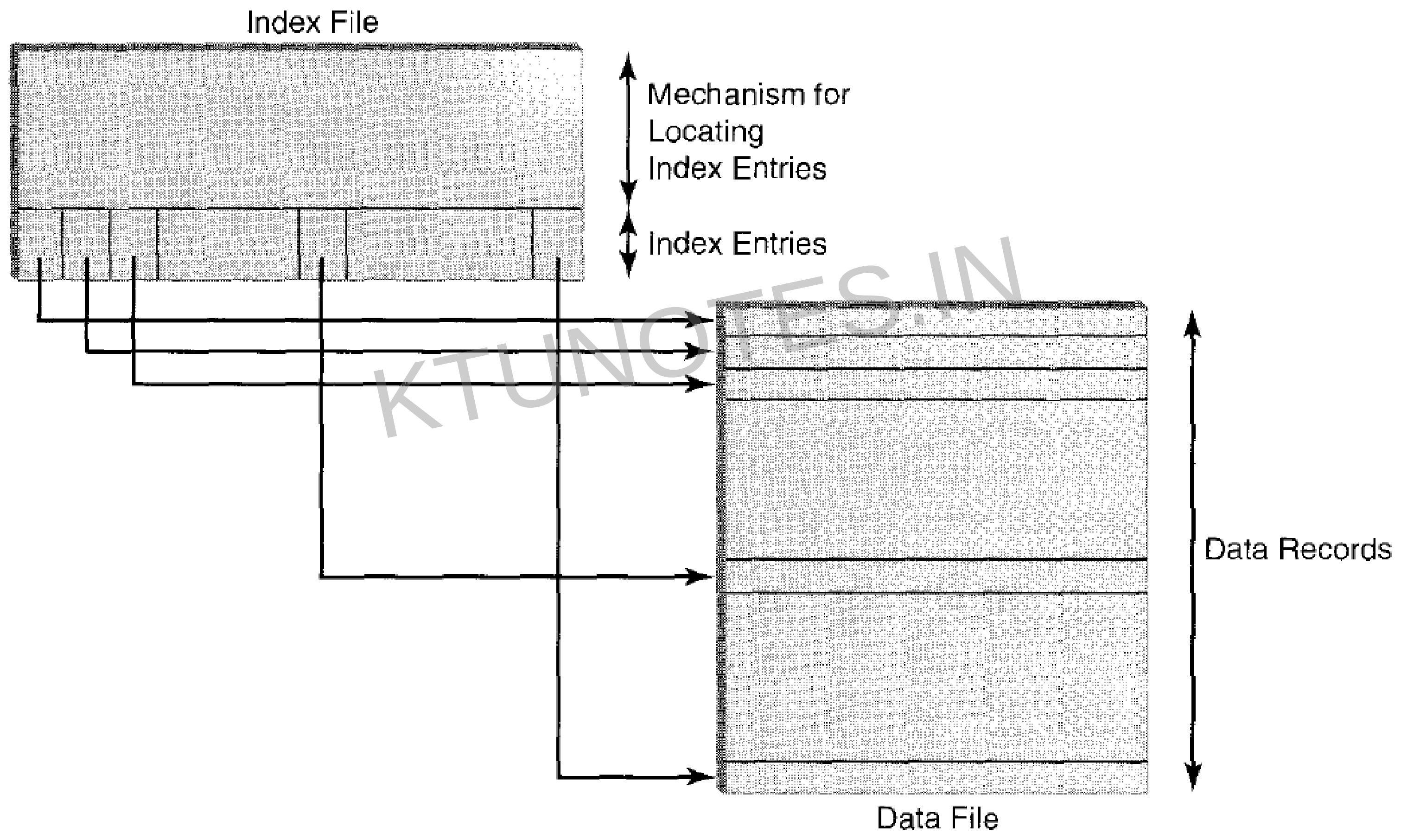
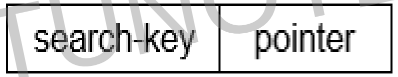
Indexing

- Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done.
- An index on a database table provides a convenient mechanism for locating a row (data record) without scanning the entire table and thus greatly reduces the time it takes to process a query.
- The property to be located is a column (or columns) of the indexed table called a search key.
- Based on a ***search key****:* rows having a particular value for the search key attributes can be quickly located
- Don’t confuse candidate key with search key:
- Candidate key: *set* of attributes; *guarantees* uniqueness
- Search key: *sequence* of attributes; *does not guarantee* uniqueness – just used for search


Index File
Mechanism for Locating Index Entries
Index Entries
Data Records
Data File

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form



- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

- CREATE INDEX TRANSGRD ON TRANSCRIPT (Grade)
- creates an index named TRANSGRD on the table TRANSCRIPT with Grade as a search key.
- Examples
- DROP INDEX CourseTran;
- CREATE INDEX CourseTran ON Transcript (CourseId);
- DROP INDEX DeptProf;
- CREATE INDEX DeptProf ON Professor (DeptId);

Advantages & Disadvantages

- Stores and organizes data into computer files.
- Makes it easier to find and access data at any given time.
- It is a data structure that is added to a file to provide faster access to the data.
- It reduces the number of blocks that the DBMS has to check.

- disadvantage of using index
- index needs to be updated periodically for insertion or deletion of records in the main table.

Structure of index

- An index is a small table having only two columns.
- The first column contains a copy of the primary or candidate key of a table and
- the second column contains a set of pointers holding the address of the disk block where that particular key value can be found.

- If the indexes are sorted, then it is called as ordered indices.

Clustered versus Unclustered Indices

- A sorted index is clustered if the index entries and the data records are sorted on the same search key; otherwise, it is said to be unclustered.
- *Clustered index*: index entries and rows are ordered in the same way
- EX:-It is like a dictionary, where all words are sorted in alphabetical order in the entire book.
- Unclustered (secondary) index: index entries and rows are not ordered in the same way
- Ex:- It is like an index in the last pages of a book, where keywords are sorted and contain the page number to the material of the book for faster reference.

- Indexing can be of the following types −
- Primary Index − Primary index is defined on an ordered data file. The data file is ordered on a key field. The key field is generally the primary key of the relation.
- Secondary Index − Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non key with duplicate values.
- Clustering Index − Clustering index is defined on an ordered data file. The data file is ordered on a non key field.
- Ordered Indexing is of two types −
- Dense Index
- Sparse Index

- **Primary Index**
- It is an ordered file whose records are of fixed length with two fields.
- Only based on the primary key.
- The total number of entries in the index is the same as the number of disk blocks in the ordered data file.
- Primary index is a king of non dense (sparse) index.
- There may be at most one primary index for a file
- Needs less storage space.

- **Secondary index**
- It provides a secondary means of accessing a file for which some primary access already exists.
- May be based on candidate key or secondary key.
- It has a large number entries due to duplication.
- Secondary index is a kind of dense index.
- There may be more than one secondary indexes for the same file.
- Needs more storage space and longer search time.

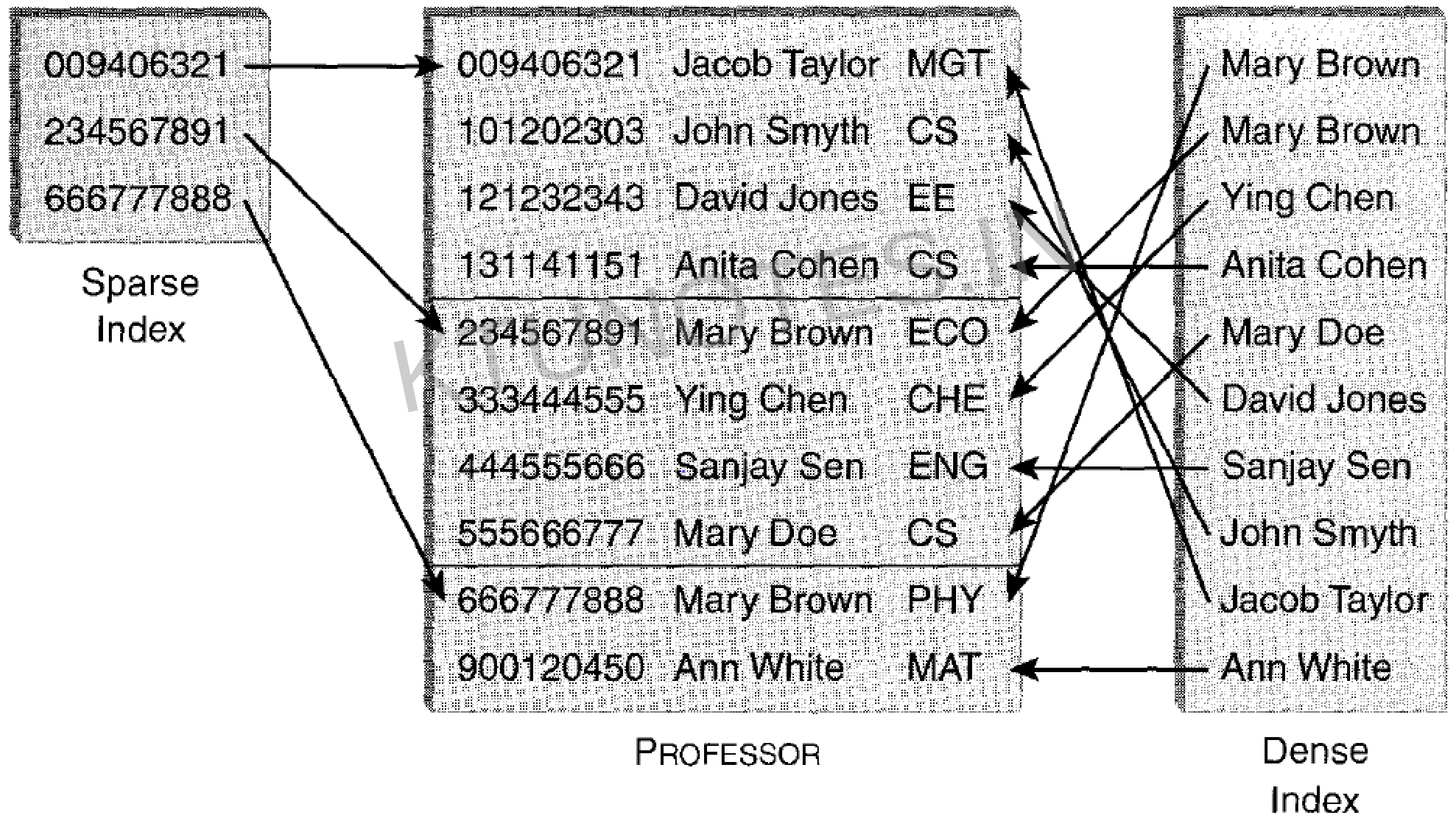
Single-Level Ordered Indexes

- Any field (or combination of fields) can be used to create an index,

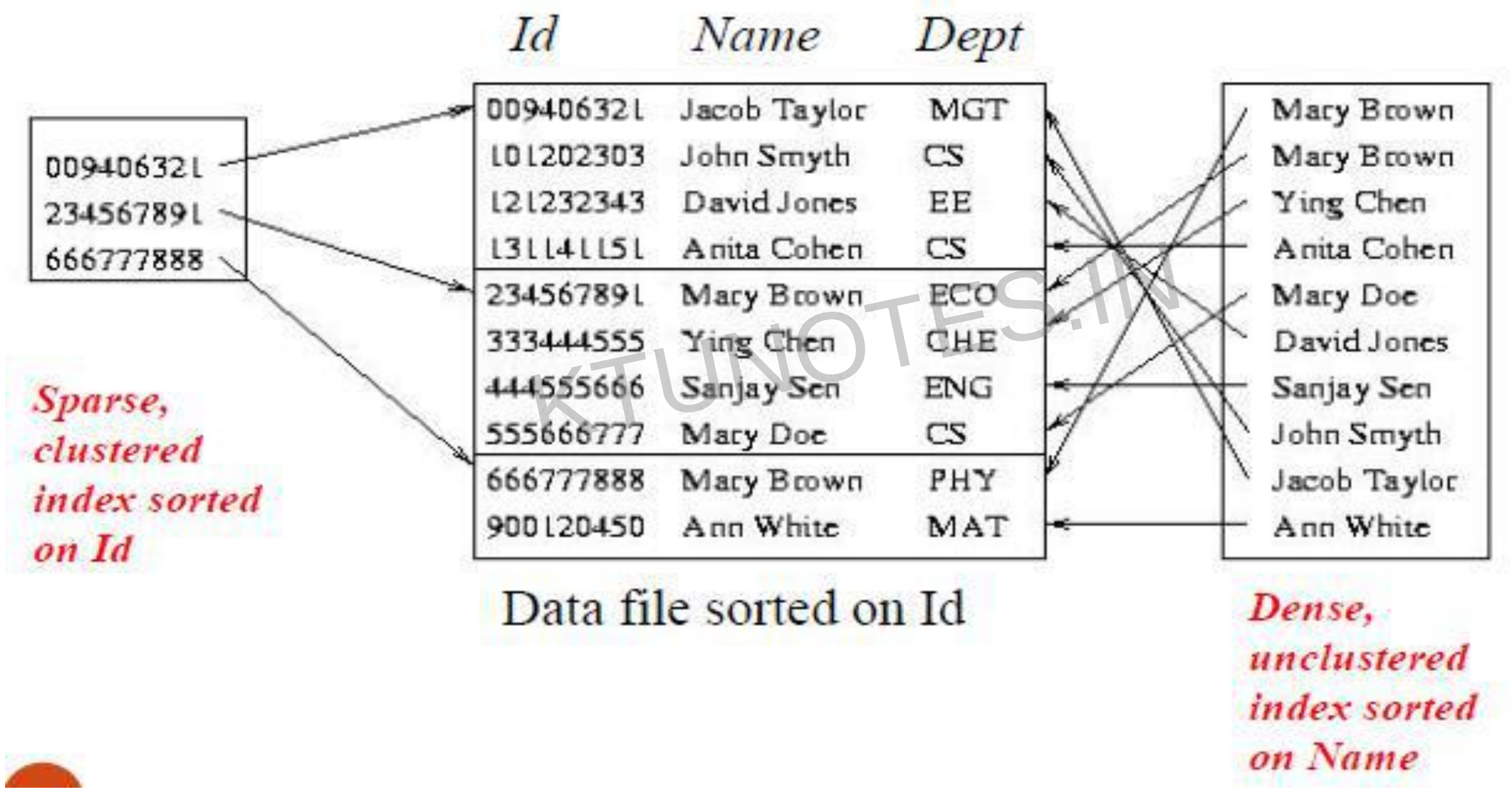
- but there will be different index types
  - depending on whether the field is a key (unique),
  - and whether the main file is sorted by it or not.
  - There can be multiple indexes on one file.

