Database System

- Database System Architecture
- System Catalog
- Query Processing and Optimization,
- Transaction Management
- Concurrency Control

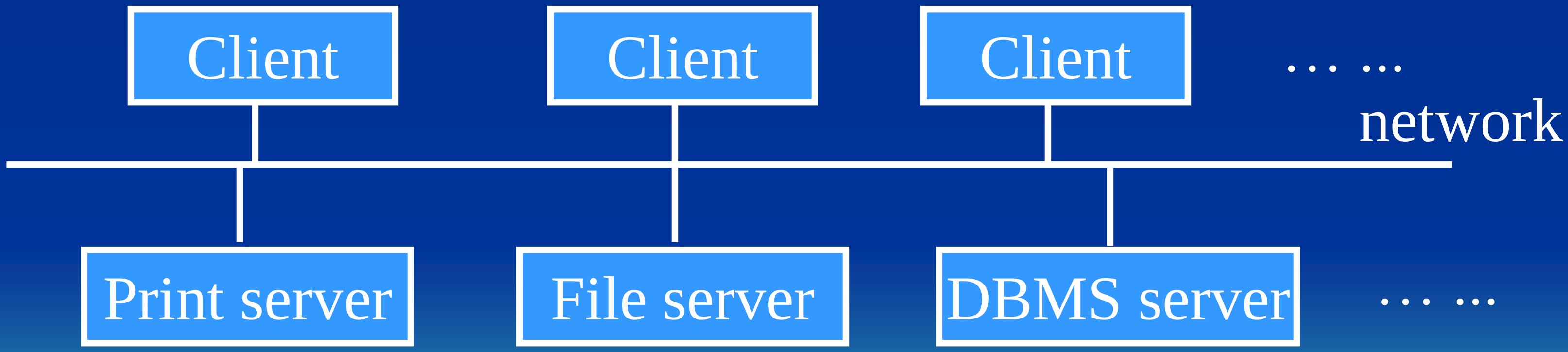
Database System Architecture

- Main frame computer and main frame database architecture
- Client-server computer architecture
- Client-server database architecture

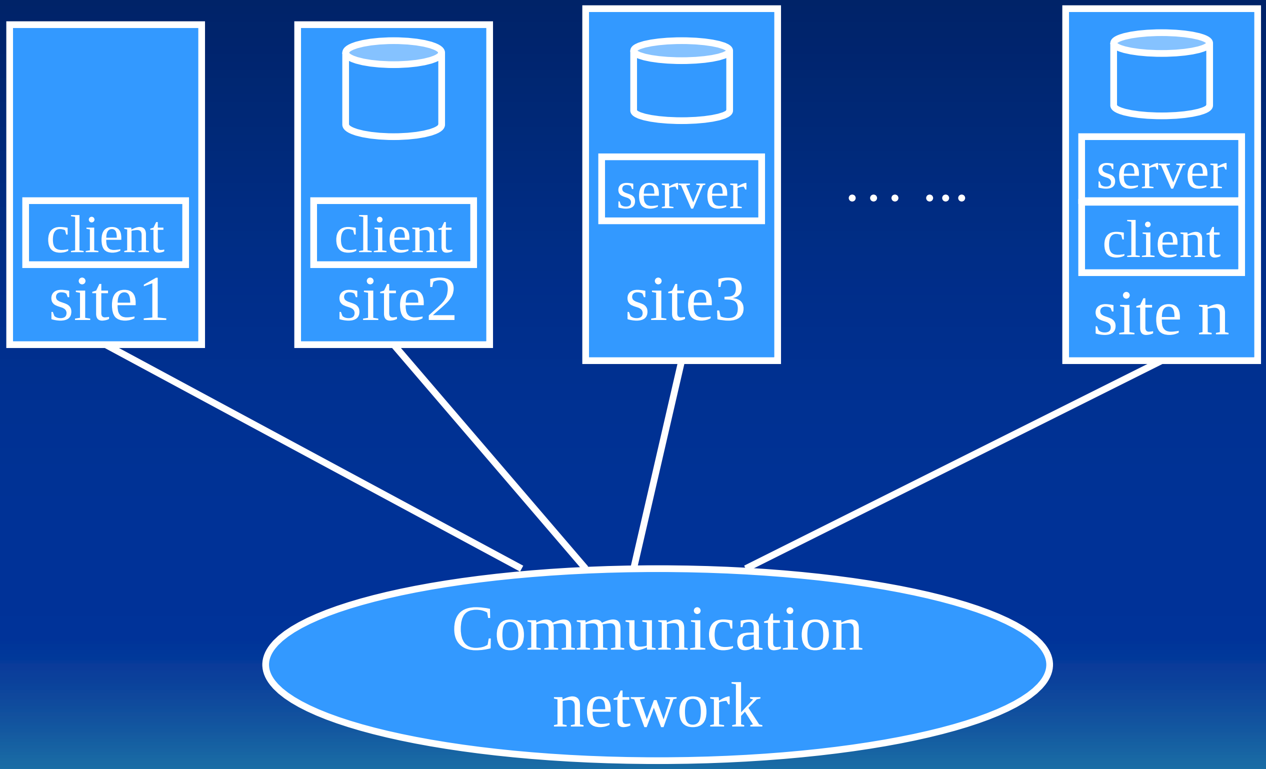
Client-Server Computer Architecture

- 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 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:

Data file: Works_on

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

123456789, 1	•
234567891, 2	•
… …	

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

SSN	Pno	hours
234567891	2	
345678912	2	
345678912	3	
456789123	1	

… …

Clustering index:

Data file: Works_on

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

123456789	•
234567891	•
345678912	•
456789123	•

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

SSN	Pno	hours
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

Select FNAME, LNAME, PNAME
From Works_on1
Where FNAME = ‘David’ and LNAME = ‘Shepperd’

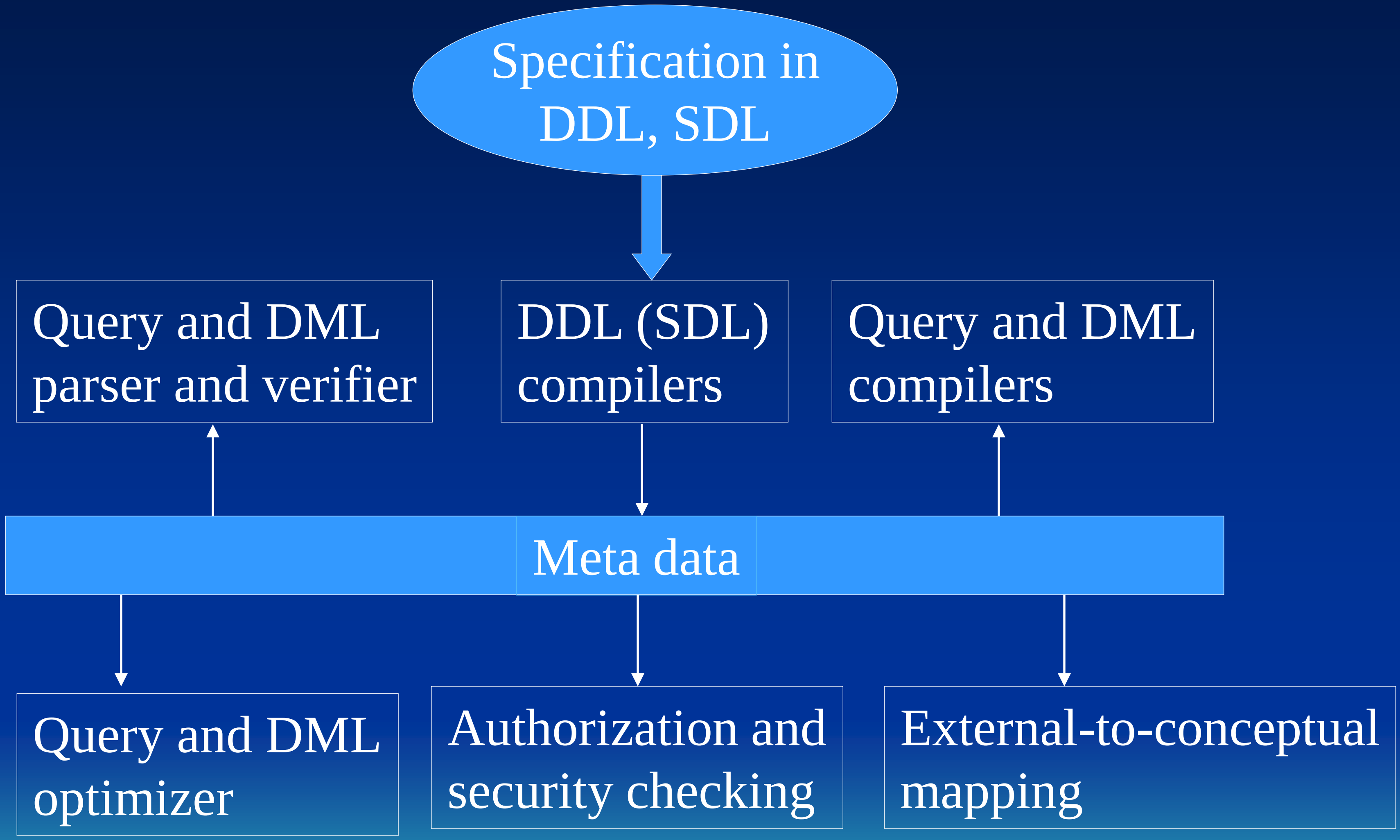
⇓

Select FNAME, LNAME, PNAME
From EMPLOYEE, PROJECT, WORKS_ON

Where ssn = essn and

Pno. = PNUMBER and

FNAME = ‘David’ and LNAME = ‘Shepperd’


