MODULE V

INDEX STRUCTURES-(SINGLE-LEVEL AND MULTILEVEL)

TYPES OF SINGLE-LEVEL ORDERED INDEXES

- The idea behind an ordered index access structure is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book.
- We can search an index to find a list of addresses-page numbers in this case-and use these addresses to locate a term in the textbook by searching the specified pages.
- For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an **indexing field.**
- The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value.
- The values in the index are ordered so that we can do a binary search on the index. The index file is much smaller than the data file, so searching the index using a binary search is reasonably efficient.
- There are several types of **single-level ordered indexes.**
- A **primary index** is specified on the ordering key field of an ordered file of records.
- An ordering key field is used to physically order the file records on disk, and every record has a unique value for that field.
- If the ordering field is not a key field-that is, if numerous records in the file can have the same value for the ordering field-another type of index, called a **clustering index**, can be used.
- A third type of index, called a **secondary index**, can be specified on any nonordering field of a file.

PRIMARY

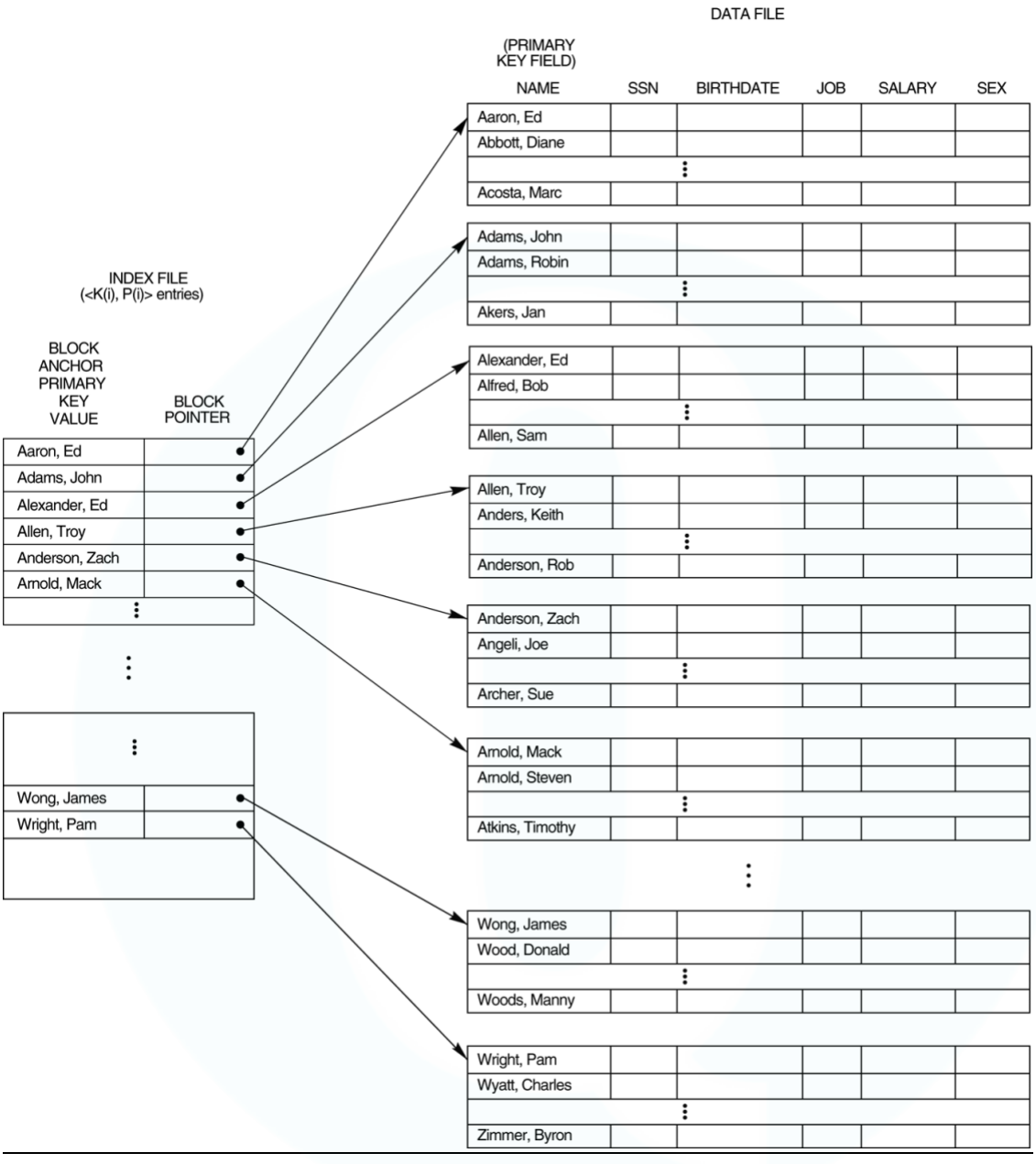


FIGURE 14.1 Primary index on the ordering key field of the file

- A primary index is an **ordered file** whose records are of fixed length with two fields.
- The first field is of the same data type as the ordering key field-called **the primary key-of the data file**, and the second field is a **pointer** to a disk block (a block address).
- There is one index entry (or index record) in the index file for each block in the data file.
- Each index entry has the value of the primary key field for the first record in a block and a pointer to that block as its two field values.

- We will refer to the two field values of index entry i as <K(i),P(i)>
- To create a primary index on the ordered file shown in the Figure, we use the NAME field as primary key, because that is the ordering key field of the file (assuming that each value of NAME is unique). Each entry in the index has a NAME value and a pointer. The first three index entries are as follows:

<K(1) = (Aaron,Ed), P(1) = address of block 1>
<K(2) = (Adams,John), P(2) = address of block 2>
<K(3) = (Alexander,Ed), P(3) = address of block 3>

- The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor.**
- Indexes can also be characterized as dense or sparse.
- A **dense** index has an index entry for every search key value (and hence every record) in the data file.
- A **sparse** (or non dense) index, on the other hand, has index entries for only some of the search values.
- A primary index is hence a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value (or every record).
- The index file for a primary index needs substantially fewer blocks than does the data file, for **two** reasons.
- **First**, there are fewer index entries than there are records in the data file.
- **Second,** each index entry is typically smaller in size than a data record because it has only two fields; consequently, more index entries than data records can fit in one block.
- A binary search on the index file hence requires fewer block accesses than a binary search on the data file.
- A major **problem** with a primary index-as with any ordered file-is **insertion and deletion** of records.
- With a primary index, the problem is compounded because, if we attempt to insert a record in its correct position in the data file, we have to not only move records to make space for the new record but also change some index entries, since **moving records will change the anchor records of some blocks.**

CLUSTERING INDICES,

- If records of a file are physically ordered on a nonkey field-which does not have a distinct value for each record-that field is called the **clustering field**.
- We can create a different type of index, called a **clustering index,** to speed up retrieval of records that have the same value for the clustering field.

- This differs from a primary index, which requires that the ordering field of the data file have a **distinct** value for each record.
- A clustering index is also an ordered file with **two** fields; the first field is of the same type as the clustering field of the data file, and the second field is a block pointer.
- There is one entry in the clustering index for each distinct value of the clustering field, containing the value and a pointer to the ***first*** block in the data file that has a record with that value for its clustering field.

