Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology

Chapter 3: Query processing and optimization

Graduate Programme in Computer Science

Dr. Vo Thi Ngoc Chau

(chauvtn@cse.hcmut.edu.vn)

Semester 2 – 2011-2012

Main references

[1] R. Elmasri, S. R. Navathe, *Fundamentals of Database Systems- 4th Edition*, Pearson- Addison Wesley, 2003.

[2] H. G. Molina, J. D. Ullman, J. Widom, *Database System Implementation*, Prentice-Hall, 2000.

[3] H. G. Molina, J. D. Ullman, J. Widom, *Database Systems: The Complete Book,* Prentice-Hall, 2002

[4] A. Silberschatz, H. F. Korth, S. Sudarshan, *Database System Concepts –3rd Edition*, McGraw-Hill, 1999.

[5] R. Ramakrishnan, J. Gehrke, *Database Management Systems- 2rd Edition,* McGraw-Hill.

[6] M. P. Papazoglou, S. Spaccapietra, Z. Tari, *Advances in Object-Oriented Data Modeling*, MIT Press, 2000.

[7]. G. Simsion, *Data Modeling: Theory and Practice*, Technics Publications, LLC, 2007.

Content

- Chapter 1: An overview of database systems
- Chapter 2: Data modeling
- **Chapter 3: Query processing and optimization**
- Chapter 4: Database security
- Chapter 5: Transaction processing and concurrency control
- Chapter 6: Database recovery
- Chapter 7: Review

3

Chapter 3: Query processing and optimization

- 3.1. An overview of query processing and optimization
- 3.2. File organization
- 3.3. Indexing
- 3.4. SQL (structured query language)
- 3.5. Algorithms for query processing
- 3.6. Rule-based query optimization
- 3.7. Cost-based query optimization
- 3.8. Semantics-based query optimization
- 3.9. Conclusion

4

3.1. An overview of query processing and optimization

- To process, optimize, and execute high-level queries
  - A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated.
    - SQL queries are decomposed into query blocks, which form the basic units 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 clauses if these are part of the block.
  - The query must also be validated, by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried.
  - An internal representation of the query is then created, usually as a tree data structure called a query tree.
  - The DBMS must then devise an execution strategy for retrieving the result of the query from the database files.
    - a *reasonably efficient strategy* for executing the query

5

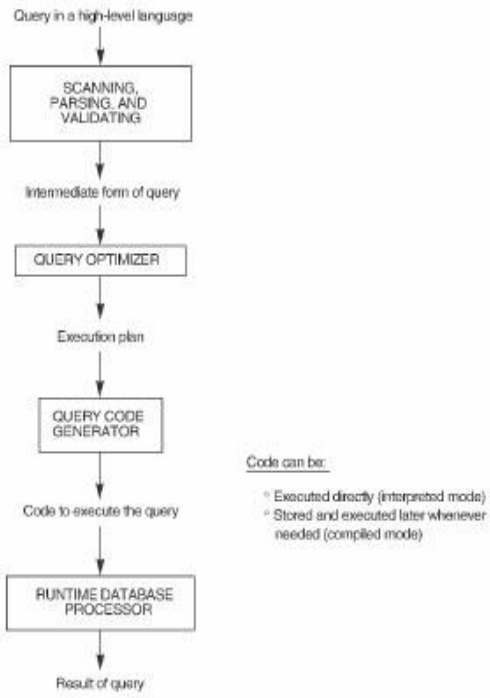


Figure 15.1. Typical steps when processing high-level queries [1], pp. 494.

6

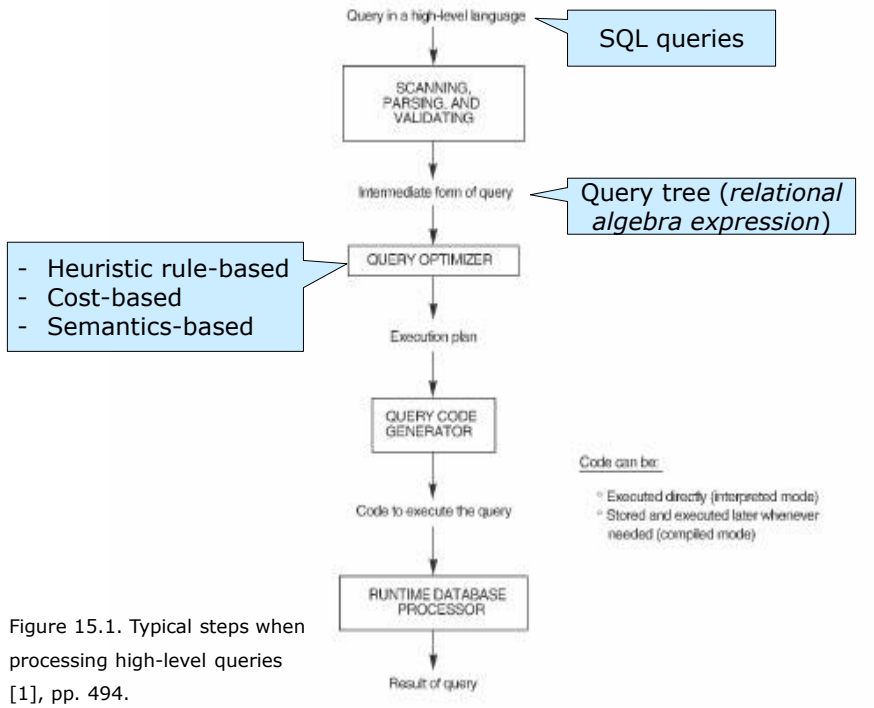


Figure 15.1. Typical steps when processing high-level queries [1], pp. 494.

7

Query: retrieve the last name and first name of each employee whose salary is greater than the max salary in department 5.

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

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

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

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

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

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

$\mathfrak{I}_{MAX\ SALARY}$
|
$\sigma_{DNO = 5}$
|
EMPLOYEE

8

3.2. File organization

- Databases are stored physically as *files of records*, which are typically stored on magnetic disks.
  - Computer storage medium
    - Data persistence
    - Volume of data
    - Cost of storage per unit of data
    - Access time
  - Data in secondary storage cannot be processed directly by the CPU; it must first be copied into primary storage.
    - Data unit used in transfer is block.

9

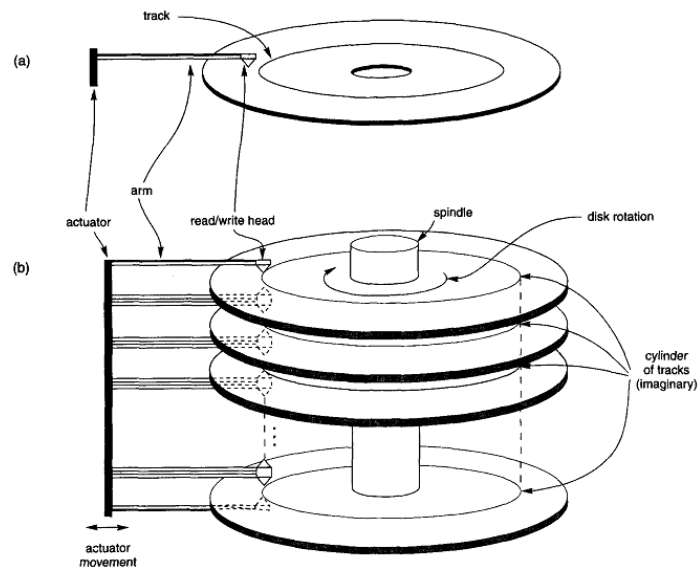
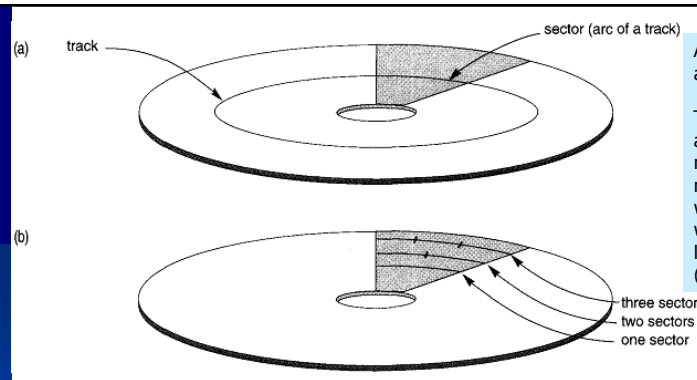


Figure 13.1 (a) A single-sided disk with read/write hardware.
(b) A disk pack with read/write hardware.

[1], pp. 416

10



A disk is a random access addressable device.

The hardware address of a block = a cylinder number + a track number (surface number within the cylinder on which the track is located) + block number (within the track)

Figure 13.2 Different sector organization on disk. (a) Sectors subtending a fixed angle. (b) Sectors maintaining a uniform recording density.

[1], pp. 417

- The division of a track into equal-sized disk blocks (or pages) is set by the operating system during disk formatting (or initialization).
- A disk with hard-coded sectors often has the sectors subdivided into blocks during initialization.
- Block size is fixed during initialization and can't be changed dynamically.
  - Typical disk block sizes range from 512 to 4096 bytes.

11

3.2. File organization

- Secondary storage technology
  - RAID (Redundant Arrays of Inexpensive (Independent) Disks) – RAID 0-6
    - Parallelism to improve disk performance
    - Redundancy to improve reliability
  - SAN (Storage Area Networks)
    - Online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner.

12

3.2. File organization

- Each *record* is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships.
  - Records should be stored on disk in a manner that makes it possible to locate them efficiently whenever they are needed.
- *Primary file organizations* determine how the records of a file are physically placed on the disk, and hence how the records can be accessed.
  - File organization + Access method
- A *secondary organization* or *auxiliary access structure* allows efficient access to the records of a file based on alternate fields than those that have been used for the primary file organization.

13

3.2. File organization

- Types of record organization
  - Unspanned
  - Spanned

→ The blocking factor (*bfr*) = (average) number of records per block for the file

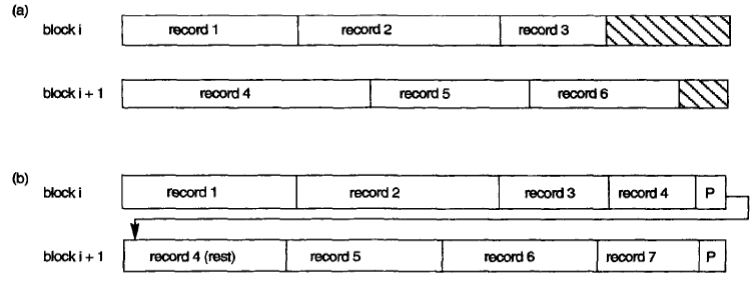


FIGURE 13.6 Types of record organization. (a) Unspanned. (b) Spanned.

14

3.2. File organization

- Primary file organizations
  - A *heap* (pile/unordered) file places the records on disk in no particular order by appending new records at the end of the file.
  - A *sorted* (ordered/sequential) file keeps the records ordered by the value of a particular field (called the sort key/ordering field).
  - A *hashed* (direct) file uses a has function applied to a particular field (called the hash key) to determine a record's placement on disk.
    - External hashing
    - Static vs. dynamic hashing

15

3.2. File organization

- Operations on files
  - Reset
  - Find (or Locate)
  - Read (or Get)
  - FindNext
  - Delete
  - Modify
  - Insert
  - FindAll
  - Find (or Locate) n
  - FindOrdered
  - Reorganize
  - Open
  - Close

16

3.2. File organization

Adapted from Table 13.2 Average Access Times for Basic File Organizations [1], pp. 434.

Type of organization	Access/Search method	Average time to access a specific record
Heap (unordered)	Sequential scan (linear search)	b/2
Ordered	Sequential scan	b/2
Ordered	Binary search	$\log_2 b$
Hashed (direct)	Static hashing	~1
Hashed (direct)	Linear (dynamic) hashing	1
Hashed (direct)	Extendible (dynamic hashing)	1-2

Note: a file with *b* blocks

17

3.3. Indexing

- Indexes
  - Additional auxiliary access structures
  - → To provide secondary access paths, which provide alternative ways of accessing the records without affecting the physical placement of records on disk
  - → To speed up the retrieval of records in response to certain search conditions
- Index types
- Index structures
  - B-tree/B+-tree
  - R-tree

18

3.3. Indexing

- Index types
  - Single-level ordered indexes/Multilevel indexes
    - Single-level ordered index: an index on a data file
    - Multilevel index: index that includes index on index, … index on index, index on a data file
  - Primary/Clustering/Secondary indexes

TABLE 14.1 TYPES OF INDEXES BASED ON THE PROPERTIES OF THE INDEXING FIELD

	INDEX FIELD USED FOR ORDERING THE FILE	INDEX FIELD NOT USED FOR ORDERING THE FILE
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

19

3.3. Indexing

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.