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

Basic algorithms

-Sorting: internal sorting and external sorting

-Implementing the SELECT operation

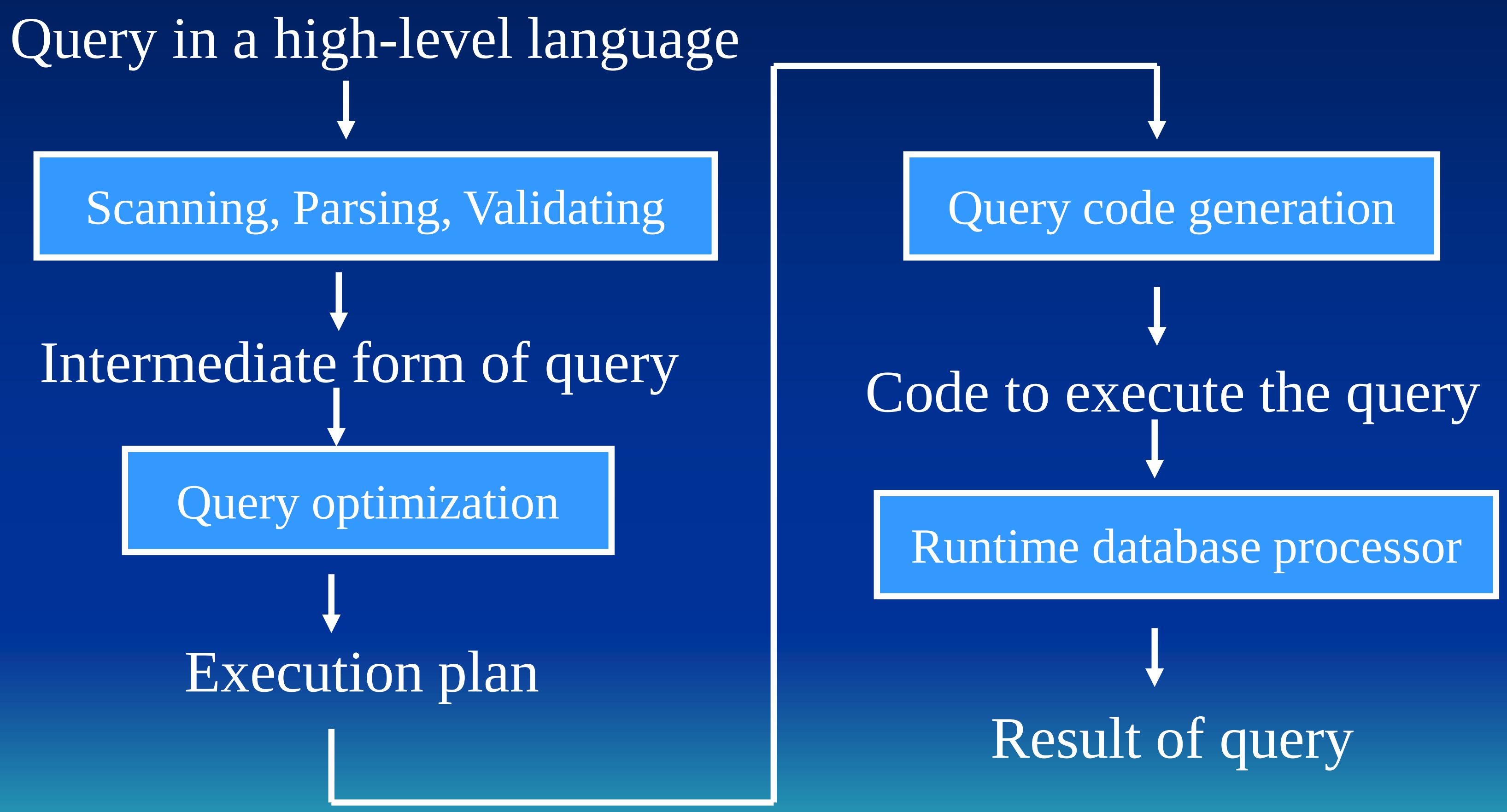
-Implementing the JOIN operation

-Implementing the PROJECT 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
```

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

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

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

- **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

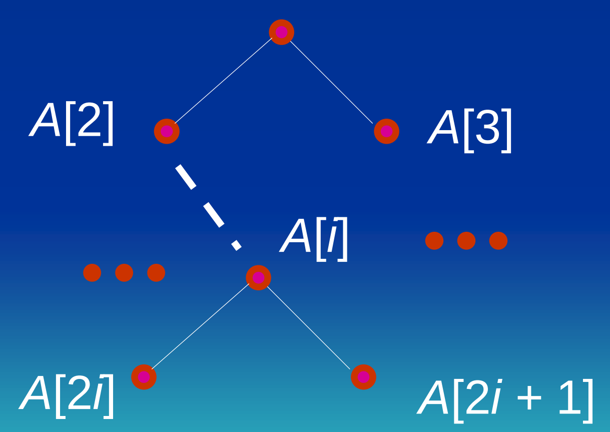
- **Sorting algorithms**
  - internal sorting - sorting in main memory:
    sort a series of integers,
    sort a series of keys
    sort a series of records
  - different sorting methods:
    simple sorting
    merge sorting
    quick sorting
    heap sorting
  - external sorting – sorting a file which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

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] – $A[\lfloor i/2 \rfloor]$

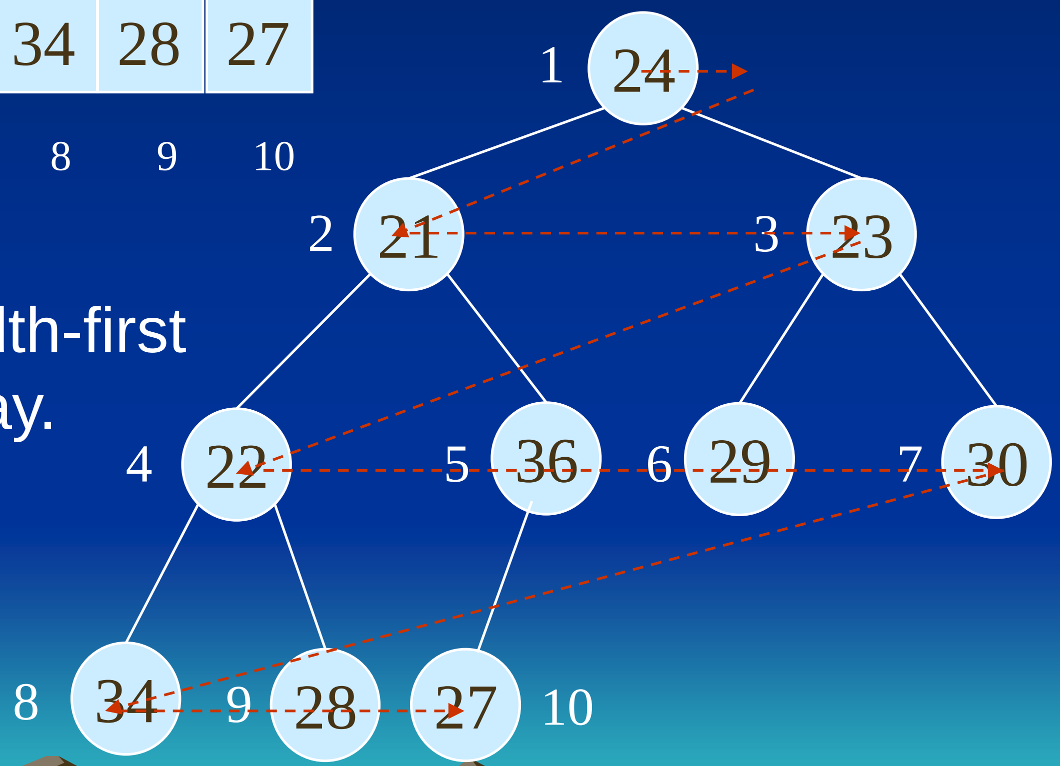


Data Structure Binary Heap

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

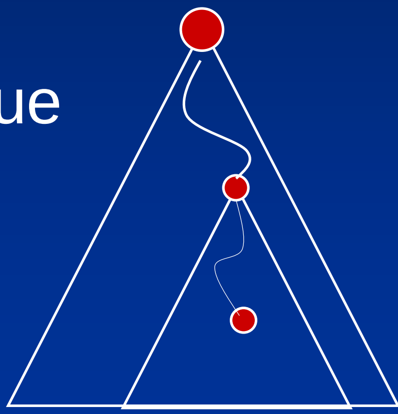
24	21	23	22	36	29	30	34	28	27
1	2	3	4	5	6	7	8	9	10

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.



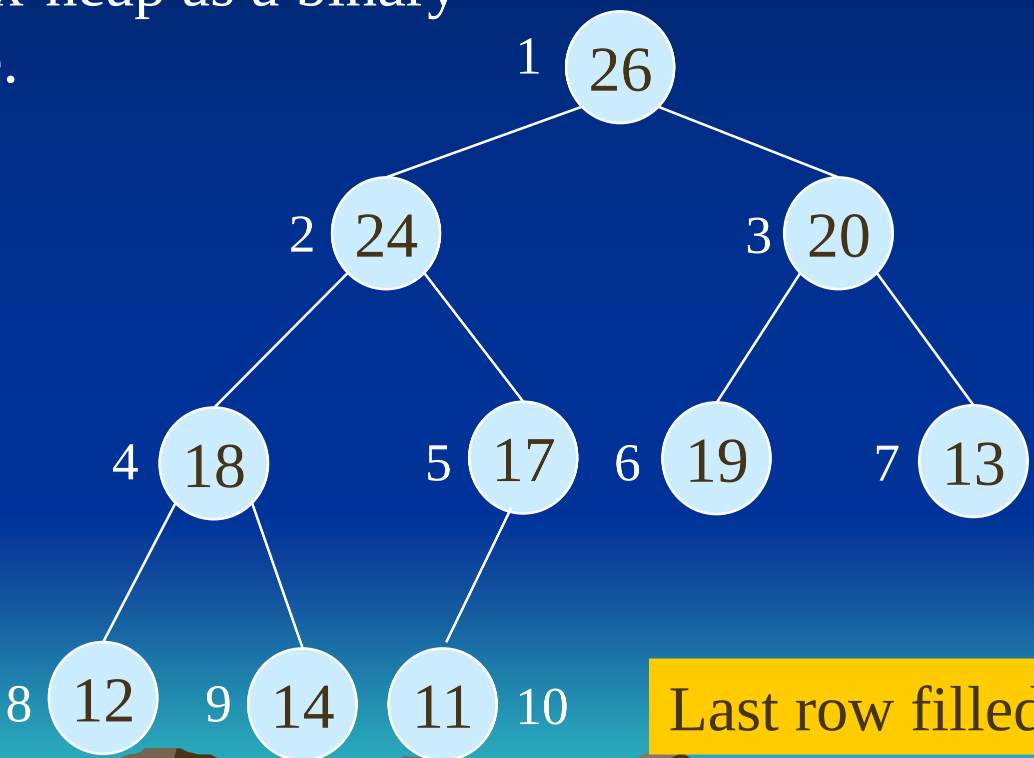
- Min-Heap
  - For every node excluding the root, the value stored in that node is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at the subtree's root

Heaps – Example

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

Max-heap as an array.

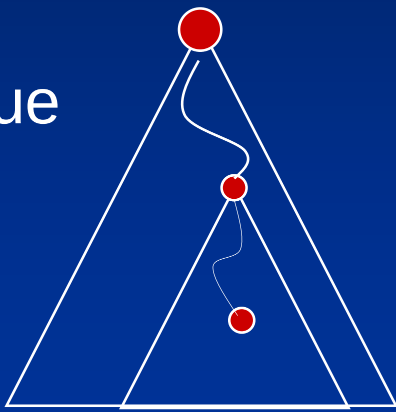
Max-heap as a binary tree.



Last row filled from left to right.

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.



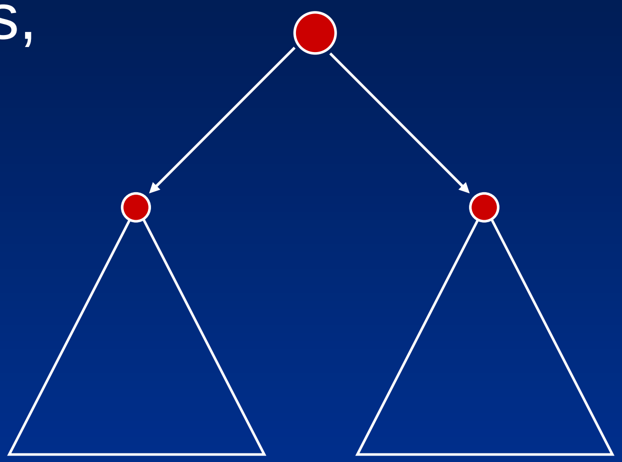
- Min-Heap
  - For every node excluding the root, the value stored in that node is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at the subtree's root

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