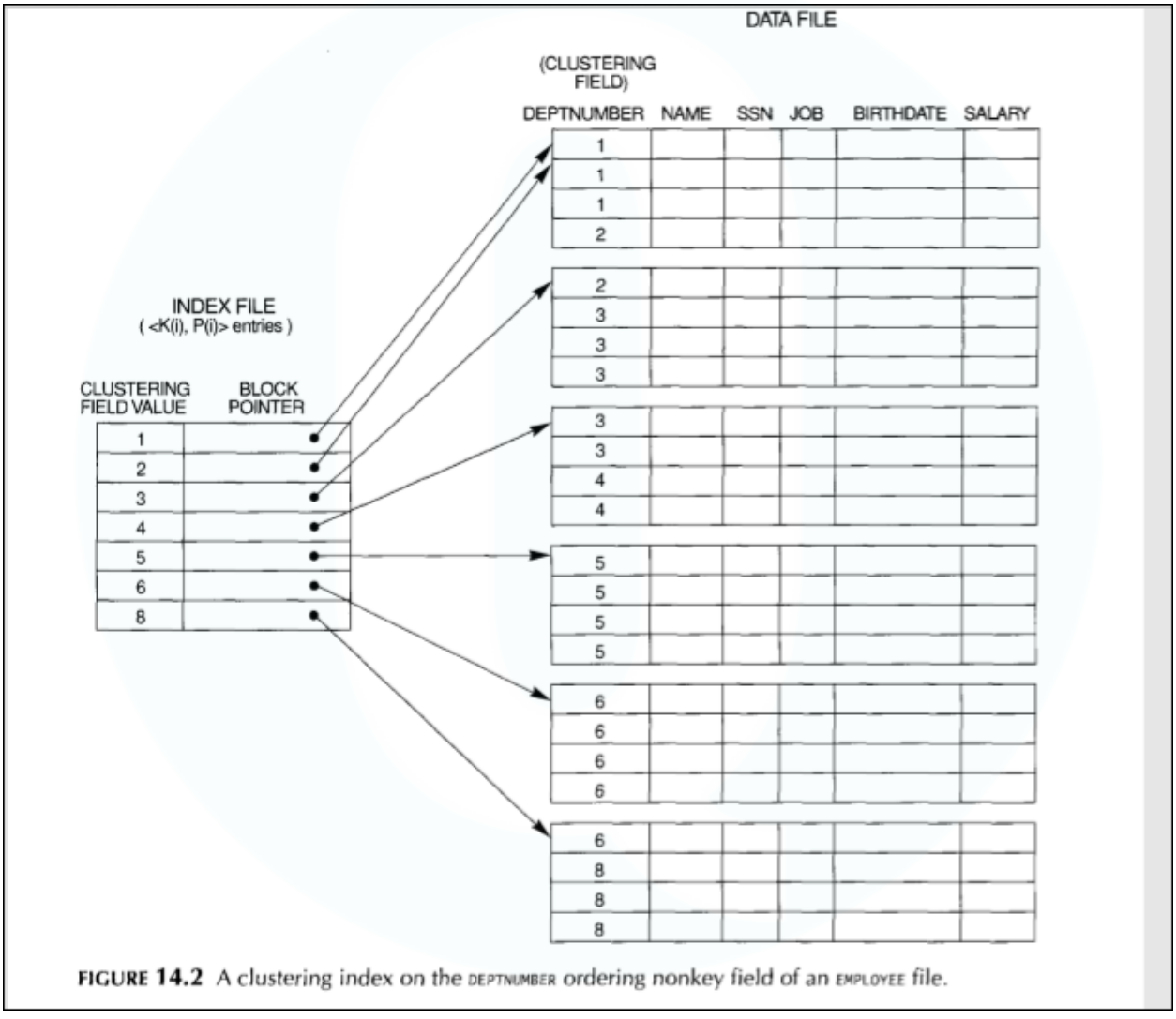


FIGURE 14.2 A clustering index on the DEPTNUMBER ordering nonkey field of an EMPLOYEE file.

- The record **insertion and deletion still cause problems,** because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for each value of the clustering field; all records

with that value are placed in the block (or block cluster). This makes insertion and deletion relatively straightforward.

- A clustering index is another example of a **nondense** index, because it has an entry for every distinct value of the indexing field which is a nonkey by definition and hence has duplicate values rather than for every record in the file.

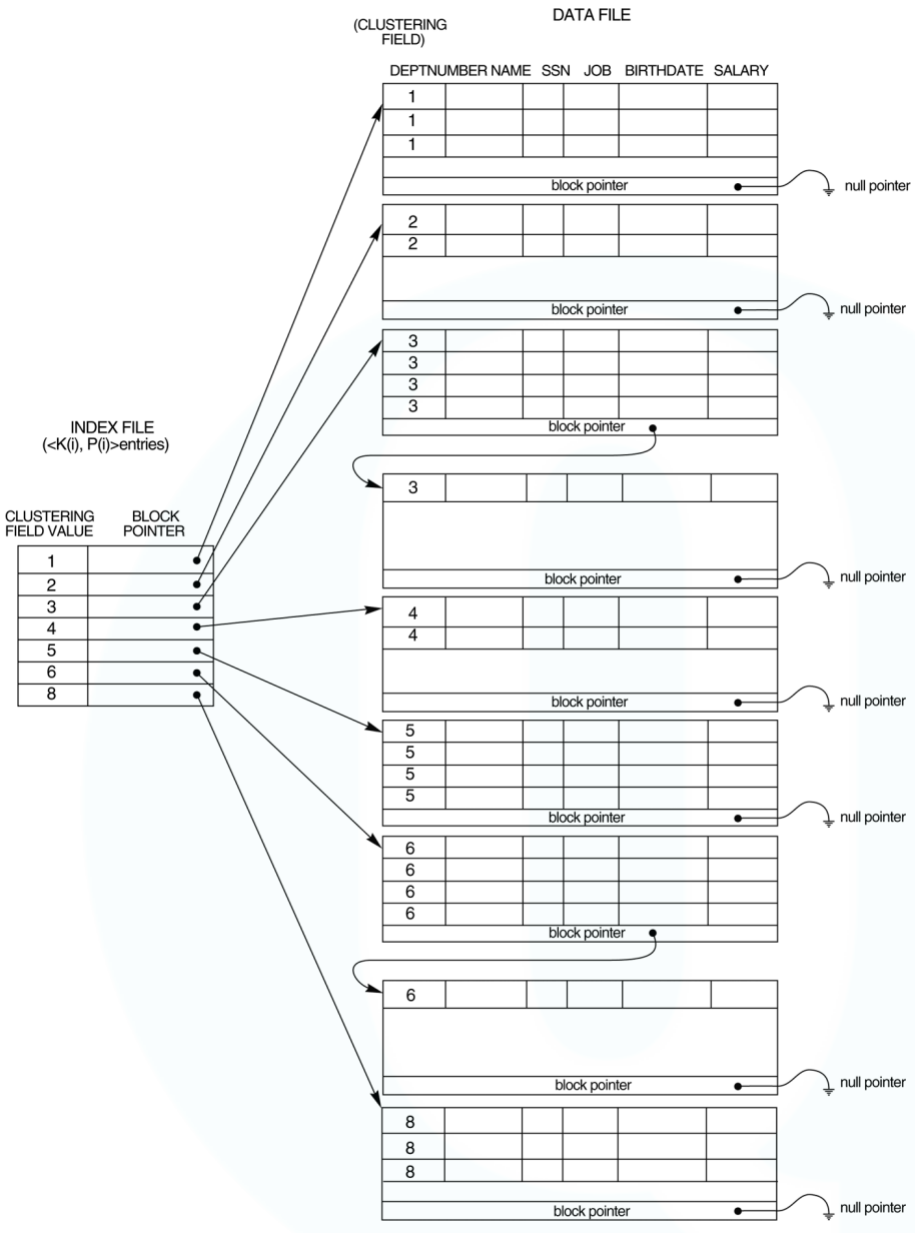


FIGURE 14.3 Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

SECONDARY

- A secondary index provides a secondary means of accessing a file for which some primary access already exists.

