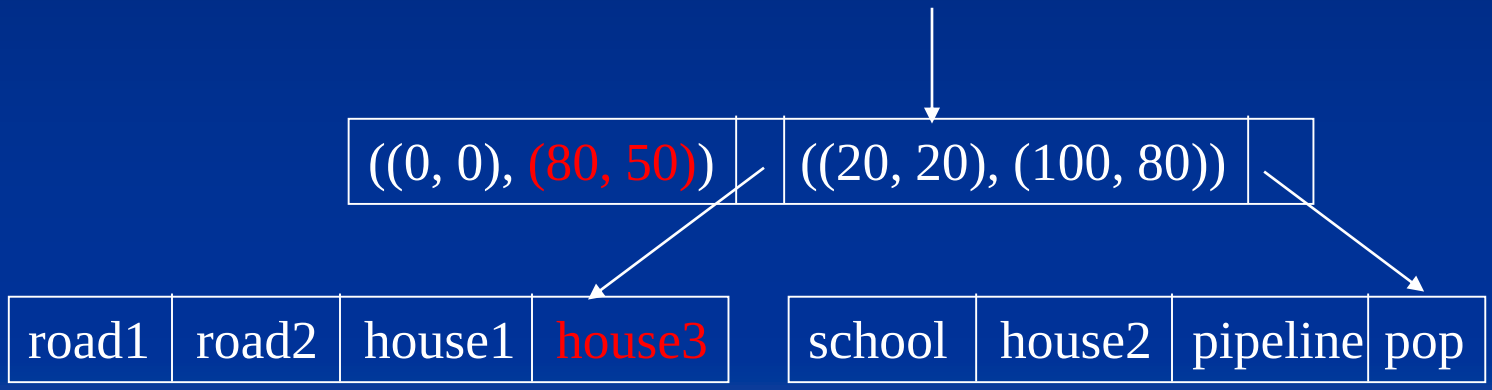
1. Search the *R*-tree, starting at the root.
2. If the encountered node is internal, find a subregion into which *r* fits.
   - If there is more than one such region, pick one and go to its corresponding child.
   - If there is no subregion that contains *r*, choose any subregion such that it needs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

((70, 5), (98, 15))



Two choices:

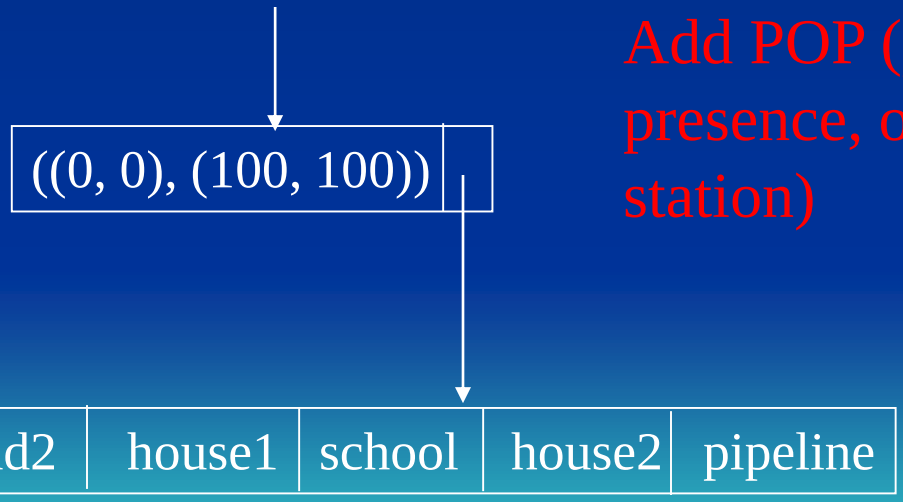
- If we expand the lower subregion, corresponding to the first leaf, then we add 1000 square units to the region.
- If we extend the other subregion by lowering its bottom by 5 units, then we add 1200 square units.



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

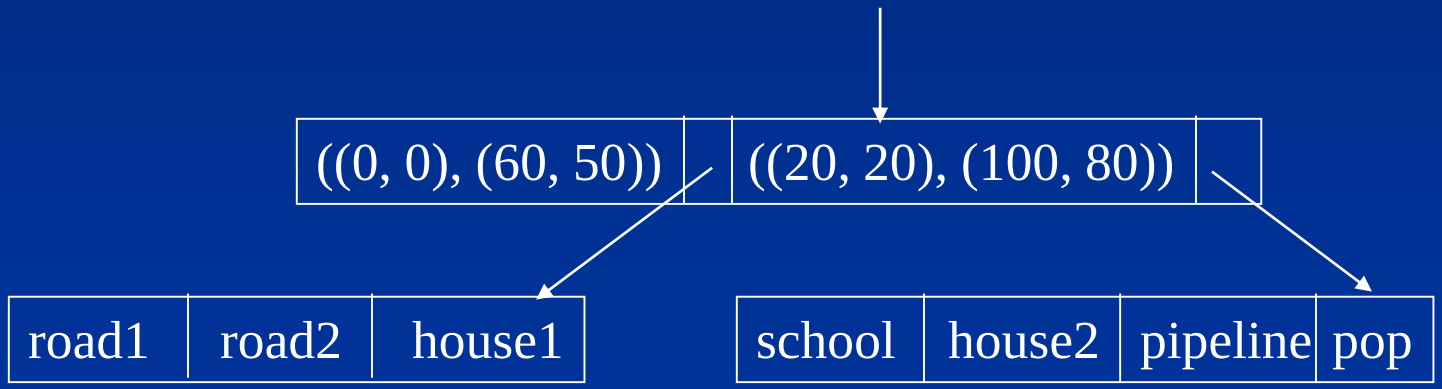
3. If the encountered node *v* is a leaf, insert *r* into it. If there is no room for *r*, split the leaf into two and distribute all subregions in them as evenly as possible. Calculate the 'parent' regions for the new leaf nodes and insert them into *v*'s parent. If there is the room at *v*'s parent,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

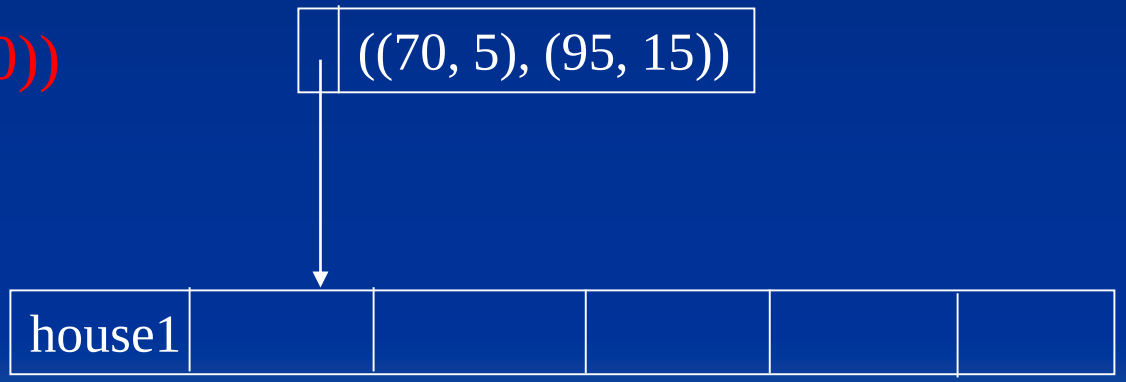


Insert the first object into an R-tree:

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

$R = \varnothing$

house2
((70, 20), (95, 50))



Bit map

1. Image that the records of a file are numbered 1, …, *n*.
2. A bitmap for a data field *F* is a collection of bit-vector 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

Query evaluation

```
Select ename
From Employee
Where age = 55 and salary = 80
```

In order to evaluate this query, we intersect the vectors for *age* = 55 and *salary* = 80.