Primary Index

- In primary index, there is a one to one relationship between the entries in the index table and the records in the main table.
- Primary index can be of two types:
- Dense index
- - one index record for every search-key value
- - faster access but higher overhead
- Sparse index
- - index records for only some of the records
- - less faster but less overhead



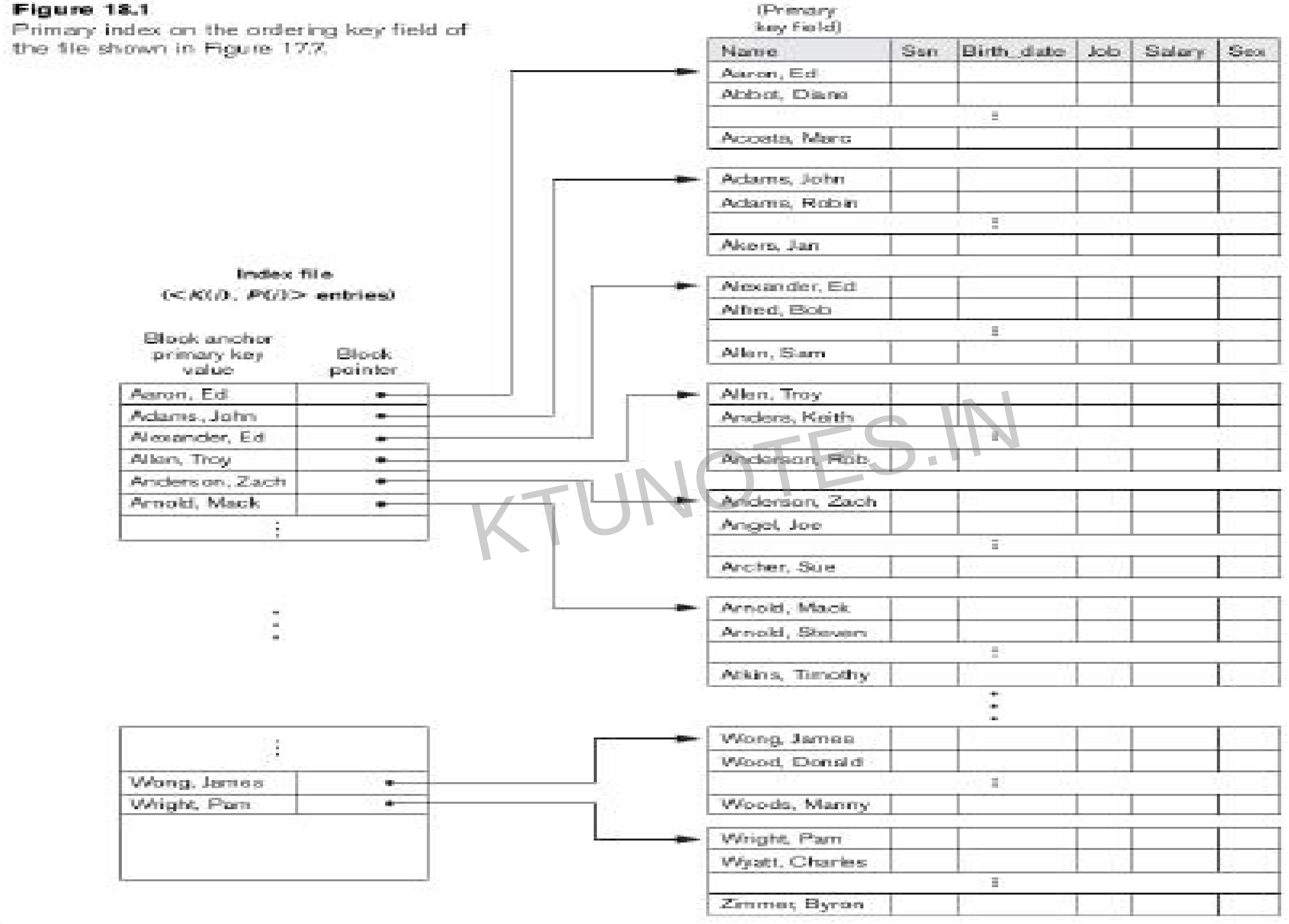
009406321
234567891
666777888
Sparse Index
009406321 Jacob Taylor MGT
101202303 John Smyth CS
121232343 David Jones EE
131141151 Anita Cohen CS
234567891 Mary Brown ECO
333444555 Ying Chen CHE
444555666 Sanjay Sen ENG
555666777 Mary Doe CS
666777888 Mary Brown PHY
900120450 Ann White MAT
PROFESSOR
Mary Brown
Mary Brown
Ying Chen
Anita Cohen
Mary Doe
David Jones
Sanjay Sen
John Smyth
Jacob Taylor
Ann White
Dense Index


Id
Name
Dept
009406321
234567891
666777888
009406321 Jacob Taylor MGT
101202303 John Smyth CS
121232343 David Jones EE
131141151 Anita Cohen CS
234567891 Mary Brown ECO
333444555 Ying Chen CHE
444555666 Sanjay Sen ENG
555666777 Mary Doe CS
666777888 Mary Brown PHY
900120450 Ann White MAT
Mary Brown
Mary Brown
Ying Chen
Anita Cohen
Mary Doe
David Jones
Sanjay Sen
John Smyth
Jacob Taylor
Ann White
Sparse, clustered index sorted on Id
Data file sorted on Id
Dense, unclustered index sorted on Name

Primary Indexes

- An index on the ordering key (often primary key) of a sorted file. The index file is a table of <key, block_pointer> pairs, also sorted, one pair for each block of the original file.
- These are called the *index entries* and recap the ordering key of the first record of their pointed-to block.
- The first record of each block is called the *anchor record*.
- To retrieve a record given its ordering key value using the index, the system does a binary search in the index file to find the index entry whose key value is ≤ the goal key's value, then retrieves the pointed-to block from the original file.
- This type of index is called a *sparse index* because it only contains entries for *some* of the records in the original file.
- If it had entries for all records in the original file it would be a *dense index*.

Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.



Example 1.

No of blocks we have to access while performing binary search

Block size (B) 1024 B

Record count (r) 30,000

Record length (R) 100 B fixed size, unspanned

The blocking factor is bfr = $\lfloor (B/R) \rfloor$ = 10 record/block.

The number of blocks needed by the file is b = $\lceil (r/bfr) \rceil$ = 3000 blocks.

A binary search on the data file would need to access approximetly $\lceil \lg 2\ b \rceil$ = 12 blocks.

Now consider a primary index on that file with these parameters:
Ordering key length (V) 9 Bytes
Block pointer length (P) 6 Bytes
Index entry length (Ri) 15 Bytes (9+6)
The blocking factor of the index file is bfri = $\lfloor (B/Ri) \rfloor$ = 68 record/block.
The total number of index entries will be the same as the number of blocks in the main file, or ri = 3000.
Thus the number of blocks needed by the index file is bi = $\lceil (ri/bfri) \rceil$ = 45 blocks.
A binary search on this need access on average only $\lceil \lg bi \rceil$ = 7 blocks.

Clustering Indexes

- If the main file is sorted on a non key field
- In a *clustering index*, the index entries are similar but
- there's one for each value of the clustering field, and its block pointer points to the first of perhaps several blocks that have records with that value of the clustering field.
- Note this is still a sparse index, since there may be (and often will be) multiple records with any value of the cluster field, but only one index entry.

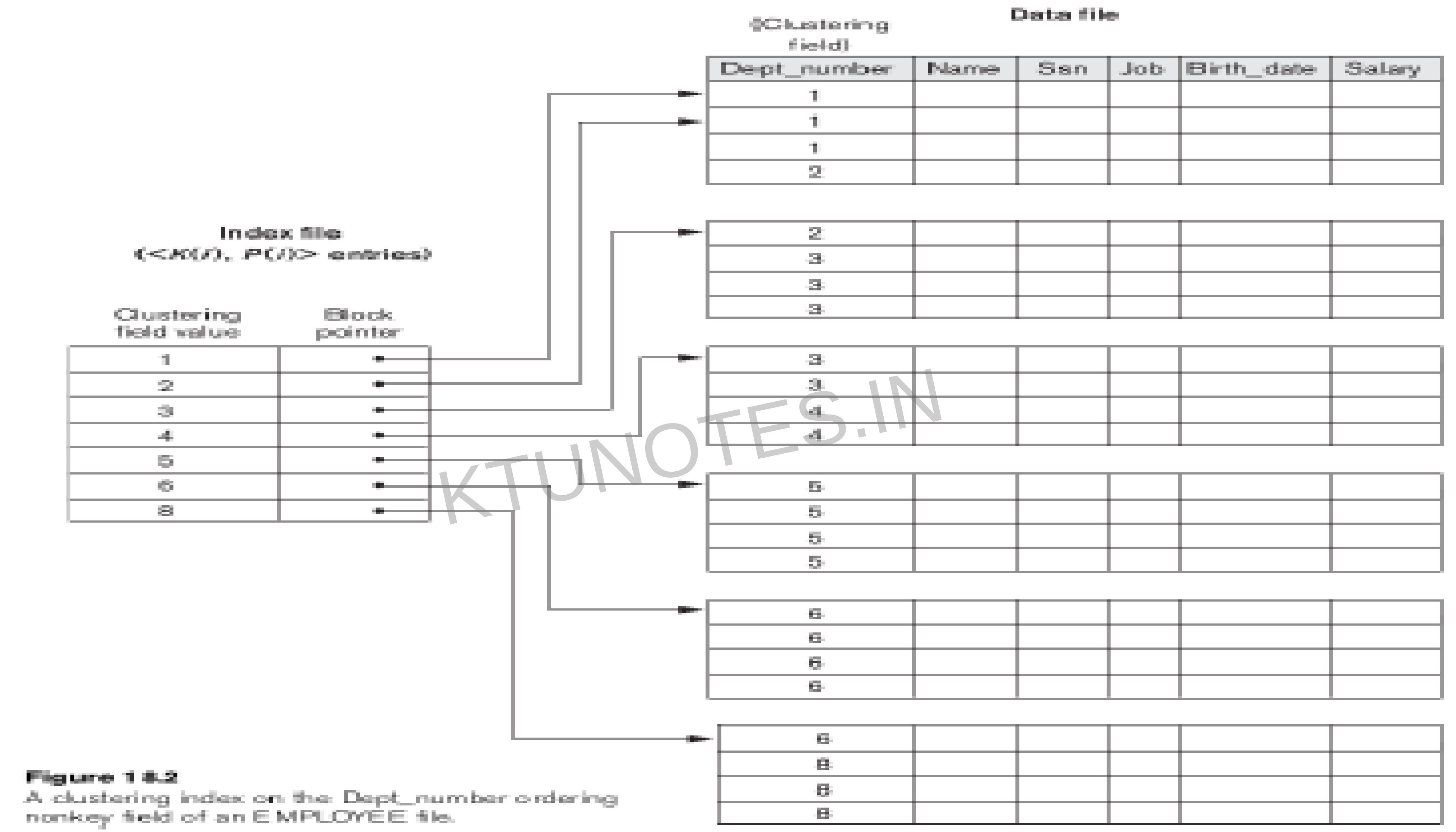
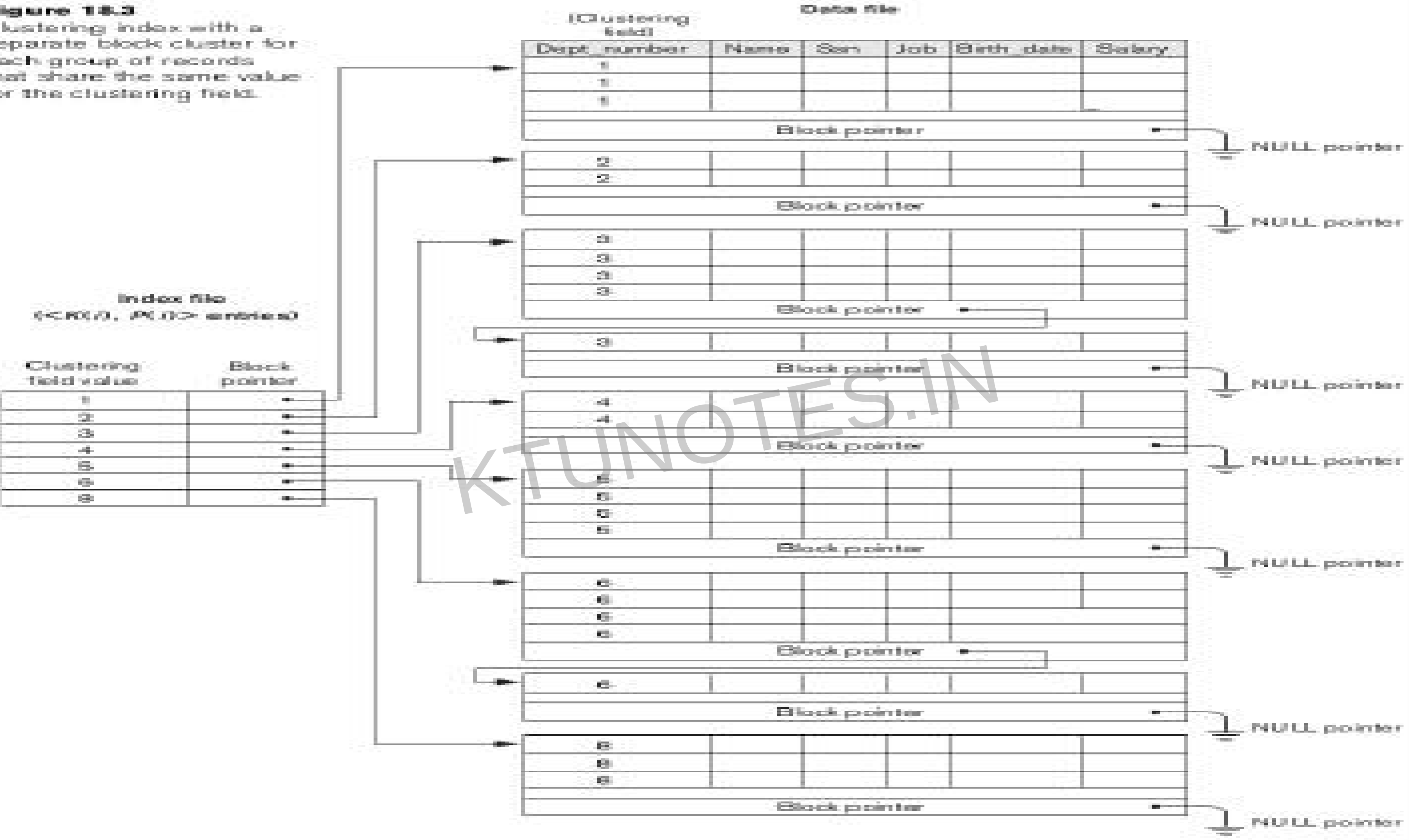