- The secondary index may be on a field which is a candidate key and has a **unique value in every record**, or a nonkey with duplicate values.
- The index is an ordered file with two fields.
- The first field is of the same data type as some **nonordering field** of the data file that is an **indexing field**. The second field is either a **block pointer** or a record pointer.
- We first consider a secondary index access structure on a key field that has a distinct value for every record. Such a field is sometimes called a **secondary key.**
- In this case there is 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. Hence, such an index is dense.
- We again refer to the two field values of index entry i as <K(i),P(i)>
- The entries are ordered by value of K(i),, so we can perform a **binary search**.
- Because the records of the data file are not physically ordered by values of the secondary key field, we cannot use block anchors.
- That is why an index entry is created for each record in the data file, rather than for each block, as in the case of a primary index
- Once the appropriate block is transferred to main memory, a search for the desired record within the block can be carried out.
- A secondary index usually needs **more storage space** and longer search time than does a primary index, because of its larger number of entries.
- However, the *improvemen*t in search time for an arbitrary record is much greater for a secondary index than for a primary index, since we would have to do a linear search on the data file if the secondary index did not exist.
- For a primary index, we could still use a binary search on the main file, even if the index did not exist.

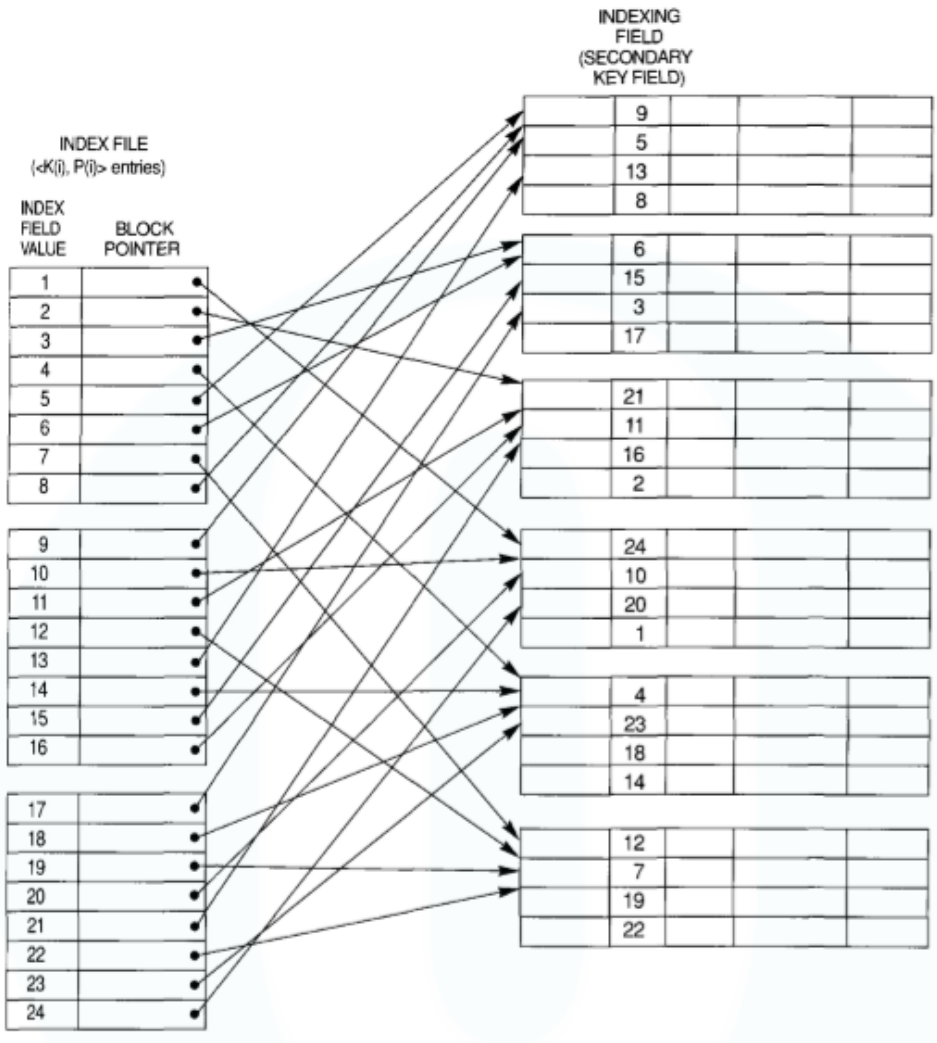


FIGURE **14.4** A dense secondary index (with block pointers) on a nonordering key field of a file.