	0000110	---------- vector for *age* = 55
∧	0000010	---------- vector for *salary* = 80
	0000010	

This indicates the 6th tuple is the answer.

Range query evaluation

```
Select ename
From Employee
Where 30 < age < 55 and 60 < 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.

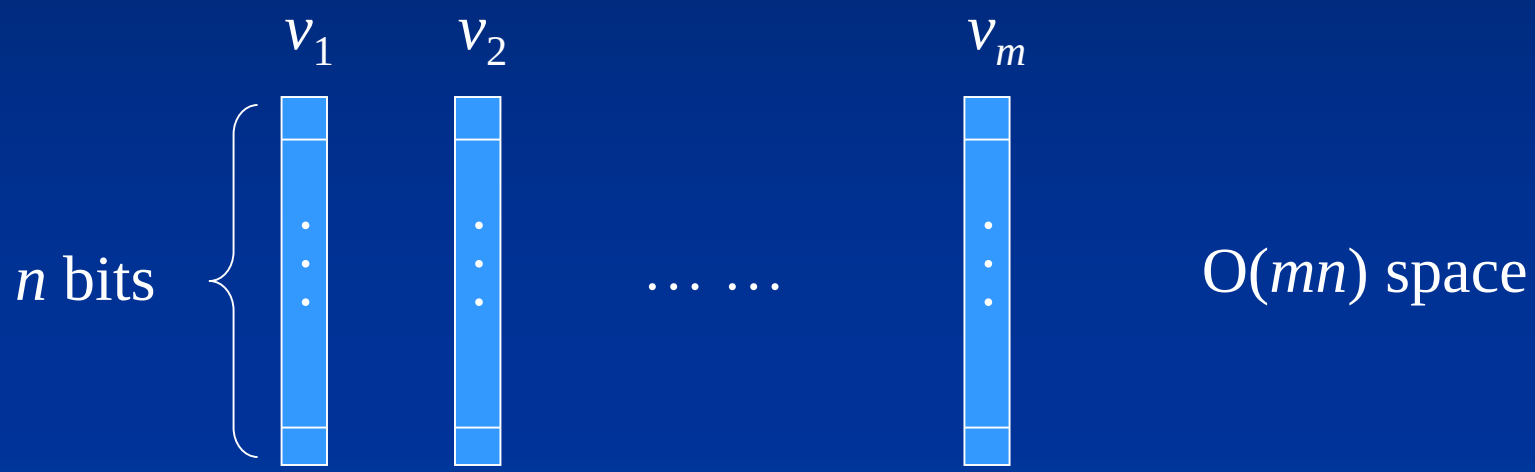
Next find the bit-vectors for the salary values in (60, 78) and take their bitwise OR: 1100000 $\vee$ 0011000 = 1111000.

$$\begin{array}{rr} & 0011000 \\ \wedge & 1111000 \\ \hline & 0011000 \end{array}$$

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:

$r_1'r_2' = 1101111011$

$r_1 = 00000001$

$r_2' = 1011$

$r_2 = 0001$

$r_1r_2 = 000000010001$

Question:

We can put all the compressed bit vectors together to get a bit sequence:

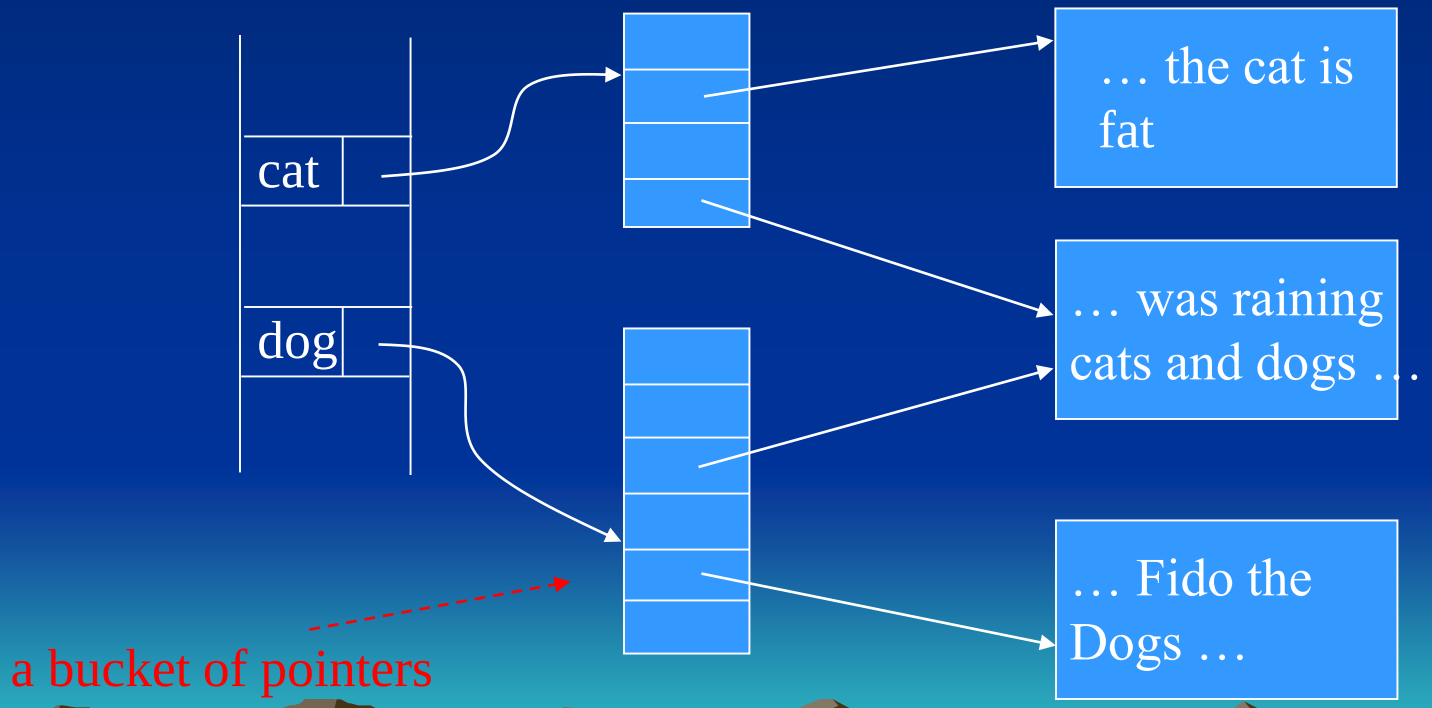
$$\boldsymbol{s} = s_1 s_2 \ldots s_m,$$

where s_i is the compressed bit string for the ith bit vector.

When decoding $\boldsymbol{s}$, how to differentiate between consecutive bit vectors?

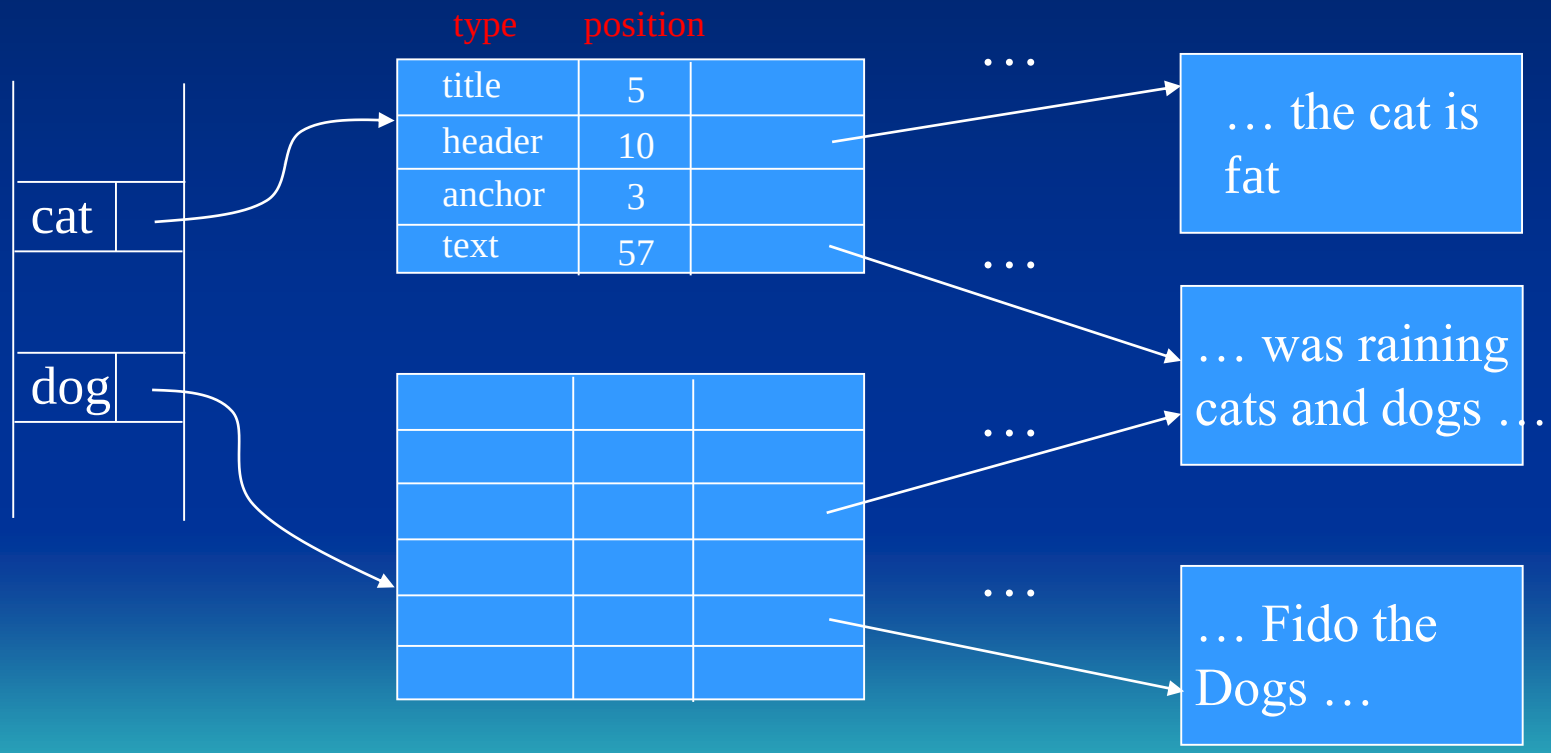
Inverted files

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



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.

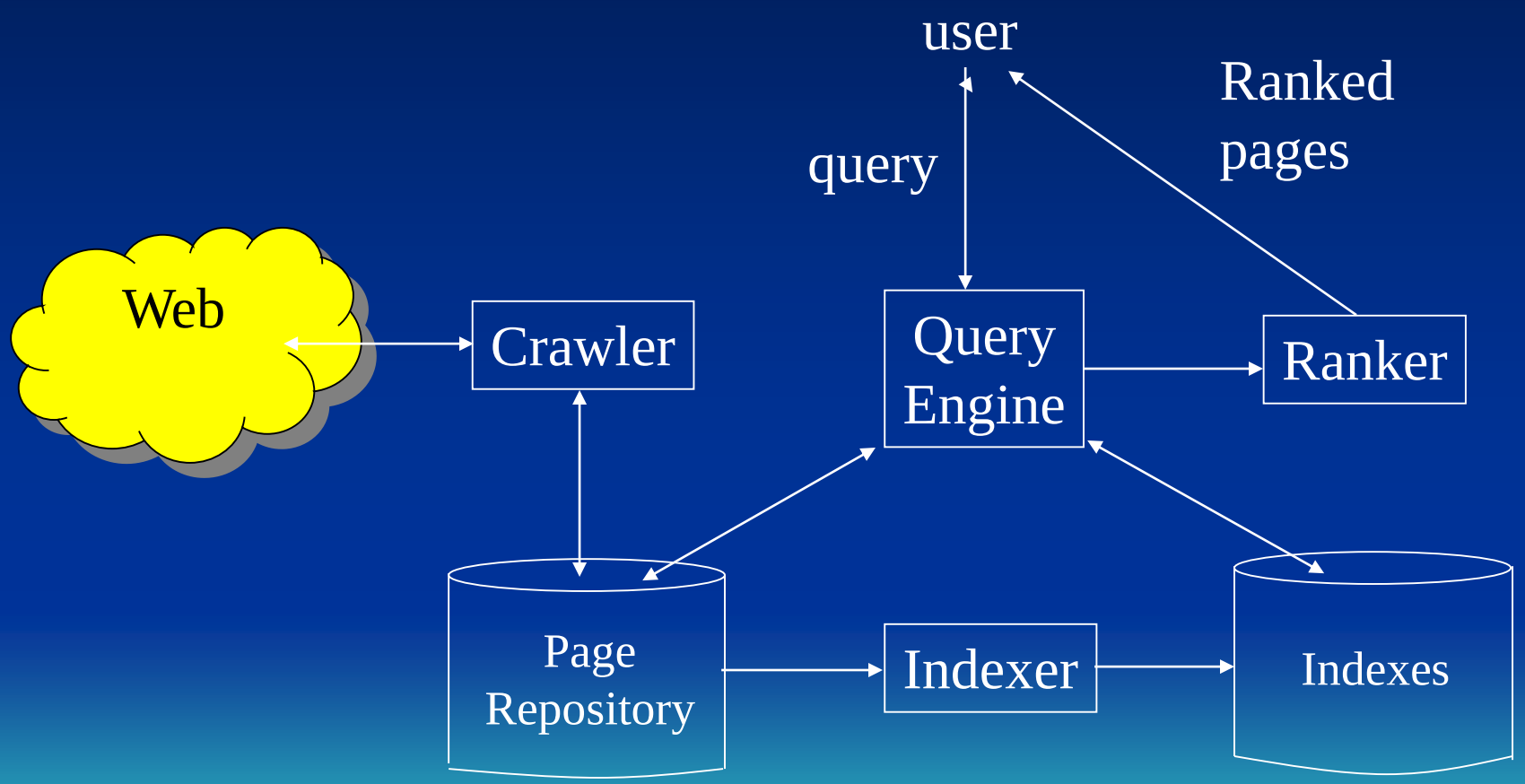


Not included in the final

Search Engine in Web Browser

- Architecture of a search engine
- PageRank for indentifying important pages
- Topic-specific PageRank
- Data streams

The Architecture of a Search Engine



The Architecture of a Search Engine

There are two main functions that a search engine must perform.

1. The Web must be crawled. That is, copies of many of the pages on the Web must be brought to the search engine and processed.
2. Queries must be answered, based on the material gathered from the Web. Usually, a query is in the form of a word or words that the desired Web pages should contain, and the answer to a query is a ranked list of the pages that contain all those words, or at least some of them.

The Architecture of a Search Engine

Crawler – interact with the Web and find pages, which will be stored in Page Repository.

Indexer – inverted file: for each word, there is a list of the pages that contain the word. Additional information in the index for the word may include its locations within the page or its role, e.g., whether the word is in the header.

Query engine – takes one or more words and interacts with indexes, to determine which pages satisfy the query.

Ranker – order the pages according to some criteria.

Web Crawler

A crawler can be a single machine that is started with a set *S*, containing the URL's of one or more Web pages to crawl. There is a repository *R* of pages, with the URL's that have already been crawled; initially *R* is empty.

Algorithm: A simple Web Crawler
Input: an initial set of URL's *S*.
Output: a repository *R* of Web pages

Web Crawler

Method: Repeatedly, the crawler does the following steps.

1. If *S* is empty, end.
2. Select a URL *r* from the set *S* to "crawl" and delete *r* from *S*.
3. Obtain a page *p*, using its URL *r*. If *p* is already in repository *R*, return to step (1) to select another URL.
4. If *p* is not already in *R*:
   (a) Add *p* to *R*.
   (b) Examine *p* for links to other pages. Insert into *S* the URL of each page *q* that *p* links to, but that is not already in *R* or *S*.
5. Go to step (1).

Web Crawler

The algorithm raises several questions.