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

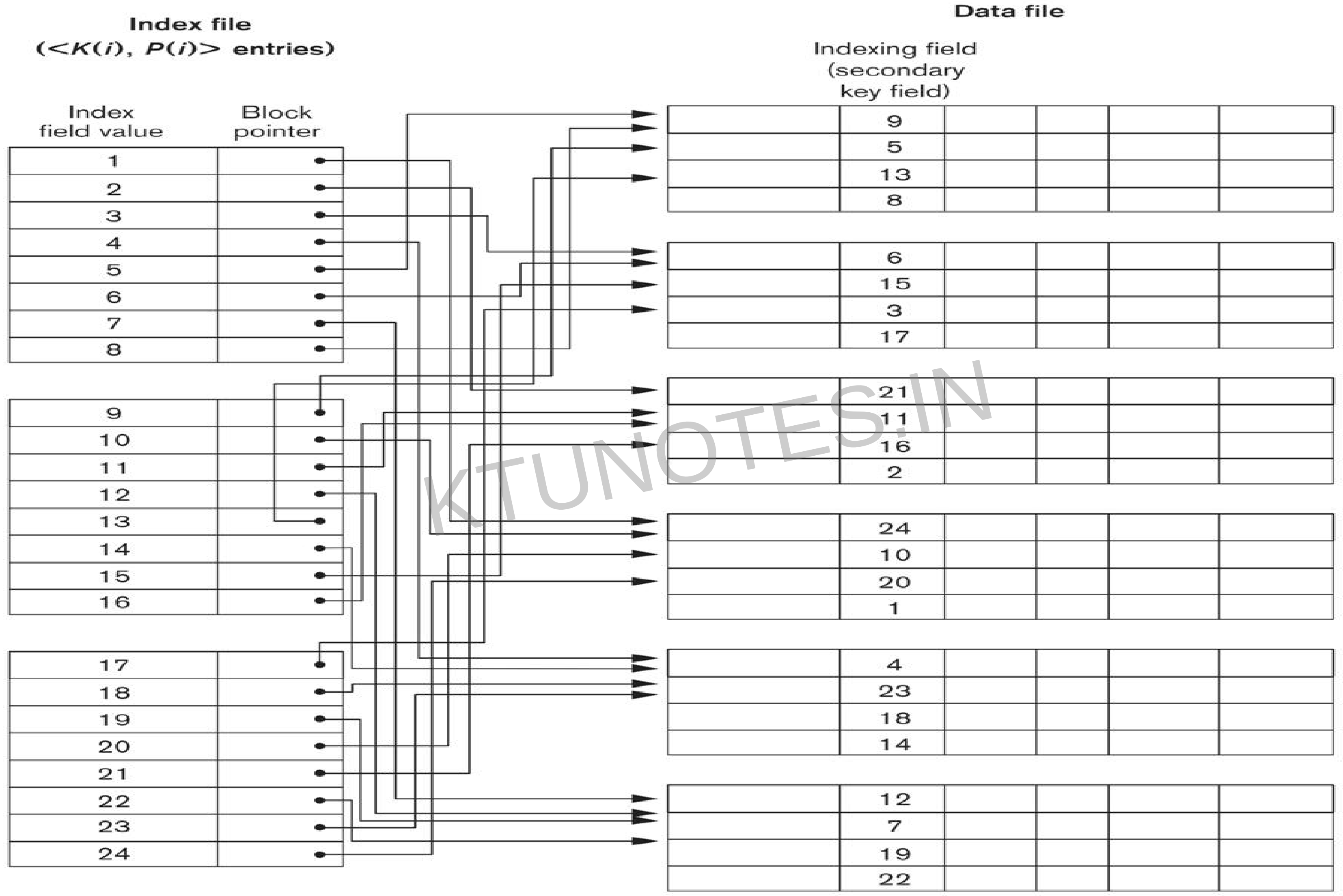
Figure 18.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



Secondary Indexes

- Similar to the other indexes, but the data file is not ordered (or not ordered by this index field).
- The index is dense, with one entry for every record in the data file.
- The index entries are sorted on the index field, with a *block* pointer to the block of the main file containing the corresponding record.
- (Block pointer because, as a key field, there's only one record to be retrieved, thus only one block, and the whole block must be retrieved anyway.)

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



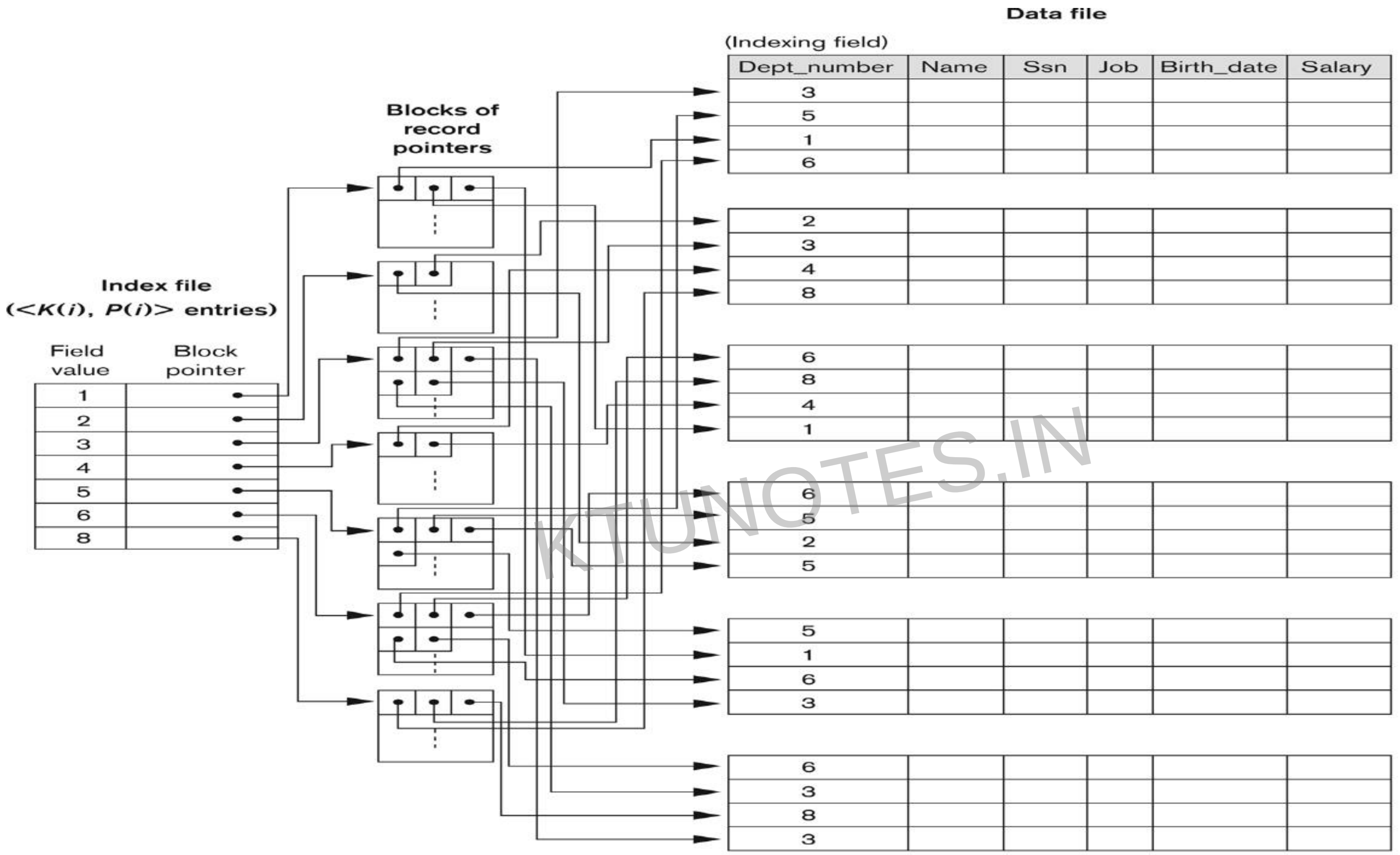


Figure 14.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

TABLE 14.2 PROPERTIES OF INDEX TYPES

TYPE OF INDEX	NUMBER OF (FIRST-LEVEL) INDEX ENTRIES	DENSE OR NONDENSE	BLOCK ANCHORING ON THE DATA FILE
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or Number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.

[b]For option 1.

[c]For options 2 and 3.

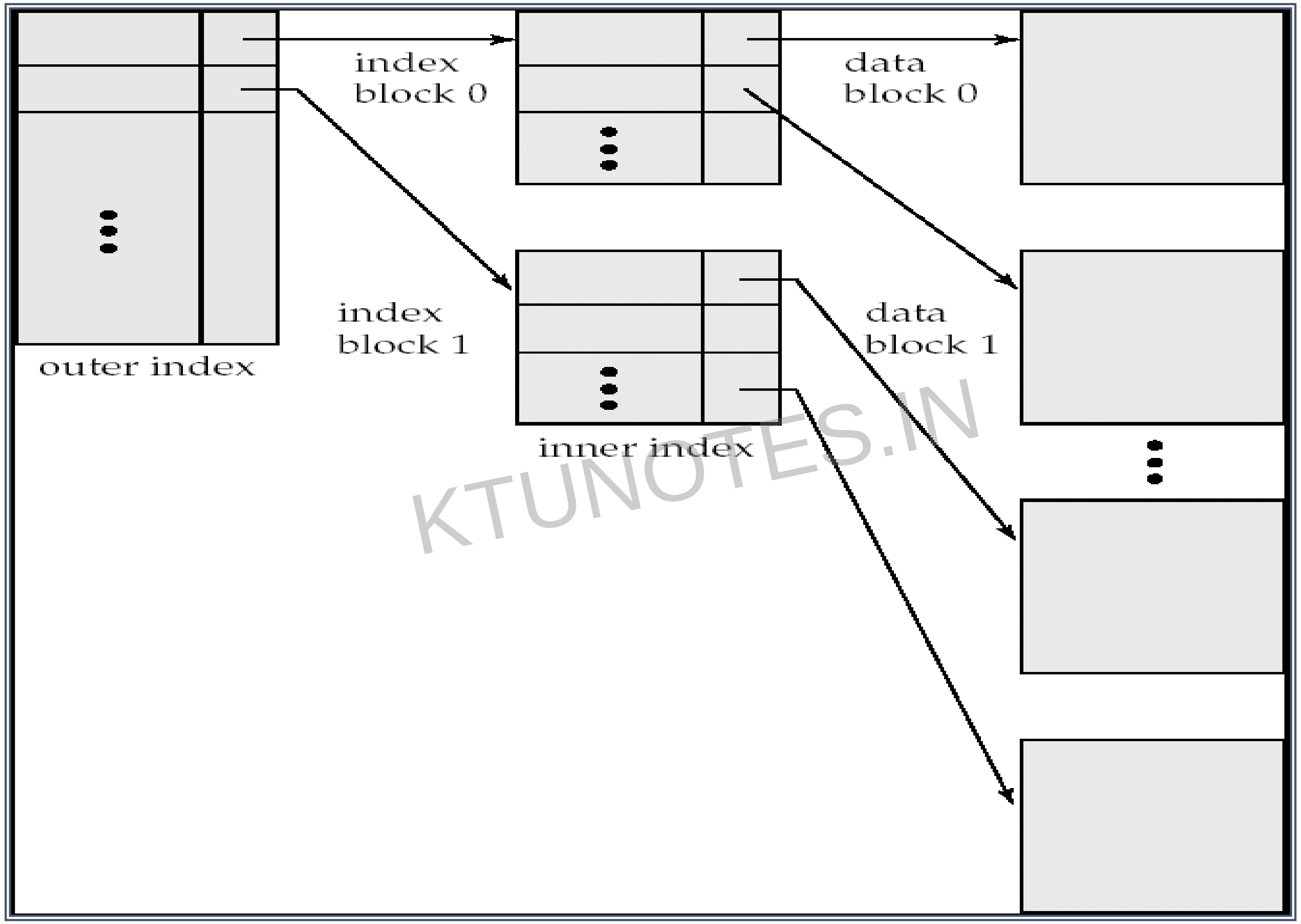
Multi-Level Indexes

- Indices with two or more levels are called multilevel indices.
- Because a single-level index is an ordered file, we can create a primary index *to the index itself*;
  - In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block

Multilevel Indexes

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
    - outer index – a sparse index of primary index
    - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

- If an index is sufficiently small to be kept in main memory search time to find an entry is low.
- If the index occupies b blocks, binary search requires as many as [log2(b)] blocks to be read.


index block 0
data block 0
index block 1
data block 1
outer index
inner index

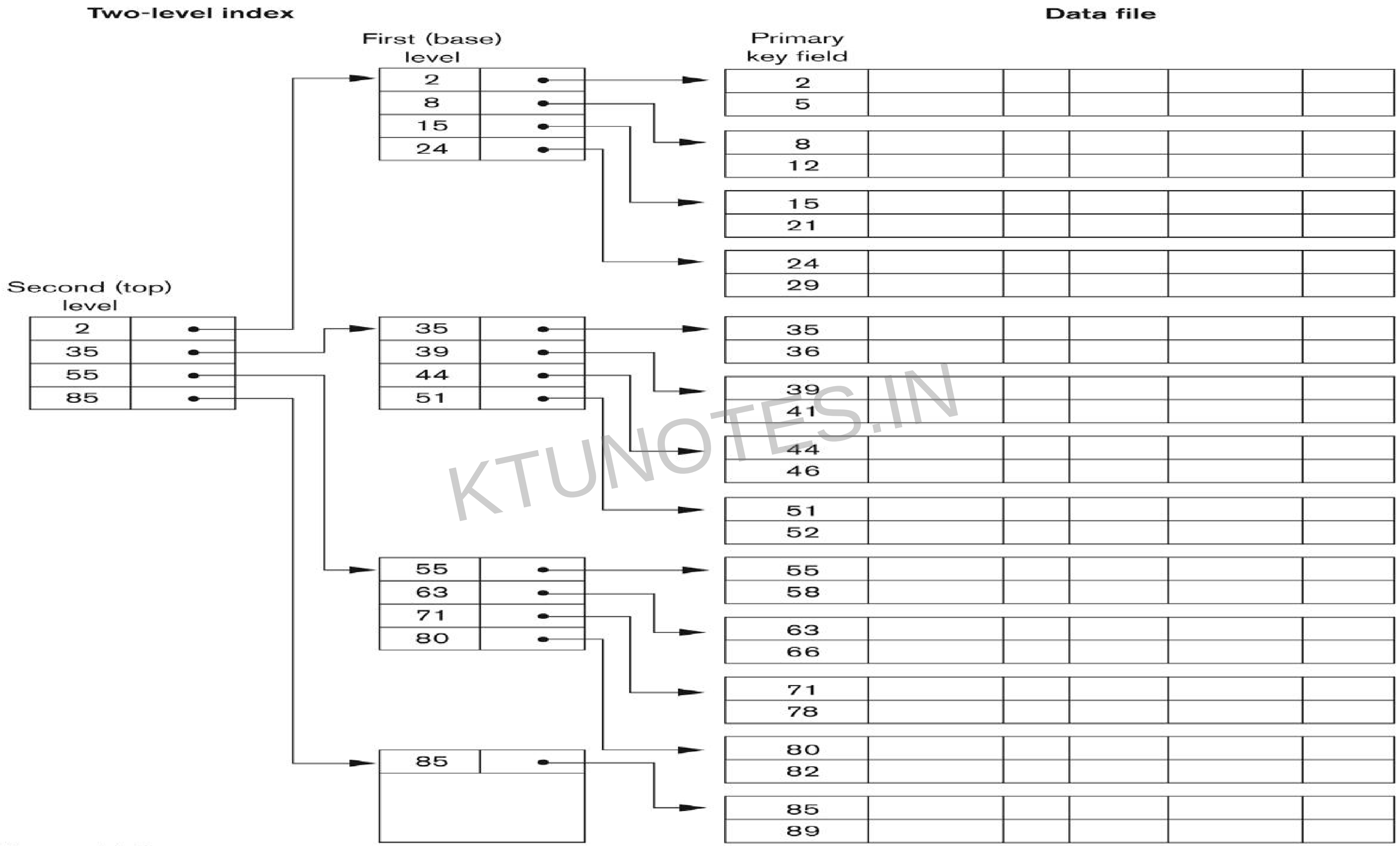


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

- Such a multi-level index is a form of *search tree*
    - However, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*.