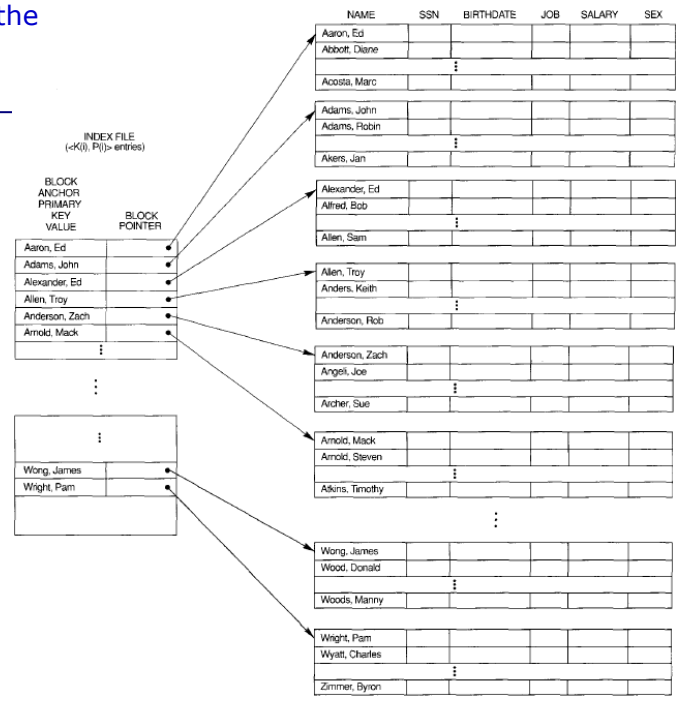
Dense index: one index entry for each record in the data file, which contains the value of the secondary key for the record and a pointer either to the block in which the record is stored or to the record itself

Block anchor (anchor record): the first record in each block of the data file

20

Primary index on the ordering key field

Figure 14.1
[1], pp. 458.



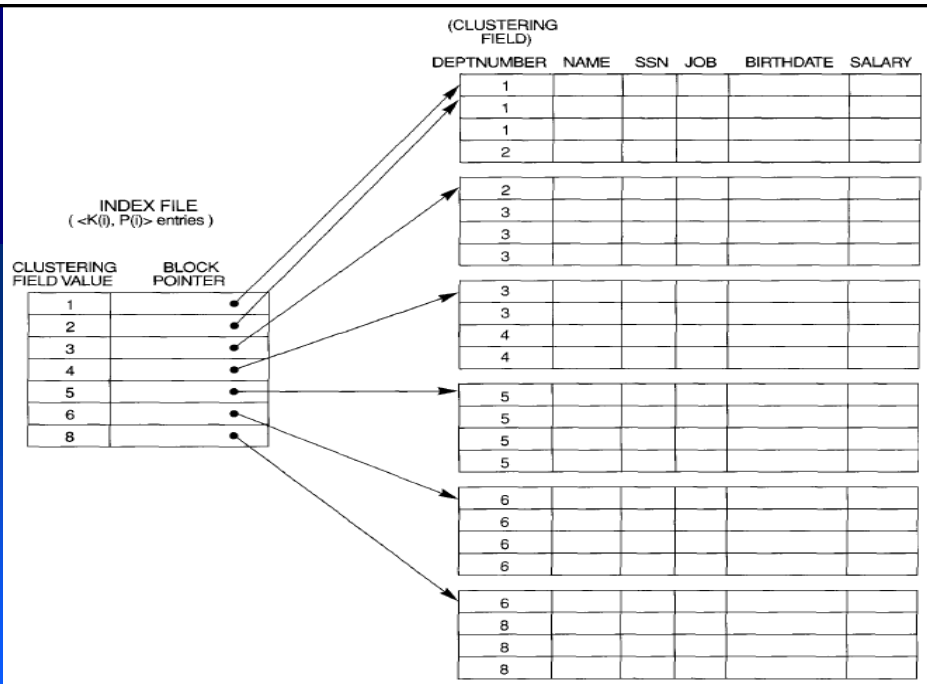


FIGURE 14.2 A clustering index on the DEPTNUMBER ordering nonkey field of an EMPLOYEE file.

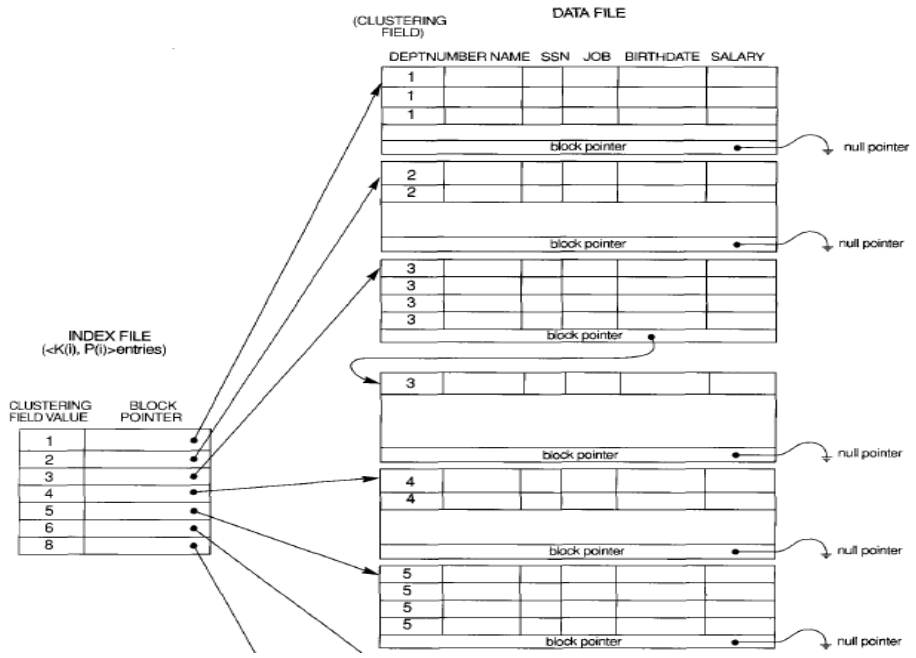


Figure 14.3. Clustering index with a separate block cluster for each group of records that share the same value for the clustering field. [1], pp. 461.

23

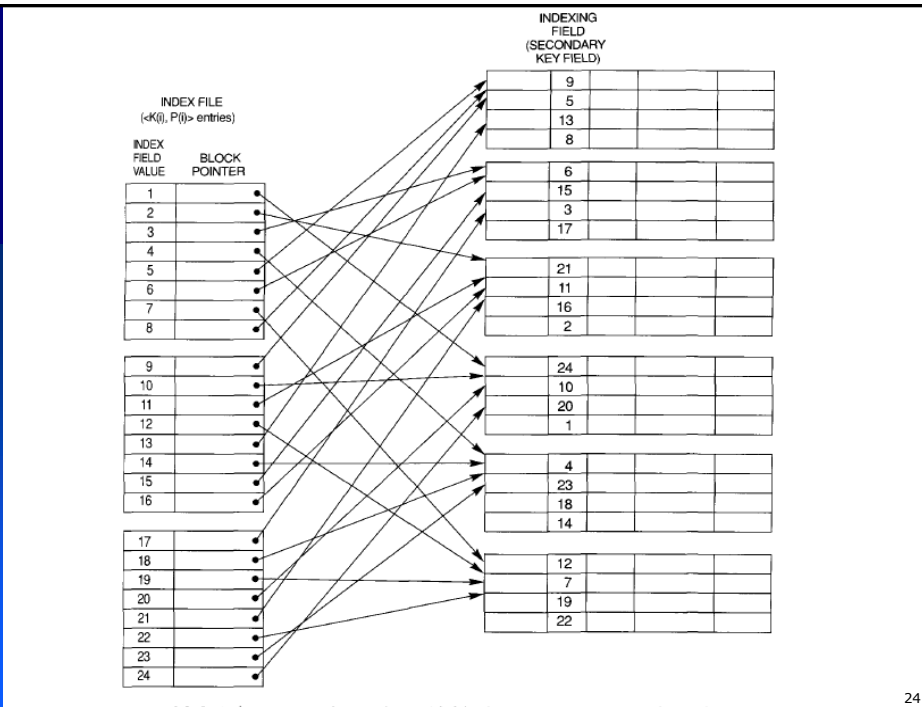


FIGURE 14.4 A dense secondary index (with block pointers) on a nonordering key field of a file.

24

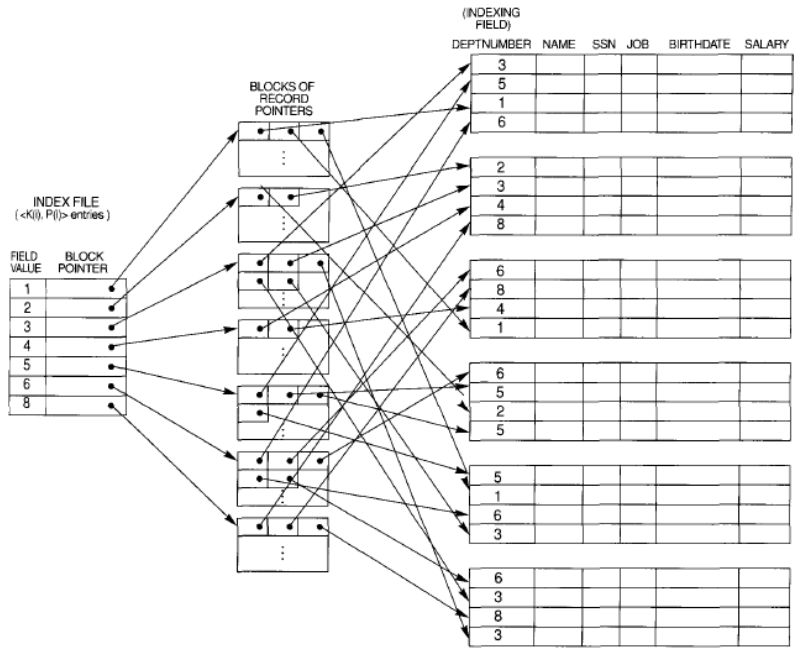


FIGURE 14.5 A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

25

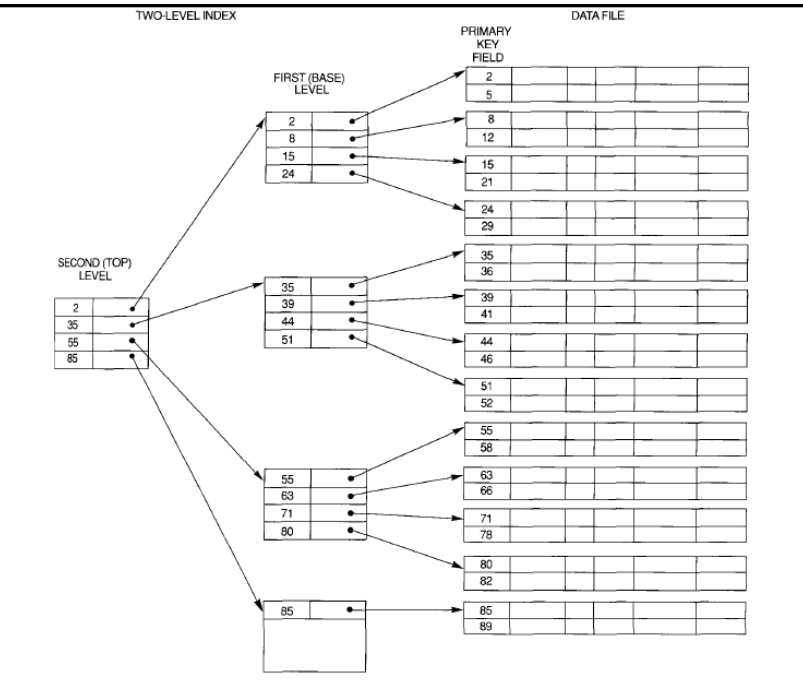


FIGURE 14.6 A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.

26

3.3. Indexing

- Single level ordered index
  - A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file having a specific index field value.
    - Search by a factor of 2: $\log_2 b_i$ block accesses for an index with b_i blocks
- Multilevel index
  - A multilevel index considers the index file, which we will now refer to as the first (or base) level of a multilevel index, as an ordered file with a distinct value for each value K(i) in index entry.
    - Search by a factor bfr_i, the blocking factor for the index (aka the fan-out fo), which is larger than 2: $\log_{fo} b_i$ block accesses

3.3. Indexing – B-tree

- A balanced tree structure used as a dynamic multilevel index to guide the search for records in a data file
  - Supported operators: =, >, >=, <, <=, between
  - Each node is kept between 50 and 100 percent full.
  - Pointers to the data blocks are stored in both internal nodes and leaf nodes.

28

3.3. Indexing – B-tree

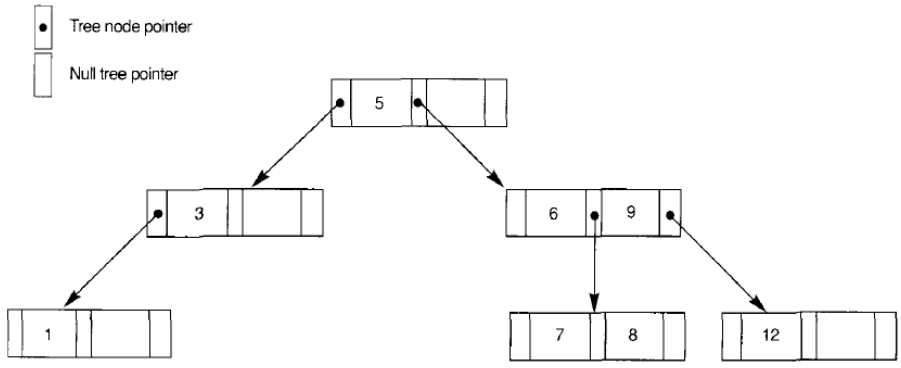


FIGURE 14.9 A search tree of order p = 3.