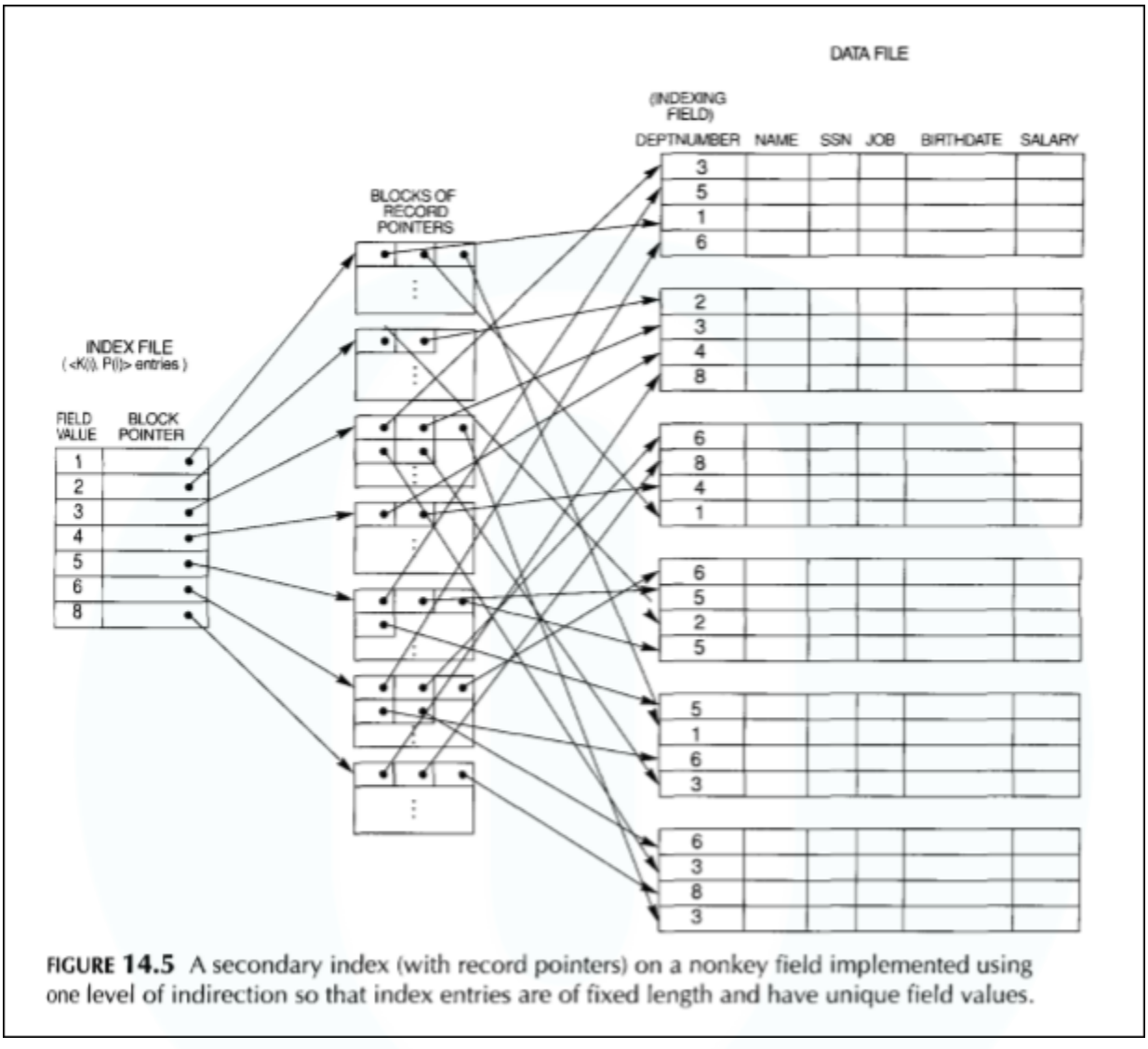


FIGURE 14.5 A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

MULTI-LEVEL INDEXING

- 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 K(i).
- Hence we can create a primary index for the first level; this index to the first level is called the second level of the multilevel index.
- Because the second level is a primary index, we can use block anchors so that the second level has one entry **for each block** of the first level.
- We can repeat this process for the second level. The third level, which is a primary index for the second level, has an entry for each second-level block.

- The multilevel scheme described here can be used on any type of index, whether it is primary, clustering, or secondary-as long as the first-level index has distinct values for K(i) and fixed-length entries.
- A common file organization used in business data processing is an ordered file witha multilevel primary index on its ordering key field. Such an organization is called an indexed sequential file and was used in a large number of early IBM systems.
- A multilevel index reduces the number of blocks accessed when searching for a record, given its indexing field value.
- We are still faced with the problems of dealing with index insertions and deletions, because all index levels are physically ordered files.
- To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index that leaves some space in each of its blocks for inserting new entries.
- This is called a **dynamic multilevel index** and is often implemented by using data structures called **B-trees and B+-trees**.

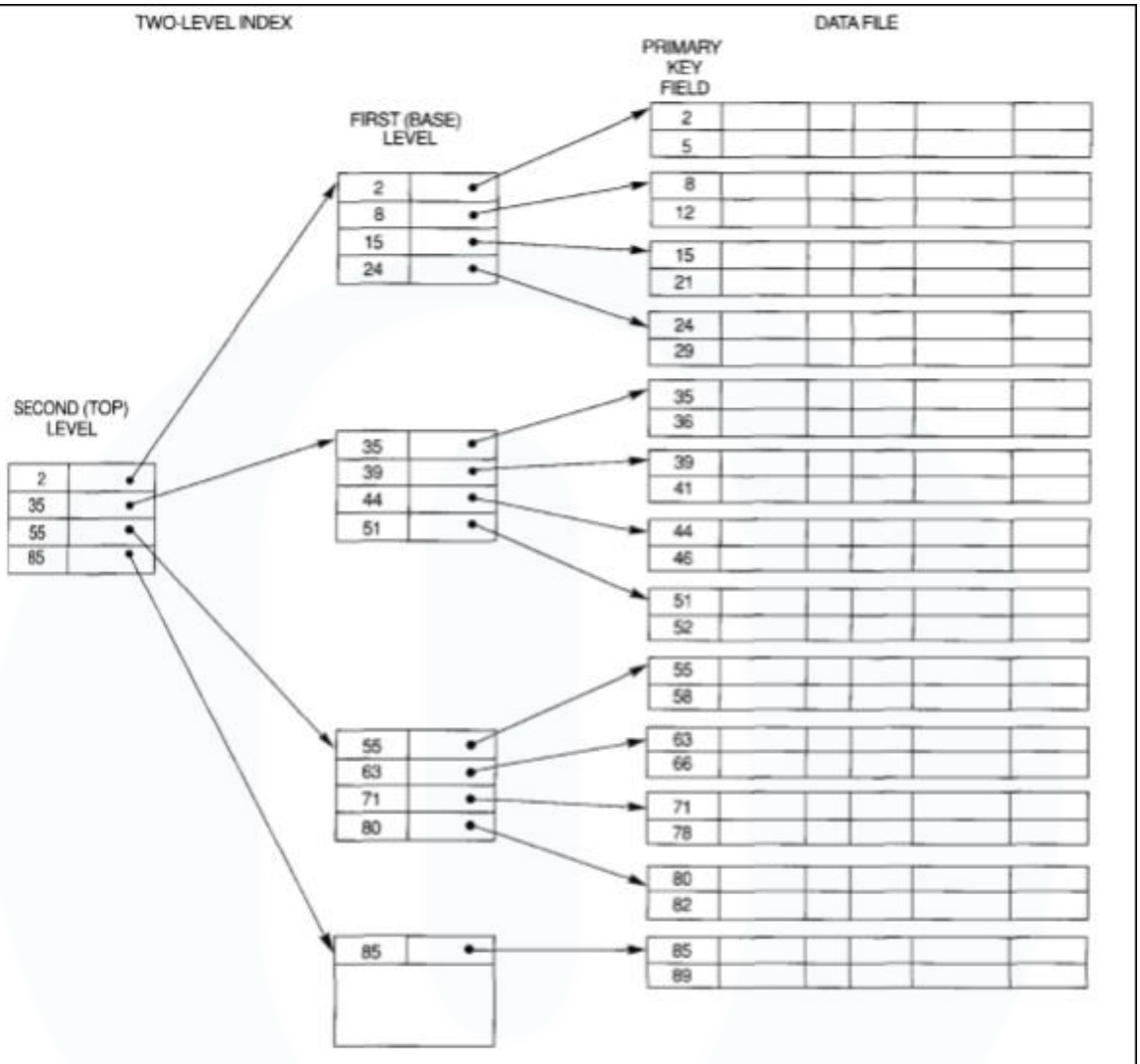


FIGURE 14.6 A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.

B TREES

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 in the B-tree (Figure 14.10a) 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 $q \leq p$. Each Pi is a **tree pointer**--a pointer to another node in the B-tree.

Each Pr_i is a **data pointer**--a pointer to the record whose search key field value is equal to Ki .

2. Within each node, $K_1 < K_2 < ... < K_{q-1}$.
3. For all search key field values X in the subtree pointed at by P_i (the i^{th} subtree, see Figure 14.10a), 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 node has at most p tree pointers.