- The value bfri is called the fan-out of multilevel indexing (*fo)*
- *Searching a multilevel index requires($\log_{fo}$ bi) block accesses.*
- *Bfri for every level is same bcoz all index entries are the same size each has one field value and one block address.*
- *If 1st level has r1 entries and bfri=fo, then 1st level needs[r1/fo] blocks*
- *Which is number of entries r2 needed*
- *R3=[r2/fo] blocks*
- *T number of index levels*
- *Top index level t=[($\log_{fo}$ r1)]*

Example 3

Data file Block size (B) 1024 B Record count (r) 30,000 Record length (R) 100 B *fixed size, unspanned* Blocking factor (bfr) 10 record/block Block size of file (b) 3000 blocks From Example 2 of secondary index	Primary index Ordering key length (V) 9 B Block pointer length (P) 6 B Index entry length (Ri) 15 B *(9+6)* Blocking factor (bfri) 68 record/block Block size of index (bi) 442blocks	If this is extended into a multilevel index, the index blocking factor is also the fan-out **(bfri = fo = 68)** index size is the b1 size= 442 So the block size of level 2 is: ⌈b1/fo⌉ = ⌈442/68⌉ = 7 blocks. block size of level 2 is: ⌈b2/fo⌉ = ⌈7/68⌉ = 1 block. So level 3 is enough. Level 3 top level of index t=3 t+1=3+1=4 blocks

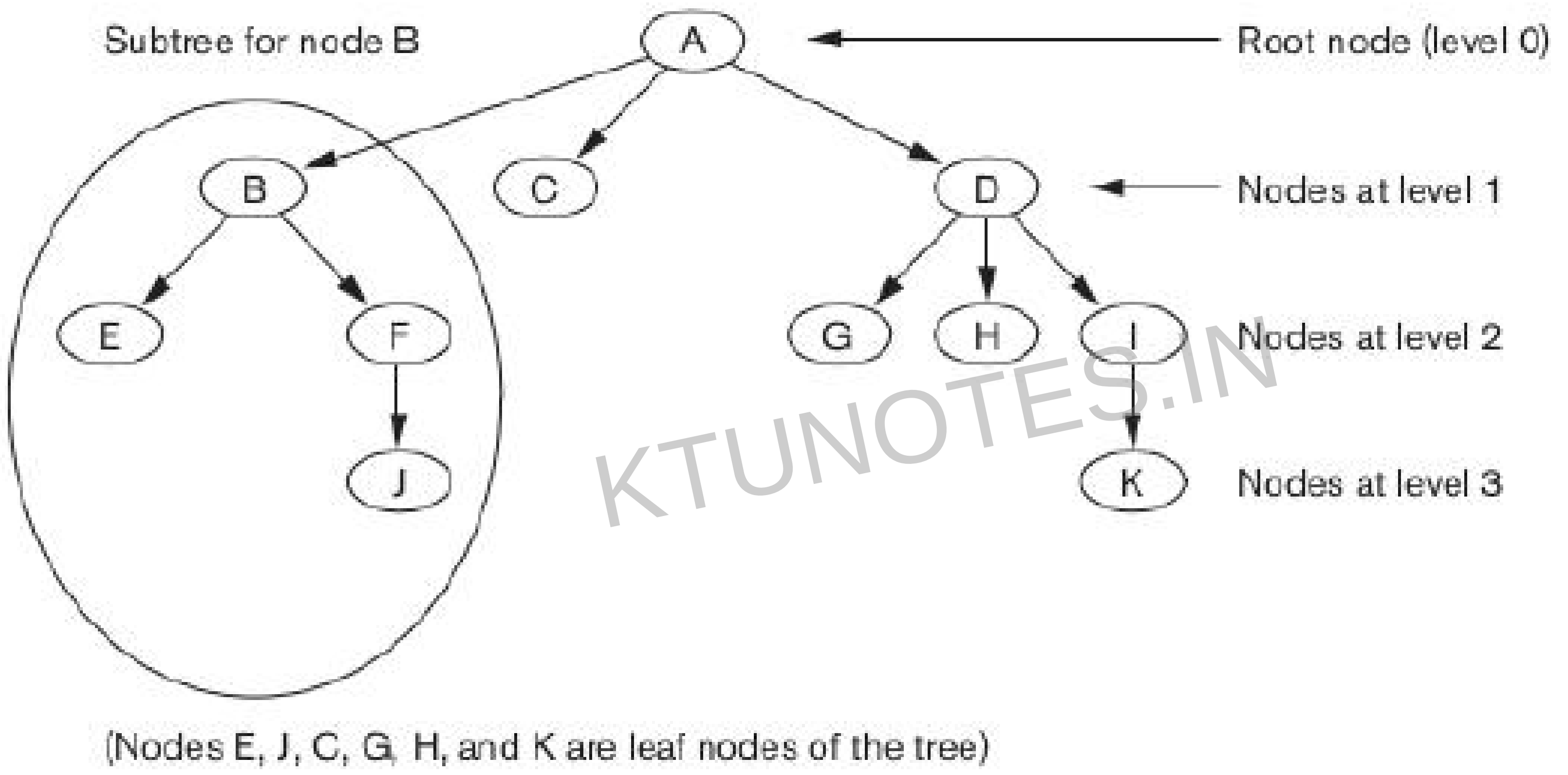
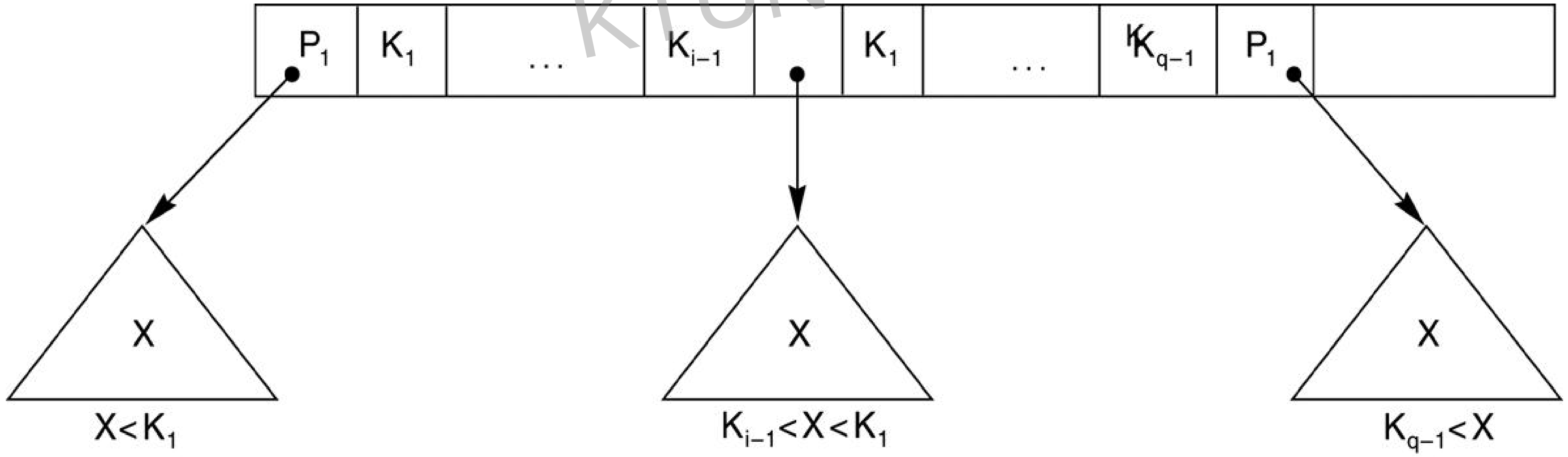


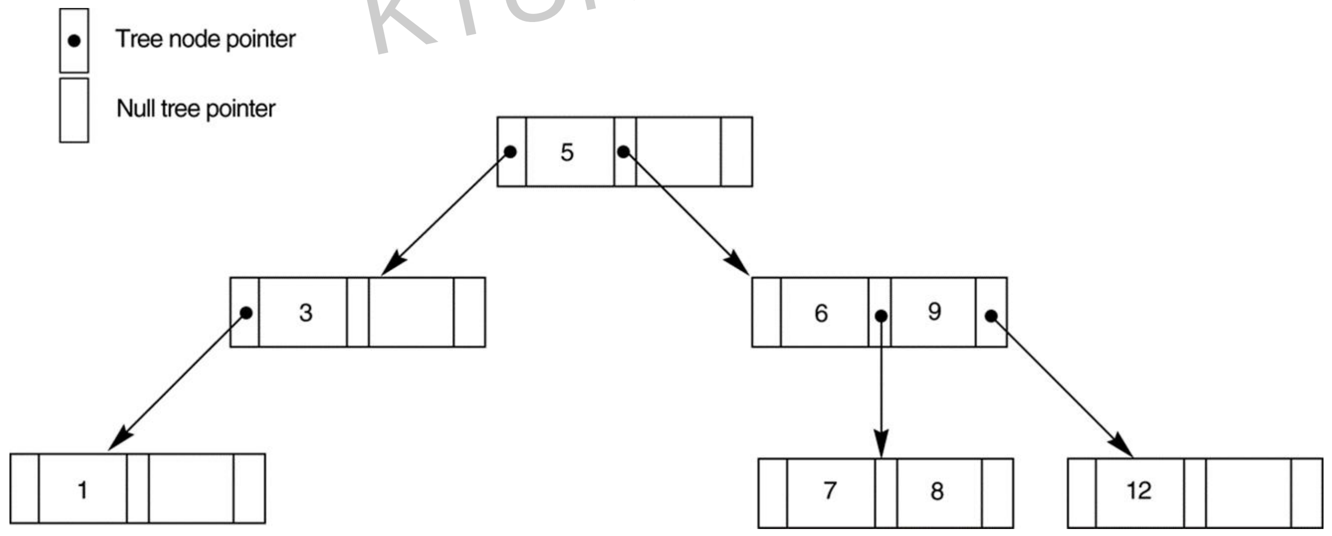
Figure 18.7
A tree data structure that shows an unbalanced tree.

Search Trees and B-Trees

- Compared to multilevel index: notice the redundancy of index values between levels, and notice the uselessness of the first entry in each block.
- Search Trees



- Each search field has the search value (index value) and a pointer to the associated record in storage (a record pointer or block pointer,
- depending).
- In practice each node is one block of storage.



B-tree Index Files

- B-tree (balanced tree)
- - a search tree with some additional constraints for efficient insertion and deletion
- - number of access is fixed
- **<u>Formal definition</u>**
- A B-tree of order n is a search tree that satisfies

1) the root has at least two children
2) all nodes other than root have at least n/2 children
3) all leaf nodes are at the same level (balanced)

B-Trees

- Each node has at most m children and at most m-1 keys.
- Nodes may leave some empty space, allowing for inserting new entries.
- Insertion into a non-full node is very efficient.
- If the node is full it causes a split.
- Spliting the root node creates two children, with only the middle value left in the original root.
- Spliting a branch node divides it into two branch nodes, and moves the middle value to the *parent* node.
- Spliting can propagate to other tree levels.
- Deletion from a node more than ½-full is very efficient.
- If the node becomes ½-full it must be merged with neighboring nodes.
- Specific approaches to deletion merging vary.