3.3. Indexing – B-tree

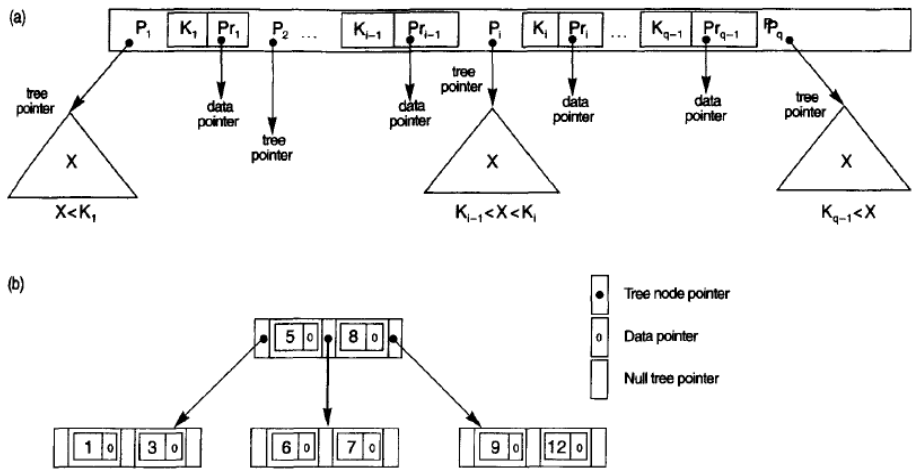


FIGURE 14.10 B-tree structures. (a) A node in a B-tree with q – 1 search values. (b) A B-tree of order p = 3. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

3.3. Indexing – B-tree

- A B-tree of order *p*, when used as an access structure on a key field to search for records in a data file, can be defined as follows:
  - 1. Each internal node is of the form:

$$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, \ldots, <K_{q-1}, Pr_{q-1}>, P_q>$$

    - Where q<=p. Each P_i is a tree pointer – a pointer to another node in the B-tree. Each Pr_j is a data pointer – a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).
  - 2. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$

31

3.3. Indexing – B-tree

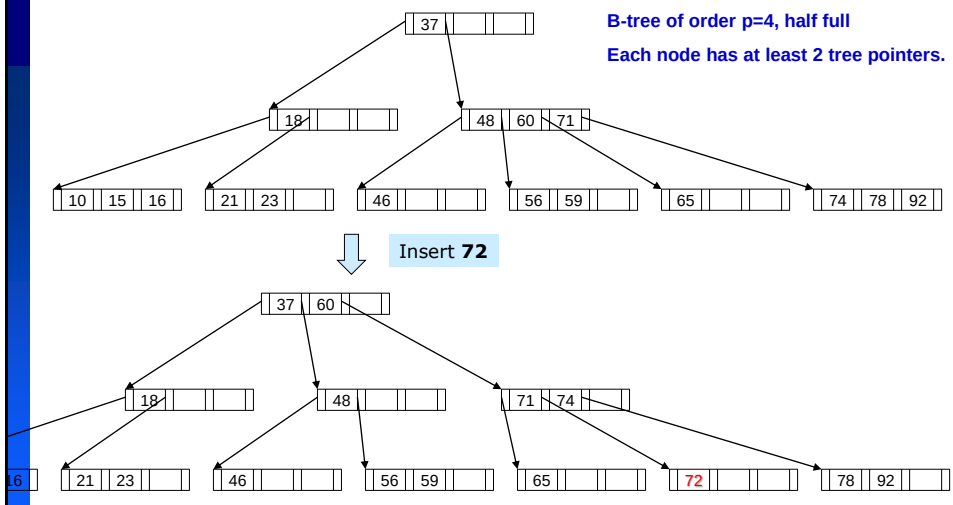
- A B-tree of order p
  - 3. For all search key field values X in the subtree pointed at by Pi, we have:

$$K_{i-1} < X < K_i \text{ for } 1 < i < q; X < K_i \text{ for } i = 1; \text{ and } K_{i-1} < X \text{ for } i = q$$

  - 4. Each node has at most p tree pointers.
  - 5. Each node, except the root and leaf nodes, has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
    - at least $\lceil p/2 \rceil$ tree pointers → how full the tree is: 69%, …
  - 6. A node with q tree pointers, q<=p, has q-1 search key field values (and hence has q-1 data pointers).
  - 7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their tree pointers Pi are null.

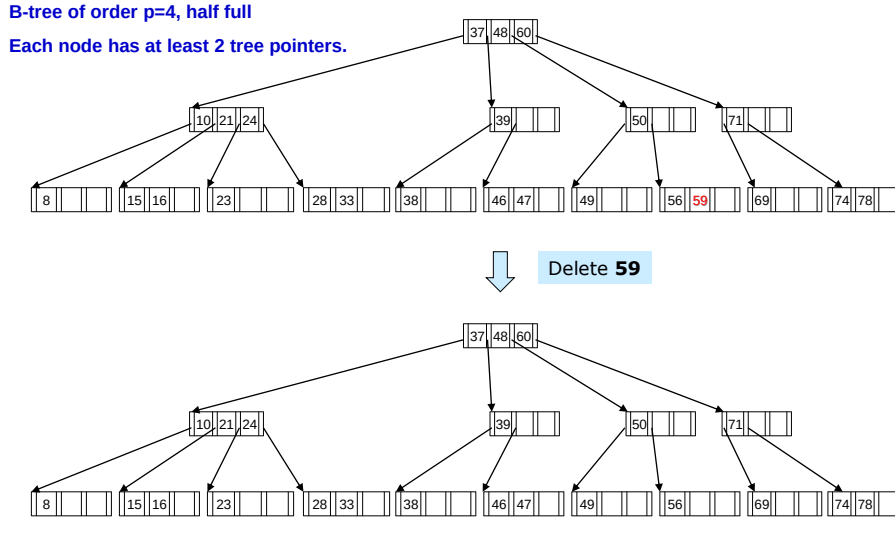
32

3.3. Indexing – B-tree



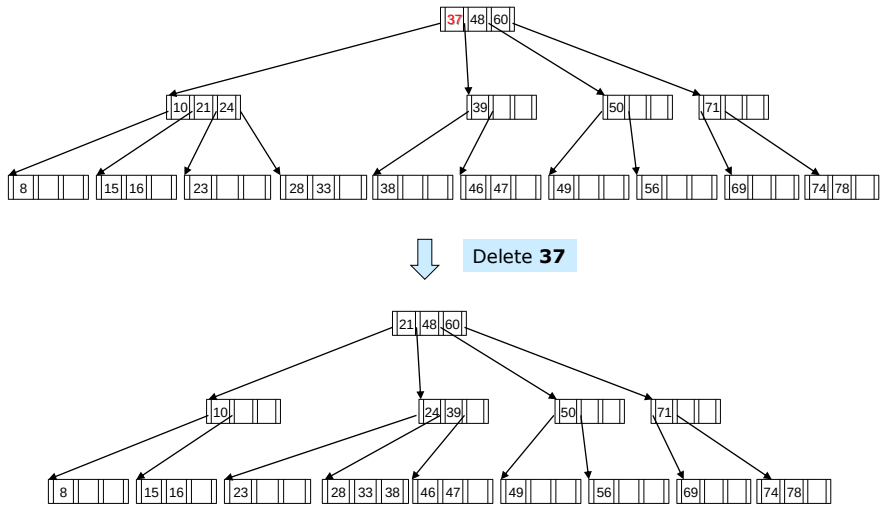
33

3.3. Indexing – B-tree



34

3.3. Indexing – B-tree



3.3. Indexing – B+-tree

- A variation of the B-tree data structure
  - Data pointers are stored only at the leaf nodes of tree.
    - The structure of leaf nodes differs from the structure of internal nodes.
  - The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field.
  - For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

3.3. Indexing – B+-tree

- A variation of the B-tree data structure
  - The leaf nodes of the B+-tree are usually linked together to provide ordered access on the search field to the records.
  - These leaf nodes are similar to the first (base) level of an index.
  - Internal nodes of the B+-tree correspond to the other levels of a multilevel index.
  - Some search field values from the leaf nodes are *repeated* in the internal nodes of the B+-tree to guide the search.

37

3.3. Indexing – B+-tree

- The structure of the *internal nodes* of a B+-tree of order p is as follows:
  - 1. Each internal node is of the form:

    $$<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$$

    - Where $q<=p$ and each P_i is a tree pointer.
  - 2. Within each internal node, $K_1 < K_2 <\ldots< K_{q-1}$.
  - 3. For all search field values X in the subtree pointed at by P_i, we have $K_{i-1}<X<=K_i$ for $1<i<q$; $X<=K_i$ for $i=1$; and $K_{i-1}<X$ for $i=q$.
  - 4. Each internal node has at most p tree pointers.
  - 5. Each internal node, except the root, has at least $\lceil p/2 \rceil$ tree pointers. The root node has at least two tree pointer if it is an internal node.
  - 6. An internal node with q pointers, $q<=p$, has q-1 search field values.

38

3.3. Indexing – B+-tree

- The structure of the leaf nodes of a B+-tree of order p is as follows:
  - 1. Each leaf node is of the form:

    $$<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{next}>$$

    - Where q<=p, each Pr_i is a data pointer, and P_{next} points to the next leaf node of the B+-tree.
  - 2. Within each leaf node, $K_1 < K_2 <...< K_{q-1}$, q<=p.
  - 3. Each Pr_i is a data pointer that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
  - 4. Each leaf node has at least $\lceil p/2 \rceil$ values.
  - 5. All leaf nodes are at the same level.

39

3.3. Indexing – B+-tree

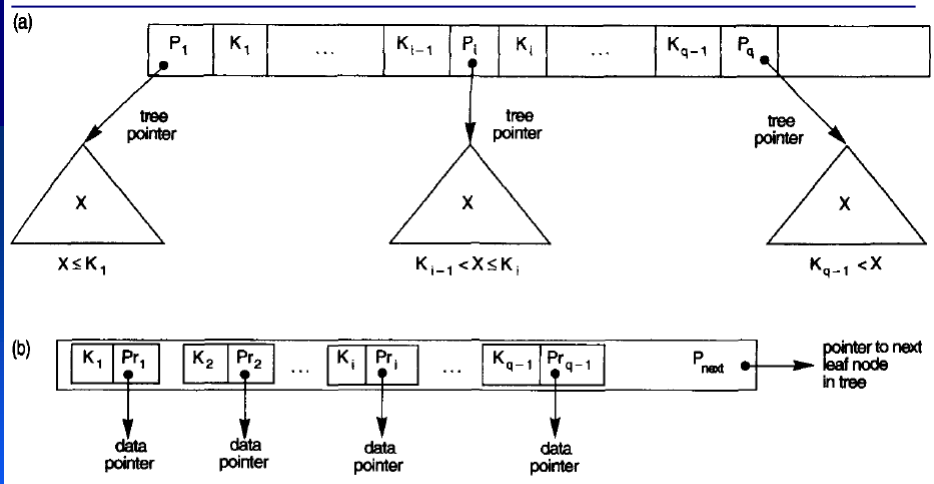
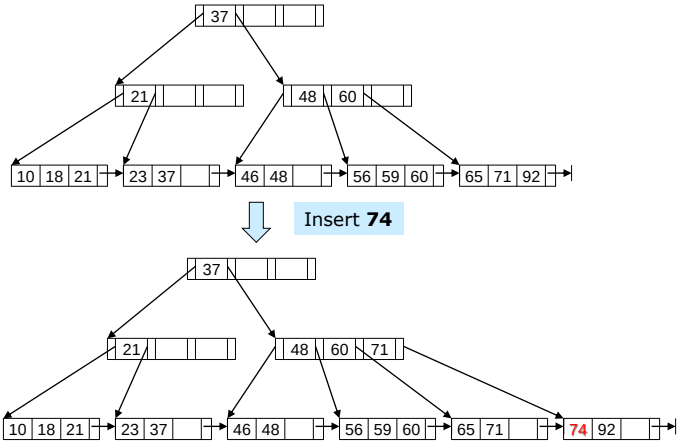


FIGURE 14.11 The nodes of a B+-tree. (a) Internal node of a B+-tree with q − 1 search values. (b) Leaf node of a B+-tree with q−1 search values and q−1 data pointers.

3.3. Indexing – B+-tree



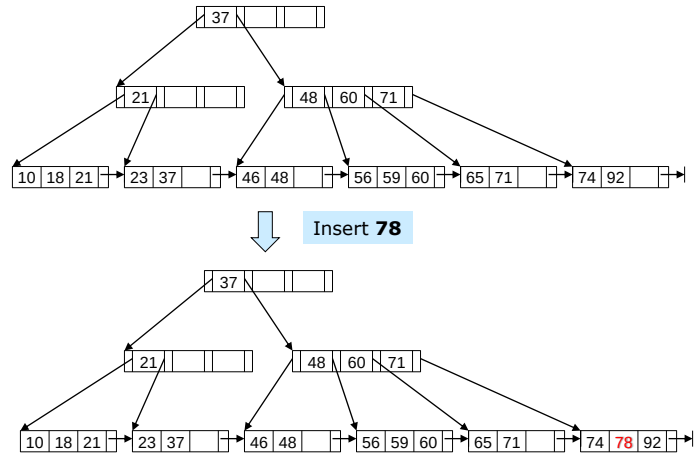
B+-tree with order p = 4, pleaf = 3, half full

Each internal node has at least 2 tree pointers.