5.A node with q tree pointers, $q \leq p$, has q - 1 search key field values (and hence has q - 1 data pointers).

6. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their tree pointers P, are null.

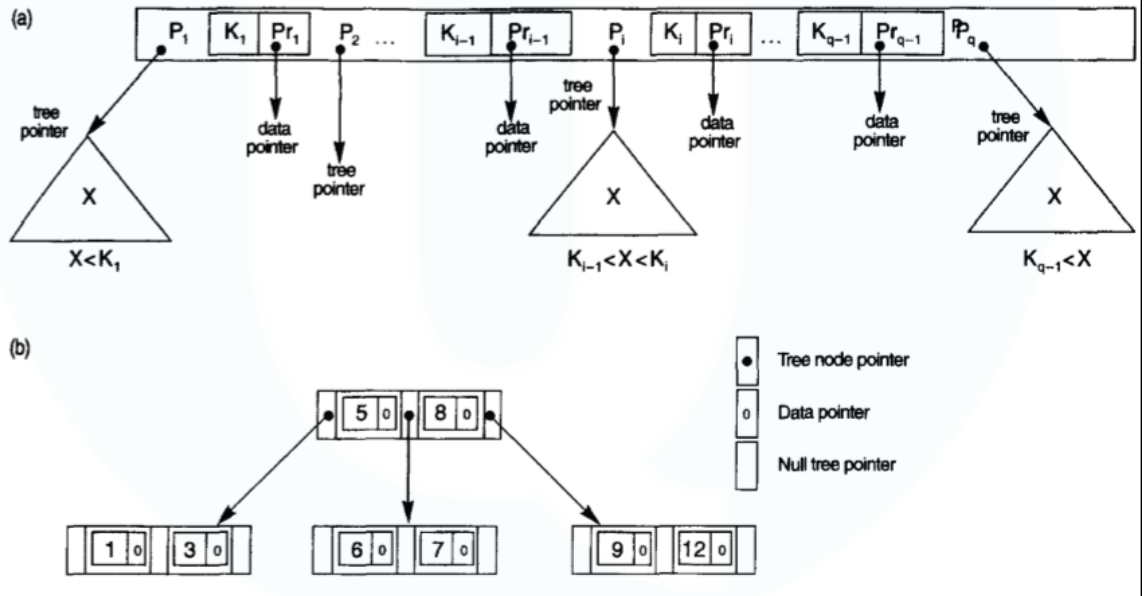


FIGURE 14.10 B-tree structures. (a) A node in a B-tree with q – 1 search values. (b) A B-tree of order p = 3. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

- A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero).
- Once the root node is full with p - 1 search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1.

- Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes.
- When a non-root node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes.
- If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split.

B+TREES (BASIC STRUCTURE ONLY, ALGORITHMS NOT NEEDED),

The structure of the **internal nodes** of a **B** $^{+}$ -tree of order p (Figure 14.11a)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 $q \leq p$ 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 $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 14.11a).[9]
4. Each internal node has at most p tree pointers.

6. An internal node with q pointers, q≤ p, has q - 1 search field values.

The structure of the leaf nodes of a W-tree oforder p (Figure 14.11b) is as follows:

1. Each leaf node is of the form

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

   where $q \leq p$, 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, $K_1 < K_2 < \ldots < K_{q-1}$, $q \leq p$.
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. All leaf nodes are at the same level.

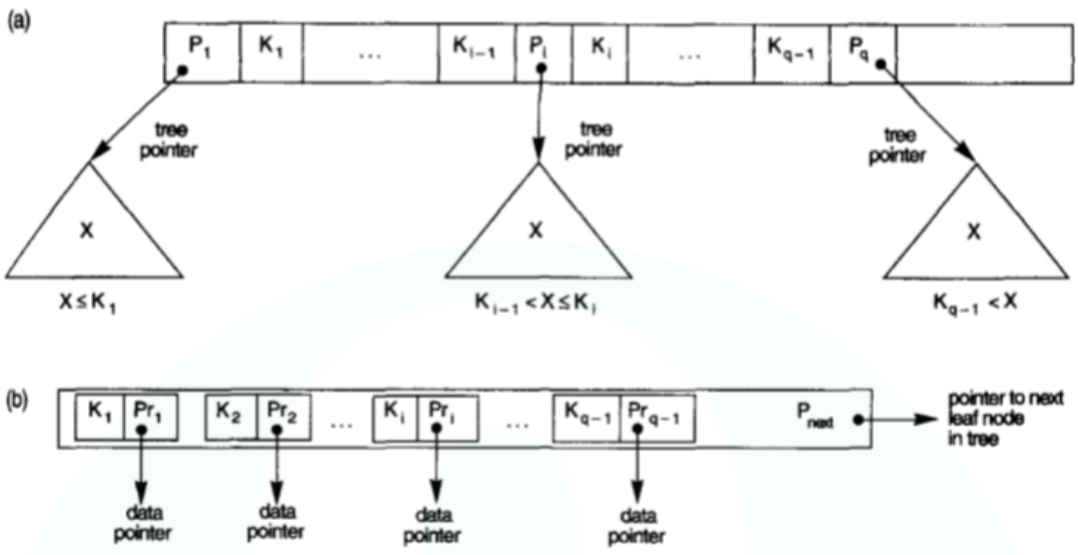


FIGURE 14.11 The nodes of a B+-tree. (a) Internal node of a B+-tree with q − 1 search values. (b) Leaf node of a B+-tree with q−1 search values and q−1 data pointers.

- Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a B+-tree.
- In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer.
- In a B+-tree, data pointers are stored only at the leaf nodes of the tree; hence, 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 non key search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.
- The leaf nodes of the B+-tree are usually linked together to provide ordered accesson 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.

QUERY OPTIMIZATION: HEURISTICS-BASED QUERY OPTIMIZATION,

- Optimization techniques are the ones that apply **heuristic rules** to modify the **internal representation of a query**-which is usually in the form of a **query tree or a query graph data structure**-to improve its expected performance.
- The parser of a high-level query first generates an initial internal representation, which is then optimized according to heuristic rules.
- Following that, a **query execution plan** is generated to **execute groups of operations** based on the access paths available on the files involved in the query.
- One of the main heuristic rules is to apply SELECT and PROJECT operations before applying the JOIN or other binary operations.
- This is because the size of the file resulting from a binary operation-such as JOIN-is usually a multiplicative function of the sizes of the input files.
- The SELECT and PROJECT operations reduce the size of a file and hence should be applied before a join or other binary operation.