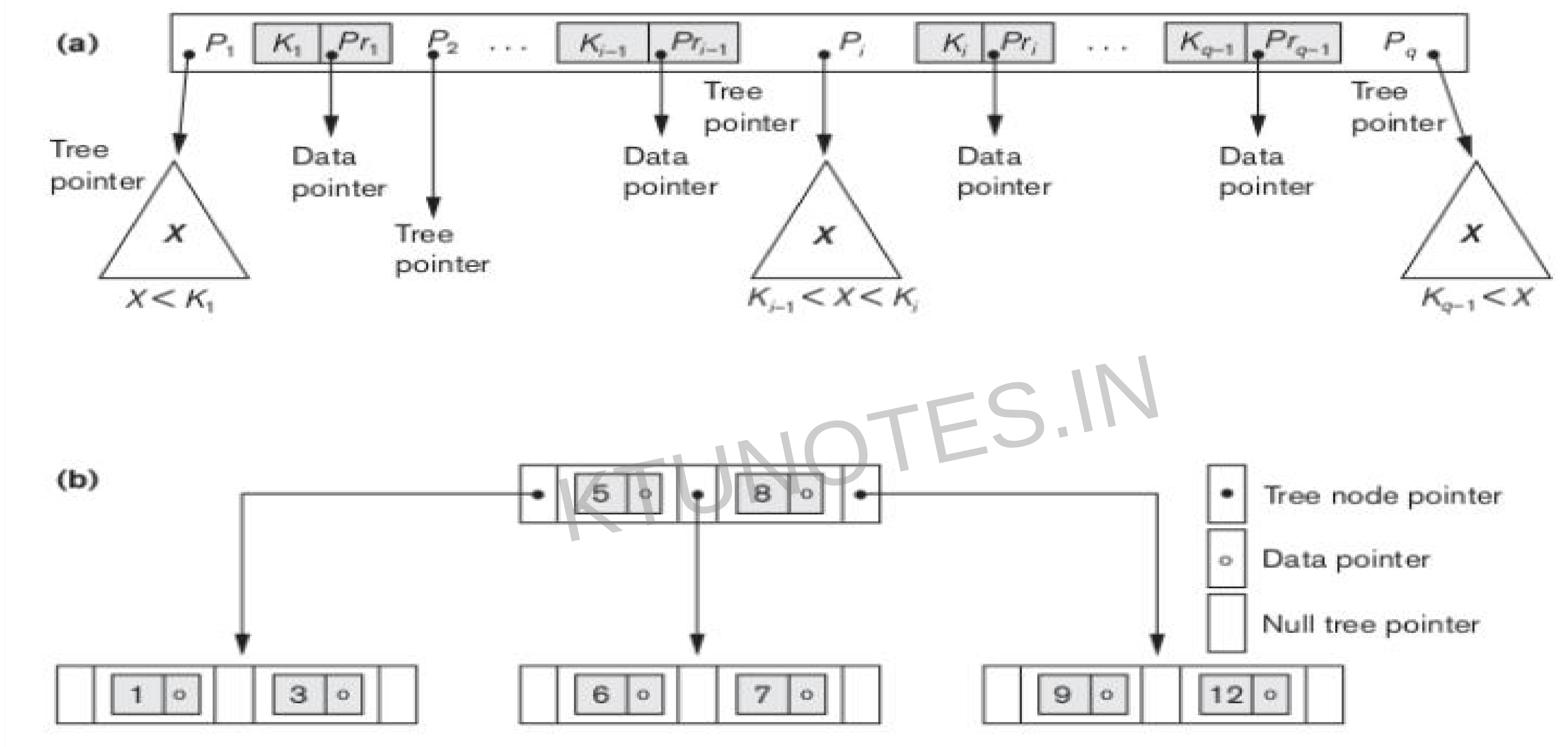


Figure 18.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

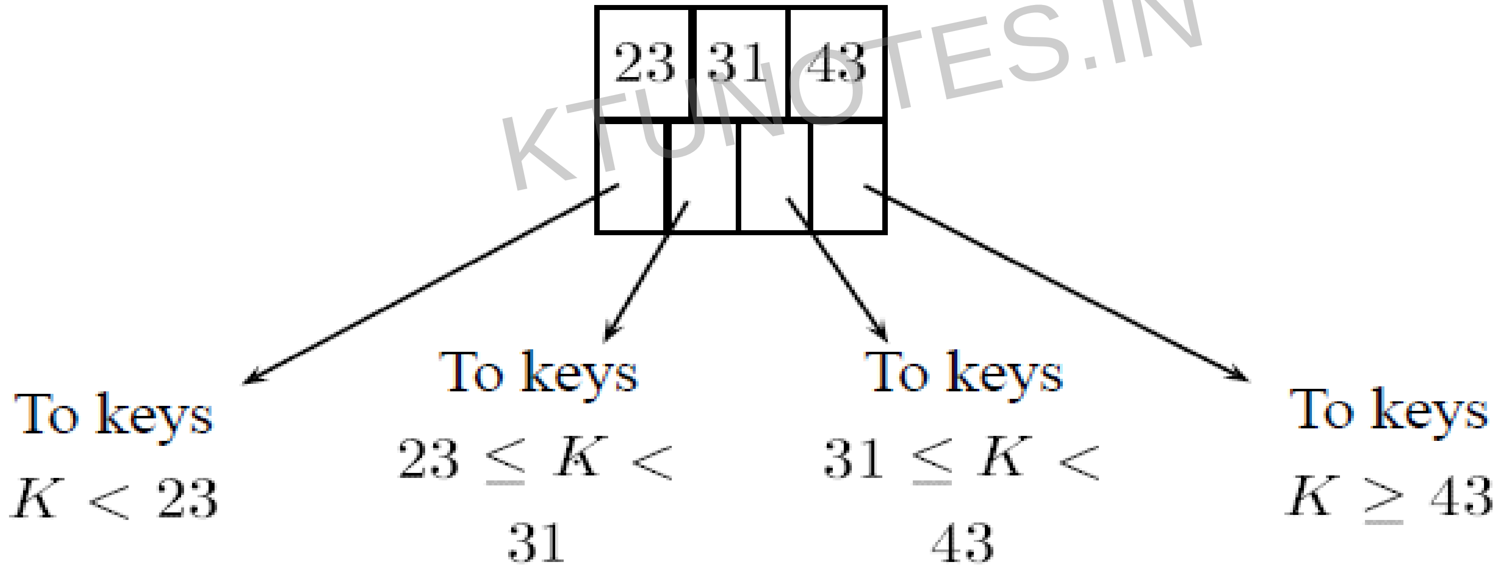
B+ tree

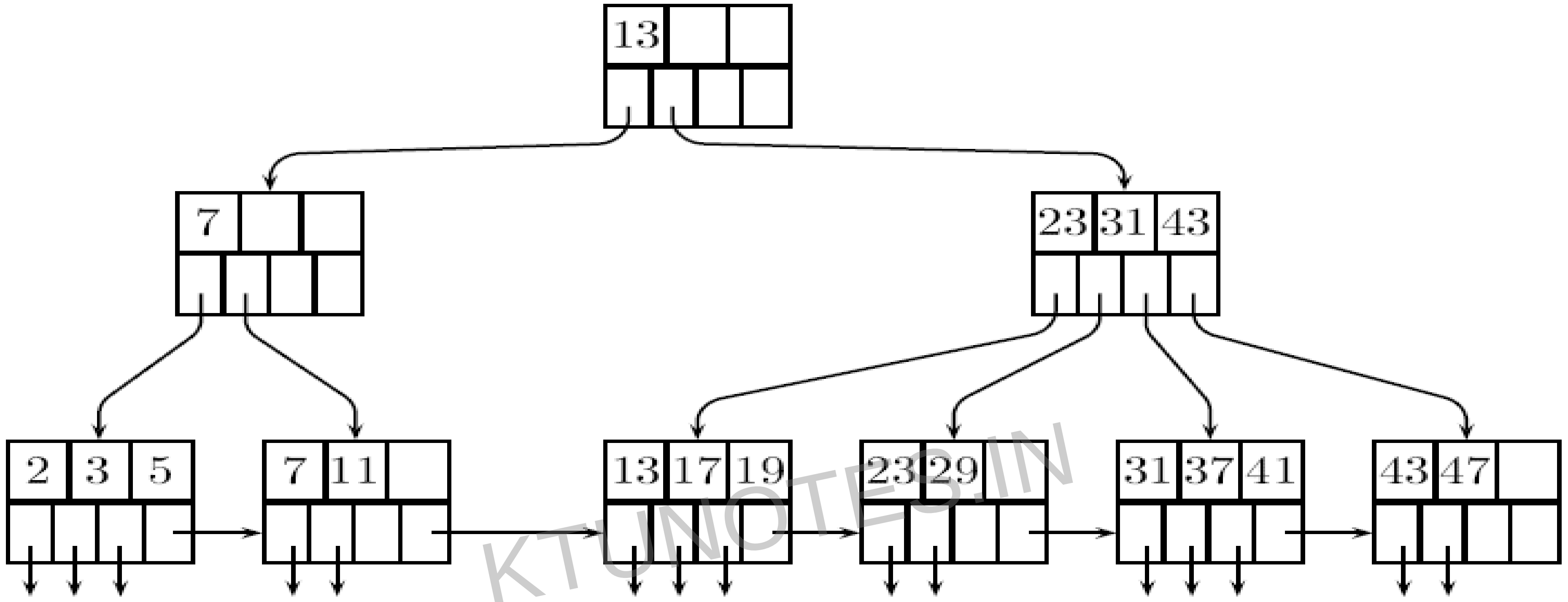
- B+-tree: a variation of B-tree data structure
- - most widely used multi-level index implementation
- **B+ Trees** A B+ Tree combines features of ISAM and B Trees. It contains index pages and data pages.
- The data pages always appear as leaf nodes in the tree. The root node and intermediate nodes are always index pages.
- These features are similar to ISAM. Unlike ISAM, overflow pages are not used in B+ trees.
- No of pointers in a node is called fanout

B-tree is usually a 3 levels tree: the root, an intermediate level, the leaves.

All the leaves are at the same level $\rightarrow$ *Balanced Tree.*

The size of each node of the B-tree is equal to a disk block. All nodes have the same format: **n keys** and **$n + 1$ pointers** $\rightarrow$ n key-pointer pairs plus 1 extra pointer.



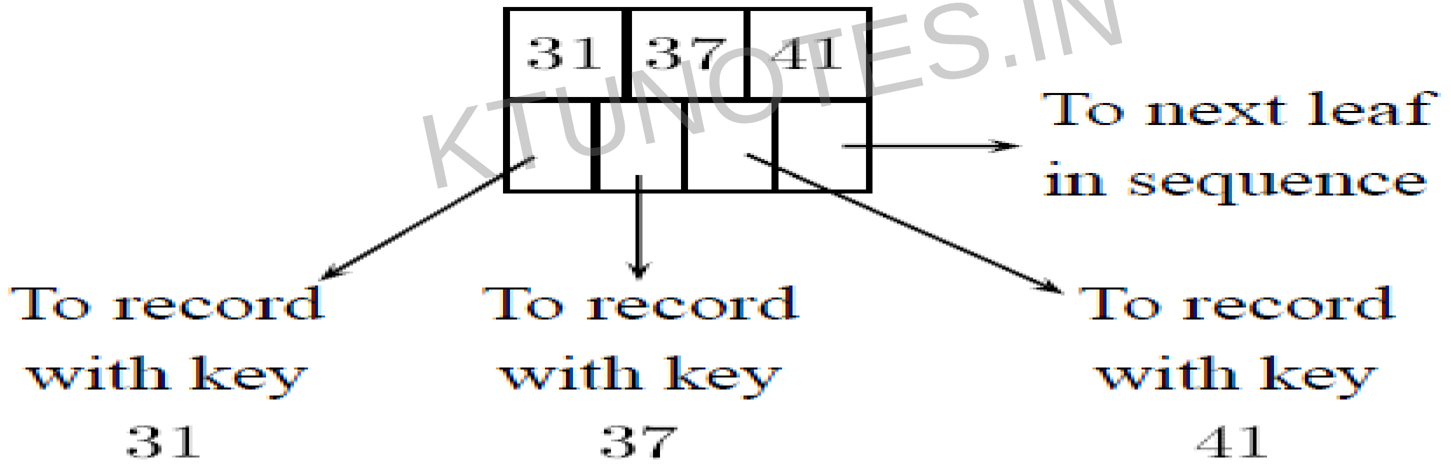


Data file where search-keys are all the prime numbers from 2 to 47.

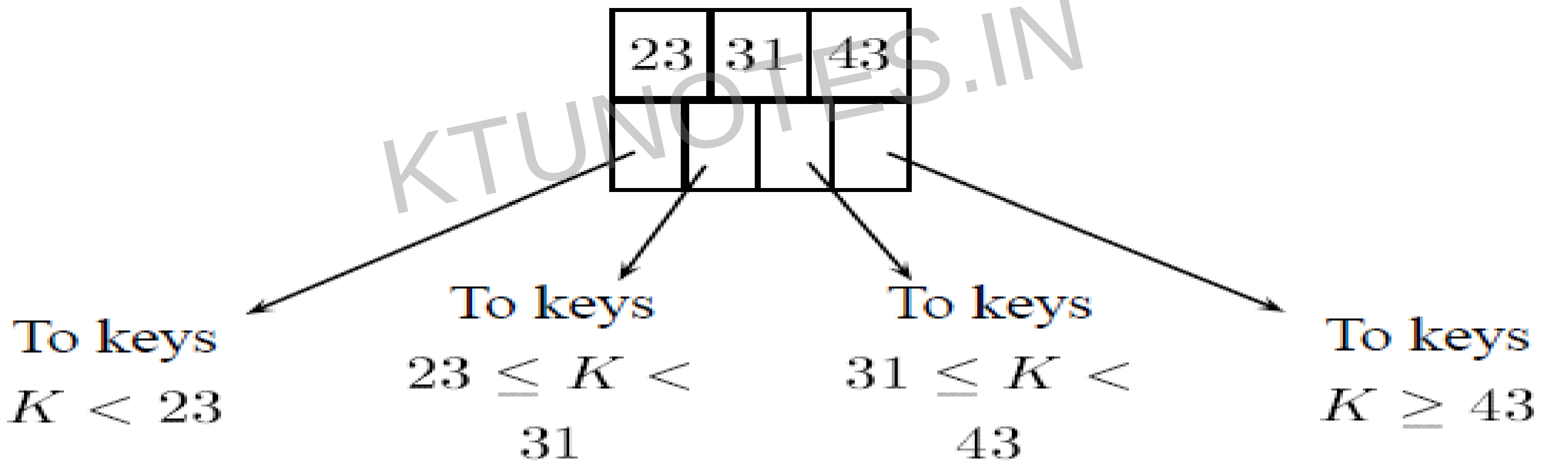
All the keys appear once (in case of a dense index), and in sorted order at the leaves.

A pointer points to either a file record (in case of a primary index structure) or to a bucket of pointers (in case of a secondary index structure).

Leaves


31
37
41
To next leaf in sequence
To record with key 31
To record with key 37
To record with key 41

Interior Nodes

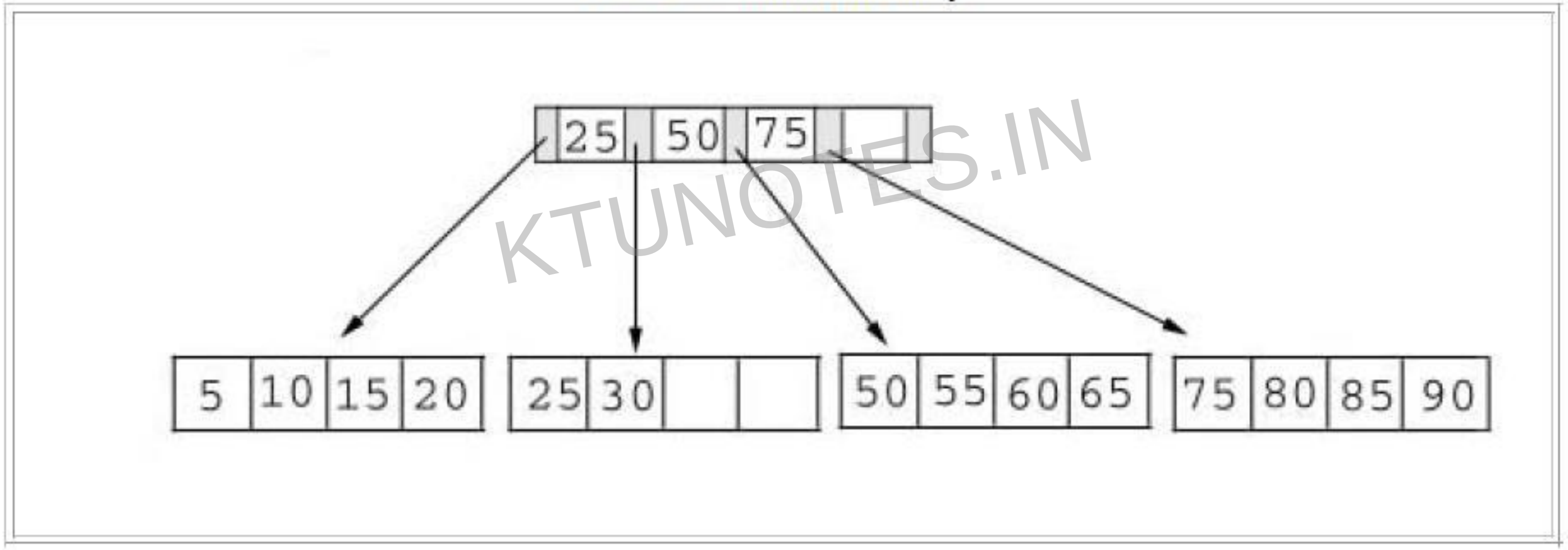


B+ Trees and B Trees use a "fill factor" to control the growth and the shrinkage. A 50% fill factor would be the minimum for any B+ or B tree. As our example we use the smallest page structure. This means that our B+ tree conforms to the following guidelines.

Number of Keys/page	4
Number of Pointers/page	5
Fill Factor	50%
Minimum Keys in each page	2

As this table indicates each page must have a minimum of two keys. The root page may violate this rule.

B+ Tree with four keys



B+-Trees Compared with B-Trees:

- Only leaf nodes store record pointers.
- All search values are stored in leaf nodes. (And may be stored at higher levels.)
- Leaf nodes are usually linked into a sorted list.
- With no data pointers in the internal nodes, fanout is increased and height is decreased.
- B+-Trees are commonly used in modern filesystems for indexing subdirectories, very large files, and so on.
- They are the basis for many related data structures like R-Tree etc.