Each leaf node has at least 2 data pointers

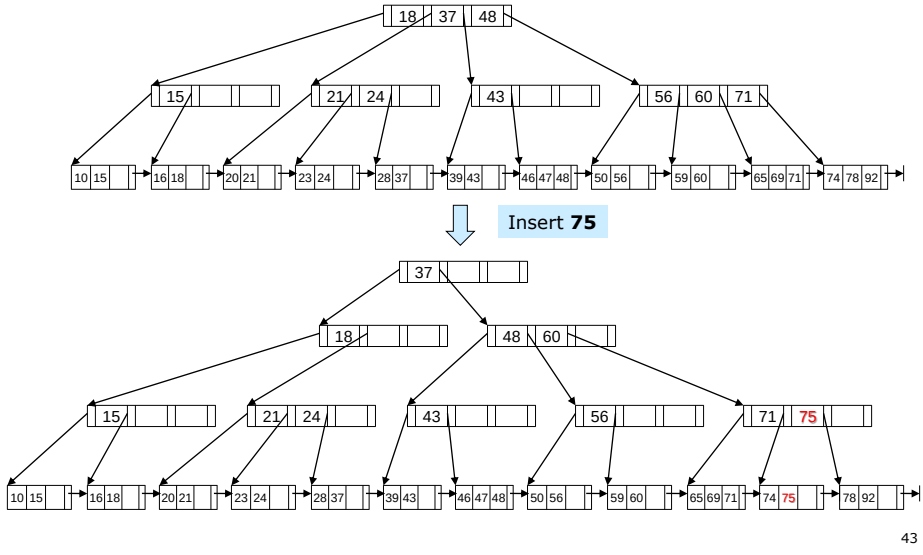
41

3.3. Indexing – B+-tree



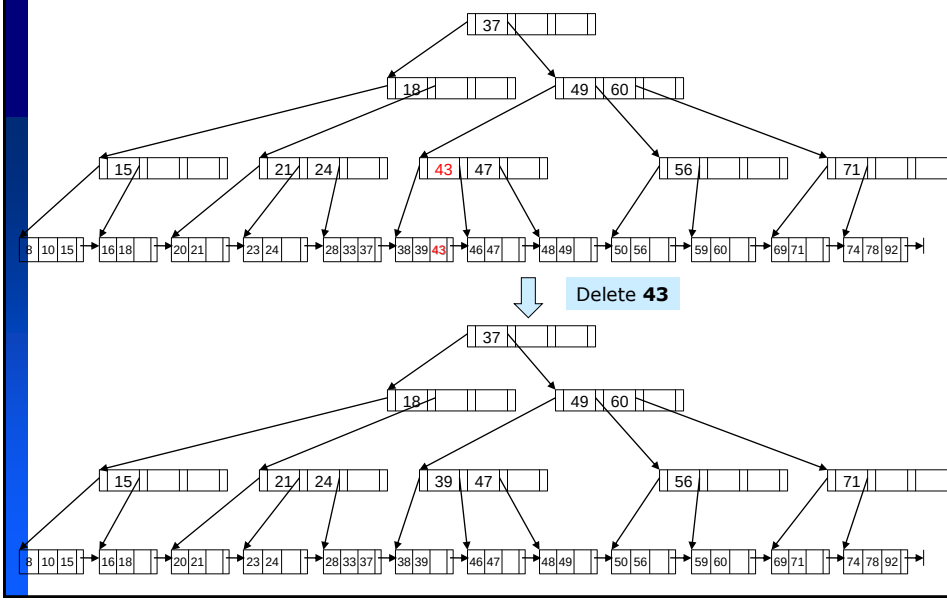
42

3.3. Indexing – B+-tree

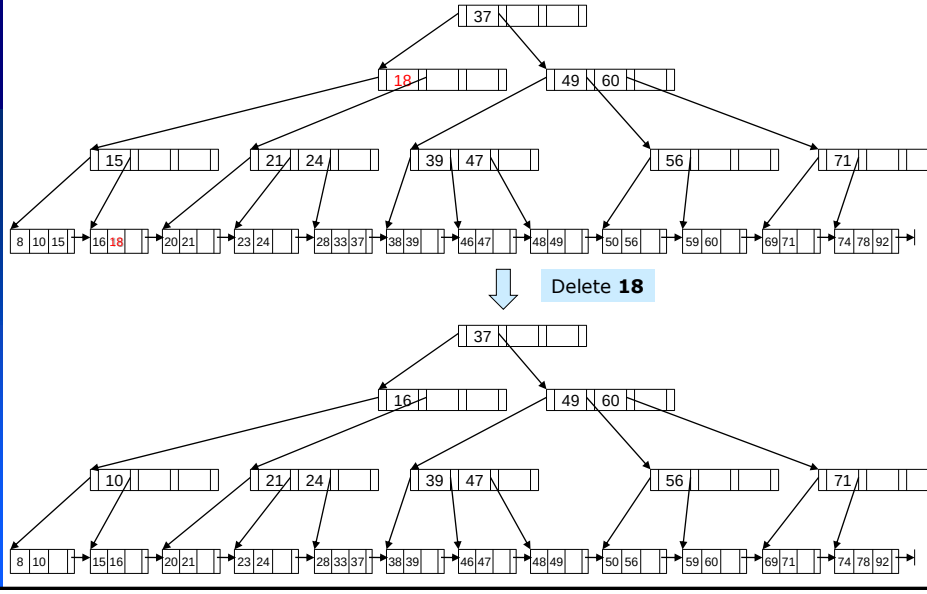


43

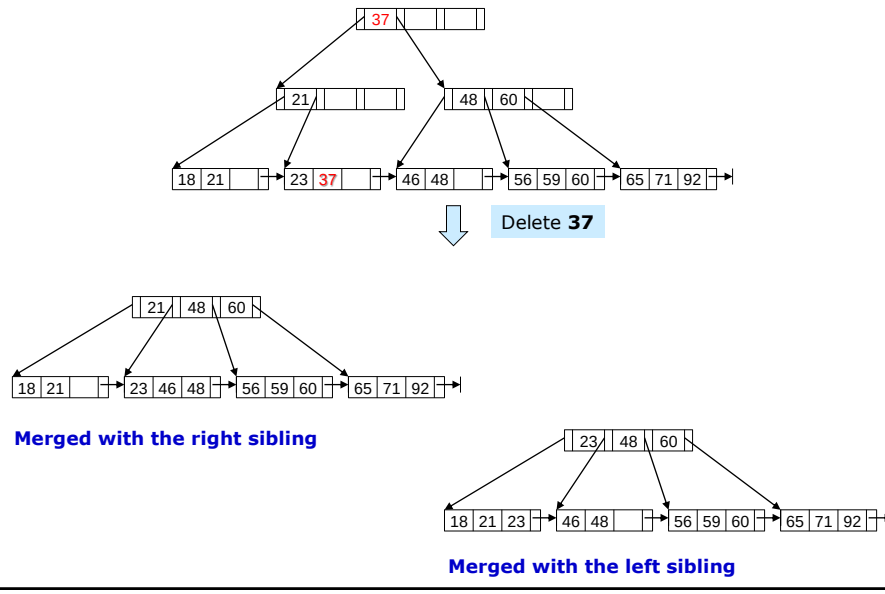
3.3. Indexing – B+-tree



3.3. Indexing – B+-tree



3.3. Indexing – B+-tree



3.3. Indexing – B-tree vs. B+-tree

EXAMPLE 4: Suppose the search field is V = 9 bytes long, the disk block size is B = 512 bytes, a record (data) pointer is P_r = 7 bytes, and a block pointer is P = 6 bytes. Each B-tree node can have *at most* p tree pointers, p − 1 data pointers, and p − 1 search key field values (see Figure 14.10a). These must fit into a single disk block if each B-tree node is to correspond to a disk block. Hence, we must have:

$$(p * P) + ((p - 1) * (P_r + V)) \leq B$$

$$(p * 6) + ((p - 1) * (7 + 9)) \leq 512$$

$$(22 * p) \leq 528$$

We can choose p to be a large value that satisfies the above inequality, which gives p = 23 (p = 24 is not chosen because of the reasons given next).

47

3.3. Indexing – B-tree vs. B+-tree

EXAMPLE 5: Suppose that the search field of Example 4 is a nonordering key field, and we construct a B-tree on this field. Assume that each node of the B-tree is 69 percent full. Each node, on the average, will have p * 0.69 = 23 * 0.69 or approximately 16 pointers and, hence, 15 search key field values. The **average fan-out** fo =16. We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

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

At each level, we calculated the number of entries by multiplying the total number of pointers at the previous level by 15, the average number of entries in each node. Hence, for the given block size, pointer size, and search key field size, a two-level B-tree holds 3840 + 240 + 15 = 4095 entries on the average; a three-level B-tree holds 65,535 entries on the average.

48

3.3. Indexing – B-tree vs. B+-tree

EXAMPLE 6: To calculate the order p of a B^+-tree, suppose that the search key field is V = 9 bytes long, the block size is B = 512 bytes, a record pointer is P_r = 7 bytes, and a block pointer is P = 6 bytes, as in Example 4. An internal node of the B^+-tree can have up to p tree pointers and p − 1 search field values; these must fit into a single block. Hence, we have:

$$(p * P) + ((p - 1) * V) \leq B$$

$$(p * 6) + ((p - 1) * 9) \leq 512$$

$$(15 * p) \leq 521$$

We can choose p to be the largest value satisfying the above inequality, which gives p = 34. This is larger than the value of 23 for the B-tree, resulting in a larger fan-out and more entries in each internal node of a B^+-tree than in the corresponding B-tree. The leaf

49

3.3. Indexing – B-tree vs. B+-tree

more entries in each internal node of a B^+-tree than in the corresponding B-tree. The leaf nodes of the B^+-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the order p_{leaf} for the leaf nodes can be calculated as follows:

$$(p_{leaf} * (P_r + V)) + P \leq B$$

$$(p_{leaf} * (7 + 9)) + 6 \leq 512$$

$$(16 * p_{leaf}) \leq 506$$

It follows that each leaf node can hold up to $p_{leaf} = 31$ key value/data pointer combinations, assuming that the data pointers are record pointers.

50

3.3. Indexing – B-tree vs. B+-tree

EXAMPLE 7: Suppose that we construct a B^+-tree on the field of Example 6. To calculate the approximate number of entries of the B^+-tree, we assume that each node is 69 percent full. On the average, each internal node will have 34 * 0.69 or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold 0.69 * p_{leaf} = 0.69 * 31 or approximately 21 data record pointers. A B^+-tree will have the following average number of entries at each level:

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

For the block size, pointer size, and search field size given above, a three-level B^+-tree holds up to 255,507 record pointers, on the average. Compare this to the 65,535 entries for the corresponding B-tree in Example 5.

51

3.3. Indexing – R-tree

- R-tree index structure
  - Proposed by A. Guttman in 1984
    - A. Guttman. *R-trees – a dynamic index structure for spatial searching*. Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 47-57, 1984.
  - An R-tree is a height-balanced tree similar to a B-tree with index records in its leaf nodes containing pointers to data objects.
  - Nodes correspond to disk pages if the index is disk-resident, and the structure is designed so that a spatial search requires visiting only a small number of nodes.
  - The index is completely dynamic; inserts and deletes can be intermixed with searches and no periodic reorganization is required.

52

3.3. Indexing – R-tree

- Leaf nodes in an R-tree contain index record entries of the form: (*I*, *tuple-identifier*).
  - *tuple-identifier*: refers to a tuple in the database.
  - *I*: an *n*-dimensional rectangle which is the bounding box of the spatial object indexed
    - $I = (I_0, I_1, \ldots, I_{n-1})$
- Non-leaf nodes contain entries of the form: (*I*, *child-pointer*).
  - *child-pointer*: is the address of a lower node in the R-tree.
  - *I*: covers all rectangles in the lower node's entries.

53

- M: the maximum number of entries that will fit in one node
- m ($m \leq M/2$): a parameter specifying the minimum number of entries in a node
- ***R-tree*** satisfies the following properties:
  - Every leaf node contains between m and M index records unless it is the root.
  - For each index record (*I*, *tuple-identifier*) in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object.
  - Every non-leaf node has between m and M children unless it is the root.
  - For each entry (*I*, *child-pointer*) in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.
  - The root node has at least two children unless it is a leaf.
  - All leaves appear on the same level.

54

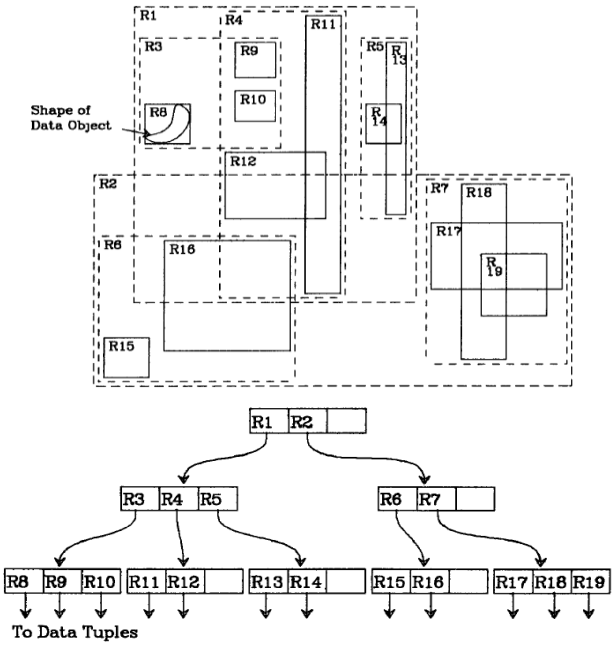


Figure 3.1.
A. Guttman. *R-trees – a dynamic index structure for spatial searching*. Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 47-57, 1984.