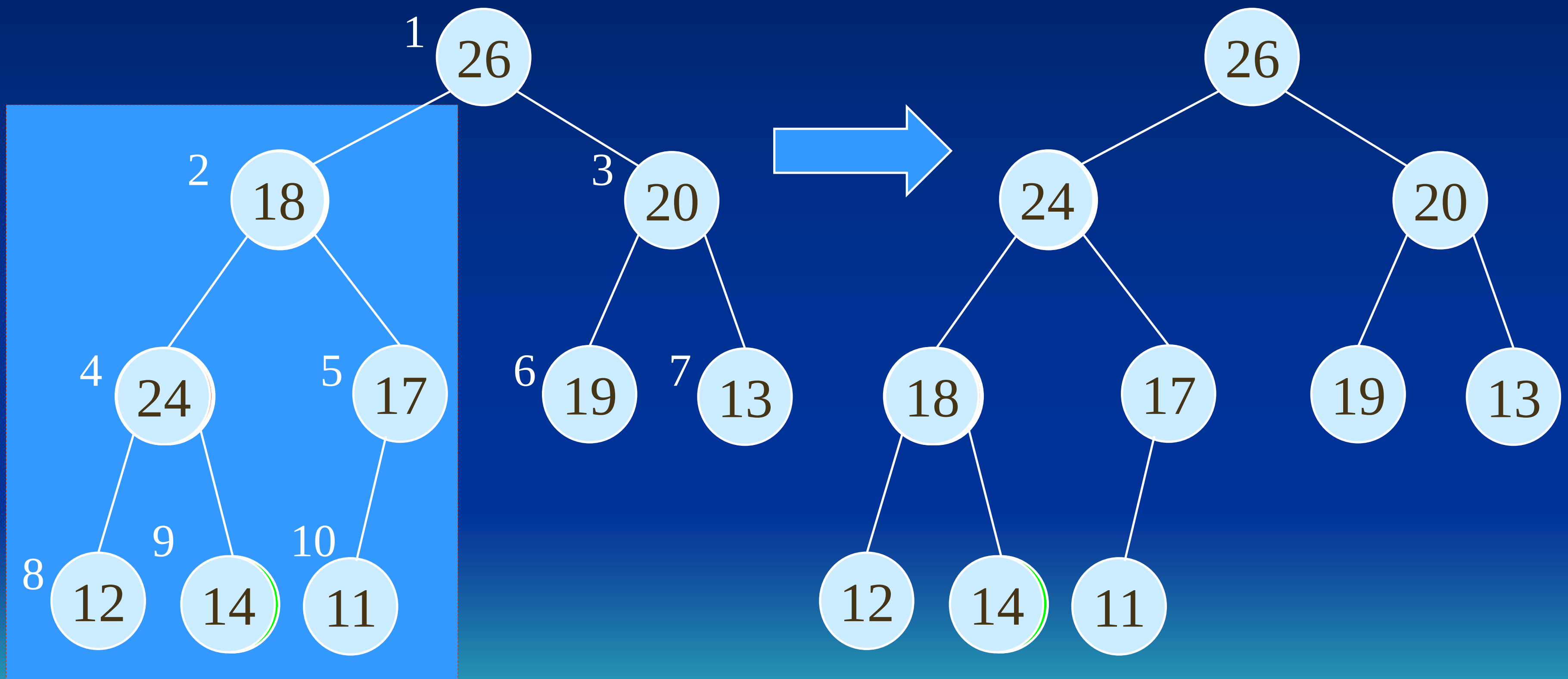
- Suppose two subtrees are max-heaps, but the root violates the max-heap property.



- 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)

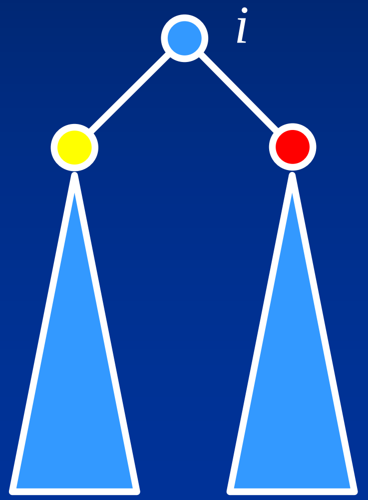


Procedure MaxHeapify

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.

<u>*BuildMaxHeap(A)*</u>

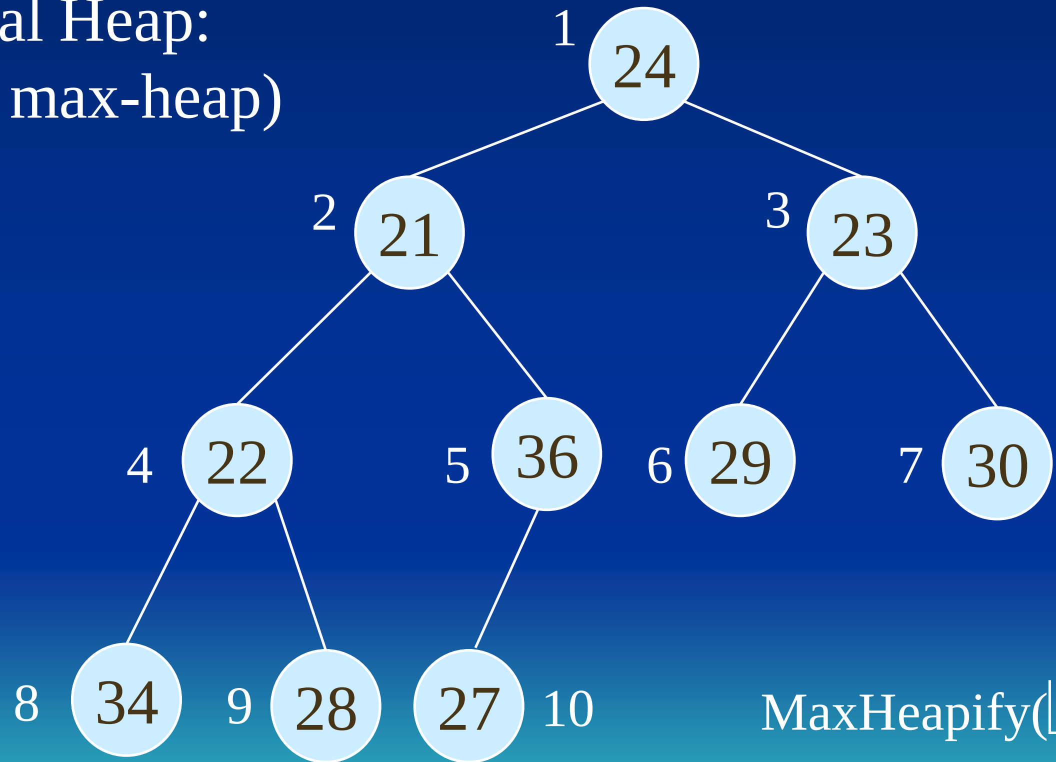
1. *heap-size*[*A*] ← *length*[*A*]
2. **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 (**A*[*length*[*A*]/2⌋ +1], *A*[*length*[*A*]/2⌋ +2], … are leaf nodes.*)
3. **do** *MaxHeapify*(*A*, *i*)

BuildMaxHeap – Example

Input Array:

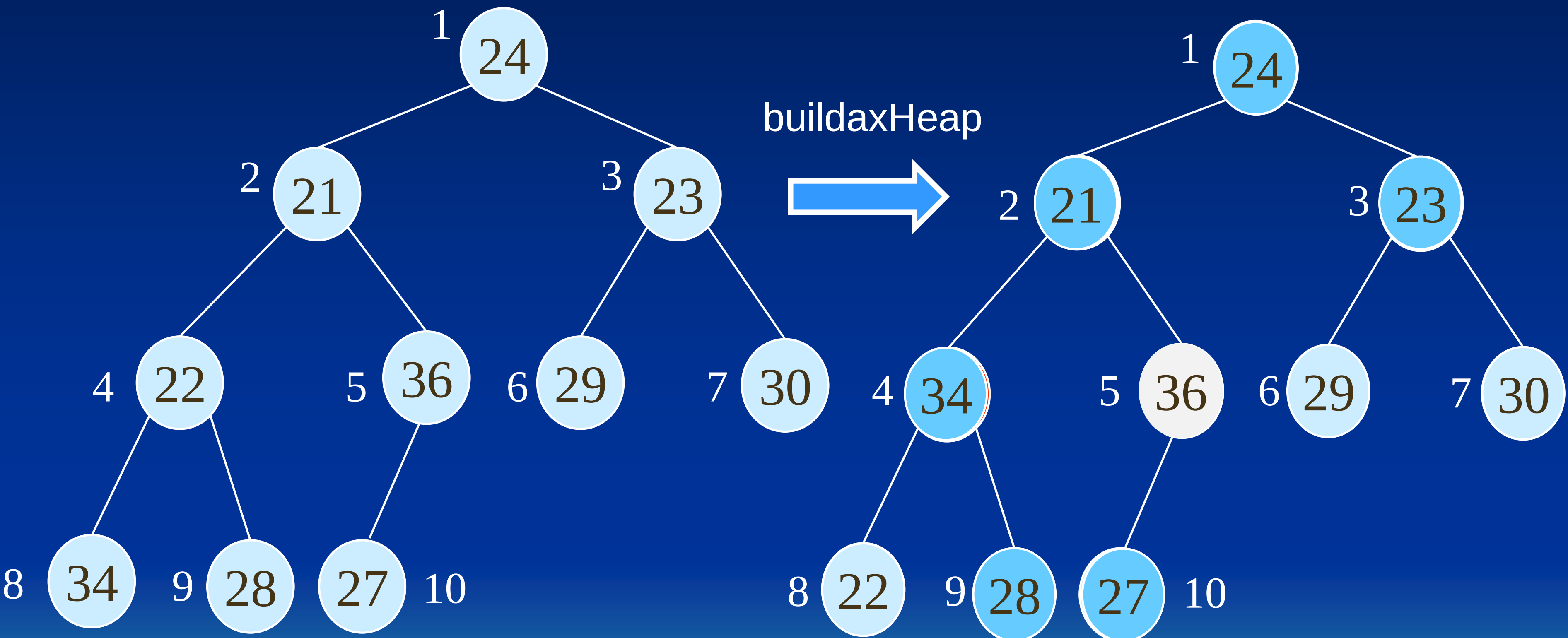
24	21	23	22	36	29	30	34	28	27

Initial Heap:
(not max-heap)



MaxHeapify(⌊10/2⌋ = 5):

BuildMaxHeap – Example



MaxHeapify($\lfloor 10/2 \rfloor$ = 5), MaxHeapify(4)

Heapsort(A)

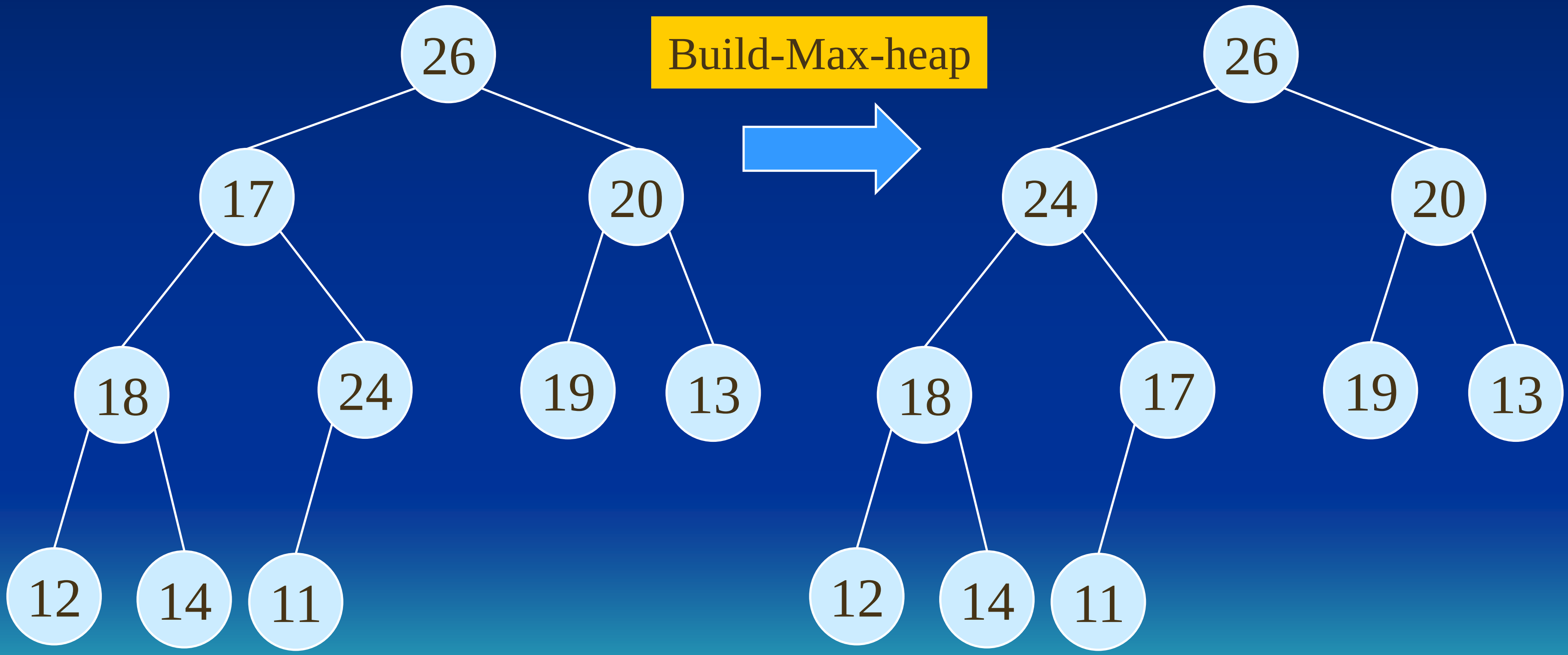
<u>*HeapSort*(A)</u>

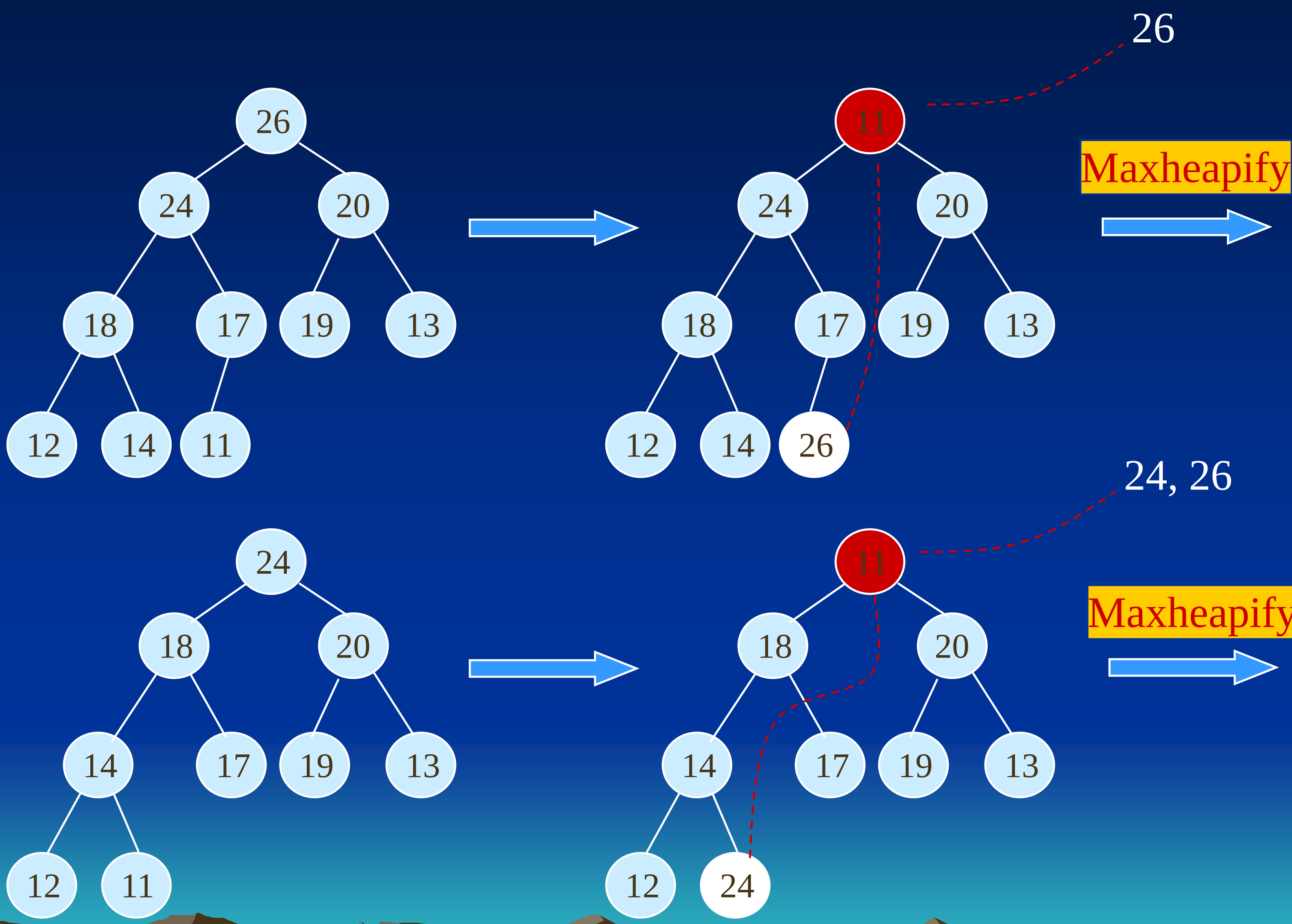
1. *BuildMaxHeap*(A)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5. *MaxHeapify*(A, 1)

Time complexity: $O(n \cdot \log n)$

Heapsort – Example

26	17	20	18	24	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10




26
Maxheapify
24, 26
Maxheapify
26 24 20 18 17 19 13 12 14 11
11 24 20 18 17 19 13 12 14 26
24 18 20 14 17 19 13 12 11
11 18 20 14 17 19 13 12 24

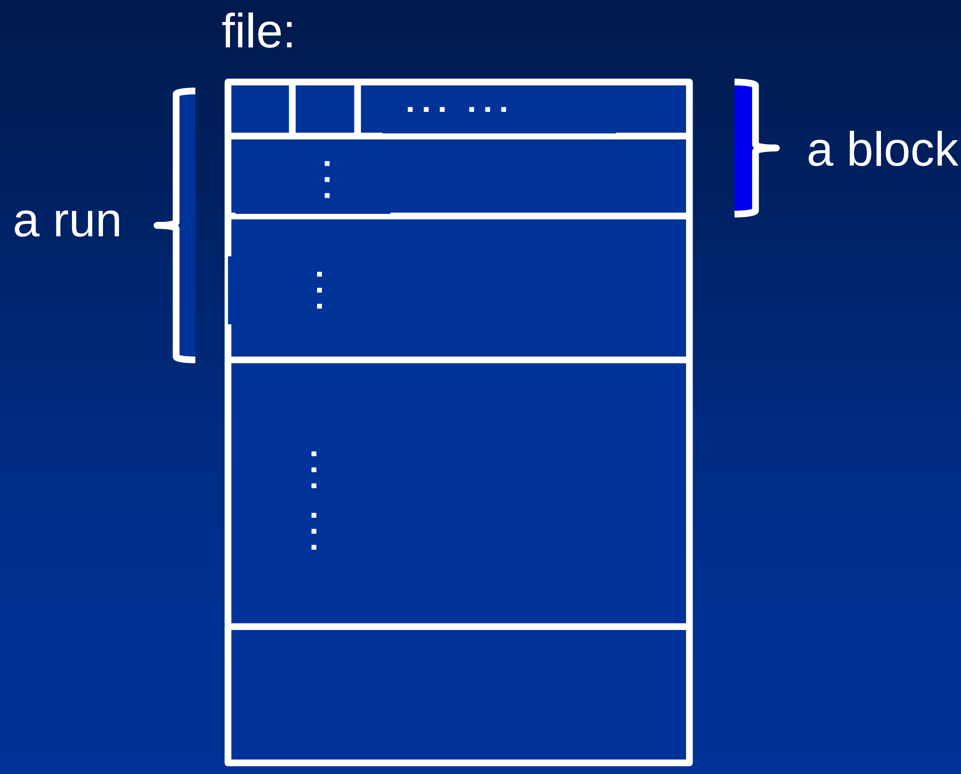
- **Basic algorithms**
  - External sorting method:

    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 (the size of each run is the same as the buffer.)

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

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

set $i \leftarrow 1$;

$j \leftarrow b$; /*size of the file in blocks*/

$k \leftarrow n_B$; /*size of buffer in blocks*/

$m \leftarrow \lceil j/k \rceil$; /*number of runs*/

/*sort phase*/

while ($i \leq m$)

do {read next k blocks of the file into the buffer or if there are less than k blocks remaining then read in the remaining blocks;