a) How to terminate the search if we do not want to search the entire Web?
b) How to check efficiently whether a page is already in repository *R?*
c) How to select an URL *r* from *S* to search next?
d) How to speed up the search, e.g., by exploiting parallelism?

Terminating Search

The search could go on forever due to dynamically constructed pages.

Set limitation:

- Set a limit on the number of pages to crawl.

  The limit could be either on each site or on the total number of pages.
- Set a limit on the depth of the crawl.

  Initially, the pages in set *S* have depth 1. If the page *p* selected for crawling at step (2) of the algorithm has depth *i*, then any page *q* we add to *S* at step 4-(b) is given depth *i* + 1. However, if *p* has depth equal to the limit, then do not examine links out of *p* at all. Rather we simply add *p* to *R* if it is not already there.

Managing the Repository

- When we add a new URL for a page *p* to the set *S*, we should check that it is not already there or among the URL's of pages in *R*.
- When we decide to add a new page *p* to *R* at step 4-(a) of the algorithm, we should be sure the page is not already there.

Page signatures:

- Hash each Web page to a signature of, say, 64 bits.
- The signatures themselves are stored in a hash table *T*, i.e., they are further hashed into a smaller number of buckets, say one million buckets.

Page signatures:

- Hash each Web page to a signature of, say, 64 bits.
- The signatures themselves are stored in a hash table *T*, i.e., they are further hashed into a smaller number of buckets, say one million buckets.
- When inserting *p* into *R*, compute the 64-bit signature *h*(*p*), and see whether *h*(*p*) is already in the hash table *T*. If so, do not store *p*; otherwise, store *p* in *T*.

Selecting the Next page

- Completely random choice of next page.
- Maintain *S* as a queue. Thus, do a breadth-first search of the Web from the starting point or points with which we initialized *S*. Since we presumably start the search from places in the Web that have "important" pages, we are assured of visiting preferentially those portions of the Web.
- Estimate the importance of pages in *S*, and to favor those pages we estimate to be the most important.
  - PageRank: number of in-links in a page

Speeding up the Crawl

- More than one crawling machine
- More crawling processes in a machine
- Concurrent access to *S*

Query Processing in Search Engine

- Search engine queries are word-oriented: a boolean combination of words
- Answer: all pages that contain such words
- Method:

  - The first step is to use the inverted index to determine those pages that contain the words in the query.
  - The second step is to evaluate the boolean expression:

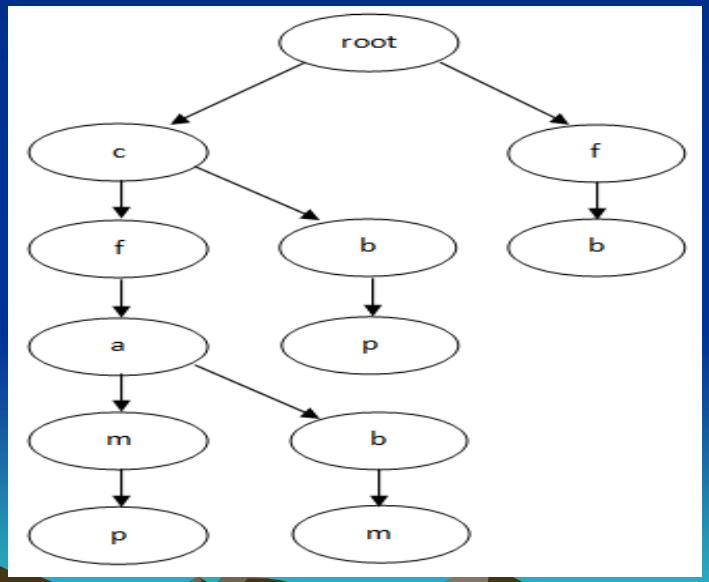
  The AND of bit vectors gives the pages containing both words.
  The OR of bit vectors gives the pages containing one or both.

$$(word1 \wedge word2) \vee (word3 \wedge word4)$$

Trie-based Method for Query Processing

- A trie is a multiway tree, in which each path corresponds to a string, and common prefixes in strings to common prefix paths.
- Leaf nodes include either the documents themselves, or links to the documents that contain the string that corresponds to the path.

Example:



A trie constructed for The following strings:

s1: cfamp
s2: cbp
s3: cfabm
s4: fb

Trie-based Method for Query Processing