55

3.3. Indexing – R-tree

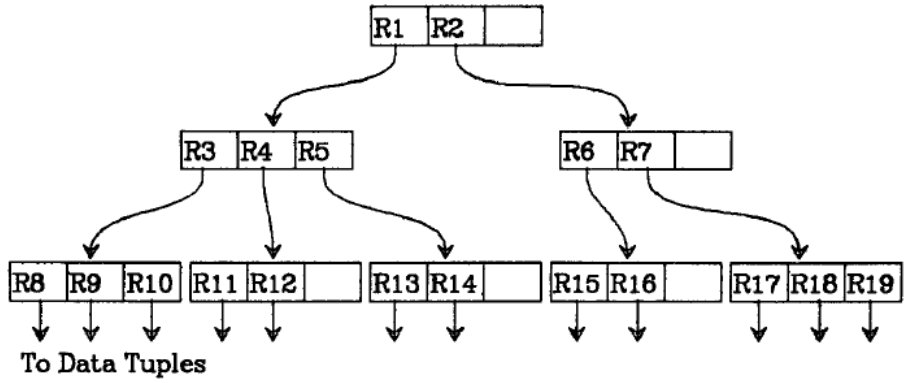
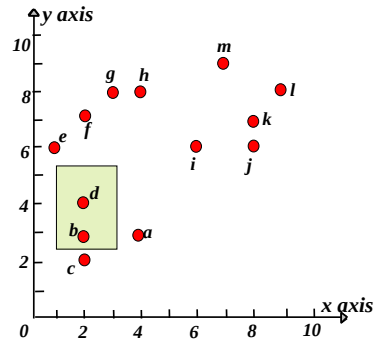


Figure 3.1. (a)
A. Guttman. *R-trees – a dynamic index structure for spatial searching*. Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 47-57, 1984.

56

3.3. Indexing – R-tree

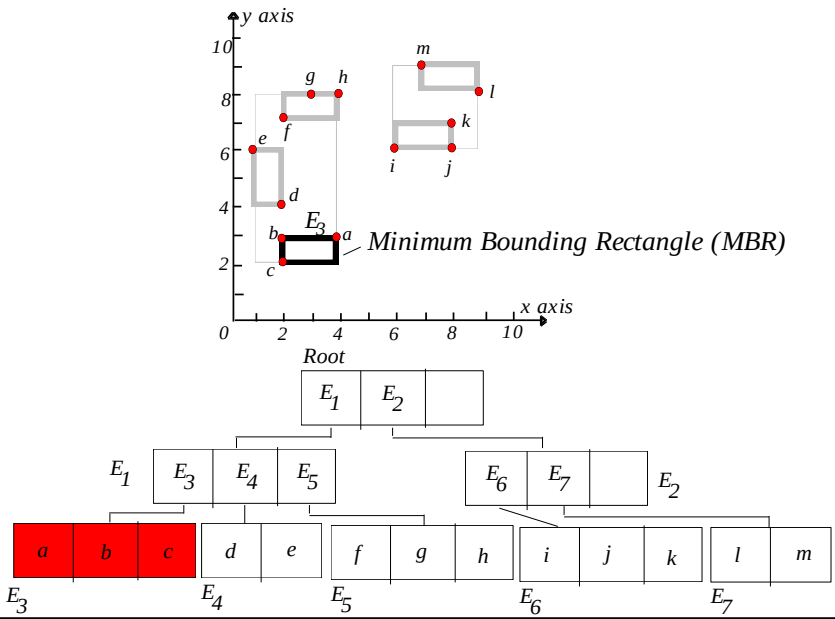


Range query: find the objects within a range.

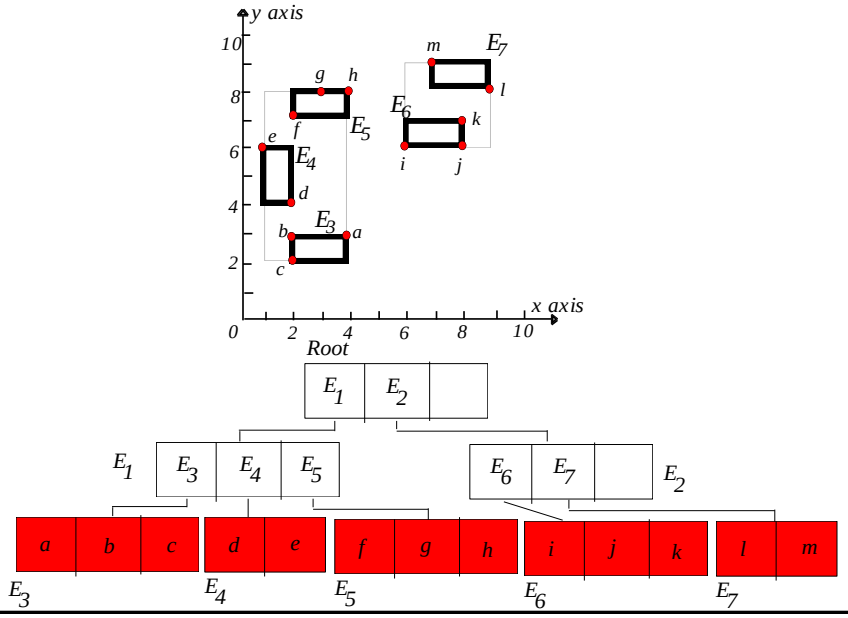
Adopted from Ch28 - Spatial data management slides (R_tree.ppt)