sort the records in the buffer and write as a temporary subfile;

$i \leftarrow i + 1$;

}

- **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

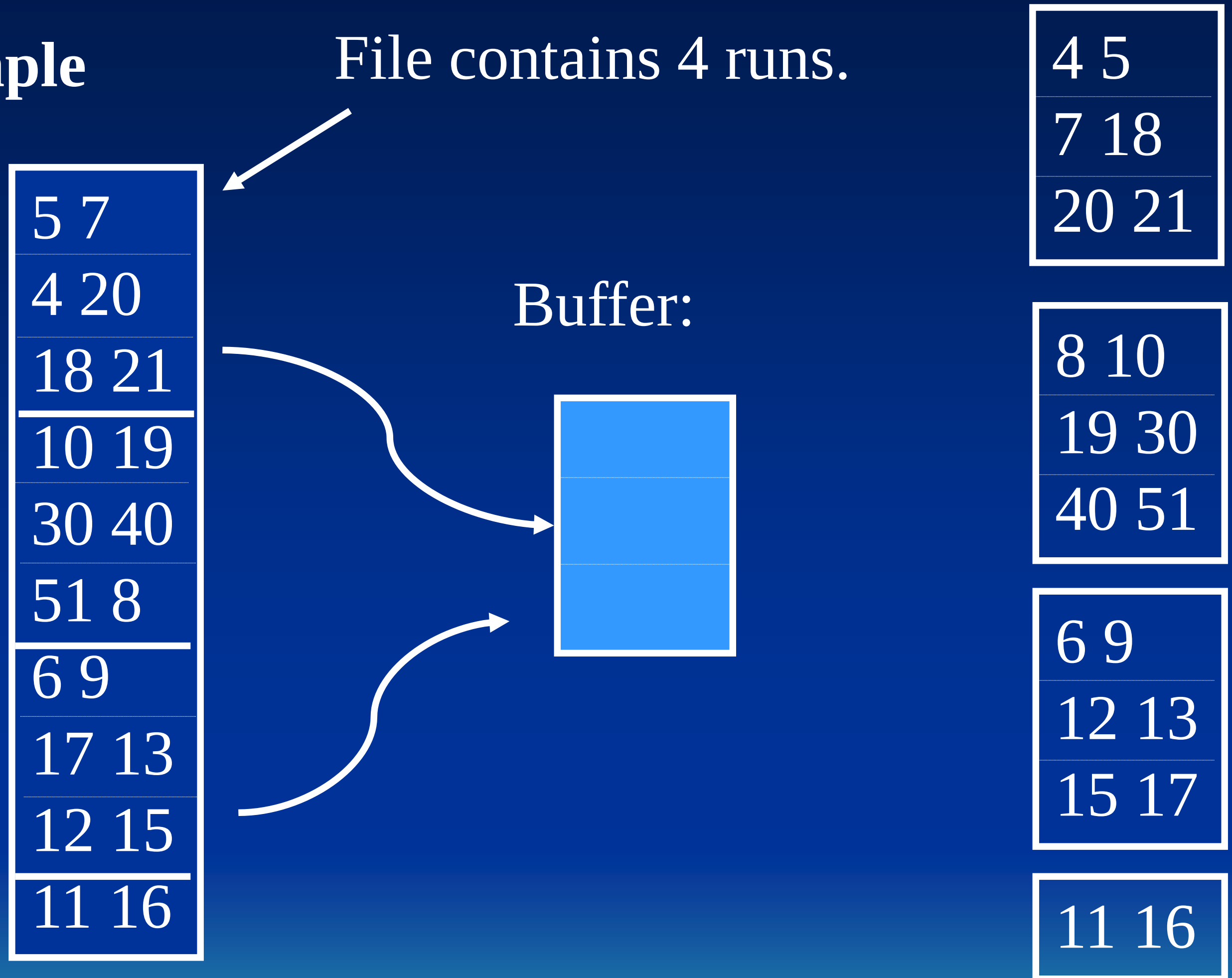
{read next k-1 subfiles or remaining subfiles (from previous pass) one block at a time;

merge and write as new subfile;

$n \leftarrow n+1$;}

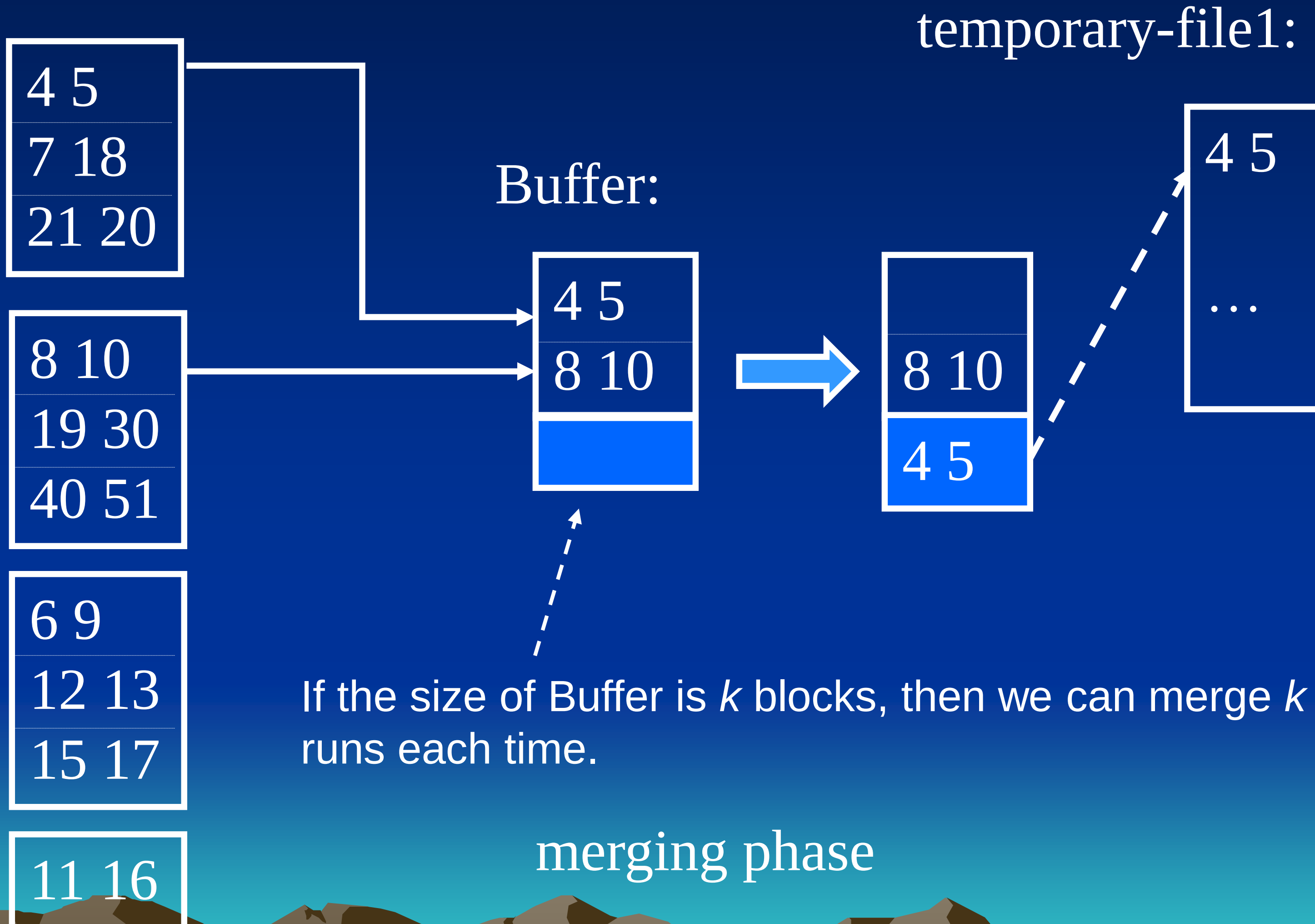
$j \leftarrow q$; $i \leftarrow i + 1$;}

- **Example**



sorting phase

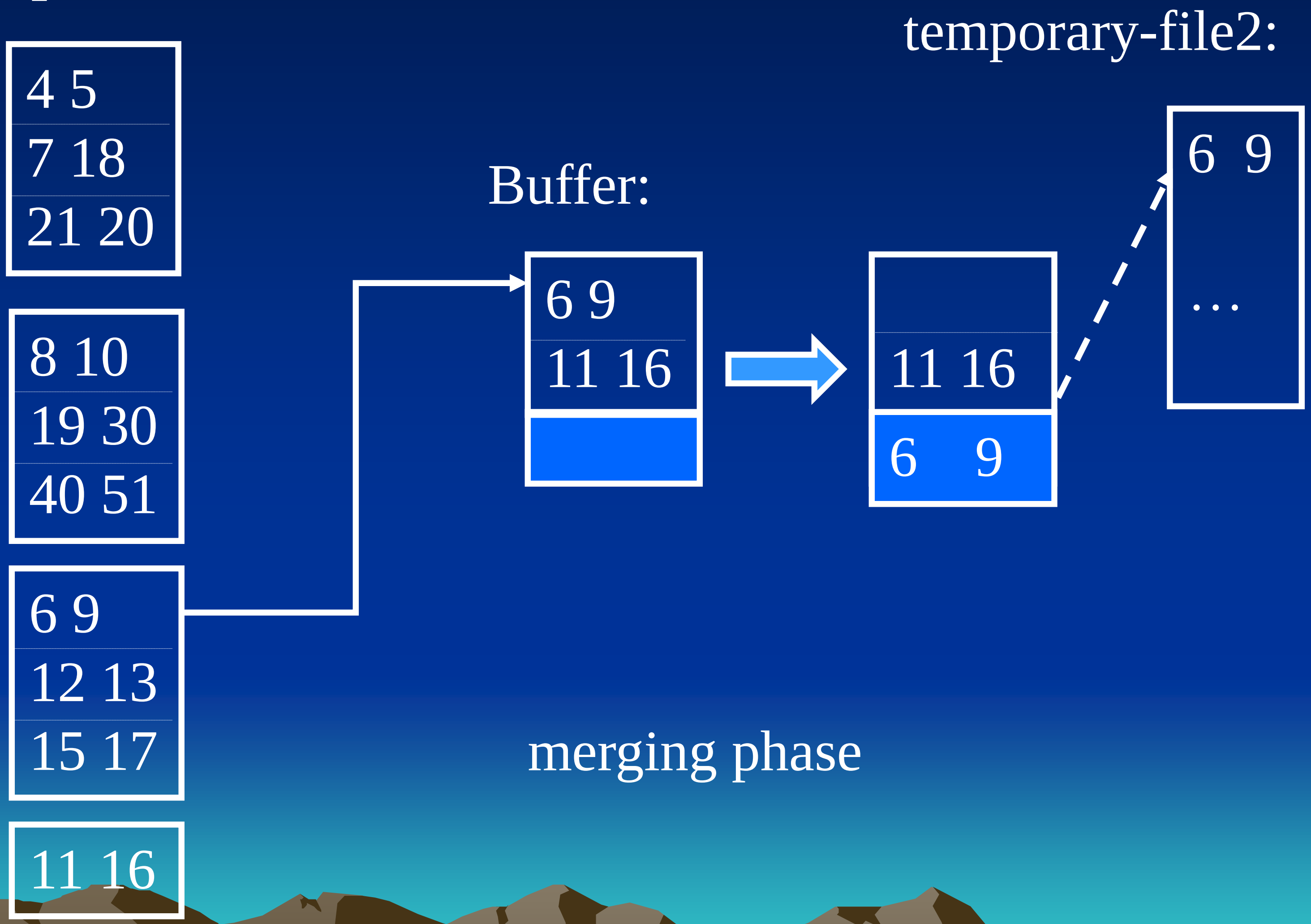
- **Example**



If the size of Buffer is *k* blocks, then we can merge *k* – 1 runs each time.

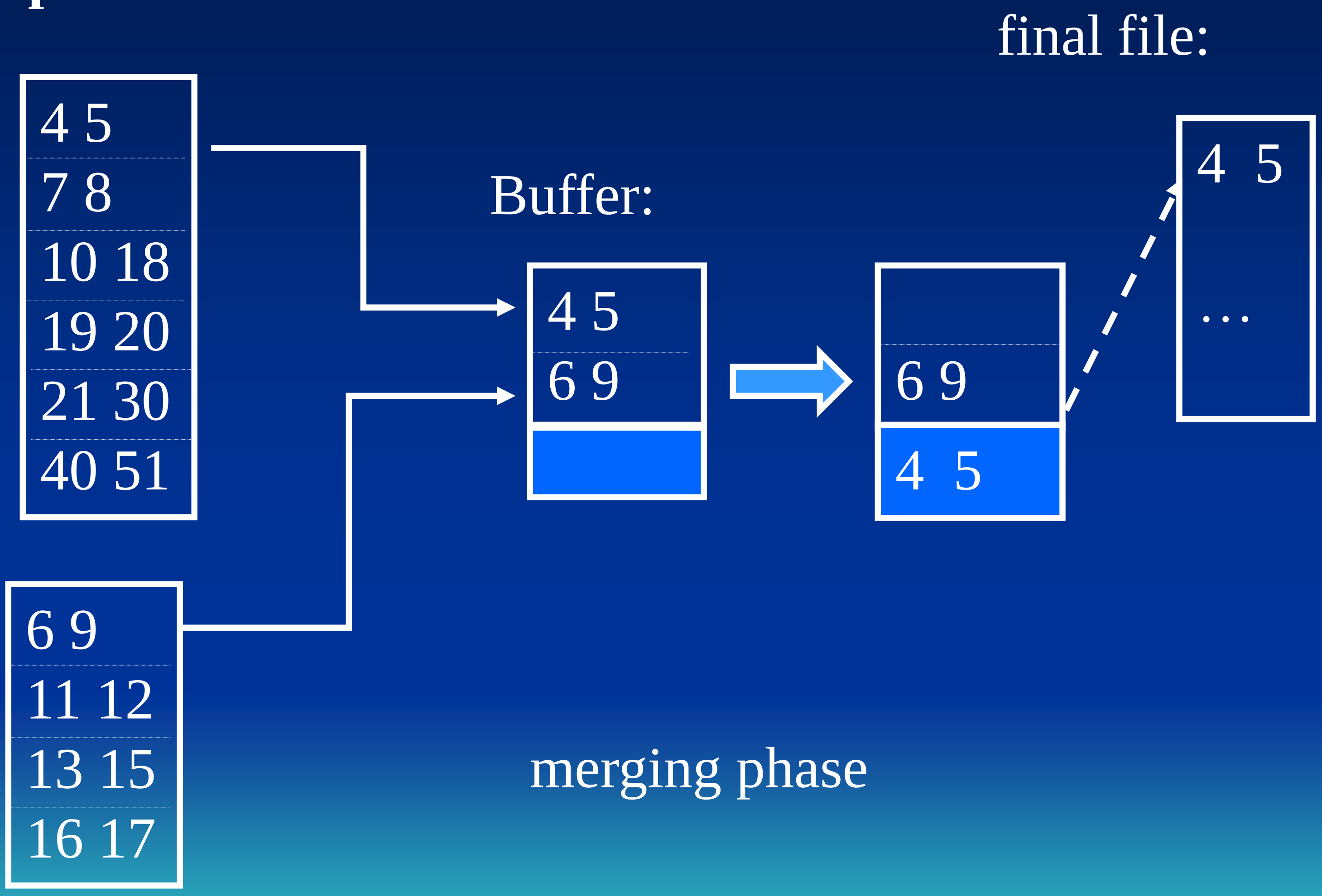
merging phase

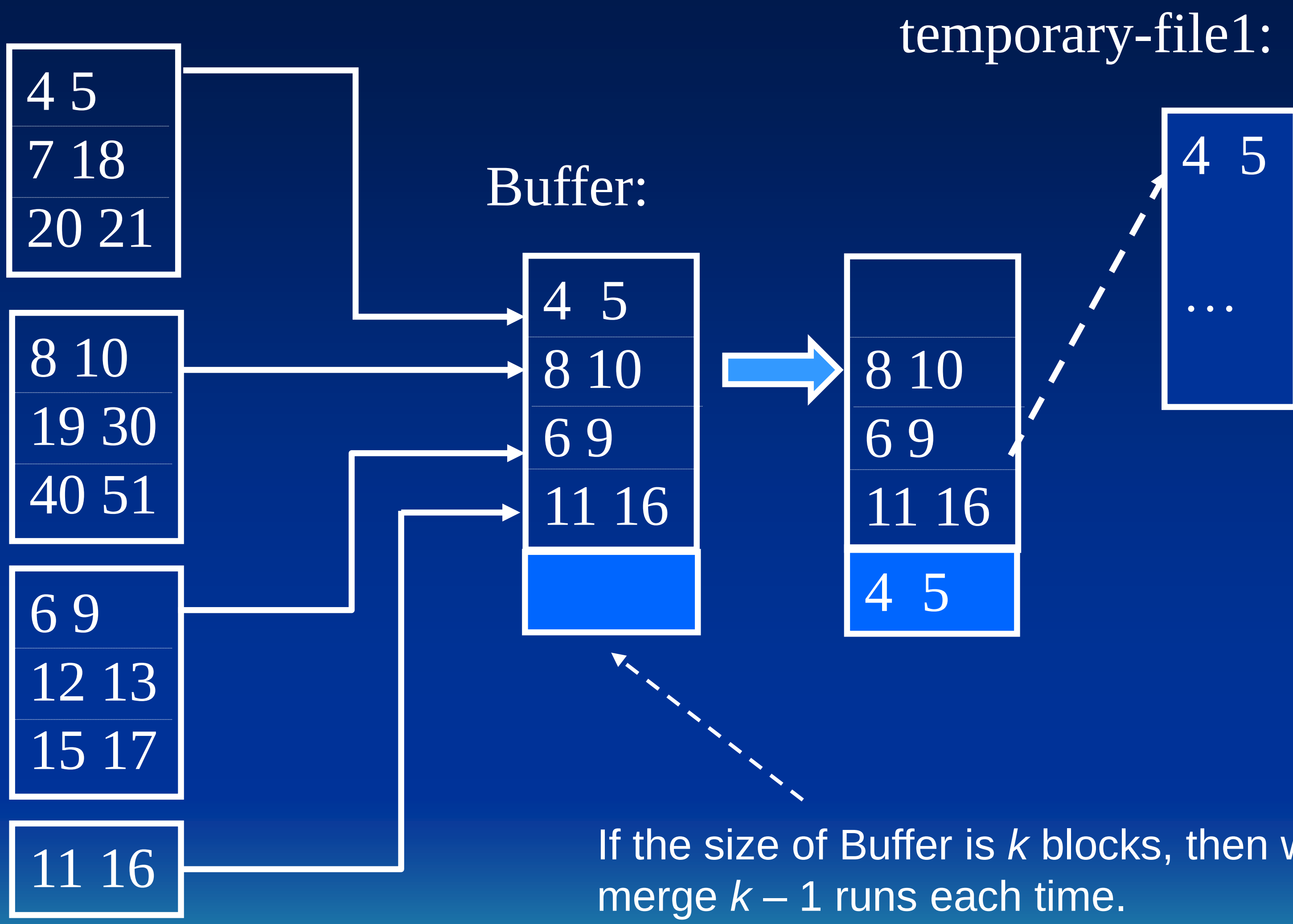
- **Example**



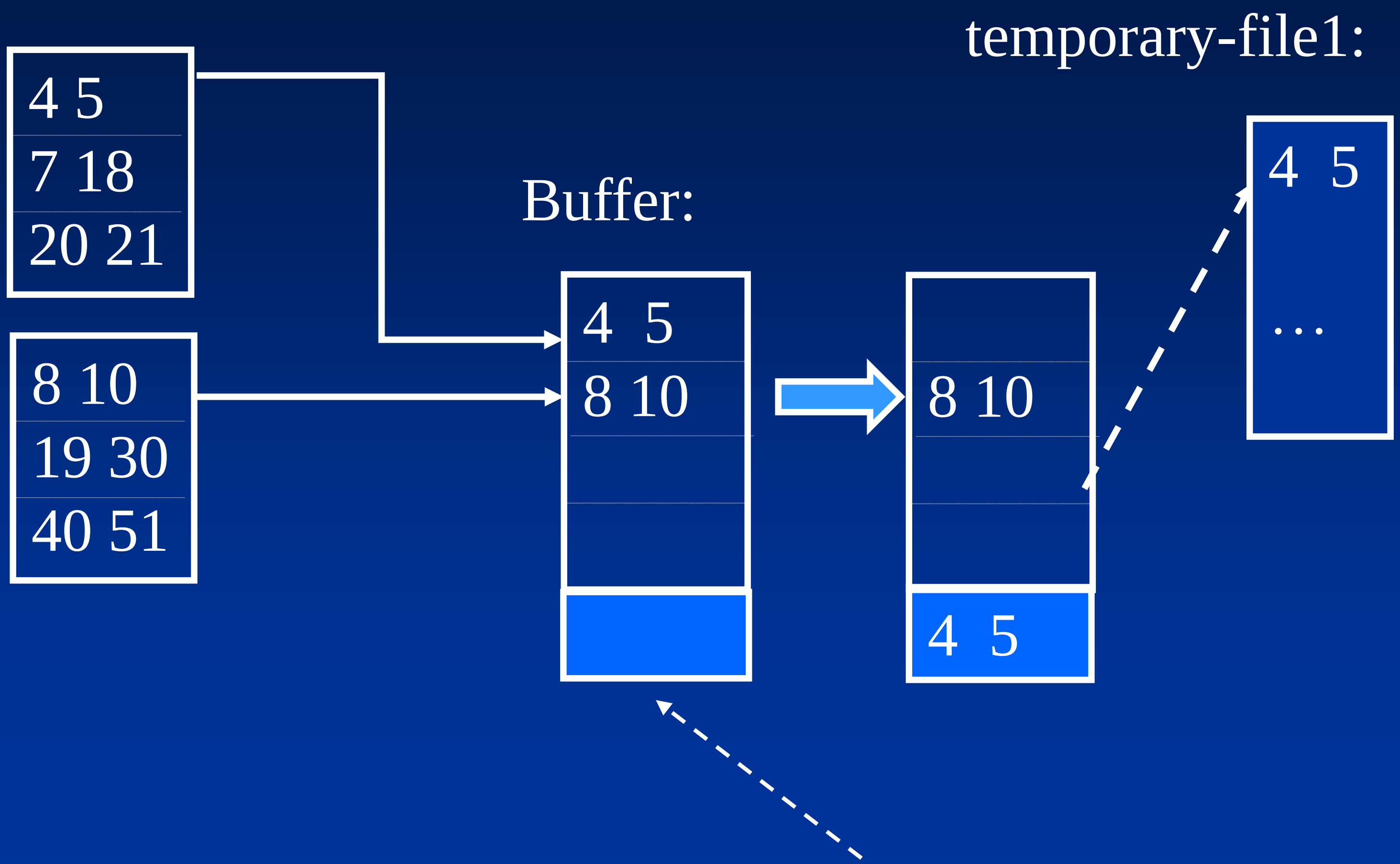
merging phase

- **Example**

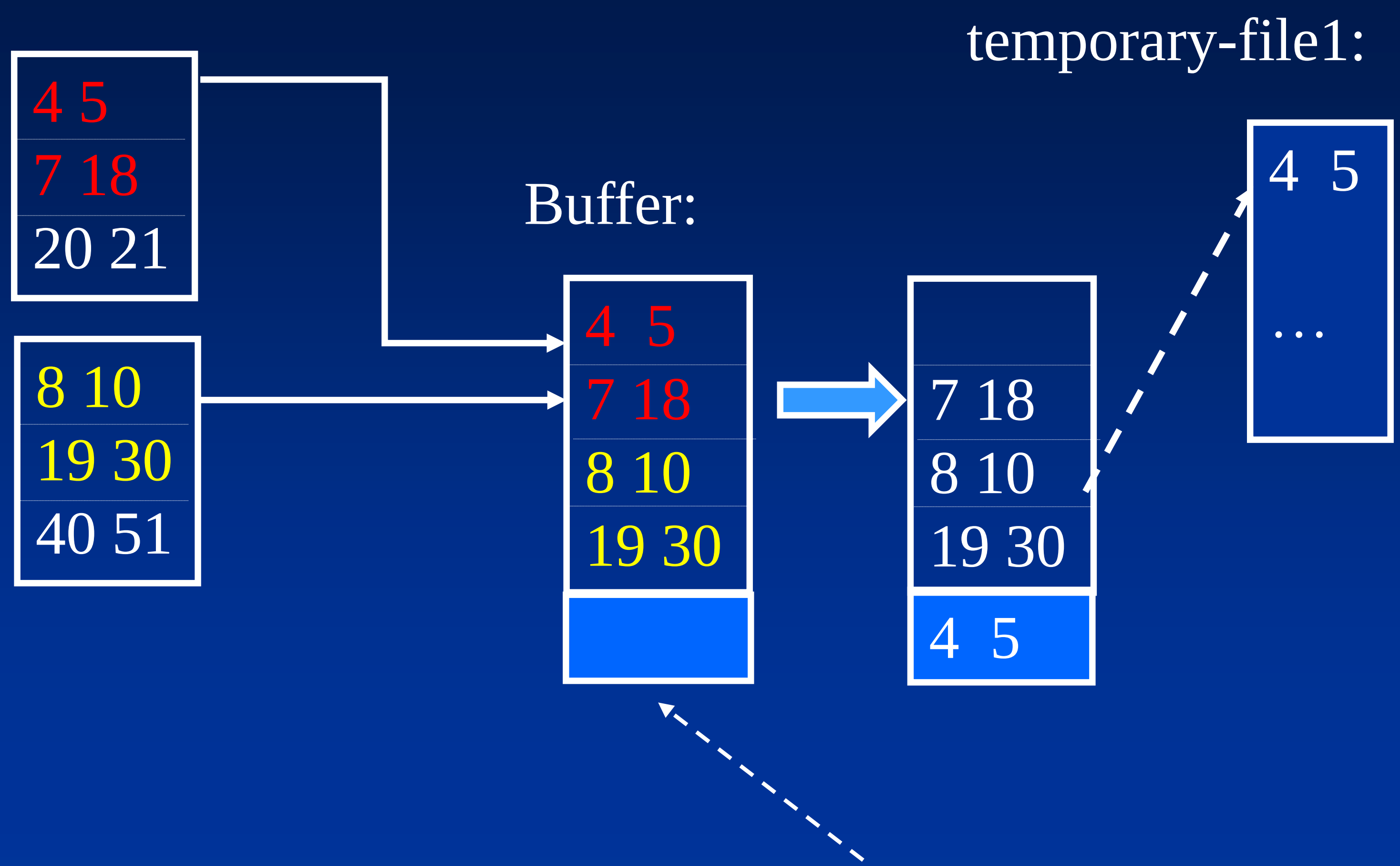

final file:
4 5
7 8
10 18
19 20
21 30
40 51
Buffer:
4 5
6 9
6 9
4 5
4 5
…
6 9
11 12
13 15
16 17
merging phase



If the size of Buffer is *k* blocks, then we can merge *k* – 1 runs each time.



If the size of Buffer is *k* blocks, then we can merge *k* – 1 runs each time.



If the size of Buffer is *k* blocks, then we can merge *k* – 1 runs each time.

- **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
      using a multiple level index to retrieve multiple records

- **Basic algorithms**
  - Using a primary index to retrieve multiple records

    If the selection condition is >, >=, <, <= on a key field with a primary index, use the index to find the record satisfying the corresponding equality condition (DNUMBER = 5, in $\sigma_{DNUMBER>5}$(DEPARTMENT)), then retrieve all subsequent records in the ordered file.

Basic algorithms

- Using a clustering index to retrieve multiple records

  If the selection condition involves an equality comparison on a non-key attribute with a clustering index (for example, DNO = 2 in $\sigma_{DNO=2}$ (EMPLOYEE)), use the index to retrieve all the records satisfying the condition.

- **Basic algorithms**
  - Searching methods for complex selection