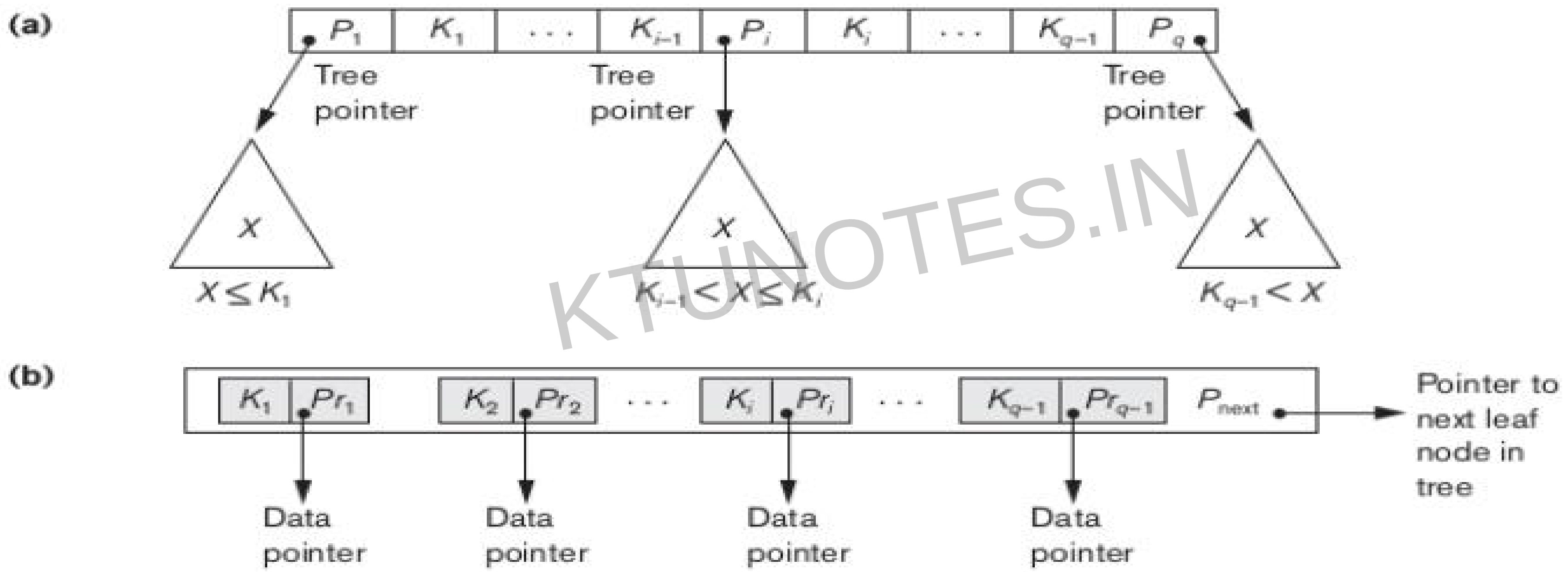
Differences of B+-tree from B-tree

1. In B+-tree, data pointers are stored only at the leaf nodes

2. Leaf and non-leaf nodes are of the same size in B+-tree, while in B-tree, non-leaf nodes are larger

3.Deletion in B-tree is more complicated

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



Dynamic Multi-Level Indexes using B+-Trees

- B+-Tree indexes are an alternative to index sequential files.
- Disadvantage of index-sequential files: performance degrades as sequential le grows, because many overflow blocks are created. Periodic reorganization of entire file is required.
- Advantage of B+-Tree index file: automatically reorganizes itself with small, local changes in the case of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B+-Trees: extra insertions and deletion overhead, space overhead.

- A B+-Tree is a rooted tree satisfying the following properties:
- All paths from the root to leaf have the same length (a B+ tree is a balanced tree).
- Each node that is not the root or a leaf node has between[n/2] and n children (where n is fixed for a particular tree).
- A leaf node has between [(n-1/2)] and n-1 values.
- Special case: if the root is not a leaf, it has at least 2 children. If the root is a leaf, it can have between 0 and n-1 values.

Typical structure of a node:

P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

- K_i are the search key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes)

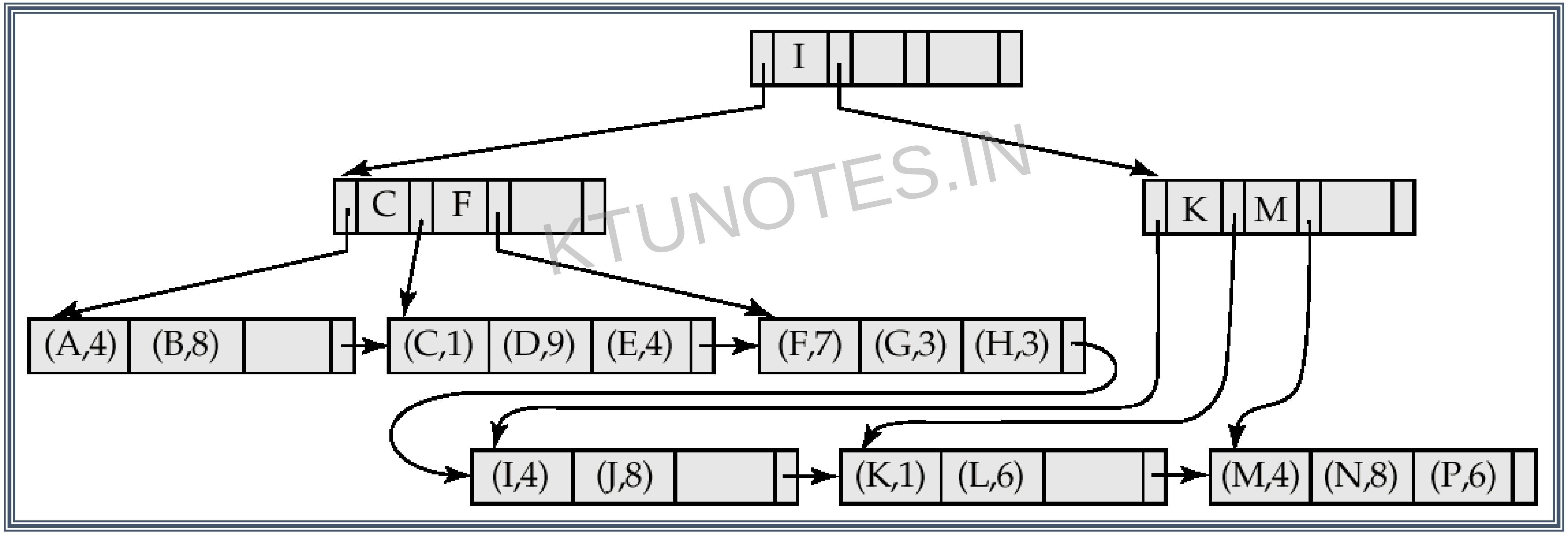
The search key values are kept in sorted order.

The search keys in a node are ordered, i.e,

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

B+ - Tree File Organization

- In a B+ - Tree file organization, the leaf nodes of the tree stores the actual record rather than storing pointers to records.
- During insertion, the system locates the block that should contain the record. If there is enough free space in the node then the system stores it. Otherwise the system splits the record in two and distributes the records.
- During deletion, the system first removes the record from the block containing it. If the block becomes less than half full as a result, the records in the block are redistributed.

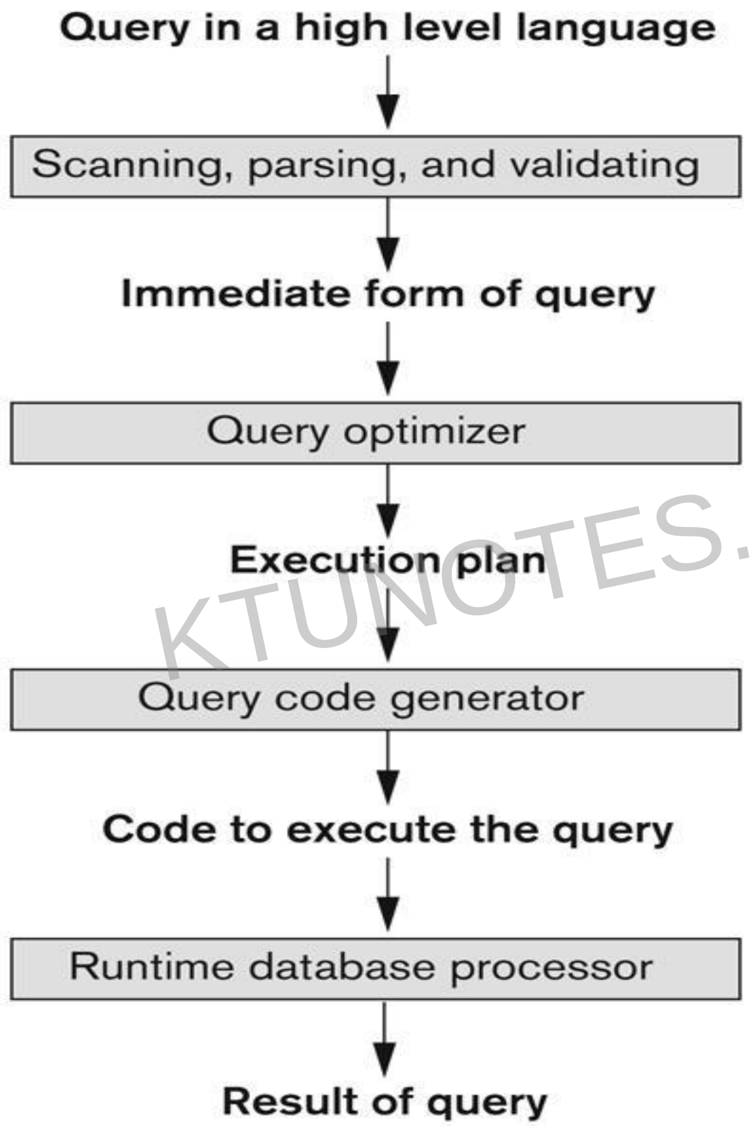

I
C
F
K
M
(A,4)
(B,8)
(C,1)
(D,9)
(E,4)
(F,7)
(G,3)
(H,3)
(I,4)
(J,8)
(K,1)
(L,6)
(M,4)
(N,8)
(P,6)

Query Processing

- **Query Processing: Activities involved in retrieving data from the database.**
- **Aims of QP:**
- **transform query written in high-level language (e.g. SQL), into correct and efficient execution strategy expressed in low-level language (implementing RA); execute the strategy to retrieve required data.**

Phases of Query Processing

- **QP has 4 main phases:**
- o **decomposition**
  - – Aims are to transform high-level query into RA query and check that query is syntactically and semantically correct.
- o **optimization**
- o **code generation**
- o **execution.**


Query in a high level language
Scanning, parsing, and validating
Immediate form of query
Query optimizer
Execution plan
Query code generator
Code to execute the query
Runtime database processor
Result of query

Code can be:

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

Figure 15.1
Typical steps when processing a high-level query.

SQL Queries and Relational Algebra

- SQL query is translated into an equivalent ***extended*** relational algebra expression --- represented as a query tree
- In order to transform a given query into a query tree, the query is decomposed into ***query blocks***
  - **Query block**:
    - The basic unit that can be translated into the algebraic operators and optimized.
  - A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clause if these are part of the block.
- The query optimizer chooses an execution plan for each block

COMPANY Relational Database Schema

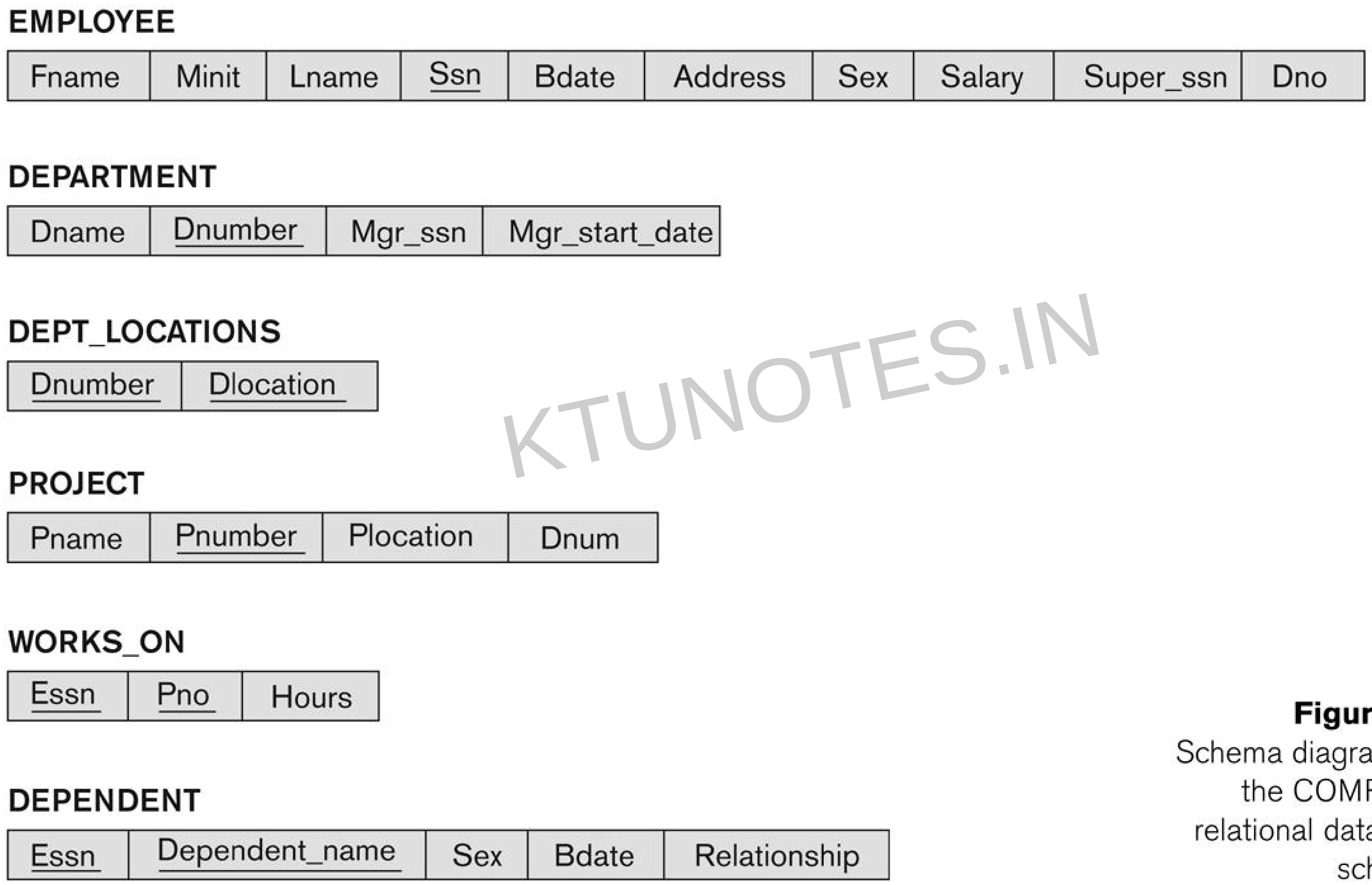

EMPLOYEE
Fname Minit Lname Ssn Bdate Address Sex Salary Super_ssn Dno
DEPARTMENT
Dname Dnumber Mgr_ssn Mgr_start_date
DEPT_LOCATIONS
Dnumber Dlocation
PROJECT
Pname Pnumber Plocation Dnum
WORKS_ON
Essn Pno Hours
DEPENDENT
Essn Dependent_name Sex Bdate Relationship

Figure 5.5
Schema diagram for the COMPANY relational database schema.