57

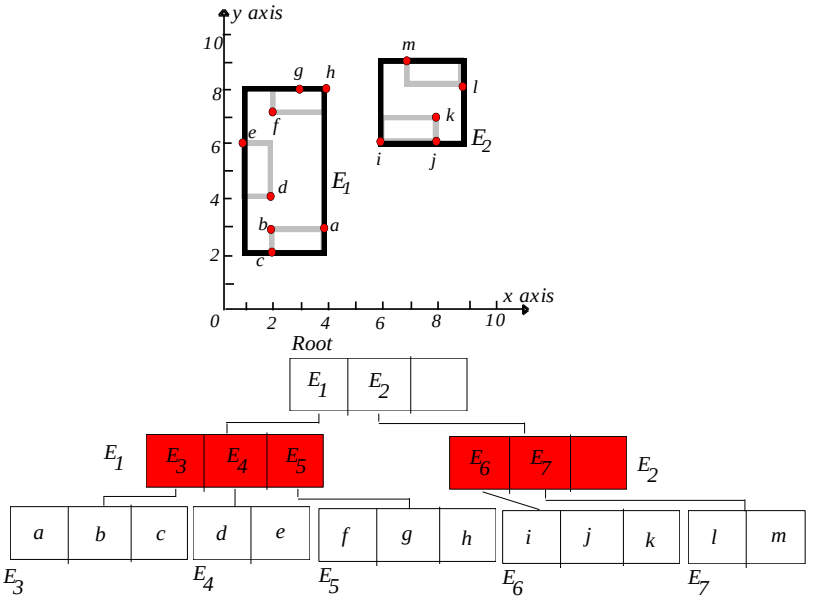
R-Tree: Clustering by Proximity



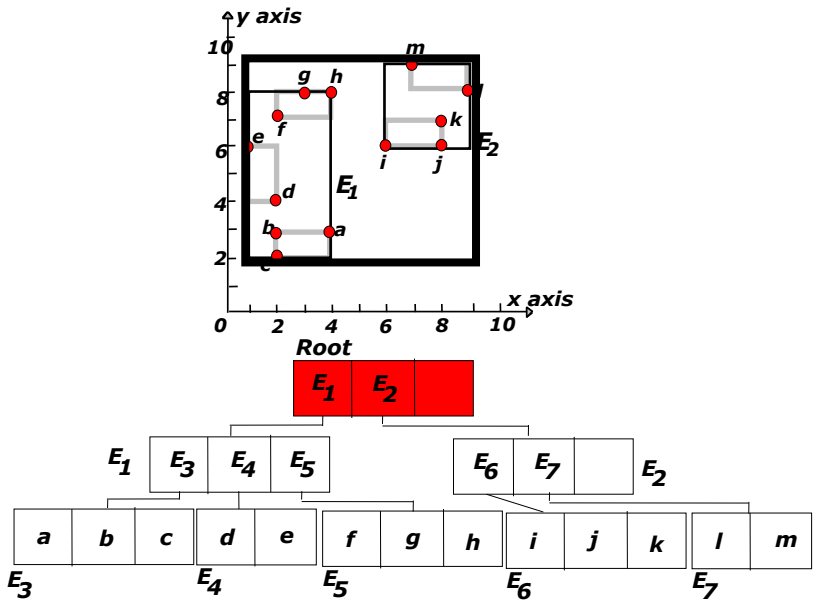
R-Tree


y axis
x axis
Root
E1
E2
E3
E4
E5
E6
E7
a
b
c
d
e
f
g
h
i
j
k
l
m

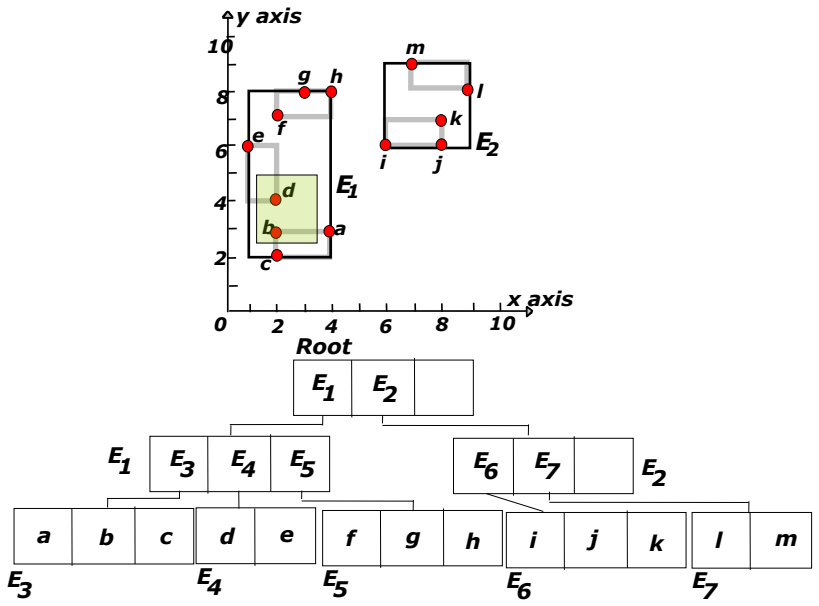
R-Tree


y axis
x axis
Root
E1
E2
E3
E4
E5
E6
E7
a
b
c
d
e
f
g
h
i
j
k
l
m

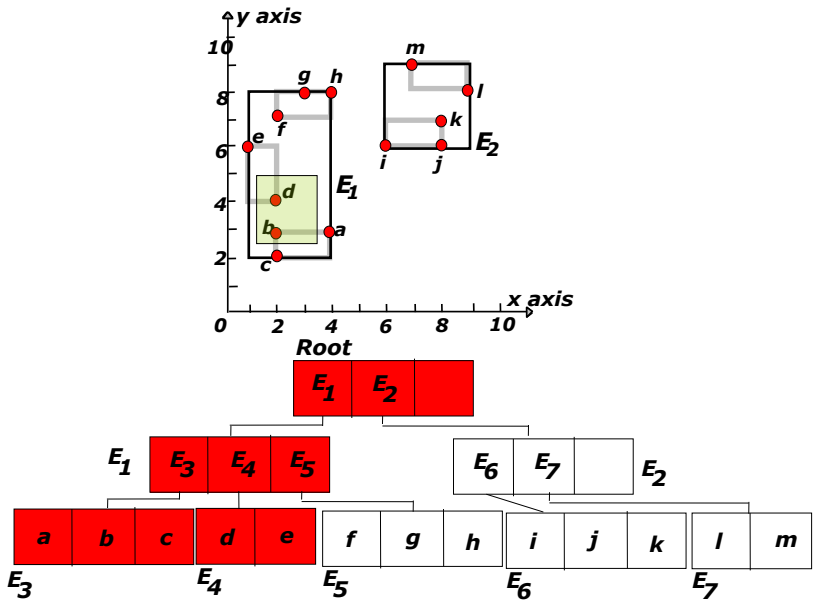
R-Tree



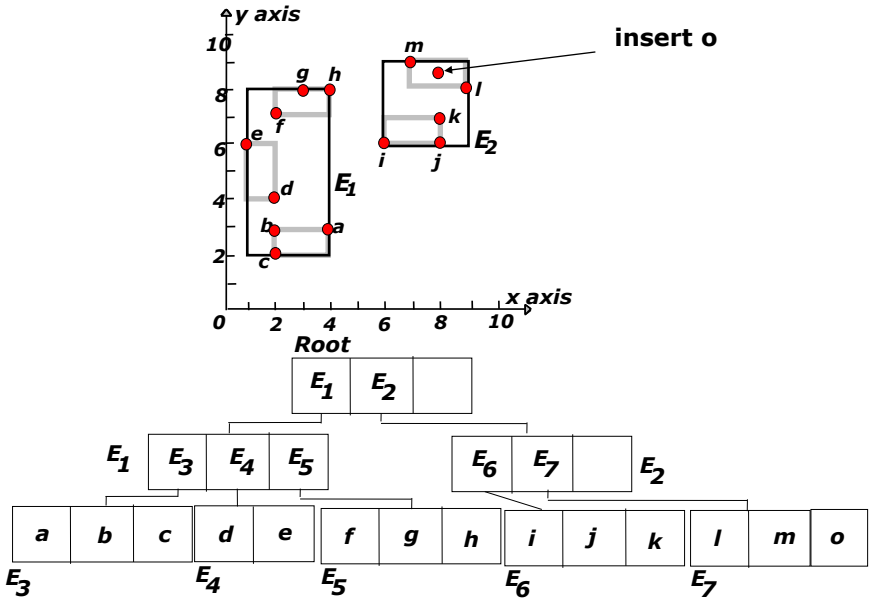
Range Query



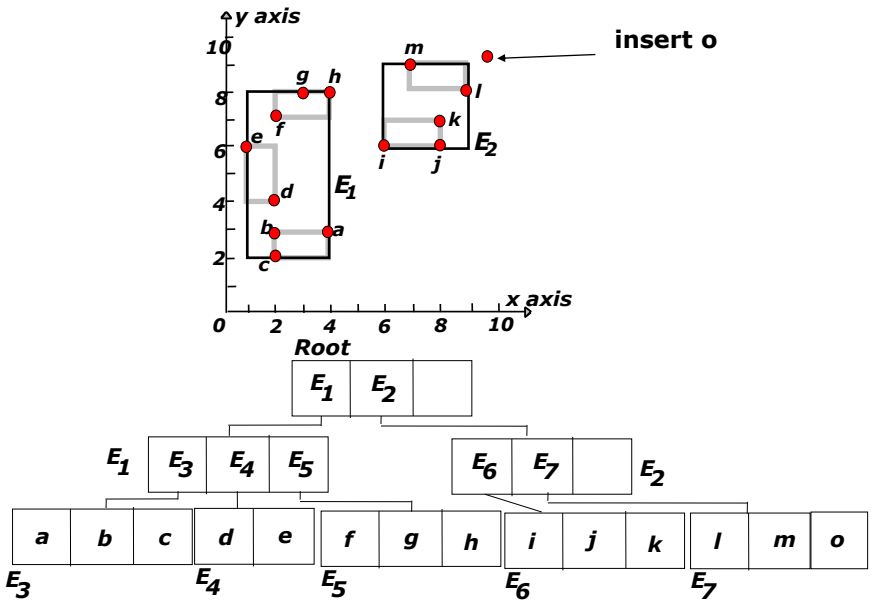
Range Query



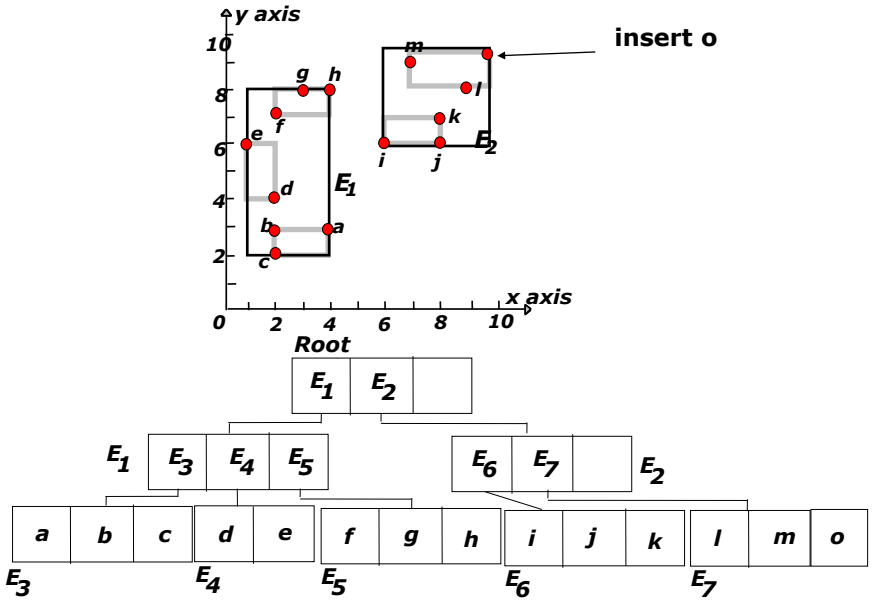
Insert: no split, no enlargement



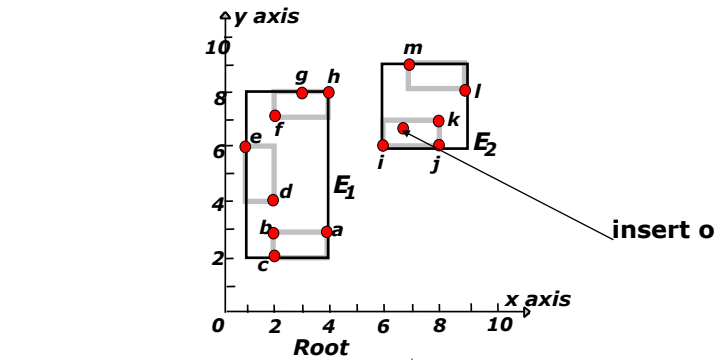
Insert: no split, but enlargement

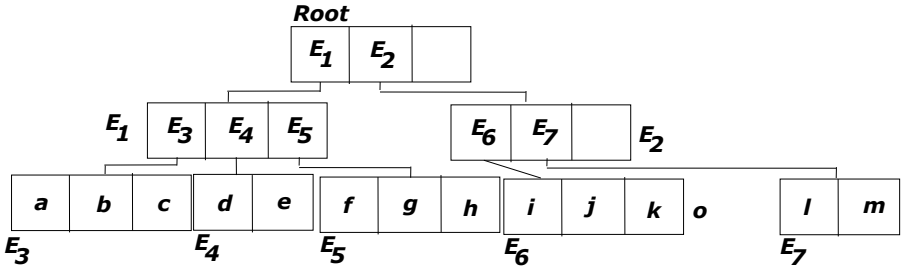


Insert: no split, but enlargement

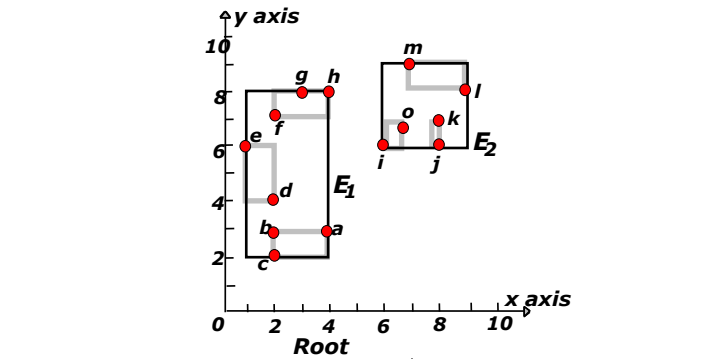


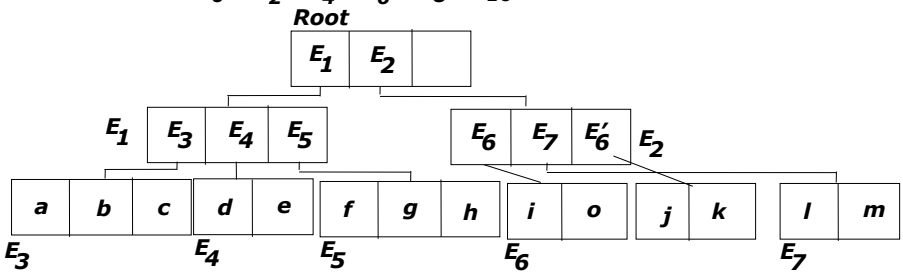
Insert: split





Insert: split





3.4. SQL (structured query language)

- Data Definition Language (DDL): Create, Alter, Drop
- Data Manipulation Language (DML): Select, Insert, Update, Delete
- Data Control Language (DCL): Commit, Rollback, Grant, Revoke

69

3.4. SQL (structured query language)

```
CREATE TABLE TableName
{(colName dataType [NOT NULL] [UNIQUE]
[DEFAULT defaultOption]
[CHECK searchCondition] [,...]}
[PRIMARY KEY (listOfColumns),]
{[UNIQUE (listOfColumns),] […,]}
{[FOREIGN KEY (listOfFKColumns)
  REFERENCES ParentTableName [(listOfCKColumns)],
  [ON UPDATE referentialAction]
  [ON DELETE referentialAction ]] [,…]}
{[CHECK (searchCondition)] [,…] })
```

70

3.4. SQL (structured query language)

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

71

3.4. SQL (structured query language)

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

72

3.4. SQL (structured query language)

SELECT [DISTINCT | ALL]

{* | [columnExpression [AS newName]] [,...] }

FROM TableName [alias] [, ...]

[WHERE condition]

[GROUP BY columnList]

[HAVING condition]

[ORDER BY columnList]

73

3.4. SQL (structured query language)

For each project *on which more than two employees work*, retrieve the project number, project name, and the number of employees who work on that project.

```
SELECT      PNUMBER, PNAME, COUNT (*)
FROM        PROJECT, WORKS_ON
WHERE       PNUMBER=PNO
GROUP BY    PNUMBER, PNAME
HAVING      COUNT (*) > 2
```

74

3.5. Algorithms for query processing

- Algorithms for external sorting
- Algorithms for select operation
- Algorithms for join operation
- Algorithms for project operation
- Algorithms for set operations

75

3.5. Algorithms for query processing

- Algorithms for external sorting
  - Sorting is one of the primary algorithms used in query processing.
    - Whenever an SQL query specifies an ORDER BY-clause, the query result must be sorted.
    - Sorting is also a key component in sort-merge algorithms used for JOIN and other operations (UNION and INTERSECTION), and in duplicate elimination algorithms for the PROJECT operation (when an SQL query specifies the DISTINCT option in the SELECT clause).
  - External sorting refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.

76

3.5. Algorithms for query processing

- Algorithms for external sorting
  - Sorting is one of the primary algorithms used in query processing.
    - Whenever an SQL query specifies an ORDER BY-clause, the query result must be sorted.
    - Sorting is also a key component in sort-merge algorithms used for JOIN and other operations (UNION and INTERSECTION), and in duplicate elimination algorithms for the PROJECT operation (when an SQL query specifies the DISTINCT option in the SELECT clause).
  - External sorting refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.

77

3.5. Algorithms for query processing

- Algorithms for external sorting
  - The typical external sorting algorithm uses a sort-merge strategy
    - Sort small subfiles – called runs – of the main file
      - Runs (portions or pieces) of the file that can fit in the available buffer space are read into main memory, sorted using an internal sorting algorithm, written back to disk as temporary sorted subfiles (or runs).
    - Merge the sorted runs, creating larger sorted subfiles that are merged in turn
      - The sorted runs are merged during one or more passes.
      - In each pass, one buffer block is needed to hold one block from each of the runs being merged, and one block is needed for containing one block of the merge result.

78

3.5. Algorithms for query processing

- Algorithms for select operation
  - Many options for executing a SELECT operation; some depend on the file having specific access paths and may apply only to certain types of selection conditions
  - Examples
    - (OP1): $\sigma_{SSN='123456789'}(EMPLOYEE)$
    - (OP2): $\sigma_{DNUMBER>5}(DEPARTMENT)$
    - (OP3): $\sigma_{DNO=5}(EMPLOYEE)$
    - (OP4): $\sigma_{DNO=5\ AND\ SALARY>30000\ AND\ SEX='F'}(EMPLOYEE)$
    - (OP5): $\sigma_{ESSN='123456789'\ AND\ PNO=10}(WORKS\_ON)$
    - (OP49): $\sigma_{DNO=5\ OR\ SALARY>30000\ OR\ SEX='F'}(EMPLOYEE)$

79

3.5. Algorithms for query processing

- Algorithms for select operation
  - S1. *Linear search (brute force)*: retrieve every record in the file, and test whether its attribute values satisfy the selection condition.
  - S2. *Binary search to retrieve a single record*: if the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search can be used.
  - S3. *Using a primary index (or hash key) to retrieve a single record*: if the selection condition involves an equality comparison on a key attribute with a primary index (or hash key), use the primary index (or hash key) to retrieve the record.

80

3.5. Algorithms for query processing

- Algorithms for select operation
  - S4. *Using a primary index to retrieve multiple records*: if the comparison condition is >, >=, <, or <= on a key field with a primary index, use the index to find the record satisfying the corresponding equality condition, then retrieve all subsequent records in the (ordered) file.
  - S5. *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, use the index to retrieve all the records satisfying the condition.

81

3.5. Algorithms for query processing

- Algorithms for select operation
  - S6. *Using a secondary (B+-tree) index on an equality comparison*: this search method can be used to retrieve a single record if the indexing field is a key (has unique values) or to retrieve multiple records if the indexing field is not a key. This can also be used for comparisons involving >, >=, <, or <= (range queries).
  - S7. *Conjunctive selection using an individual index*: if an attribute involved in any single simple condition in the conjunctive condition has an access path that permits the use of one of the methods S2-S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.

82

3.5. Algorithms for query processing

- Algorithms for select operation
  - S8. *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, use the index directly.
  - S9. *Conjunctive selection by intersection of record pointers*: if secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the set of record pointers that satisfy the individual condition. The intersection of these sets of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfied the remaining conditions.

3.5. Algorithms for query processing

- Algorithms for select operation
  - Selection with a *disjunctive* condition
    - Compared to a conjunctive selection condition, a disjunctive condition (where simple conditions are connected by the OR logical connective rather than by AND) is much harder to process and optimize.
    - Little optimization can be done because the records satisfying the disjunctive condition are the union of the records satisfying the individual conditions.
    - If any one of the conditions does not have an access path, use the brute force linear search approach.
    - Only if an access path exists on every condition, optimize the selection by retrieving the records satisfying each condition and then applying the union operation to eliminate duplicates.

3.5. Algorithms for query processing

- Algorithms for select operation
  - Examples
    - (OP1): $\sigma_{SSN='123456789'}$(EMPLOYEE)
    - (OP2): $\sigma_{DNUMBER>5}$(DEPARTMENT)
    - (OP3): $\sigma_{DNO=5}$(EMPLOYEE)
    - (OP4): $\sigma_{DNO=5\ AND\ SALARY>30000\ AND\ SEX='F'}$(EMPLOYEE)
    - (OP5): $\sigma_{ESSN='123456789'\ AND\ PNO=10}$(WORKS_ON)
    - (OP49): $\sigma_{DNO=5\ OR\ SALARY>30000\ OR\ SEX='F'}$(EMPLOYEE)

  → What selection methods should be applied on each example?