- Item sequences sorted by appearance frequency (*af*) in documents.

DocID	Items	Sorted item sequence
1	*f, a, c, m, p*	*c, f, a, m, p*
2	*a, b, c, f*	*c, f, a, b, m*
3	*b, f*	*f, b*
4	*b, c, p*	*c, b, p*
5	*a, f, c, m, p*	*c, f, a, m, p*

$$af(w) = \frac{\text{No. of doc. Containing } w}{\text{No. of doc.}}$$

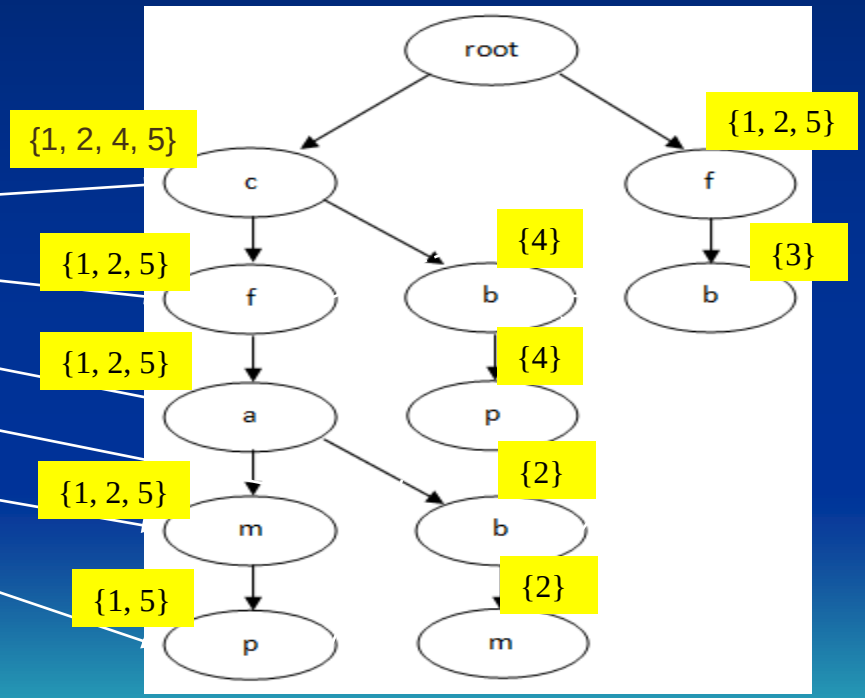
- View each sorted item sequence as a string and construct a trie over them, in which each node is associated with a set of document IDs each containing the substring represented by the corresponding prefix.

Trie-based Method for Query Processing

- View each sorted item sequence as a string and construct a trie over them.

Header table:

items	links
c	
f	
a	
b	
m	
p	



Trie-based Method for Query Processing

- Evaluation of queries

  - Let $Q = \text{word}_1 \wedge \text{word}_2 \ldots \wedge \text{word}_k$ be a query
  - Sort the words in Q according to the appearance frequency:

    $$\text{word}_{i_1} \wedge \ldots \wedge \text{word}_{i_k}$$

  - Find a node in the trie, which is labeled with word_{i_1}
  - If the path from the root to word_{i_1} contains all word_i ($i = 1, \ldots, k$), Return the document identifiers associated with word_{i_1}
  - The check can be done by searching the path bottom-up, starting from word_{i_1}. In this process, we will first try to find word_{i_2}, and then word_{i_3}, and so on.

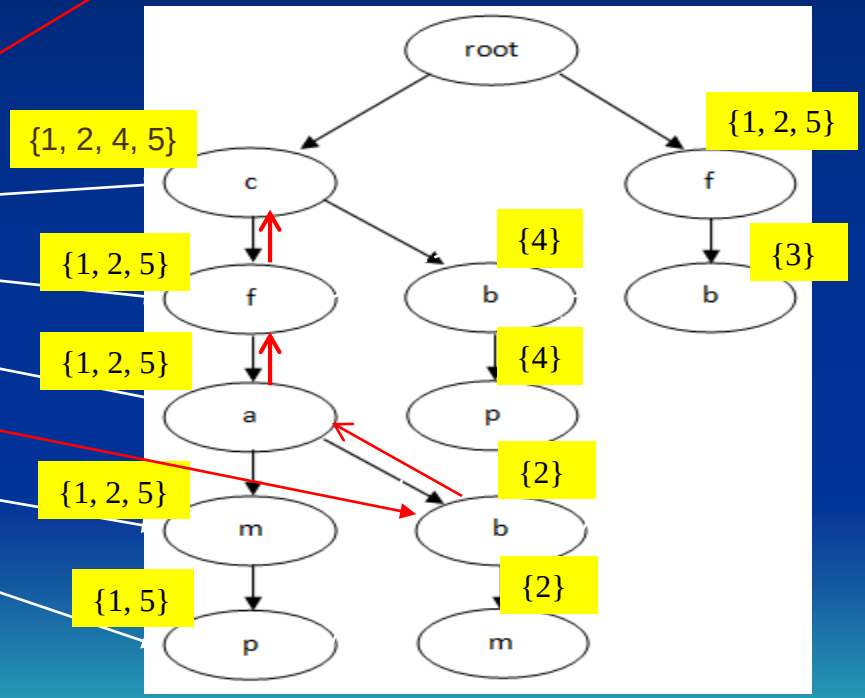
Trie-based Method for Query Processing

- Example

query: c ∧ b ∧ f —sorting→ b ∧ f ∧ c

Header table:

items	links
c	
f	
a	
b	
m	
p	



Ranking Pages

Once the set of pages that match the query is determined, these pages are ranked, and only the highest-ranked pages are shown to the user.

Measuring PageRank:

- The presence of all the query words
- The presence of query words in important positions in the page
- Presence of several query words near each other would be a more favorable indication than if the words appeared in the page, but widely separated.
- Presence of the query words in or near the anchor text in links leading to the page in question.

PageRank for Identifying Important Pages

One of the key technological advances in search is the PageRank algorithm for identifying the "importance" of Web pages.

The Intuition behind PageRank

When you create a page, you tend to link that page to others that you think are important or valuable

A Web page is important if many important pages link to it.

Recursive Formulation of PageRank

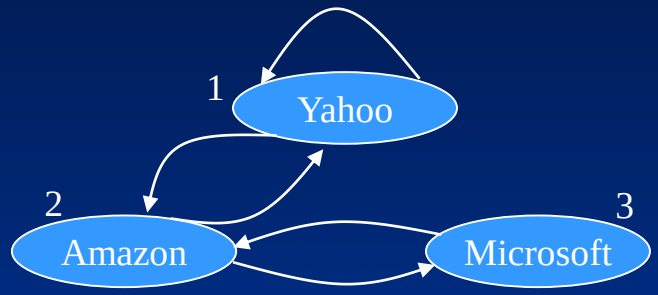
The Web navigation can be modeled as random walker move. So we will maintain a *transition matrix* to represent links.

- Number the pages 1, 2, …, *n*.
- The transition matrix **M** has entries m_{ij} in row *i* and column *j*, where:
  1. $m_{ij} = 1/r$ if page *j* has a link to page *i*, and there are a total $r \geq 1$ pages that *j* links to.
  2. $m_{ij} = 0$ otherwise.

- If every page has at least one link out, then **M** is *stochastic* – elements are nonnegative, and its columns each sum to exactly 1.
- If there are pages with no links out, then the column for that page will be all 0's. **M** is said to be *substochastic* if all columns sum to at most 1.

Solutions to the equation:

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix}$$



- If (y_0, a_0, m_0) is a solution to the equation, then (cy_0, ca_0, cm_0) is also a solution for any constant c.

- $y_0 + a_0 + m_0 = 1$.