COMPANY Relational Database Schema

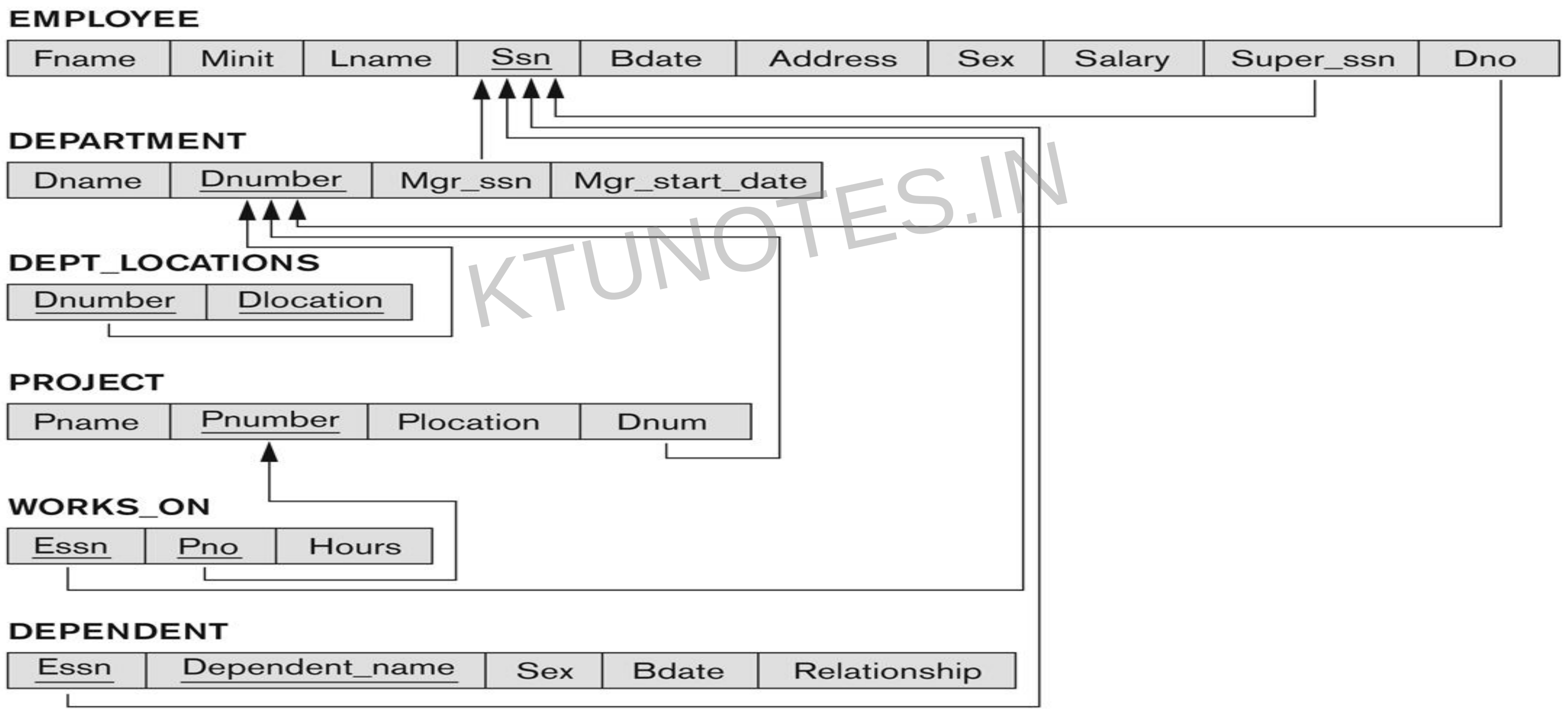


Figure 5.7
Referential integrity constraints displayed on the COMPANY relational database schema.

- Example

```
SELECT  Lname, Fname
FROM EMPLOYEE
WHERE Salary > ( SELECT MAX(Salary)
                 FROM EMPLOYEE
                 WHERE Dno = 5      )
```

- Inner block and outer block

Translating SQL Queries into Relational Algebra

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

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

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

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

$$\mathcal{F}_{MAX\ SALARY}\ (\sigma_{DNO=5}\ (EMPLOYEE))$$

- Example

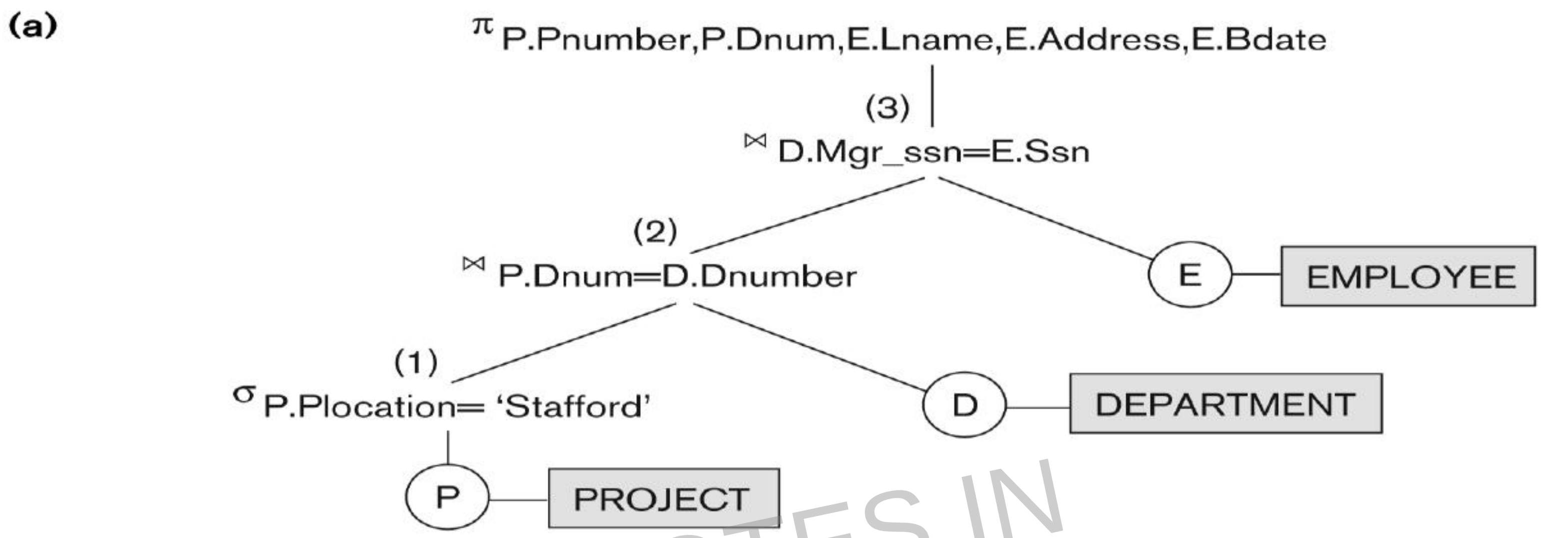
  For every project located in 'Stafford', retrieve the project number, the controlling department number and the department manager's last name, address and birthdate.

- SQL query:

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

- Relation algebra:

$$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}\,(((\sigma_{\text{PLOCATION='STAFFORD'}}(\text{PROJECT}))\ \bowtie_{\text{DNUM=DNUMBER}}\ (\text{DEPARTMENT}))\ \bowtie_{\text{MGRSSN=SSN}}\ (\text{EMPLOYEE}))$$



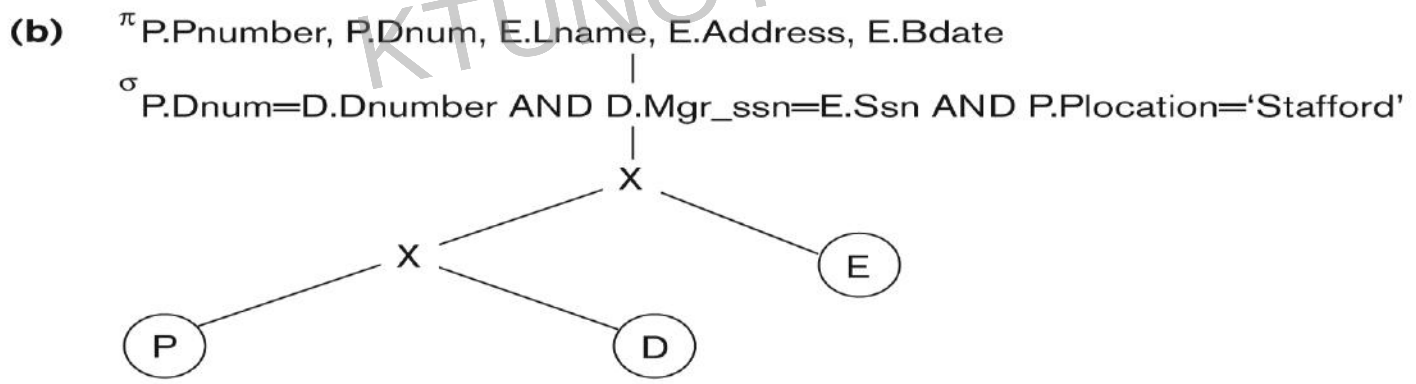

(b)
π P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
σ P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
X
X
E
P
D

Figure 15.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

QUERY OPTIMIZATION

- **Query Optimization: Activity of choosing an efficient execution strategy for processing query.**
- **Converting a query into an equivalent form which is more efficient to execute.**
- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every legal database instance
- **As there are many equivalent transformations of same high-level query, aim of QO is to choose one that minimizes resource usage.**
- **Generally, reduce total execution time of query.**
- **Problem computationally intractable with large number of relations, so strategy adopted is reduced to finding near optimum solution.**

Example 1 - Different Strategies

Find all Managers that work at a London branch:

```
SELECT *
FROM staff s, branch b
WHERE s.bno = b.bno AND
(s.position = 'Manager' AND b.city = 'London');
```

3 equivalent RA queries are:

$\sigma_{(position='Manager') \wedge (city='London') \wedge (staff.bno=branch.bno)} (Staff \times Branch)$

$\sigma_{(position='Manager') \wedge (city='London')}(Staff \bowtie Branch)$

$(\sigma_{position='Manager'}(Staff)) \bowtie (\sigma_{city='London'} (Branch))$

- **Two main techniques for query optimization:**
- – heuristic rules that order operations in a query.
- – comparing different strategies based on relative costs, and selecting one that minimizes resource usage.

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Heuristic-Based Query Optimization

- Outline of heuristic algebraic optimization algorithm
  1. Break up SELECT operations with conjunctive conditions into a cascade of SELECT operations
  2. Using the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition
  3. Using commutativity and associativity of binary operations, rearrange the leaf nodes of the tree
  4. Combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition
  5. Using the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed
  6. Identify sub-trees that represent groups of operations that can be executed by a single algorithm

Heuristic-Based Query Optimization: Example

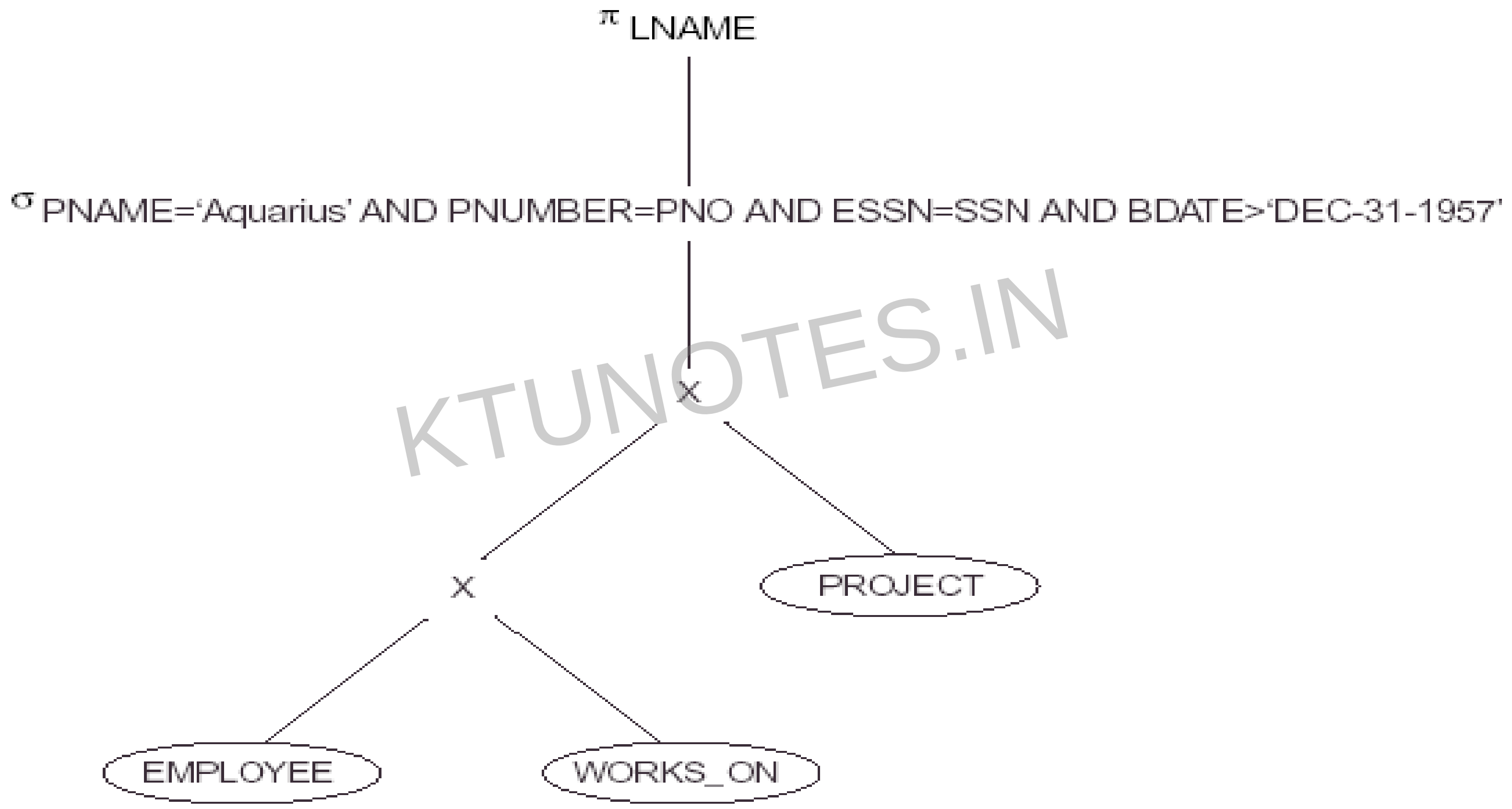
- Query

  "Find the last names of employees born after 1957 who work on a project named ‘Aquarius’."