85

3.5. Algorithms for query processing

- Algorithms for join operation
  - Form: R $\triangleright\triangleleft_{A=B}$ S
    - A and B are domain-compatible attributes of R and S, respectively
  - Example:
    - (OP6): EMPLOYEE $\triangleright\triangleleft_{DNO=DNUMBER}$DEPARTMENT
    - (OP7): DEPARTMENT $\triangleright\triangleleft_{MGRSSN=SSN}$EMPLOYEE
  - Methods:
    - J1. Nested-loop join (brute force)
    - J2. Single-loop join
    - J3. Sort-merge join
    - J4. Hash-join

86

3.5. Algorithms for query processing

- Algorithms for join operation
  - J1. *Nested-loop join (brute force)*: For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition t[A] = s[B].
  - J2. *Single-loop join (using an access structure to retrieve the matching records)*: If an index (or hash key) exists for one of the two join attributes-say, B of S-retrieve each record t in R, one at a time (single loop), and then use the access structure to retrieve directly all matching records s from S that satisfy s[B] =t[A].

87

3.5. Algorithms for query processing

- Algorithms for join operation
  - J3. *Sort-merge join*: If the records of R and S are *physically sorted* (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B. If the files are not sorted, they may be sorted first by using external sorting. In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file-unless both A and B are nonkey attributes, in which case the method needs to be modified slightly.

88

3.5. Algorithms for query processing

- Algorithms for join operation
  - J4. *Hash-join*: The records of files R and S are both hashed to the same hash file, using the same hashing function on the join attributes A of R and B of S as hash keys. First, a single pass through the file with fewer records (say, R) hashes its records to the hash file buckets; this is called the partitioning phase, since the records of R are partitioned into the hash buckets. In the second phase, called the probing phase, a single pass through the other file (S) then hashes each of its records to *probe* the appropriate bucket, and that record is combined with all matching records from R in that bucket. This simplified description of hash-join assumes that the smaller of the two files *fits entirely* into *memory buckets* after the first phase.

3.5. Algorithms for query processing

- Algorithms for project operation
  - A PROJECT operation $\pi_{<\text{attribute list}>}(R)$ is straightforward to implement if <attribute list> includes a key of relation R, because in this case the result of the operation will have the same number of tuples as R, but with only the values for the attributes in <attribute list> in each tuple.
  - If <attribute list> does not include a key of R, *duplicate tuples must be eliminated.*
    - Sorting the result of the operation and then eliminating duplicate tuples, which appear consecutively after sorting.
    - Hashing: as each record is hashed and inserted into a bucket of the hash file in memory, it is checked against those already in the bucket; if it is a duplicate, it is not inserted.

3.5. Algorithms for query processing

- Algorithms for set operations (U, ∩, -)
  - Apply only to union-compatible relations, which have the same number of attributes and the same attribute domains
  - Use variations of the *sort-merge* technique: the two relations are sorted on the same attributes, and, after sorting, a single scan through each relation is sufficient to produce the result
    - Implement the UNION operation, RUS, by scanning and merging both sorted files concurrently, and whenever the same tuple exists in both relations, only one is kept in the merged result
    - For the INTERSECTION operation, R∩S, keep in the merged result only those tuples that appear in *both relations*

91

3.6. Rule-based query optimization

- *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*:

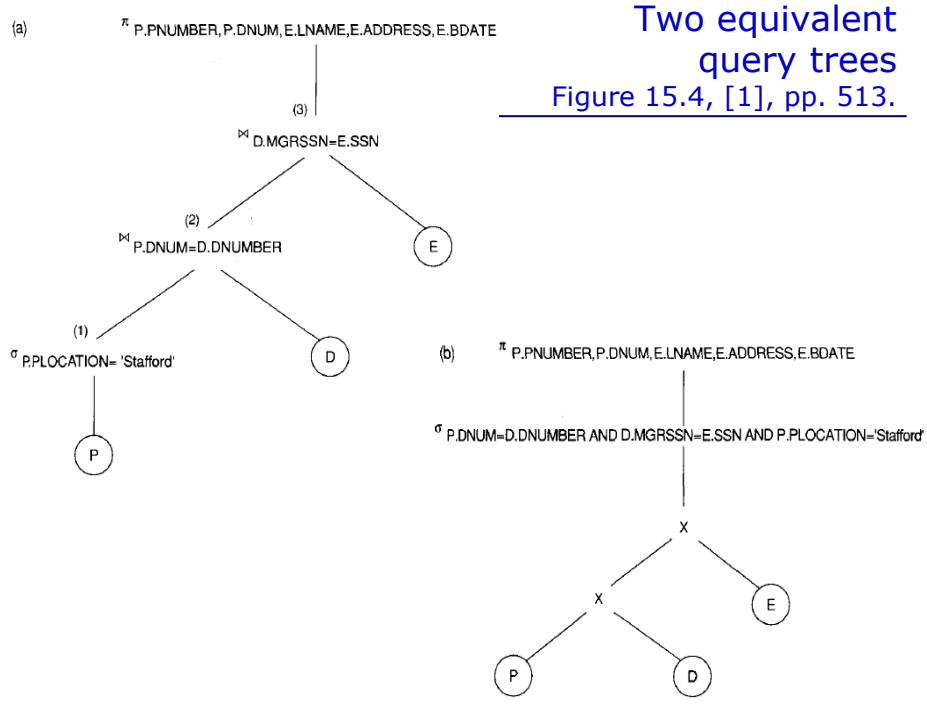
```
SELECT  P.NUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE
FROM PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE  P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN AND
    P.PLOCATION='STAFFORD';
```

- *Relational algebra expression*:

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

Two equivalent query trees

Figure 15.4, [1], pp. 513.



3.6. Rule-based query optimization

- 1. The main heuristic is to apply first the operations that reduce the size of intermediate results.
- 2. Perform SELECT operations as early as possible to reduce the number of tuples and perform PROJECT operations as early as possible to reduce the number of attributes. (This is done by moving SELECT and PROJECT operations as far down the tree as possible.)
- 3. The SELECT and JOIN operations that are *most restrictive* should be executed before other similar operations. (This is done by reordering the leaf nodes of the tree among themselves and adjusting the rest of the tree appropriately.)

94

General Transformation Rules for Relational Algebra Operations:

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

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

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

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

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

$$\pi_{List1}(\pi_{List2}(\ldots(\pi_{Listn}(R))\ldots)) = \pi_{List1}(R)$$

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

$$\pi_{A1, A2,., An}(\sigma_c(R)) = \sigma_c(\pi_{A1, A2,., An}(R))$$

95

General Transformation Rules for Relational Algebra Operations:

5. Commutativity of ⋈ (and ×): The operation is commutative as the × operation:

$$R \bowtie S = S \bowtie R; \quad R \times S = S \times R$$

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

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

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

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

96

General Transformation Rules for Relational Algebra Operations:

7. Commuting π with $\bowtie$ (or $\times$): Suppose that the projection list is L = {A1, ..., An, B1, ..., Bm}, where A1, ..., An are attributes of R and B1, ..., Bm are attributes of S. If the join condition *c* involves only attributes in *L*, the two operations can be commuted as follows:

$$\pi_L(R \bowtie_C S) = (\pi_{A1, ..., An}(R)) \bowtie_C (\pi_{B1, ..., Bm}(S))$$

If the join condition *c* contains additional attributes not in *L*, these must be added to the projection list, and a final operation is needed.

97

General Transformation Rules for Relational Algebra Operations:

8. Commutativity of set operations: The set operations $\cup$ and $\cap$ are commutative but – is not.

9. Associativity of $\bowtie$, x, $\cup$, and $\cap$: These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have

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

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

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

98

General Transformation Rules for Relational Algebra Operations:

11. The π operation commutes with $\cup$.

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

12. Converting a (σ, $\times$) sequence into $\rhd\lhd$: If the condition *c* of a σ that follows a $\times$ corresponds to a join condition, convert the (σ, $\times$) sequence into a $\rhd\lhd$ as follows:

$$(\sigma_C(R \times S)) = (R \rhd\lhd_C S)$$

13. Other transformations for NOT, AND, OR.

99

Outline of a Heuristic Algebraic Optimization Algorithm

1. Using rule 1, break up any select operations with conjunctive conditions into a cascade of select operations.

2. Using rules 2, 4, 6, and 10 concerning the commutativity of select with other operations, move each select operation as far down the query tree as is permitted by the attributes involved in the select condition.

3. Using rule 9 concerning associativity of binary operations, rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive select operations are executed first in the query tree representation.

100

Outline of a Heuristic Algebraic Optimization Algorithm

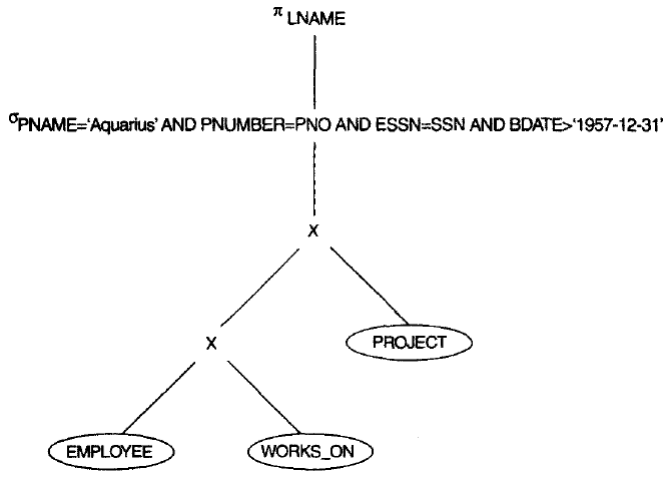
4. Using Rule 12, combine a cartesian product operation with a subsequent select operation in the tree into a join operation.

5. Using rules 3, 4, 7, and 11 concerning the cascading of project and the commuting of project with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new project operations as needed.

6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

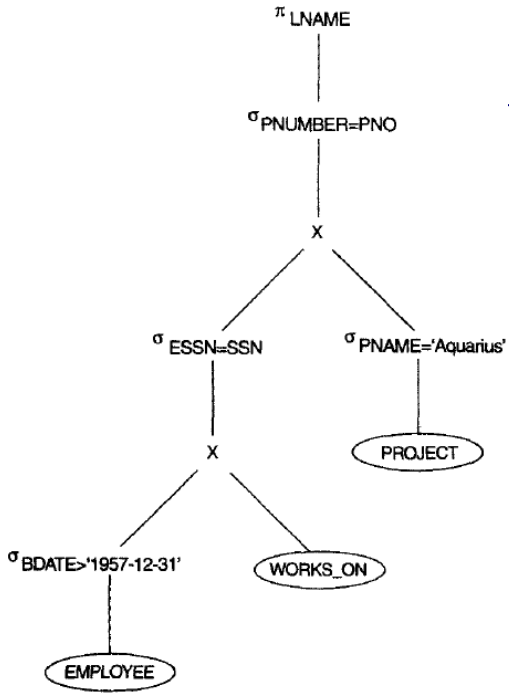
101

3.6. Rule-based query optimization



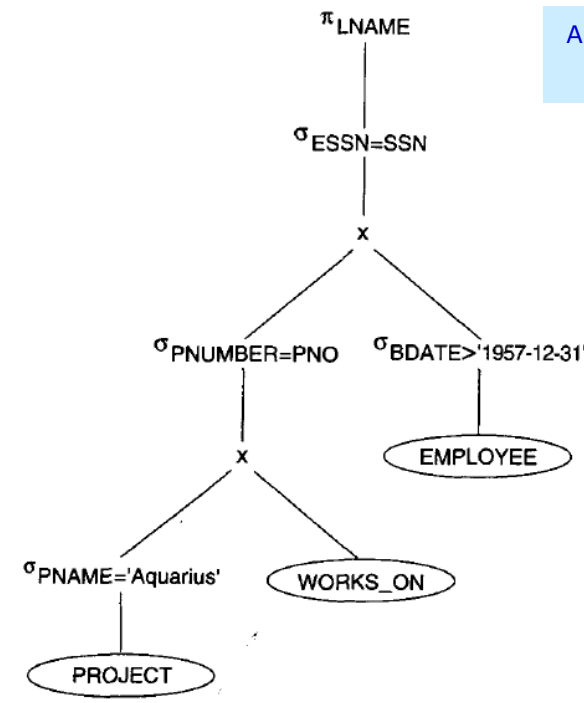
Assume: original query tree.