Notation for Query Trees and Query Graphs

- A query tree is a tree data structure that corresponds to a **relational algebra expression**.
- It represents the input **relations**(tables) of the query as **leaf nodes** of the tree, and represents the **relational algebra operations** as **internal nodes**.
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.
- The execution **terminates** when the **root node** is executed and produces the result relation for the query.
- Consider the query Q2
  For every project located in 'Stafford', retrieve the project number, the controlling department number,

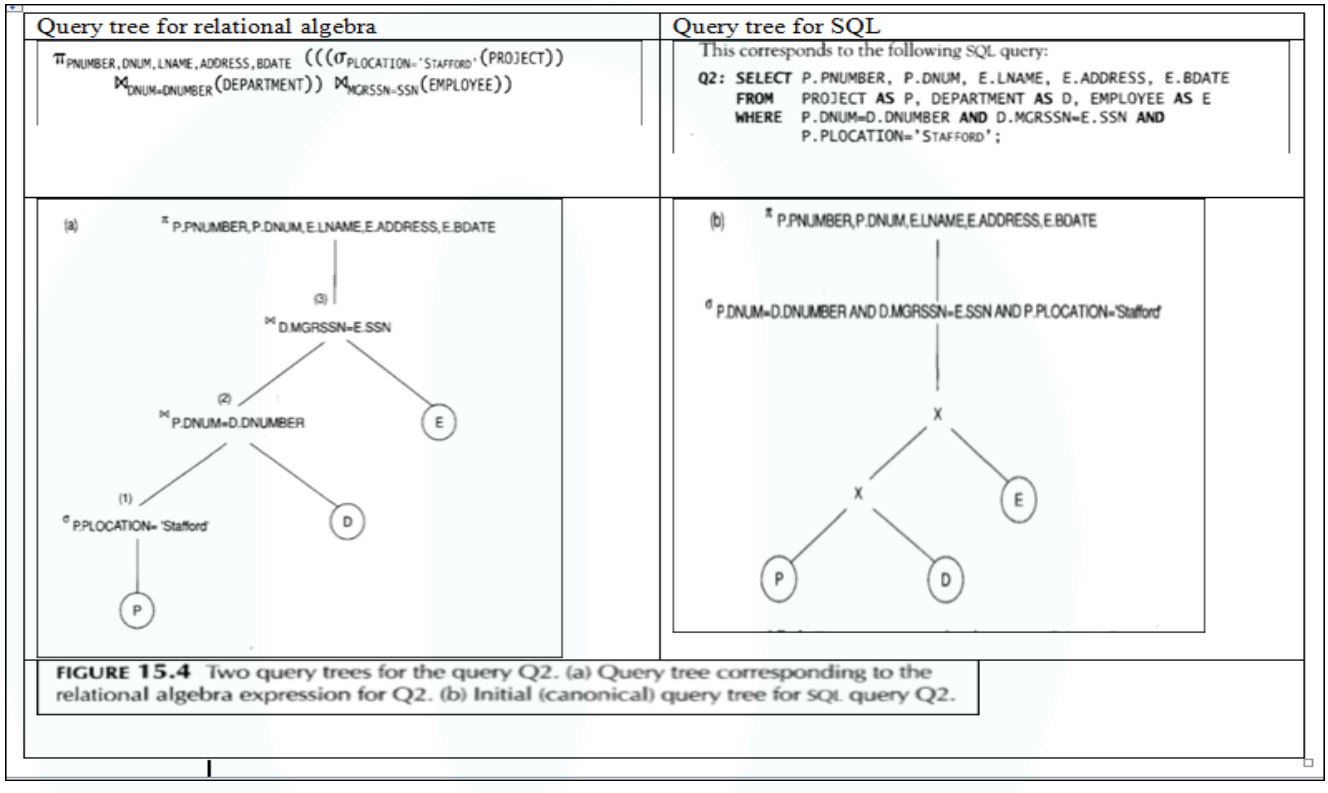


FIGURE 15.4 Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2.

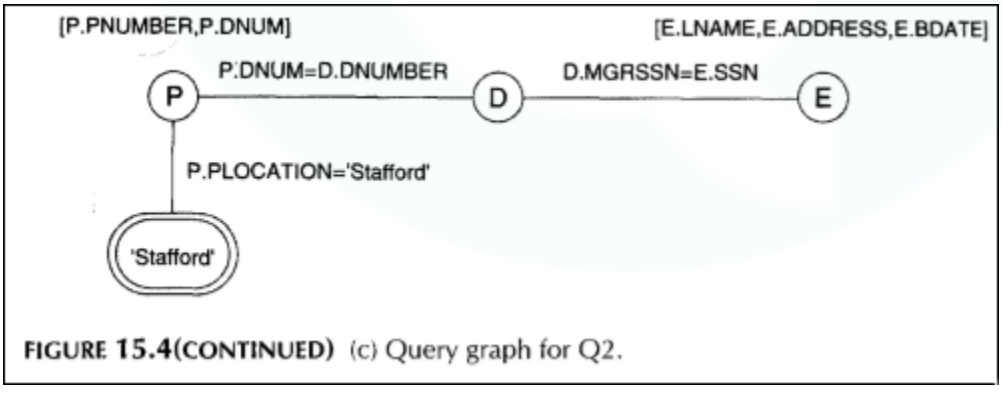


FIGURE 15.4(CONTINUED) (c) Query graph for Q2.

- As we can see, the **query tree represents a specific order** of operations for executing a query.

- A more neutral representation of a query is the query graph notation. Figure 15.4c shows the **query graph** for query Q2.
- **Relations(tables)** in the query are represented by relation nodes, which are displayed as **single circles.**
- Constant values, typically from the **query selection conditions**, are represented by constant nodes, which are displayed as **double circles or ovals.**
- **Selection and join conditions** are represented by the **graph edges.**
- Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

Heuristic Optimization of Query Trees

- The query parser will typically generate a **standard initial query tree** to correspond to an SQL query, without doing any optimization.
- For example, for a select-project-join query, such asQ2, the initial tree is shown in Figure 15.4b. The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes.
- Such a canonical query tree represents a relational algebra expression that is very *inefficient* if *executed directly*, because of the CARTESIAN PRODUCT (X) operations.It produces more no.of unnecessary tuples.
- It is now the job of the It is now the job of the **heuristic query optimizer** to transform this **initial query tree** into a **final query tree** that is efficient to execute.