Gaussian elimination method – $O(n^3)$. If n is large, the method cannot be used. (Consider billions pages!)

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix}$$

$$y = ½ \cdot y + ½ \cdot a + 0 \cdot m$$

$$a = ½ \cdot y + 0 \cdot a + 1 \cdot m$$

$$m = 0 \cdot y + ½ \cdot a + 0 \cdot m$$

$$y = ½ \cdot y + ½ \cdot a + 0 \cdot m \qquad P(y) = ½ \cdot P(y) + ½ \cdot P(a) + 0 \cdot P(m)$$

$$a = ½ \cdot y + 0 \cdot a + 1 \cdot m \qquad P(a) = ½ \cdot P(y) + 0 \cdot a\, P(a) + 1 \cdot P(y)$$

$$m = 0 \cdot y + ½ \cdot a + 0 \cdot m \qquad P(m) = 0 \cdot P(y) + ½ \cdot P(a) + 0 \cdot P(m)$$

$$P(y) = P(y \mid y) \cdot P(y) + P(y \mid a) \cdot P(a) + P(y \mid m) \cdot P(m)$$

$$P(a) = P(a \mid y) \cdot P(y) + P(a \mid a) \cdot P(a) + P(a \mid m) \cdot P(m)$$

$$P(m) = P(m \mid y) \cdot P(y) + P(m \mid a) \cdot P(a) + P(m \mid m) \cdot P(m)$$

Conditional probability

Approximation by the method of *relaxation*:

- Start with some estimate of the solution and repeatedly multiply the estimate by **M.**
- As long as the columns of **M** each add up to 1, then the sum of the values of the variables will not change, and eventually they converge to the distribution of the walker's location.
- In practice, 50 to 100 iterations of this process suffice to get very close to the exact solution.

Suppose we start with $(y, a, m) = (1/3, 1/3, 1/3)$. We have

$$\begin{pmatrix} 2/6 \\ 3/6 \\ 1/6 \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$$

At the next iteration, we multiply the new estimate (2/6, 3/6, 1/6) by **M**, as:

$$\begin{pmatrix} 5/12 \\ 4/12 \\ 3/12 \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} 2/6 \\ 3/6 \\ 1/6 \end{pmatrix}$$

If we repeat this process, we get the following sequence of vectors:

$$\begin{pmatrix} 9/24 \\ 11/24 \\ 4/24 \end{pmatrix}, \begin{pmatrix} 20/48 \\ 17/48 \\ 11/48 \end{pmatrix}, \ldots, \begin{pmatrix} 2/5 \\ 2/5 \\ 1/5 \end{pmatrix}$$

Spider Traps and Dead Ends

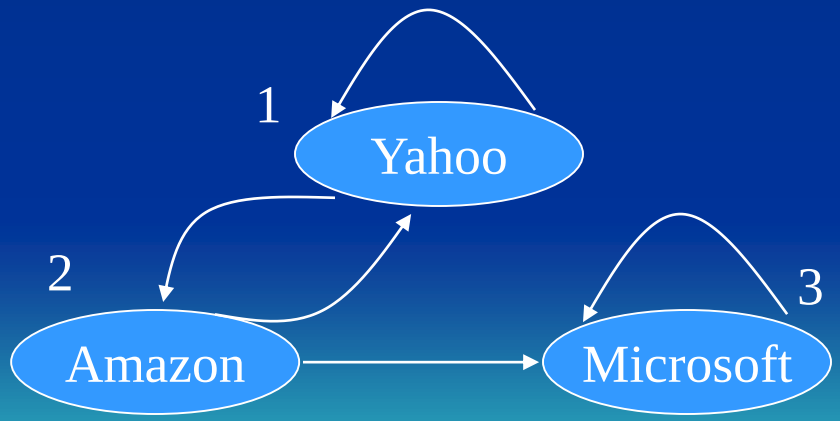
- Spider traps. There are sets of Web pages with the property that if you enter that set of pages, you can never leave because there are no links from any page in the set to any page outside the set.
- Dead ends. Some Web pages have no out-links. If the random walker arrives at such a page, there is no place to go next, and the walk ends.
  - Any dead end is, by itself, a spider trap. Any page that links only to itself is a spider trap.
  - If a spider trap can be reached from outside, then the random walker may wind up there eventually and never leave.

Spider Traps and Dead Ends

Problem:

Applying relaxation to the matrix of the Web with spider traps can result in a limiting distribution where all probabilities outside a spider trap are 0.

Example.



$$\mathbf{M} = \begin{array}{c} \begin{matrix} p1 & p2 & p3 \end{matrix} \\ \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 1 \end{pmatrix} \end{array}$$

Solutions to the equation:

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix}$$

Initially, $\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$

$$\begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} \begin{pmatrix} 2/6 \\ 1/6 \\ 3/6 \end{pmatrix} \begin{pmatrix} 3/12 \\ 2/12 \\ 7/12 \end{pmatrix} \begin{pmatrix} 5/24 \\ 3/24 \\ 16/24 \end{pmatrix} \begin{pmatrix} 8/48 \\ 5/48 \\ 35/48 \end{pmatrix} \text{, …,} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