    Conjunctive selection using an individual index

    If an attribute involved in any single simple condition in the conjunctive has an access path that permits one of the methods discussed above, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.

$$\sigma_{DNO=1 \wedge salary > 50000}(EMPLOYEE)$$

- **Basic algorithms**
  - Searching methods for complex selection

    Conjunctive selection using a composite index

    If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined fields - for example, if an index has been created on the composite key (SSN value and PNO value) of the WORKS_ON file - we can use the index directly.

$$\sigma_{SSN=123456789 \wedge PNO = 3}(WORKS\_ON)$$

- **Basic algorithms**
  - Searching methods for complex selection

    Conjunctive selection by intersection of record pointers

    - Secondary indexes (indexes on any *nonordering* field of a file, which is not a key) are available on more than one of the fields
    - The indexes include record pointers (rather than block pointers)
    - Each index can be used to retrieve the set of record pointers that satisfy the individual condition.
    - The intersection of these sets of records pointers gives the record pointers that satisfy the conjunctive condition.

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

$$R \bowtie_{A=B} S$$

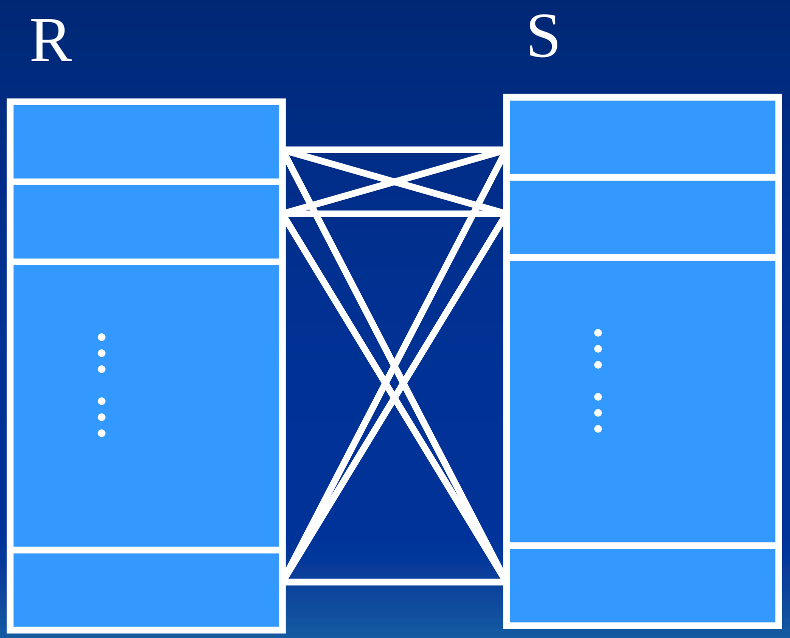
Example:

$$\text{(OP6): EMPLOYEE} \bowtie_{\text{DNO=DNUMBER}} \text{DEPARTMENT}$$

$$\text{(OP7): DEPARTMENT} \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE}$$

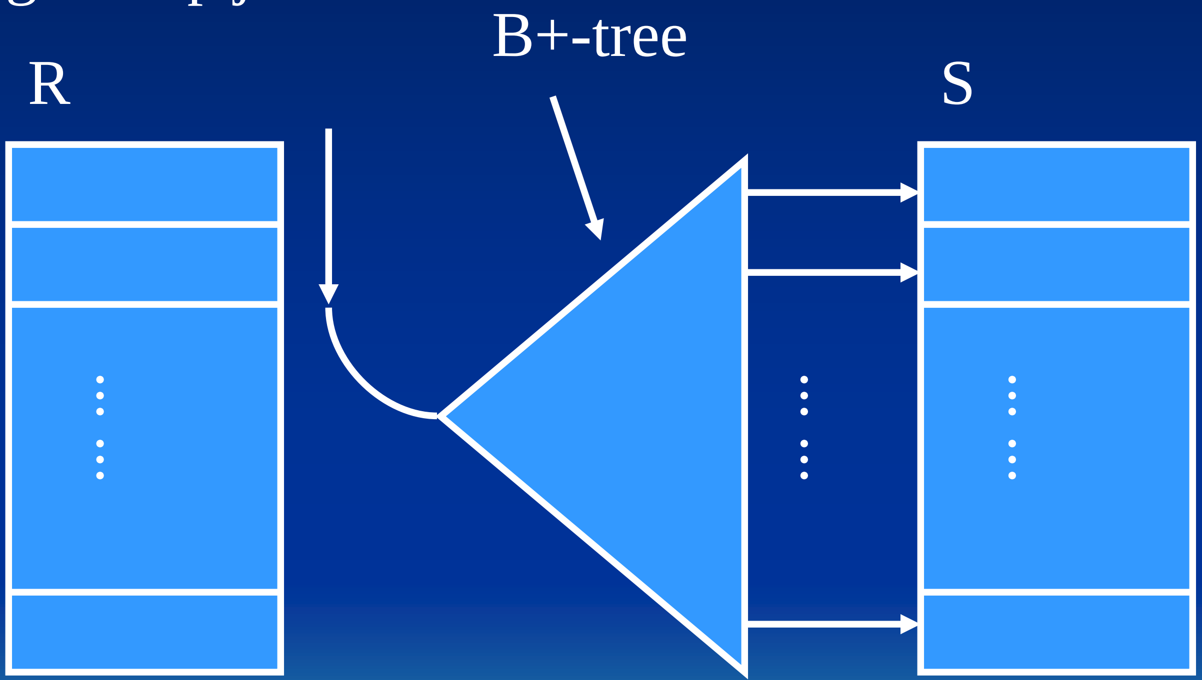
- **Basic algorithms**
  - Methods for implementing JOINs

    Nested-loop join:



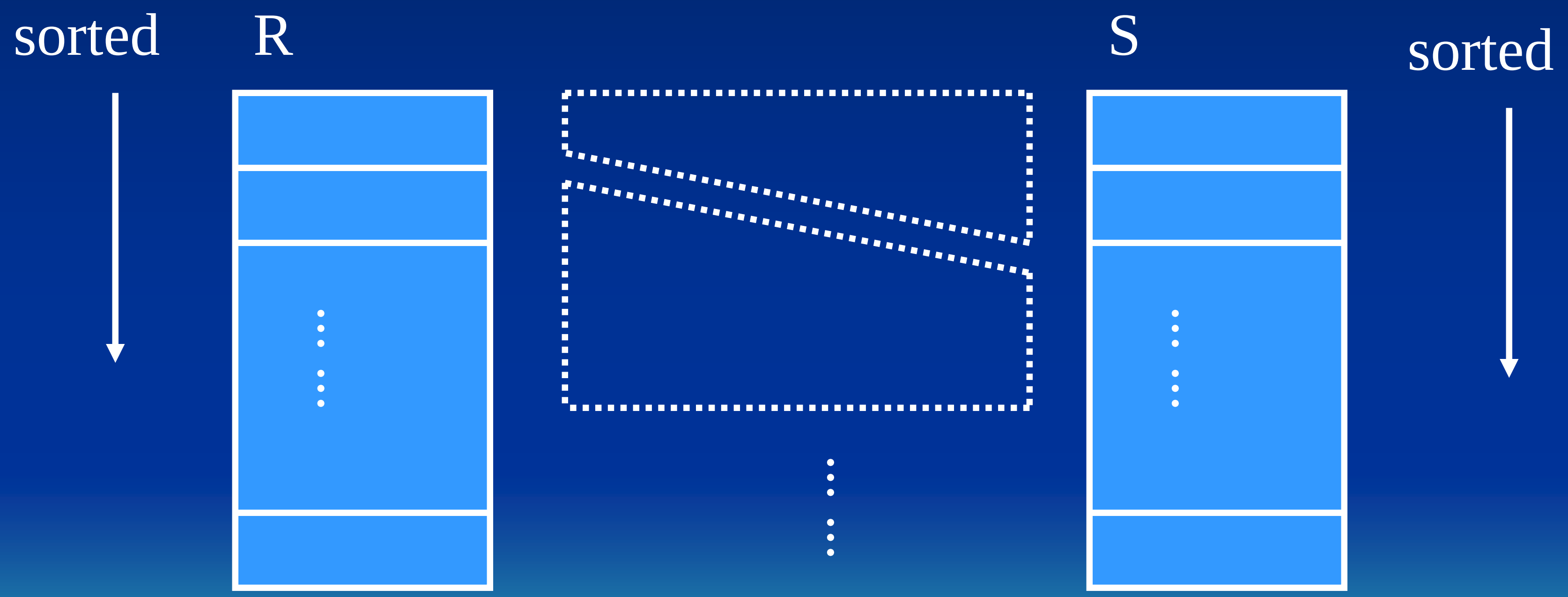
- **Basic algorithms**
  - Methods for implementing JOINs

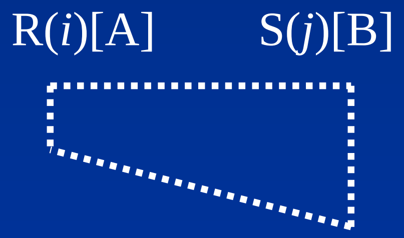
    Single-loop join:



- **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; l ← l + 1;}
                       set k ← k +1;}
              set i ← k, j ← l;}}
```