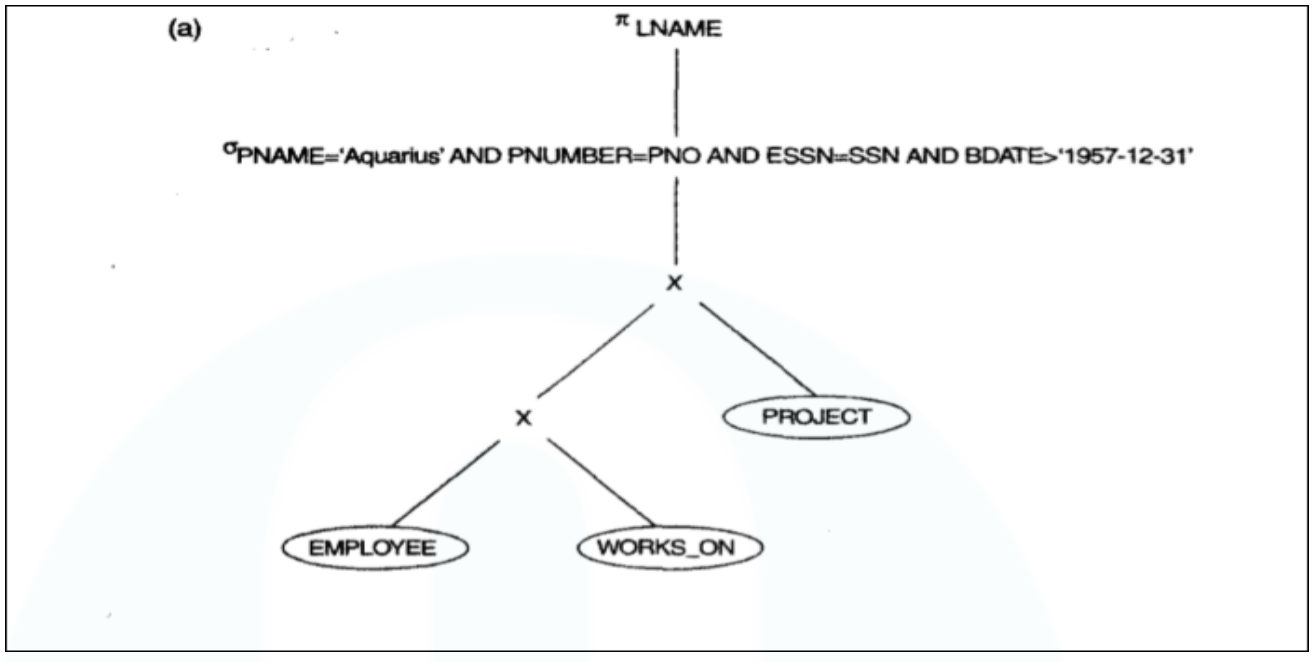
Example of Transforming a Query

- Consider the example query

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

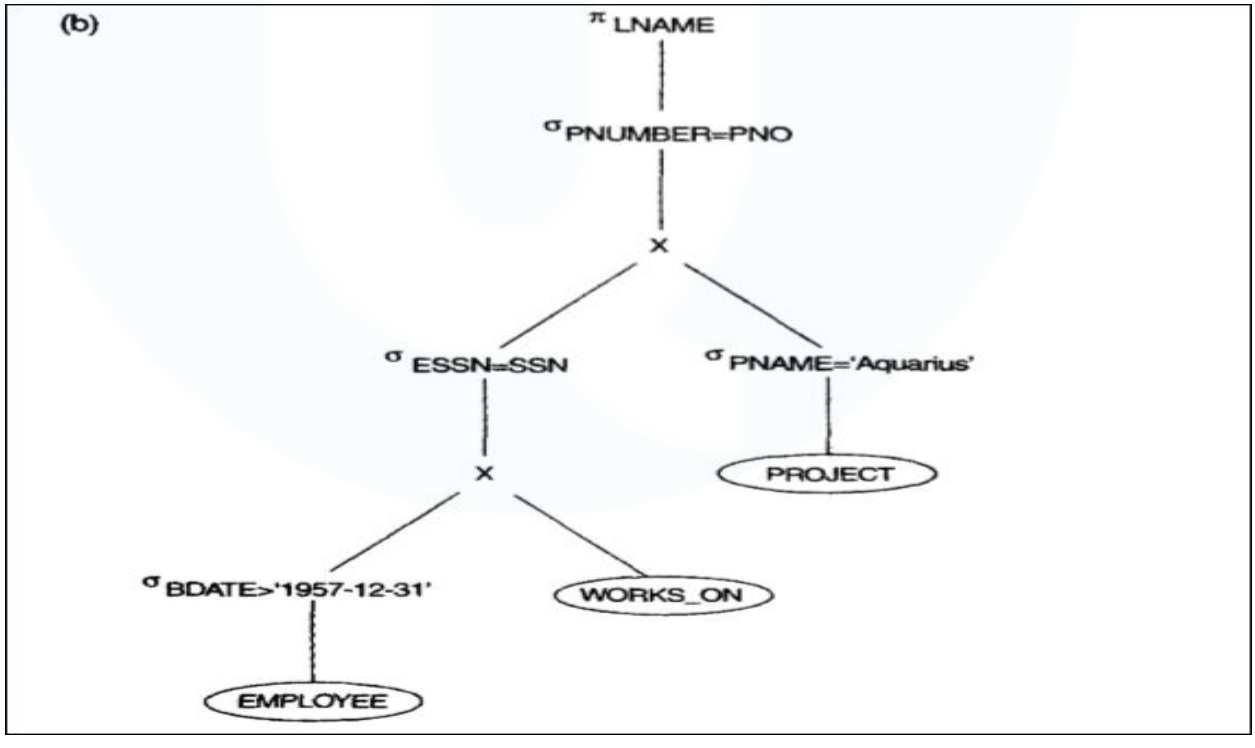
Step 1

- The initial query tree for Q is shown in Figure 15.5a.
- Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ ON, and PROJECT files.
- However, this query needs only one record from the PROJ ECT relation for the 'Aquarius' project-and only the EMPLOYEE records for those whose date of birth is after '1957-12-31'.



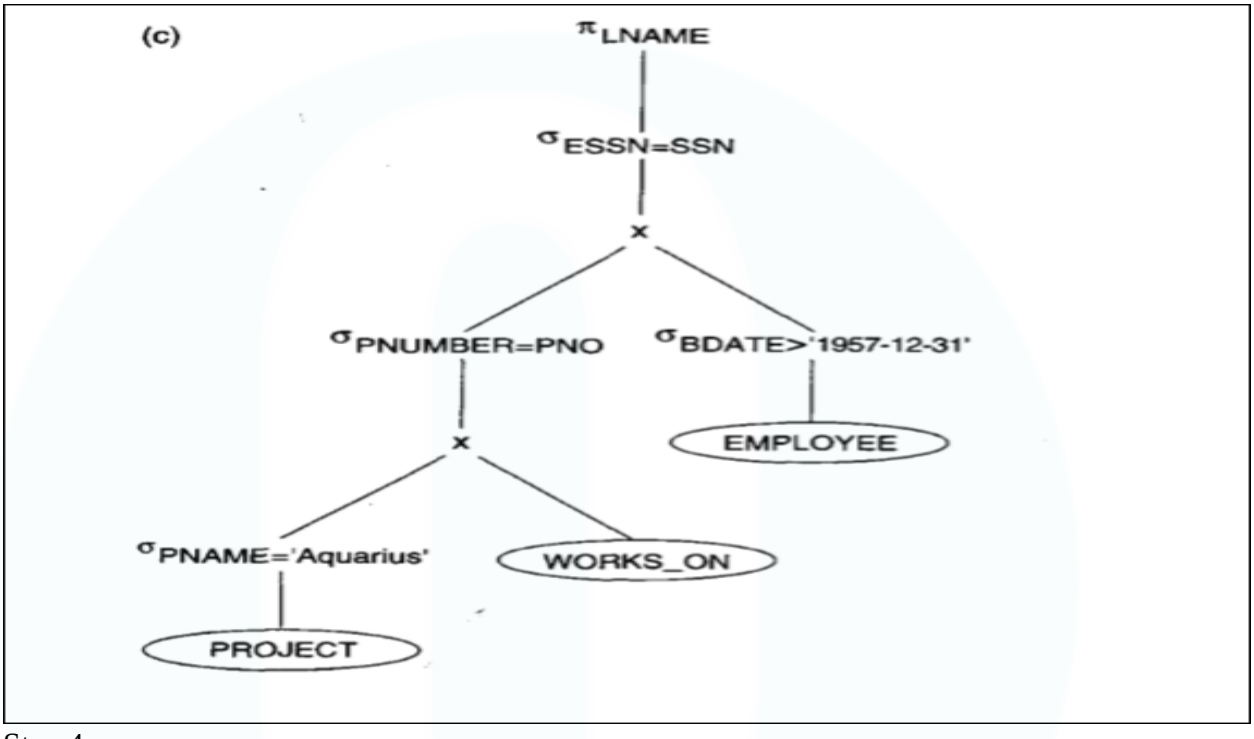
Step 2

Figure 15.5b shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.