This shows that with probability 1, the walker will eventually wind up at the Microsoft page (page 3) and stay there.

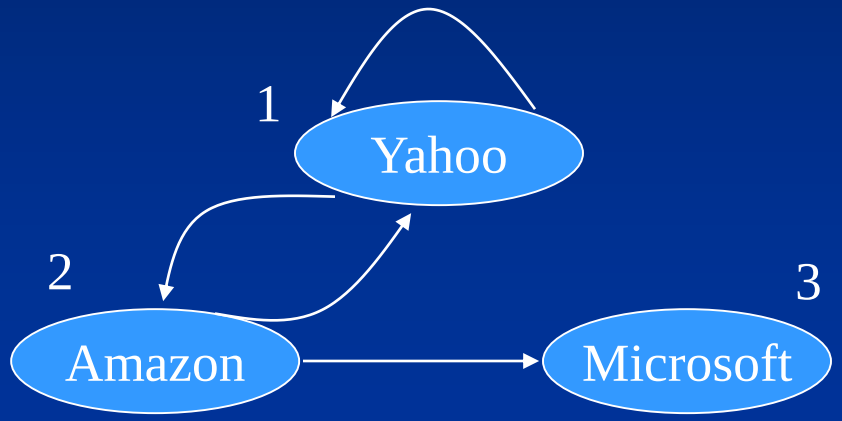
Problem Caused by Spider Traps

- If we interpret these PageRank probabilities as “importance” of pages, then the Microsoft page has gathered all importance to itself simply by choosing not to link outside.
- The situation intuitively violates the principle that other pages, not you yourself, should determine your importance on the Web.

Problem Caused by Dead Ends

- The dead end also cause the PageRank not to reflect importance of pages.

Example.



$$\mathbf{M} = \begin{array}{c} \begin{matrix} p1 & p2 & p3 \end{matrix} \\ \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 0 \end{pmatrix} \end{array}$$

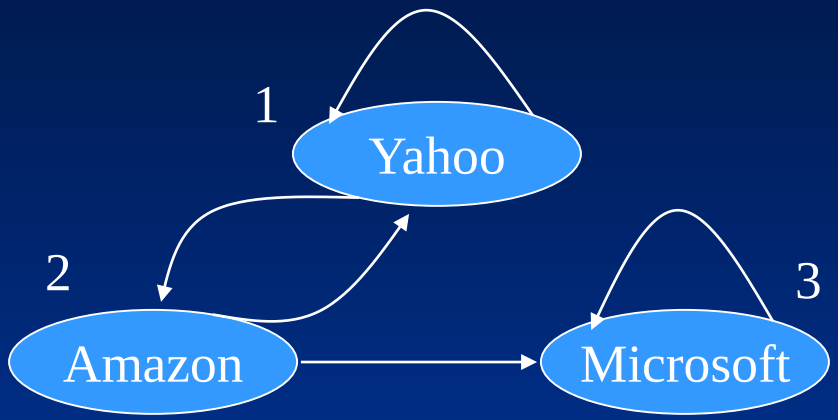
$$\begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} \begin{pmatrix} 2/6 \\ 1/6 \\ 1/6 \end{pmatrix} \begin{pmatrix} 3/12 \\ 2/12 \\ 1/12 \end{pmatrix} \begin{pmatrix} 5/24 \\ 3/24 \\ 2/24 \end{pmatrix} \begin{pmatrix} 8/48 \\ 5/48 \\ 3/48 \end{pmatrix} , \ldots, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

PageRank Accounting for Spider Traps and Dead Ends

Limiting random walker is allowed to wander at random. We let the walker follow a random out-link, if there is one, with probability β (normally, $0.8 \leq \beta \leq 0.9$). With probability $1 - \beta$ (called the taxation rate), we remove that walker and deposit a new walker at a randomly chosen Web page.

- If the walker gets stuck in a spider trap, it doesn't matter because after a few time steps, that walker will disappear and be replaced by a new walker.
- If the walker reaches a dead end and disappears, a new walker takes over shortly.

Example.



$$\mathbf{M} = \begin{array}{c} \begin{matrix} p1 & p2 & p3 \end{matrix} \\ \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 1 \end{pmatrix} \end{array}$$

Let $\mathbf{P}_{new}$ and $\mathbf{P}_{old}$ be the new and old distributions of the location of the walker after one iteration, the relationship between these two can be expressed as:

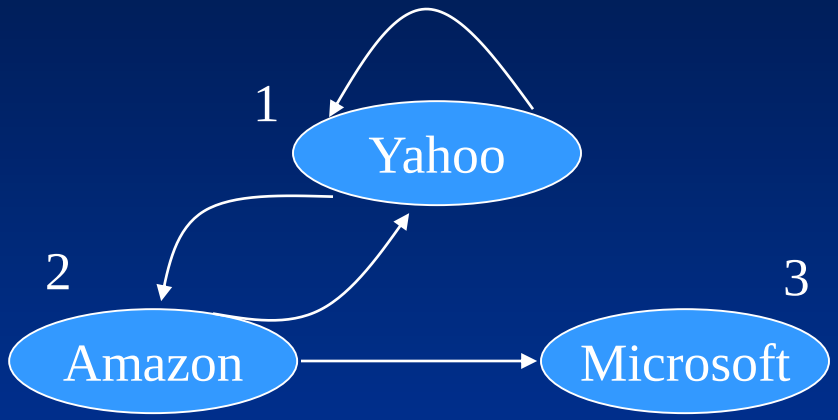
$$\mathbf{P}_{new} = \underset{\beta}{0.8} \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 1 \end{pmatrix} \mathbf{P}_{old} + \underset{1-\beta}{0.2} \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$$

The meaning of the above equation is:
With probability 0.8, we multiply $\mathbf{P}_{old}$ by the matrix of the Web to get the new location of the walker, and with probability 0.2 we start with a new walker at a random place.

If we start with $\mathbf{P}_{old}$ = (1/3, 1/3, 1/3) and repeatedly compute $\mathbf{P}_{new}$ and then replace $\mathbf{P}_{old}$ by $\mathbf{P}_{new}$, we get the following sequence of approximation to the asymptotic distribution of the walker:

$$\begin{pmatrix} .333 \\ .333 \\ .333 \end{pmatrix} \begin{pmatrix} .333 \\ .200 \\ .467 \end{pmatrix} \begin{pmatrix} .280 \\ .300 \\ .520 \end{pmatrix} \begin{pmatrix} .259 \\ .179 \\ .563 \end{pmatrix} , \ldots, \begin{pmatrix} 7/33 \\ 5/33 \\ 21/33 \end{pmatrix}$$

Example.



$$\mathbf{M} = \begin{array}{c} \\ p1 \\ p2 \\ p3 \end{array} \begin{pmatrix} \text{p1} & \text{p2} & \text{p3} \\ ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 0 \end{pmatrix}$$

$$\mathbf{P}_{new} = \underset{\beta}{0.8} \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 0 \\ 0 & ½ & 0 \end{pmatrix} \mathbf{P}_{old} + \underset{1-\beta}{0.2} \begin{pmatrix} 1/3 \\ 1/3 \\ 1/3 \end{pmatrix}$$