- **Basic algorithms**
  - PROJECT operation

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

Example:

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

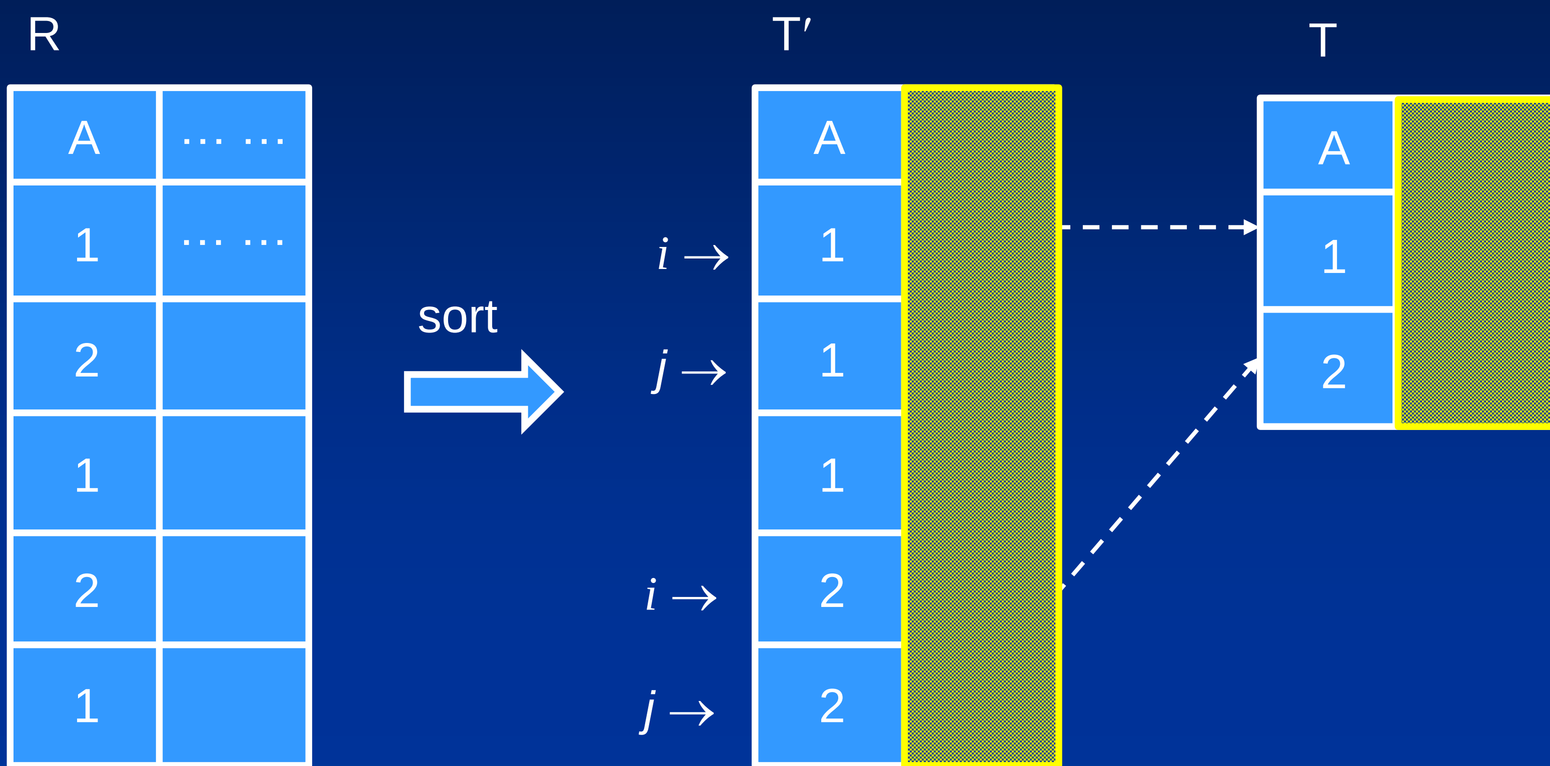
Algorithm:

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

- **Basic algorithms**
  - PROJECT operation

For each tuple t in R, create a tuple t[<Attribute list>] in T′
/*T′ contains the projection result before duplication elimination*/
if <Attribute list> includes a key of R then T ← T′
else { sort the tuples in T′;
set $i \leftarrow 1$, $j \leftarrow 2$;
while $i \leq n$
do { output the tuple T′[i] to T;
while T′[i] = T′[j] and $j \leq n$ do $j \leftarrow j + 1$;
$i \leftarrow j$; $j \leftarrow j + 1$;
}
}

$\pi_A(R)$:



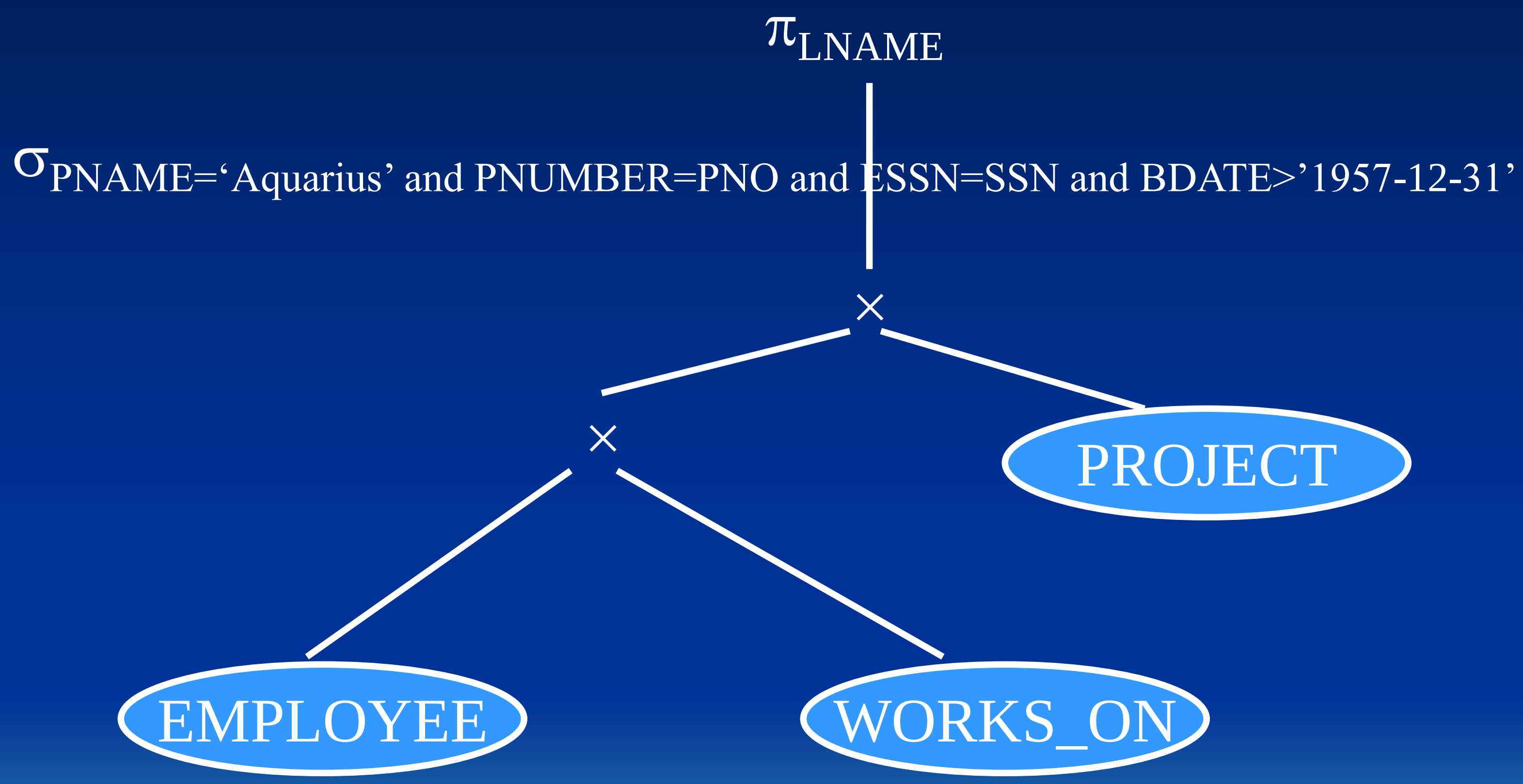
- **Heuristics for query optimization**
  - Query trees and query graph
  - Heuristic optimization of query trees
  - General transformation rules for relational algebra operations
  - Outline of a heuristic algebraic optimization algorithm

- **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.