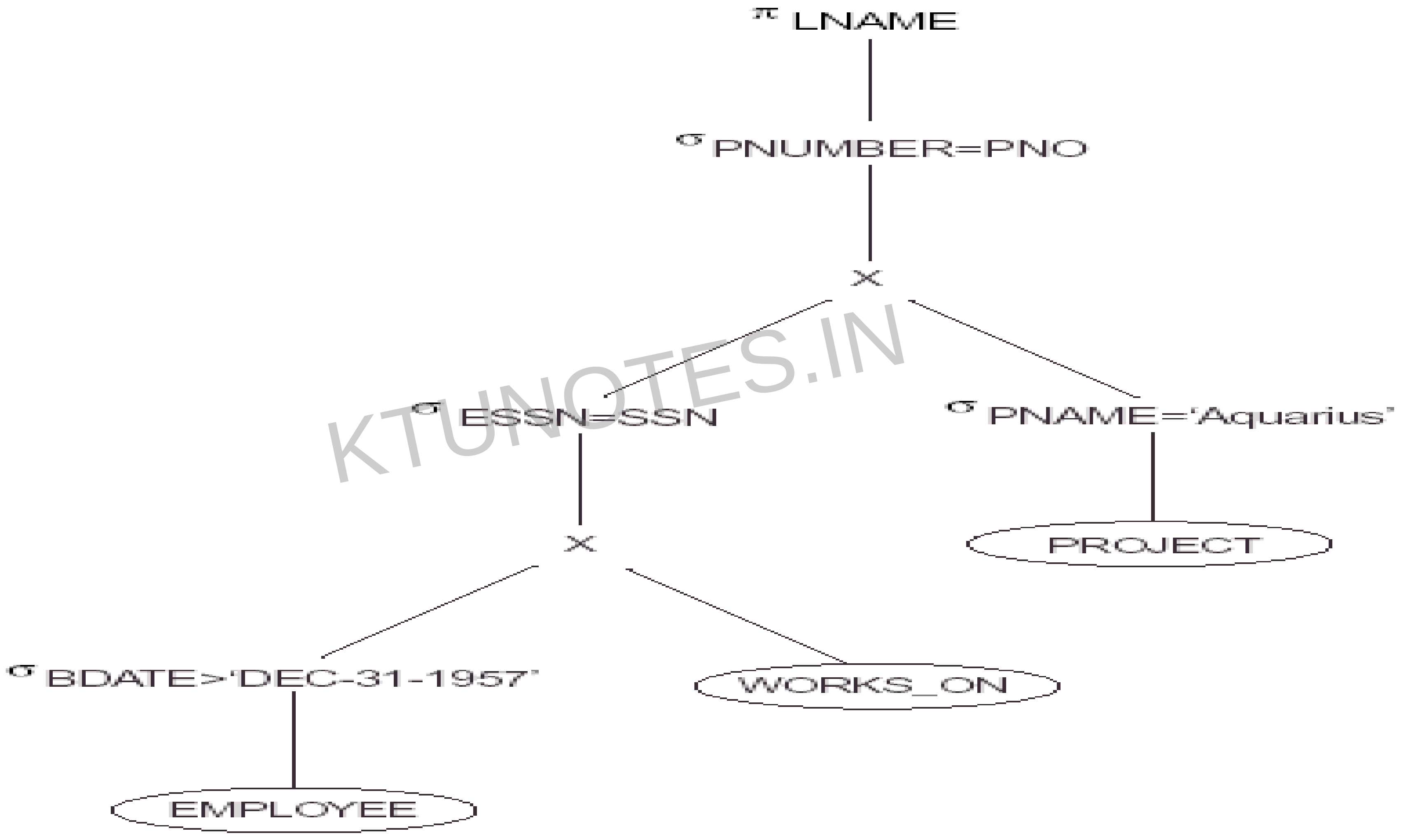
- SQL

```
SELECT LNAME
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE PNAME=‘Aquarius’ AND PNUMBER=PNO AND ESSN=SSN AND
   BDATE.‘1957-12-31’;
```

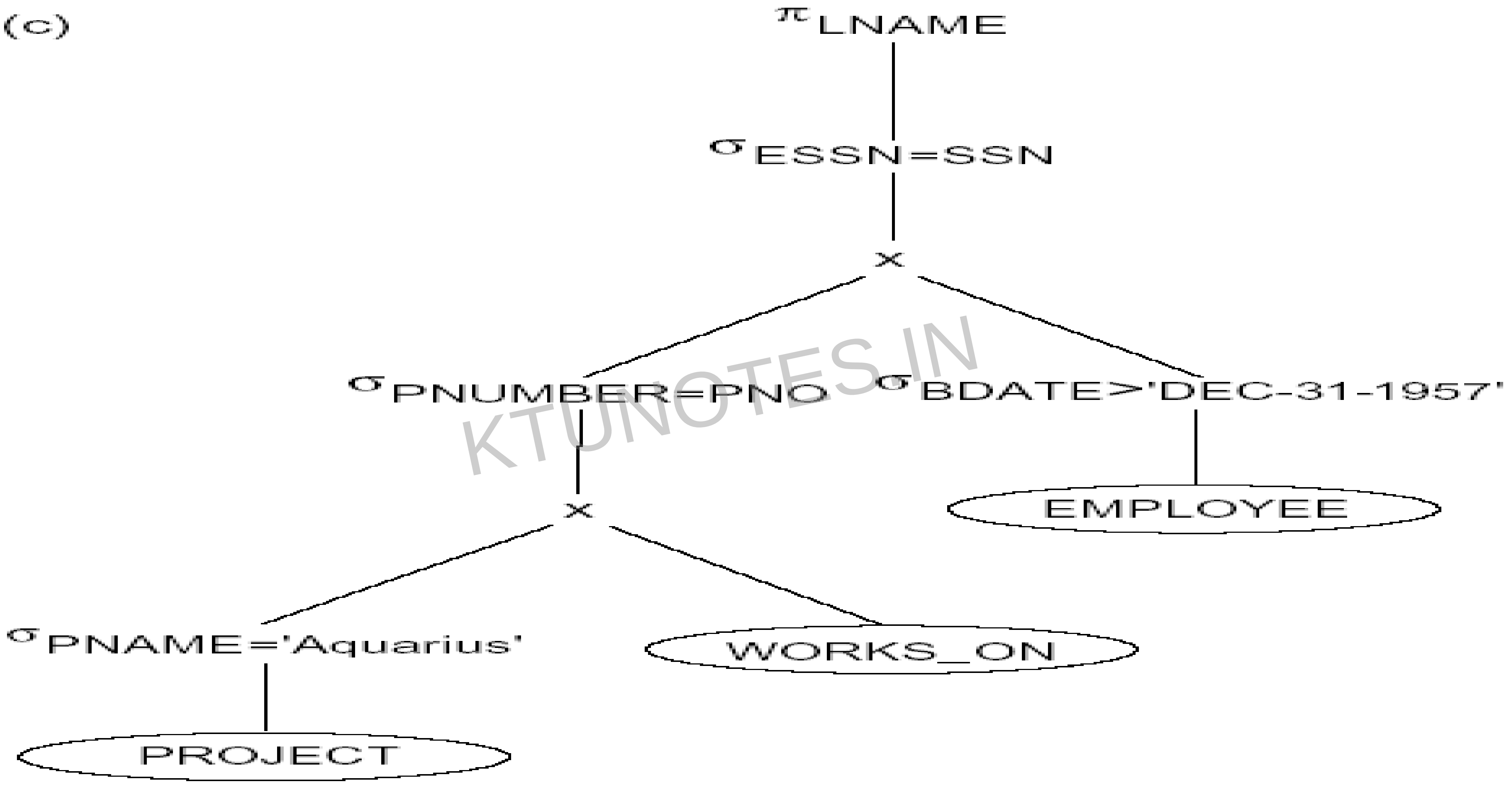
(a)


π LNAME
σ PNAME='Aquarius' AND PNUMBER=PNO AND ESSN=SSN AND BDATE>'DEC-31-1957'
X
X
PROJECT
EMPLOYEE
WORKS_ON

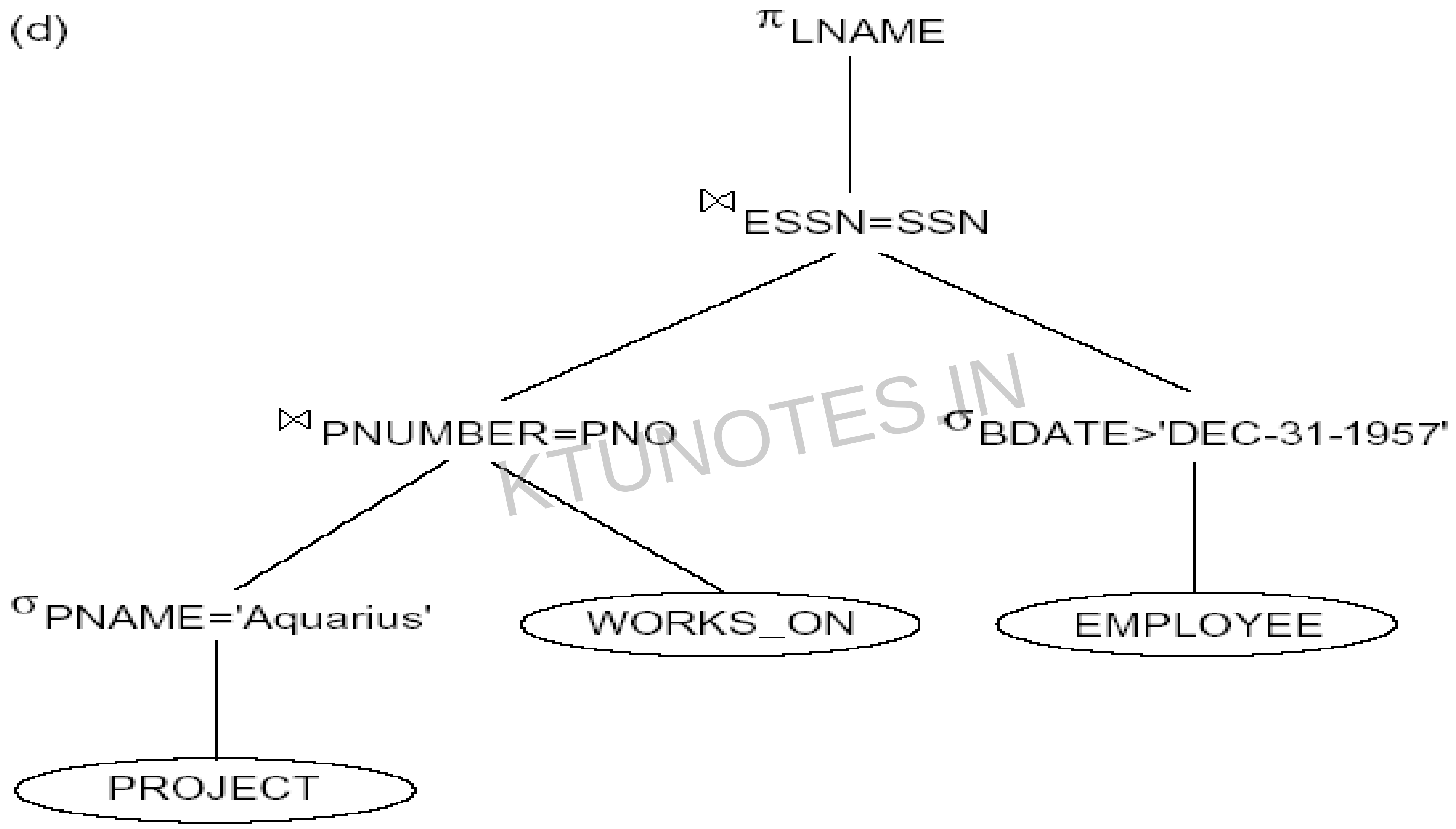
(b)


π LNAME
σ PNUMBER=PNO
X
σ ESSN=SSN
σ PNAME='Aquarius'
X
PROJECT
σ BDATE>'DEC-31-1957'
WORKS_ON
EMPLOYEE

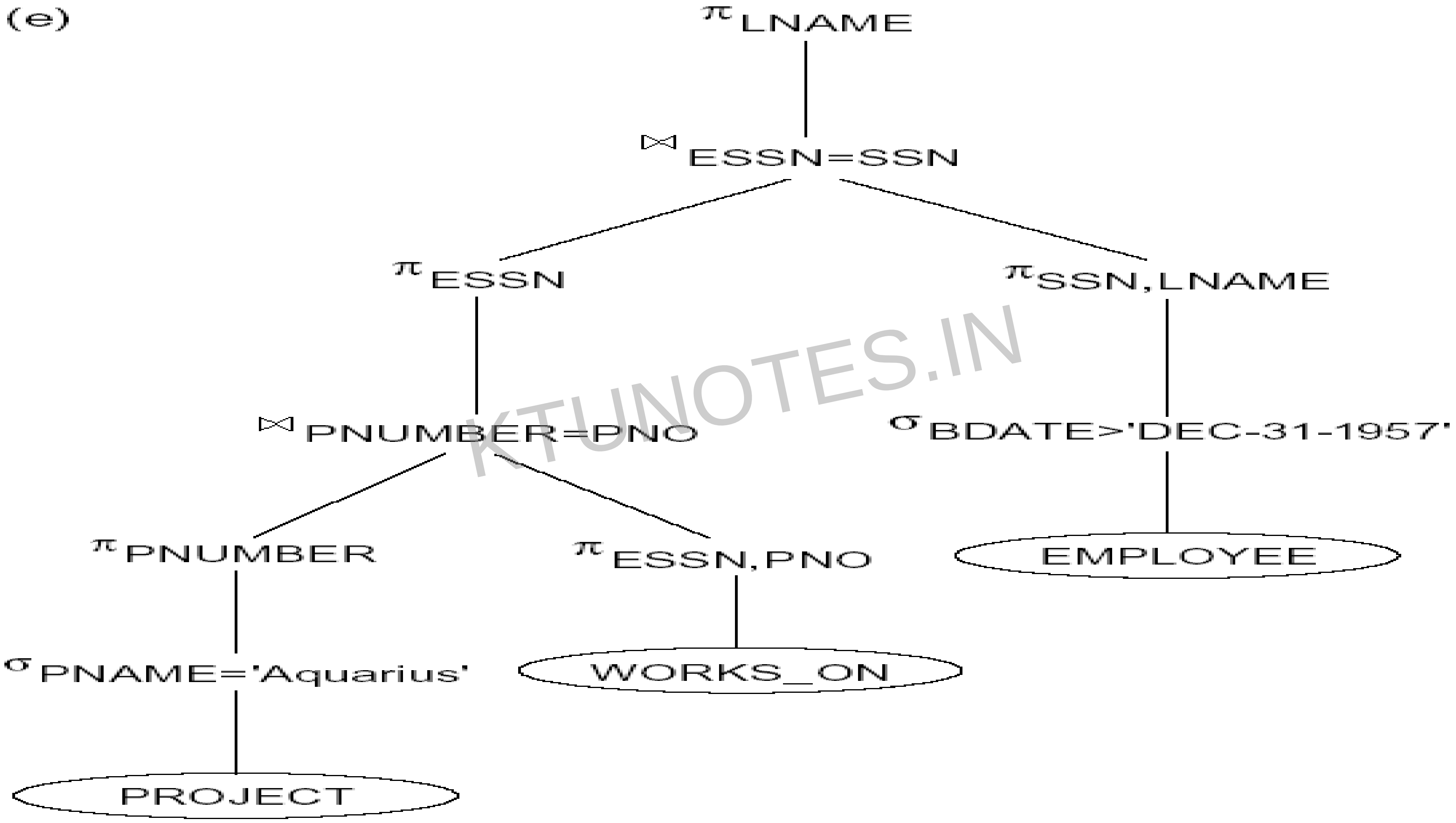
(c)


π_{LNAME}
$\sigma_{ESSN=SSN}$
X
$\sigma_{PNUMBER=PNO}$
$\sigma_{BDATE>'DEC-31-1957'}$
X
EMPLOYEE
$\sigma_{PNAME='Aquarius'}$
WORKS_ON
PROJECT

(d)


π LNAME
$\bowtie$ ESSN=SSN
$\bowtie$ PNUMBER=PNO
σ BDATE>'DEC-31-1957'
σ PNAME='Aquarius'
WORKS_ON
EMPLOYEE
PROJECT

(e)



π LNAME
⋈ ESSN=SSN
π ESSN
π SSN, LNAME
⋈ PNUMBER=PNO
σ BDATE>'DEC-31-1957'
π PNUMBER
π ESSN, PNO
EMPLOYEE
σ PNAME='Aquarius'
WORKS_ON
PROJECT