If we start with $\mathbf{P}_{old}$ = (1/3, 1/3, 1/3) and repeatedly compute $\mathbf{P}_{new}$ and then replace $\mathbf{P}_{old}$ by $\mathbf{P}_{new}$, we get the following sequence of approximation to the asymptotic distribution of the walker:

$$\begin{pmatrix} .333 \\ .333 \\ .333 \end{pmatrix} \begin{pmatrix} .333 \\ .200 \\ .200 \end{pmatrix} \begin{pmatrix} .280 \\ .200 \\ .147 \end{pmatrix} \begin{pmatrix} .259 \\ .179 \\ .147 \end{pmatrix} , \ldots, \begin{pmatrix} 35/165 \\ 25/165 \\ 21/165 \end{pmatrix}$$

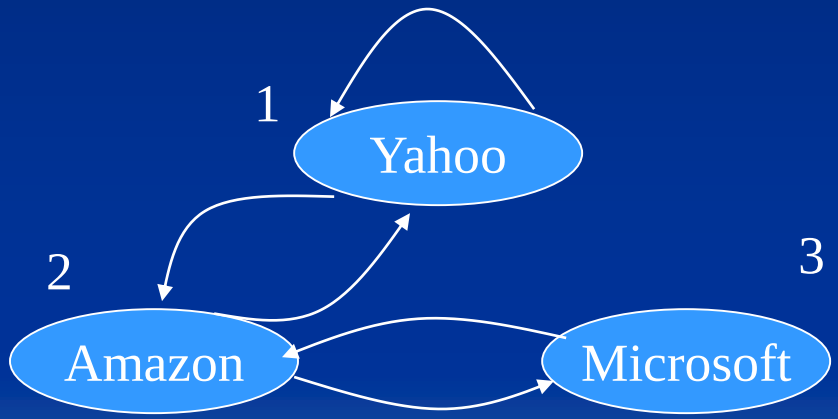
Notice that these probabilities do not sum to one, and there is slightly more than 50% probability that the walker is "lost" at any given time. However, the ratio of the importance of Yahoo!, and Amazon are the same as in the above example. That makes sense because in both the cases there are no links from the Microsoft page to influence the importance of Yahoo! or Amazon.

Topic-Specific PageRank

The calculation o PageRank should be biased to favor certain pages.

Teleport Sets

Choose a set of pages about a certain topic (e.g., sport) as a teleport set.



Assume that we are interested only in retail sales, so we choose a teleport set that consists of Amazon alone.

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix}$$

$$\begin{pmatrix} y \\ a \\ m \end{pmatrix} = 0.8 \begin{pmatrix} ½ & ½ & 0 \\ ½ & 0 & 1 \\ 0 & ½ & 0 \end{pmatrix} \begin{pmatrix} y \\ a \\ m \end{pmatrix} + 0.2 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

The entry for Amazon is set to 1.

Topic-Specific PageRank

The *general rule* for setting up the equations in a topic-specific PageRank problem is as follows.

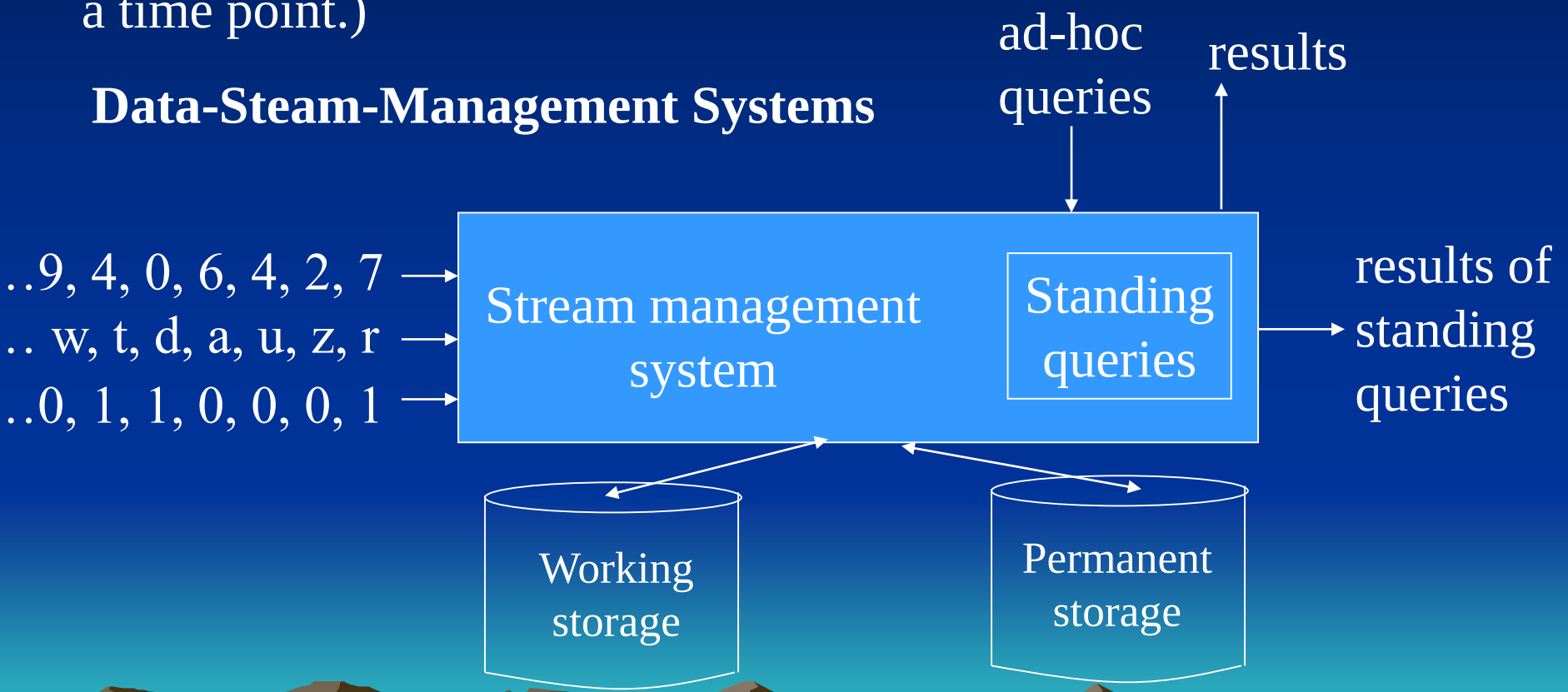
Suppose there are k pages in the teleport set. Let **T** be a column-vector that has $1/k$ in the positions corresponding to members of the teleport set and 0 elsewhere. Let **M** be the transition matrix of the Web. Then, we must solve by relaxation the following iterative rule:

$$\mathbf{P}_{new} = \beta\mathbf{M}\mathbf{P}_{old} + (1 - \beta)\mathbf{T}$$

Data Streams

A data steam is a sequence of tuples, which may be unbounded. (Note that a relation is a set of tuples. The set is always bounded at a time point.)

Data-Steam-Management Systems



Data Streams

The system accepts data streams as input, and also accepts queries.
Two kinds of queries:

1. Conventional ad-hoc queries.
2. Standing queries that are stored by the system and run on the input streams at all times.

Example.

Suppose we are receiving streams of radiation levels from sensors around the world.

1. DSMS stores a *sliding window* of each input stream in the "working storage". All readings from all sensors for the past 24 hours.
2. Data from further back in time could be dropped, summarized, or copied in its entirety to the permanent store (archive)

Stream Applications

1. Click streams. A Web site might wish to analyze the clicks it receives. (An increase in clicks on a link may indicate that link is broken, or that it has become of much more interest recently.)
2. Packet streams. We may wish to analyze the sources and destinations of IP packets that pass through a switch. An unusual increase in packets for a destination may warn of a denial-of-service attack.