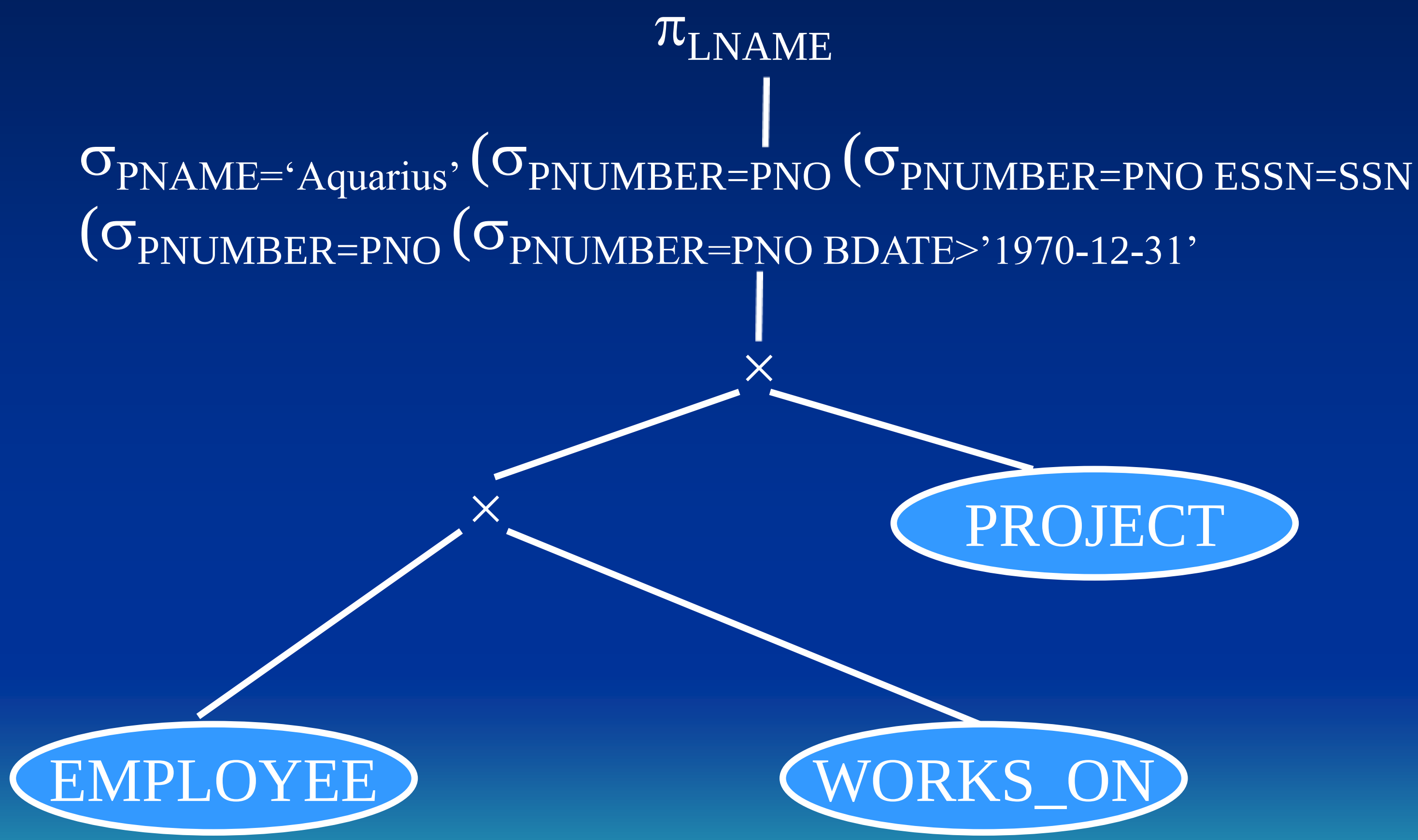
Example:

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

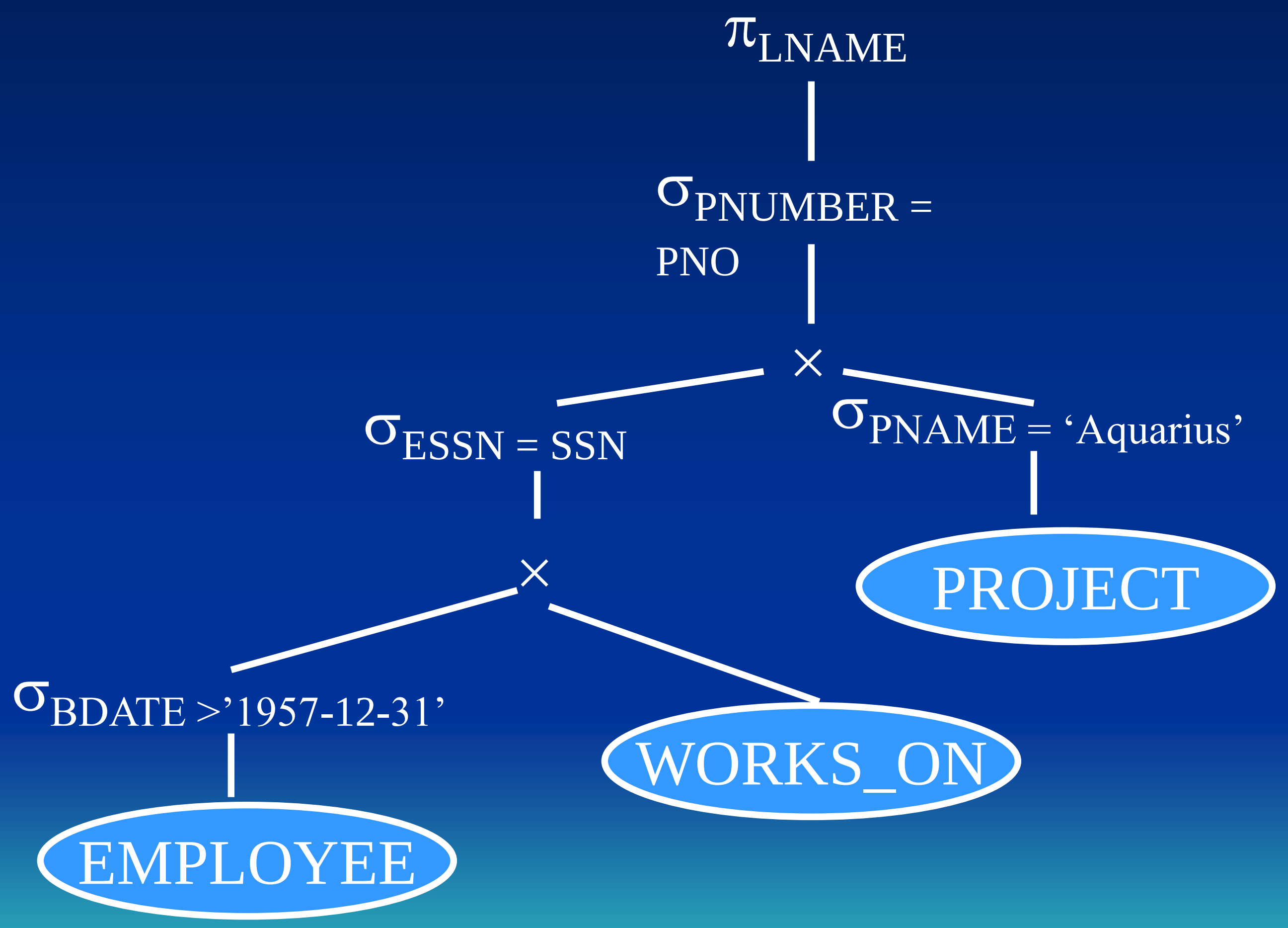
Initial query tree:



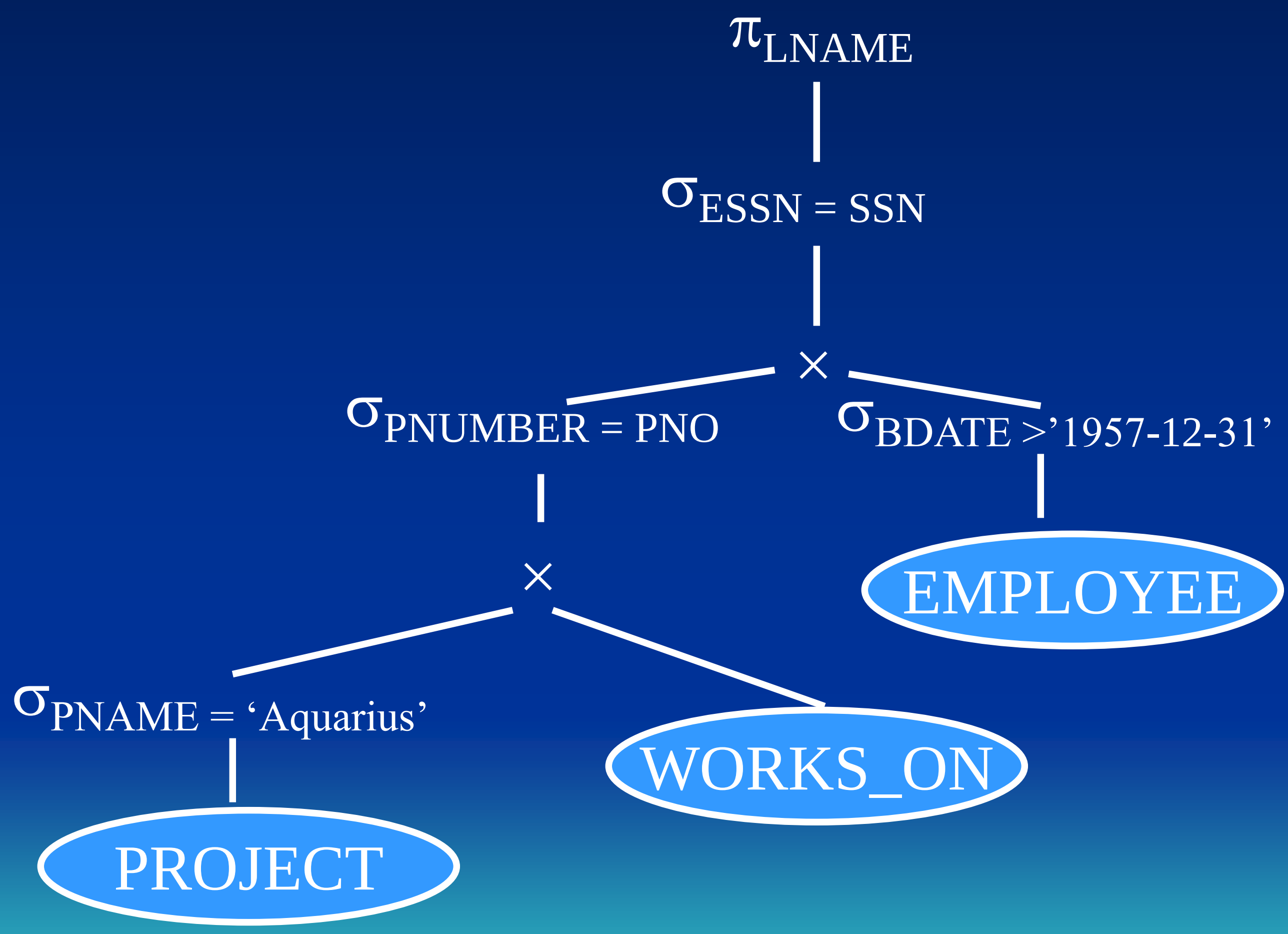
First transformation:



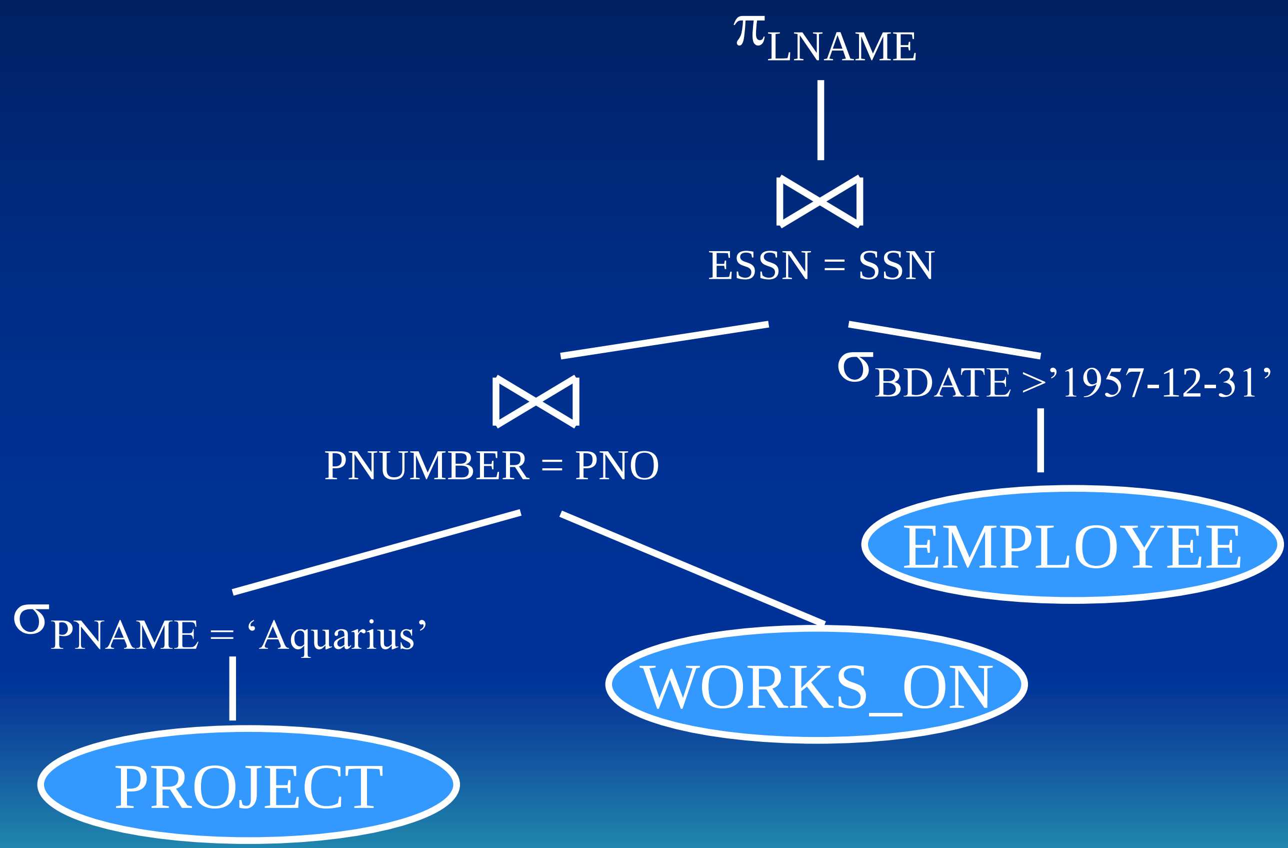
First transformation:



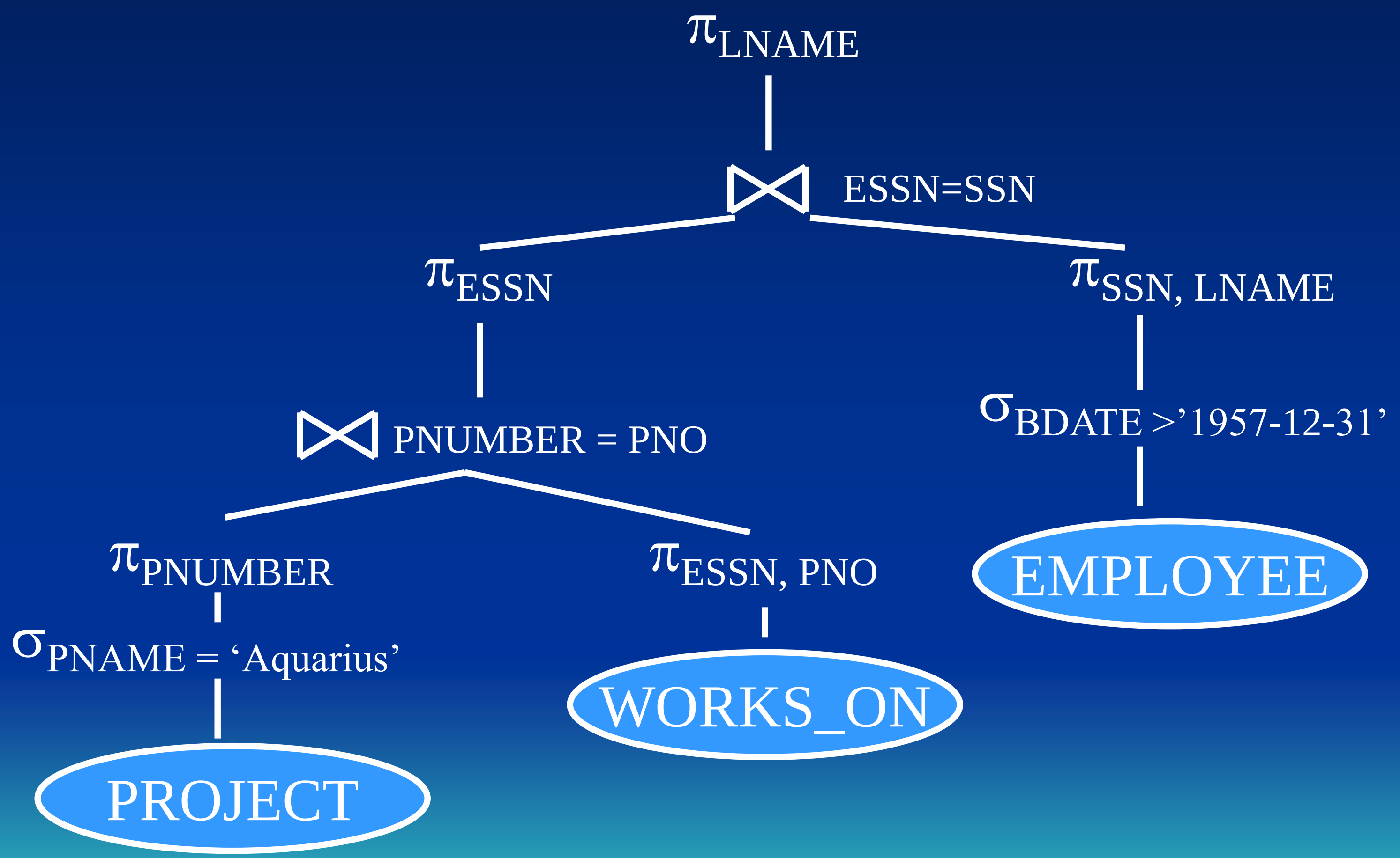
Second transformation:



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:

   $\pi_{list1}(\pi_{list2}(\ldots(\pi_{listn}(R))\ldots)) \equiv \pi_{list1}(R)$

   if $list1 \subseteq list2 \subseteq \ldots \subseteq list\textit{n}$.

- **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 "$\cup$", "$\cap$" 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 $\cup$:

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

$$\pi_L(R \cap S) \neq (\pi_L(R)) \cap (\pi_L(S))?$$

R

A	B
2	3
1	3

S

A	B
1	3
2	1

$\pi_A(R \cap S) =$

A
1

$\pi_A(R) \cap \pi_A(S) =$

A
1
2

$$\pi_L(R - S) \neq \pi_L(R) - \pi_L(S)?$$

12. Converting a $(\sigma, \times)$ sequence into $\bowtie$: If the condition c of a σ that follows a $\times$ corresponds to a join condition, convert then $(\sigma, \times)$ sequence into $\bowtie$ 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)

C1	C2	Not (C1 and C2)
0	0	1
0	1	1
1	0	1
1	1	0

≡

C1	C2	(Not C1) or (notC2)
0	0	1
0	1	1
1	0	1
1	1	0


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

ACID principles:

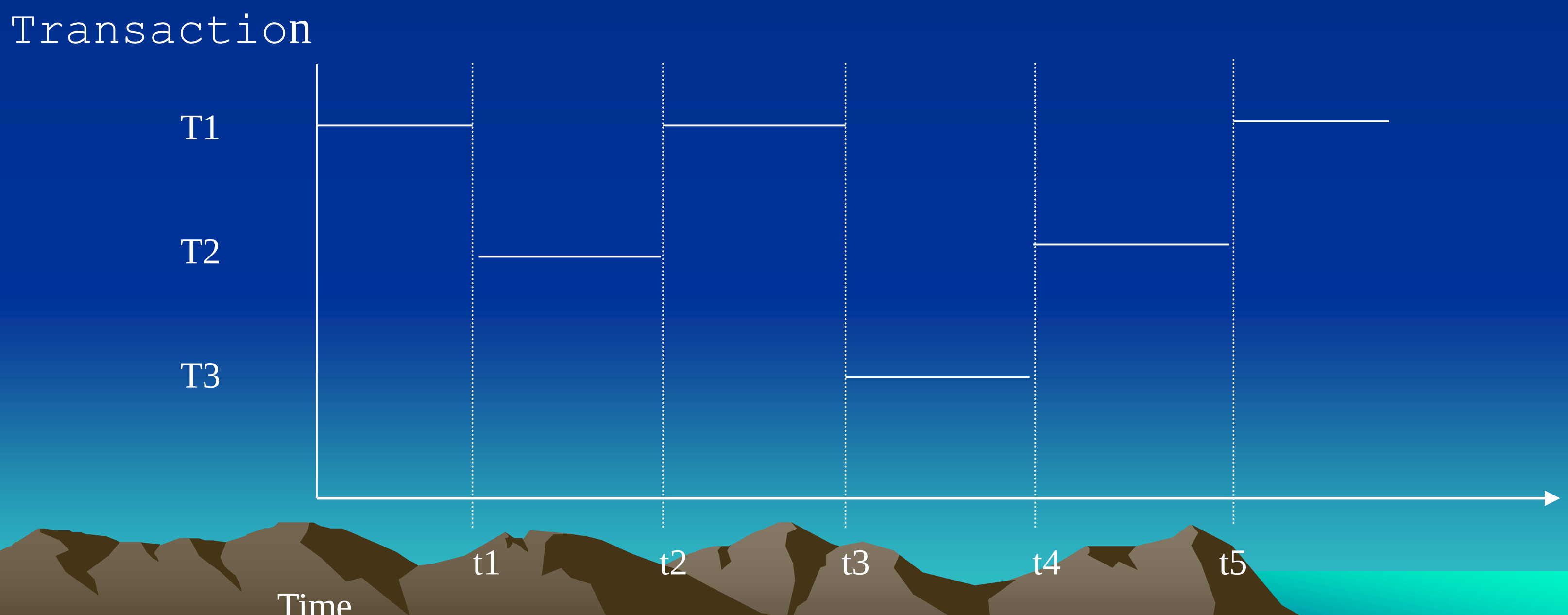
To generate faith in the computing system, a transaction will have the **ACID** properties:

- 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

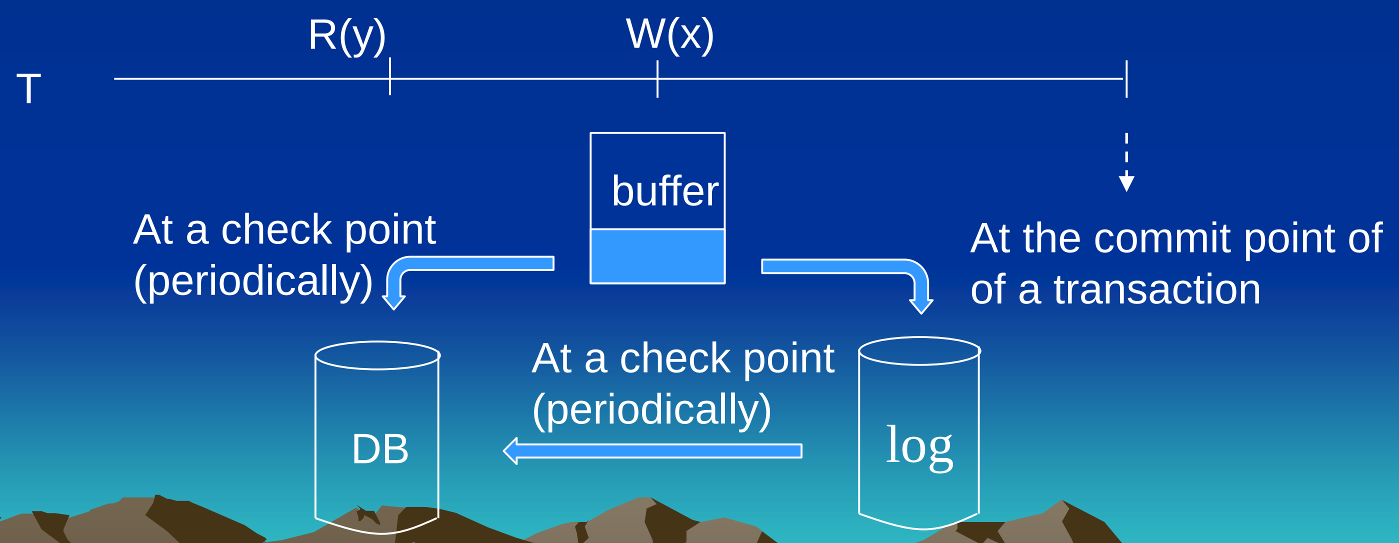
1.[start_transaction, T] Indicates that transaction T has started execution.
2.[write_item, T, X, old_value, new_value] Indicates that transaction T has changed the value of database item X from old_value to new_value.
3.[Read_item, T, X] Indicates that transaction T has read the value of database item 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.

Commit Point

A transaction has committed when it reaches its Commit Point (when the commit command is explicitly performed).

At this point:

- The DBMS force-writes all changes/updates made by a transaction to the log
- Then the DBMS force-writes a commit record for the transaction



Checkpoint

A DBMS will execute a checkpoint in order to simplify the recovery process. The checkpoints occur periodically, arranged by a DBA (DB Administrator).

At a checkpoint any committed transactions will have their database writes (updates/changes) physically written to the database.
(The changes made by unaccomplished transactions may also be written to the database.)

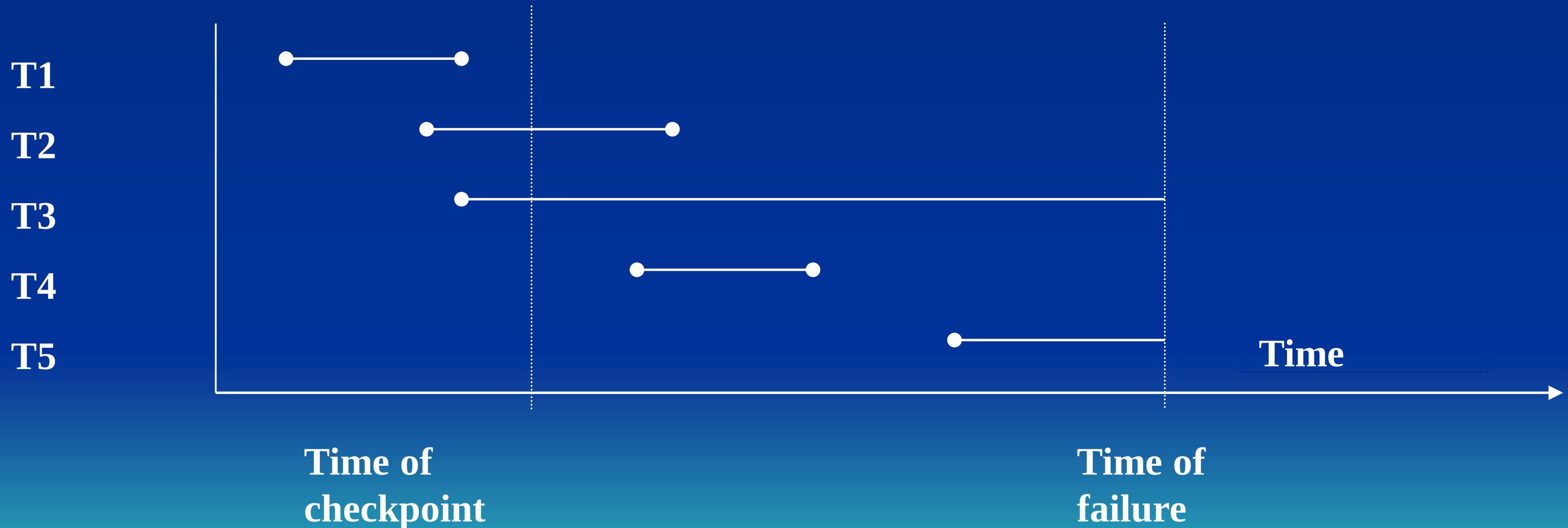
This is a four-step process

- Suspend transaction execution temporarily
- The DBMS force-writes all database changes to the database
- The DBMS writes a checkpoint record to the log and force-writes the log to disk
- Transaction execution is resumed

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

Schedules

Example:

S_1: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$; $W_1(Y)$; C_1; C_2;
(recoverable)

S_2: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$; C_2; A_1;
(non-recoverable)

S_3: $R_1(X)$; $R_2(X)$; $W_1(X)$; $W_2(X)$; A_1; C_2;
(cascadeless)

S_4: $R_1(X)$; $W_1(X)$; $R_2(X)$; $R_1(Y)$; $W_2(X)$;$W_1(Y)$; A_1; A_2;
(recoverable)

S_5: $R_1(X)$; $W_1(X)$; $R_2(Y)$; $W_2(Y)$; C_1; $R_2(X)$; $W_2(X)$; C_2;
(strict)

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

-they have the same operations (coming from the same set of transactions)

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

Binary Locks: data structures

- lock(X) can have one of two values:
  0 or *1*
  unlocked or *locked*
  etc

- We require a Wait Queue where we keep track of suspended transactions

Lock Table

item	lock	trx_id
X	1	1
Y	1	2

Wait Queue

item	transaction
X	2
Y	3

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: data structures

- For any data item X, lock(X) can have one of three values:
  read-locked, *write-locked*, *unlocked*
- For any data item X, we need a counter (no_of_readers) to know when all "readers" have relinquished access to X
- We require a Wait Queue where we keep track of suspended transactions

Lock Table

item	lock	no_of_readers	trx_ids
X	1	2	{1, 2}

Wait Queue

item	transaction
X	3

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) or write_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

Shared and Exclusive Locks (2PL)

Figure 18.4

T1'	T2'
read_lock(Y)	**read_lock**(X)
read_item(Y)	read_item(X)
write_lock(X)	**write_lock**(Y)
unlock(Y)	**unlock**(X)
read_item(X)	read_item(Y)
X:=X+Y	Y:=X+Y
write_item(X)	write_item(Y)
unlock(X)	**unlock**(Y)

These transactions obey the 2PL protocol

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
- undoing a transaction can be efficiently conducted.

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.

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

deadlock

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.