102



Moving SELECT operations *down* the query tree

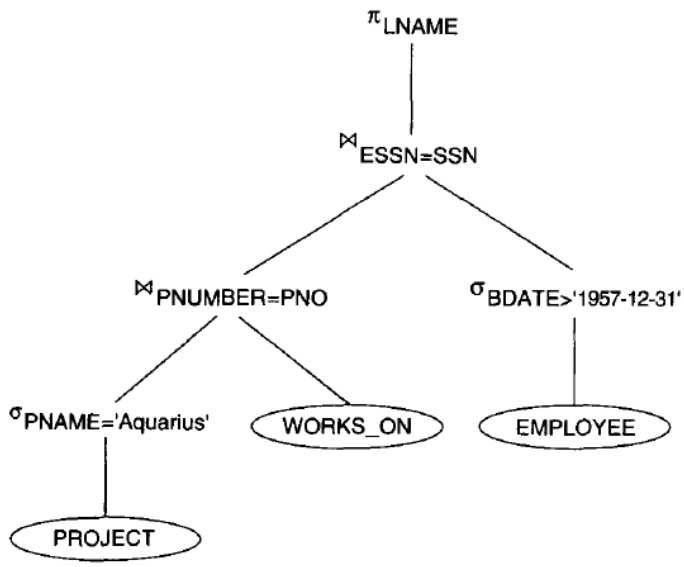
103



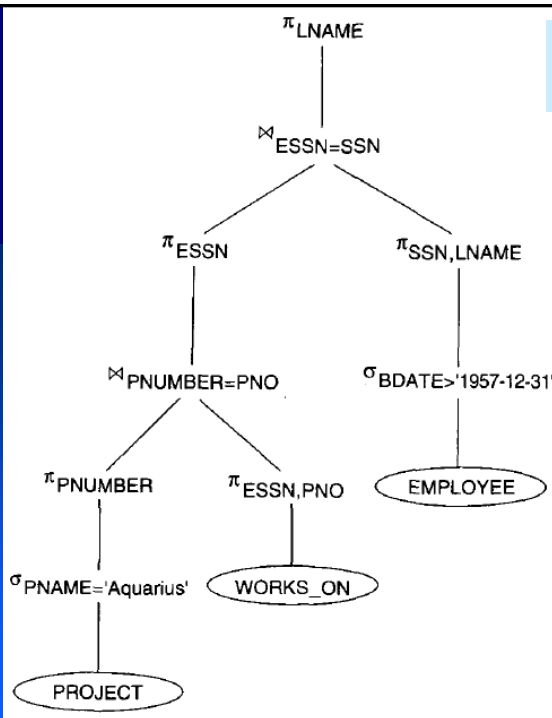
Applying the *more restrictive* SELECT operation first

104

Replacing CARTESIAN PRODUCT and SELECT with JOIN operations



105



Moving PROJECT operations *down* the query tree

106

3.7. Cost-based query optimization

- **Cost-based query optimization:** Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate.
- **Issues**
  - Cost function
  - Number of execution strategies to be considered

3.7. Cost-based query optimization

Cost Components for Query Execution

- 1. Access cost to secondary storage
- 2. Storage cost
- 3. Computation cost
- 4. Memory usage cost
- 5. Communication cost

Note: Different database systems may focus on different cost components.

3.7. Cost-based query optimization

Catalog Information Used in Cost Functions

- Information about the size of a file
  - **number of records (tuples) (r)**
  - **record size (R)**
  - **number of blocks (b)**
  - **blocking factor (bfr)**
- Information about indexes and indexing attributes of a file
  - **Number of levels (x)** of each multilevel index
  - **Number of first-level index blocks (bl1)**
  - **Number of distinct values (d)** of an attribute
  - **Selectivity (sl)** of an attribute
  - **Selection cardinality (s)** of an attribute. (s = sl * r)

109

3.7. Cost-based query optimization

Cost Functions for SELECT

- **S1. Linear search (brute force) approach:**

  C_{S1a}= b;

  For an equality condition on a key, C_{S1b} = (b/2) if the record is found; otherwise C_{S1a}= b.
- **S2. Binary search**:

  C_{S2}= $\log_2 b$ + ⌈(s/bfr)⌉ - 1

  For an equality condition on a unique (key) attribute,

  C_{S2} =$\log_2 b$
- **S3. Using a primary index (S3a) or hash key (S3b) to retrieve a single record:**

  C_{S3a}= x + 1; C_{S3b} = 1 for static or linear hashing;

  C_{S3b} = 2 for extendible hashing

110

3.7. Cost-based query optimization

Cost Functions for SELECT (cont.)

- **S4. Using an ordering index to retrieve multiple records:**

  For the comparison condition on a key field with an ordering index, $C_{S4} = x + (b/2)$

- **S5. Using a clustering index to retrieve multiple records for an equality condition**:

  $C_{S5} = x + \lceil (s/bfr) \rceil$

- **S6. Using a secondary (B^+-tree) index:**

  For an equality comparison, $C_{S6a} = x + s$

  For a comparison condition such as >, <, >=, or <=,

  $C_{S6b} = x + (bI1/2) + (r/2)$

111

3.7. Cost-based query optimization

Cost Functions for SELECT (cont.)

- **S7. Conjunctive selection:**

  Use either S1 or one of the methods S2 to S6 to solve.

  For the latter case, use one condition to retrieve the records and then check in the memory buffer whether each retrieved record satisfies the remaining conditions in the conjunction.

- **S8. Conjunctive selection using a composite index**:

  Same as S3a, S5 or S6a, depending on the type of index.

112

Example

- r_E = 10,000 , b_E = 2000 , bfr_E = 5
- Access paths:
  - 1. A clustering index on SALARY, with levels x_{SALARY} = 3 and average selection cardinality S_{SALARY} = 20.
  - 2. A secondary index on the key attribute SSN, with x_{SSN} = 4 (S_{SSN} = 1).
  - 3. A secondary index on the nonkey attribute DNO, with x_{DNO}= 2 and first-level index blocks b_{I1DNO}= 4. There are d_{DNO} = 125 distinct values for DNO, so the selection cardinality of DNO is S_{DNO} = (r/d_{DNO}) = 80.
  - 4. A secondary index on SEX, with x_{SEX} = 1. There are d_{SEX} = 2 values for the sex attribute, so the average selection cardinality is S_{SEX} = (r/d_{SEX}) = 5000.

113

Example

- (op1): $\sigma_{SSN='123456789'}$ (EMPLOYEE)
  - C_{S1b} = 1000
  - C_{S6a} = x_{SSN} + 1 = 4+1 = 5
- (op2): $\sigma_{DNO>5}$ (EMPLOYEE)
  - C_{S1a} = 2000
  - C_{S6b} = x_{DNO} + ($b_{I1DNO}/2$) + ($r/2$) = 2 + 4/2 + 10000/2 = 5004

114

Example

- (op3): $\sigma_{DNO=5}$ (EMPLOYEE)
  - $C_{S1a} = 2000$
  - $C_{S6a} = x_{DNO} + s_{DNO} = 2 + 80 = 82$
- (op4): $\sigma_{DNO=5 \text{ AND } SALARY>30000 \text{ AND } SEX='F'}$ (EMPLOYEE)
  - $C_{S6a\text{-}DNO} = 82$
  - $C_{S4\text{-}SALARY} = x_{SALARY} + (b/2) = 3 + 2000/2 = 1003$
  - $C_{S6a\text{-}SEX} = x_{SEX} + s_{SEX} = 1 + 5000 = 5001$
  - => chose DNO=5 first and check the other conditions

115

3.7. Cost-based query optimization

Cost Functions for JOIN

- **Join selectivity (js)**

  $js = | (R \rhd\lhd_C S) | / | R \times S | = | (R \rhd\lhd_C S) | / (|R| * |S|)$

  If condition C does not exist, js = 1.

  If no tuples from the relations satisfy condition C, js = 0.

  Usually, **0 <= js <= 1** .

Size of the result file after join operation

$| (R \rhd\lhd_C S) | = js * |R| * |S|$

116

3.7. Cost-based query optimization

Cost Functions for JOIN (cont.)

- **J1. Nested-loop join:**

  $C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|)/bfr_{RS})$

  (Use *R* for outer loop)

- **J2. Single-loop join**(using an access structure to retrieve the matching record(s))

  If an index exists for the join attribute *B* of *S* with index levels x_B, we can retrieve each record *s* in *R* and then use the index to retrieve all the matching records *t* from *S* that satisfy t[B] = s[A].

  The cost depends on the type of index.

117

3.7. Cost-based query optimization

Cost Functions for JOIN (cont.)

- **J2. Single-loop join (cont.)**

  For a secondary index,

  $C_{J2a} = b_R + (|R| * (x_B + s_B)) + ((js * |R| * |S|)/bfr_{RS})$;

  For a clustering index,

  $C_{J2b} = b_R + (|R| * (x_B + (s_B/bfr_B))) + ((js * |R| * |S|)/bfr_{RS})$;

  For a primary index,

  $C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS})$;

  If a hash key exists for one of the two join attributes — B of S

  $C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|)/bfr_{RS})$;

  h: the average number of block accesses to retrieve a record, given its hash key value, h>=1

- **J3. Sort-merge join:**

  $C_{J3a} = C_S + b_R + b_S + ((js * |R| * |S|)/bfr_{RS})$;

  (C_S: Cost for sorting files)

118

Example

- Suppose that we have the EMPLOYEE file described in the previous example
- Assume that the DEPARTMENT file of $r_D = 125$ and $b_D = 13$, $x_{DNUMBER} = 1$, secondary index on MGRSSN of DEPARTMENT, $s_{MGRSSN} = 1$, $x_{MGRSSN} = 2$, $js_{OP6} = (1/IDEPARTMENTI) = 1/125$, $bfr_{ED} = 4$
- (op6): EMPLOYEE $\triangleright\triangleleft_{DNO=DNUMBER}$ DEPARTMENT
- (op7): DEPARTMENT $\triangleright\triangleleft_{MGRSSN=SSN}$EMPLOYEE

Example

J1: nested-loop join
J2: single-loop join

- (op6): EMPLOYEE $\triangleright\triangleleft_{DND=DNUMBER}$ DEPARTMENT
  - Method J1 with Employee as outer:
    - $C_{J1} = b_E + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED})$
    - $= 2000 + (2000 * 13) + (((1/125) * 10,000 * 125)/4) = 30,500$
  - Method J1 with Department as outer:
    - $C_{J1} = b_D + (b_E * b_D) + (((js_{OP6} * r_E * r_D)/bfr_{ED})$
    - $= 13 + (13 * 2000) + (((1/125) * 10,000 * 125/4) = 28,513$
  - Method J2 with EMPLOYEE as outer loop:
    - $C_{J2c} = b_E + (r_E * (x_{DNUMBER} + 1)) + ((js_{OP6} * r_E * r_D)/bfr_{ED}$
    - $= 2000 + (10,000 * 2) + (((1/125) * 10,000 * 125/4) = 24,500$
  - Method J2 with DEPARTMENT as outer loop:
    - $C_{J2a} = b_D + (r_D * (x_{DNO} + s_{DNO})) + ((js_{OP6} * r_E * r_D)/bfr_{ED})$
    - $= 13 + (125 * (2 + 80)) + (((1/125) * 10,000 * 125/4) = 12,763$

Assume that the DEPARTMENT file of $r_D = 125$ and $b_D = 13$, $x_{DNUMBER} = 1$, secondary index on MGRSSN of DEPARTMENT, $s_{MGRSSN} = 1$, $x_{MGRSSN} = 2$, $js_{OP7} = (1/|EMPLOYEE|) = 1/r_E = 1/10{,}000$, $bfr_{ED} = 4$

A secondary index on the key attribute SSN of EMPLOYEE, with $x_{SSN} = 4$ ($S_{SSN} = 1$).

(op7): DEPARTMENT $\triangleright\!\triangleleft_{MGRSSN=SSN}$ EMPLOYEE

- Method J1 with Employee as outer:
  - $C_{J1} = b_E + (b_E * b_D) + ((js_{OP7} * r_E * r_D)/bfr_{ED})$
    $= 2000 + (2000 * 13) + (((1/10{,}000) * 10{,}000 * 125)/4) = \lceil 28{,}031.25 \rceil = 28{,}032$
- Method J1 with Department as outer:
  - $C_{J1} = b_D + (b_E * b_D) + (((js_{OP7} * r_E * r_D)/bfr_{ED})$
    $= 13 + (13 * 2000) + (((1/10{,}000) * 10{,}000 * 125/4) = \lceil 26{,}044.25 \rceil = 26{,}045$
- Method J2 with EMPLOYEE as outer loop:
  - $C_{J2c} = b_E + (r_E * (x_{MGRSSN} + s_{MGRSSN})) + ((js_{OP7} * r_E * r_D)/bfr_{ED}$
    $= 2000 + (10{,}000 * (2+1)) + (((1/10{,}000) * 10{,}000 * 125/4) = \lceil 32{,}031.25 \rceil = 32{,}032$
- Method J2 with DEPARTMENT as outer loop:
  - $C_{J2a} = b_D + (r_D * (x_{SSN} + s_{SSN})) + ((js_{OP7} * r_E * r_D)/bfr_{ED})$
    $= 13 + (125 * (4 + 1)) + (((1/10{,}000) * 10{,}000 * 125/4) = \lceil 669.25 \rceil = 670$

3.8. Semantics-based query optimization

- Uses *constraints* specified on the database schema - such as unique attributes and other more complex constraints - in order to modify one query into another query that is more efficient to execute
  - → Searching through many constraints to find those that are applicable to a given query and that may semantically optimize it can also be quite time-consuming.
  - → Inclusion of *active rules* in database systems

122

3.8. Semantics-based query optimization

- Example: a query retrieves the names of employees who earn more than their supervisors.
  - a constraint on the database schema that stated that no employee can earn more than his or her direct supervisor

  → not execute the query at all because it knows that the result of the query will be empty

```
SELECT E.LNAME, M.LNAME
FROM   EMPLOYEE AS E, EMPLOYEE AS M
WHERE  E.SUPERSSN=M.SSN AND E.SALARY > M.SALARY
```

123

3.9. Conclusion

- Query processing and optimization for high-level SQL queries
  - The most reasonable execution plan
    - Heuristic rules
    - Cost
    - Semantics
  - → How data are stored on secondary storage devices
  - → How data are accessed from secondary storage devices
- Index structures: B-tree, B+-tree, R-tree

124

Questions ???

Review

- Review questions and exercises in [1], chapters 13-15.

Next - Chapter 4: Database security

- *4.1. An overview of database security*
- *4.2. Discretionary access control based on granting and revoking privileges*
- *4.3. Mandatory access control and role-based access control for multilevel security*
- *4.4. Encryption and public key infrastructures*
- *4.5. Conclusion*

127