3. Sensor data. There are many kinds of sensors whose outputs need to be read and considered collectively, e.g., tsunami warning sensors that record ocean levels at subsecond frequencies or the signals that come from seismometers around the world.

Stream Applications

4. Satellite data. Satellites send back to the earth incredible streams of data, often petabytes per day.
5. Financial data. Trades of stocks, commodities, and other financial instruments are reported as a stream of tuples, each representing one financial transaction. These streams are analyzed by software that looks for events or patterns that trigger actions by traders.

A Data-Stream Data Model

- Each stream consists of a sequence of tuples. The tuples have a fixed relation schema (list of attributes), just as the tuples of a relation do. However, unlike relations, the sequence of tuples in a stream may be unbounded.
- Each tuple has an associated arrival time, at which time it becomes available to DSMS for processing. The DSMS has the option of placing it in the working storage or in the permanent storage, or of dropping the tuple from memory altogether. The tuple may also be processed in simple ways before storing it.

A Data-Stream Data Model

For any stream, we can define a sliding window, which is a set consisting of the most recent tuples to arrive.

- Time-based. It consists of the tuples whose arrival time is between the current time t and $t - \tau$, where τ is a constant.
- Tuple-based. It consists of the most recent n tuples to arrive for some fixed n.

For a certain stream S, we use the notation $S[W]$ to represent a window, where W is:

1. Row n, meaning the most recent n tuples of the stream; or
2. Range τ, meaning all tuples that arrived within the previous amount of time τ.

Example.

Let Sensors(sensID, temp, time) be stream, each of whose tuples represent a temperature reading of temp at a certain time by the sensor named sensID.

Sensors[Row 1000]

describes a window on the Sensor stream consisting of the most recent 1000 tuples.

Sensors[Range 10 seconds]

describes a window on the Sensor stream consisting of all tuples that arrived in the past 10 seconds.

Handling Streams as Relations

Each stream window can be handled as a relation, whose content changes rapidly.
Suppose we would like to know, for each sensor, the highest recorded temperature to arrive at the DSMS in the past hour.

```
SELECT sensID, MAX(temp)
FROM Sensors[Range 1 hour]
GROUP BY sensID;
```

Handling Streams as Relations

Suppose that besides the stream Sensors, we also maintain an ordinary relation:

Calibrate(sensID, mult, add),

which gives a multiplicative factor and additive term that are used to correct the reading from each sensor.

```
SELECT MAX(mult*temp + add)
FROM Sensors[Range 1 hour], Calibrate
WHERE Sensors.sensID = Calibrate.sensID
```

The query finds the highest, properly calibrated temperature reported by any sensor in the past hour.

Handling Streams as Relations

Suppose we wanted to give, for each sensor, its maximum temperature over the past hour, but we also wanted the resulting tuples to give the most recent time at which that maximum temperature was recorded.

```
SELECT s.sensID, s.temp, s.time
FROM Sensors[Range 1 Hour] s
WHERE NOT EXISTS (
        SELECT * FROM Sensors[Range 1 Hour]
        WHERE sensID = s.sensID AND (
                temp > s.temp OR
                (temp = s.temp AND time > s.time)
));
```