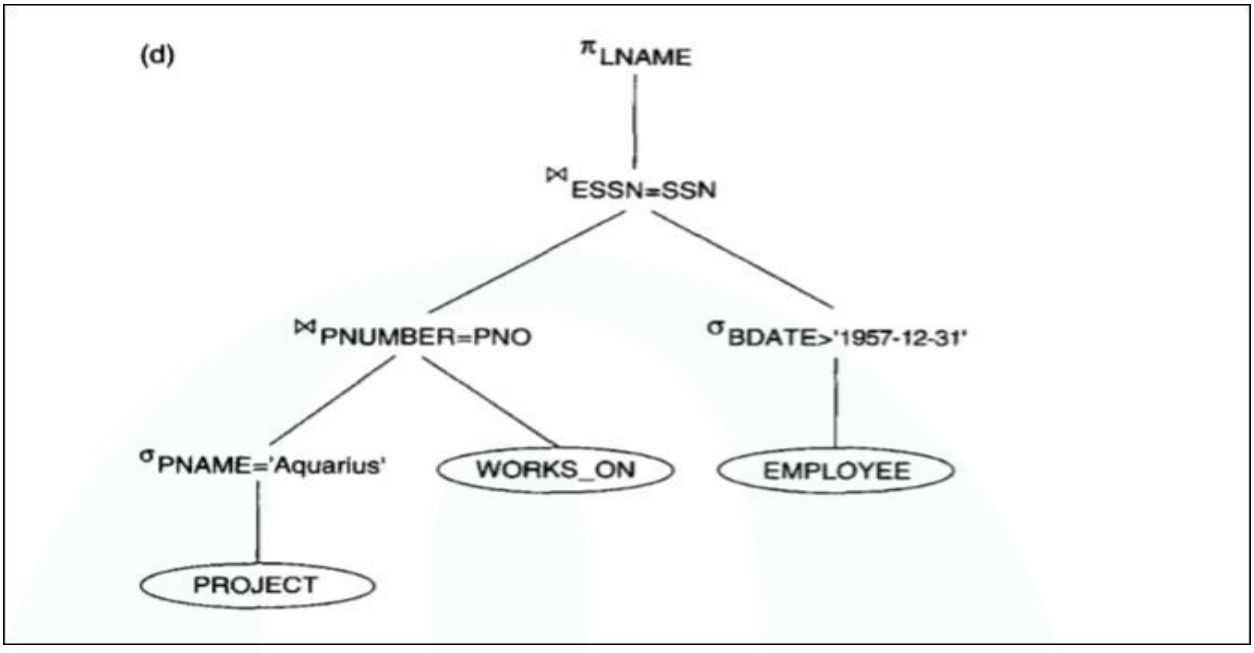
Step 3

A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure 15.5c. This uses the information that PNUMBER is a key attribute of the project relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only.



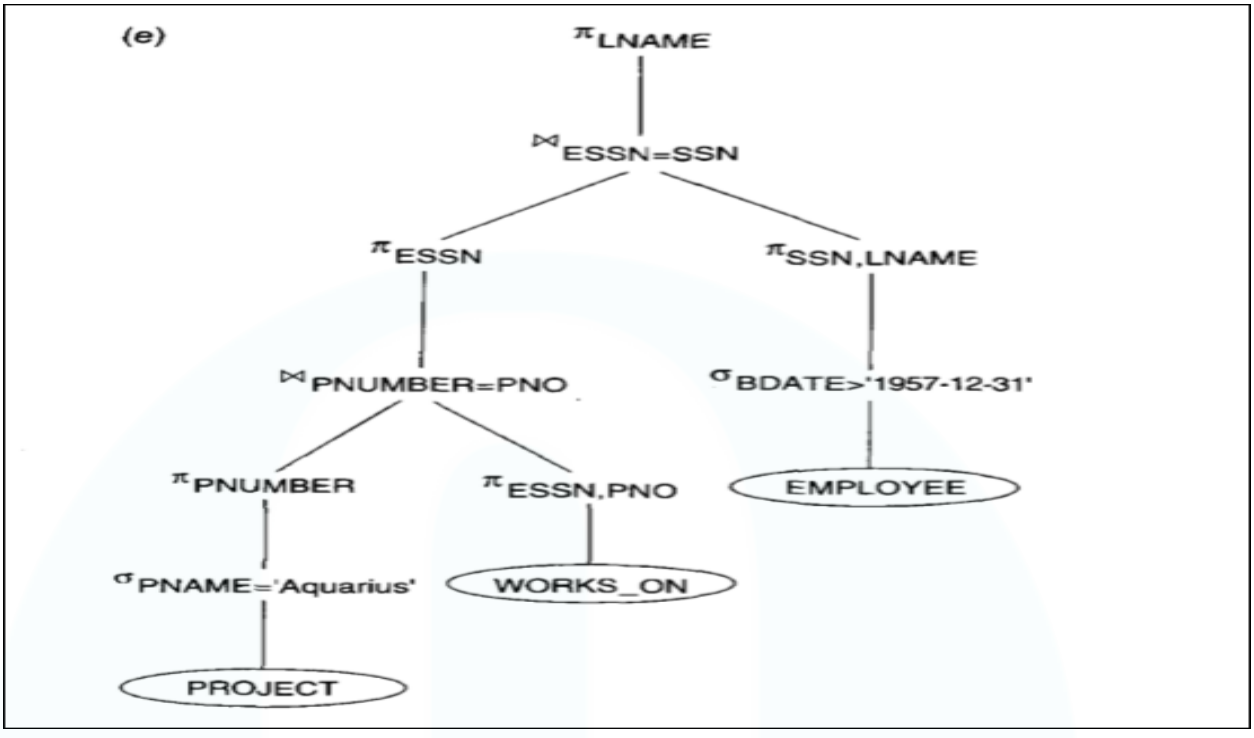
Step 4

We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition with a JOIN operation, as shown in Figure (d)



Step 5

- Another improvement is to keep only the attributes needed by subsequent operations in the intermediate relations, by including PROJECT (7r) operations as early as possible in the query tree,as shown in Figure(e).
- This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).



Outline of a Heuristic Algebraic Optimization Algorithm

- Break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations
- Move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition
- First, position the leaf node relations with the most restrictive SELECT (produce a relation with the fewest tuples) operations so they are executed first in the query tree representation. (or with the smallest absolute size.It may be desirable to change the order of leaf nodes to avoid Cartesian products.
- Combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.
- Break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.

Converting Query Trees into Query Execution Plans

An **execution plan** for a relational algebra expression represented as a query tree **includes** information about the **access methods** available for each **relation**(tables) as well as the **algorithms to be used in computing** the relational operators represented in the tree.

Consider the query

$$\pi_{FNAME, LNAME, ADDRESS}(\sigma_{DNAME='RESEARCH'}(DEPARTMENT) \bowtie_{DNUMBER=DNO} EMPLOYEE)$$

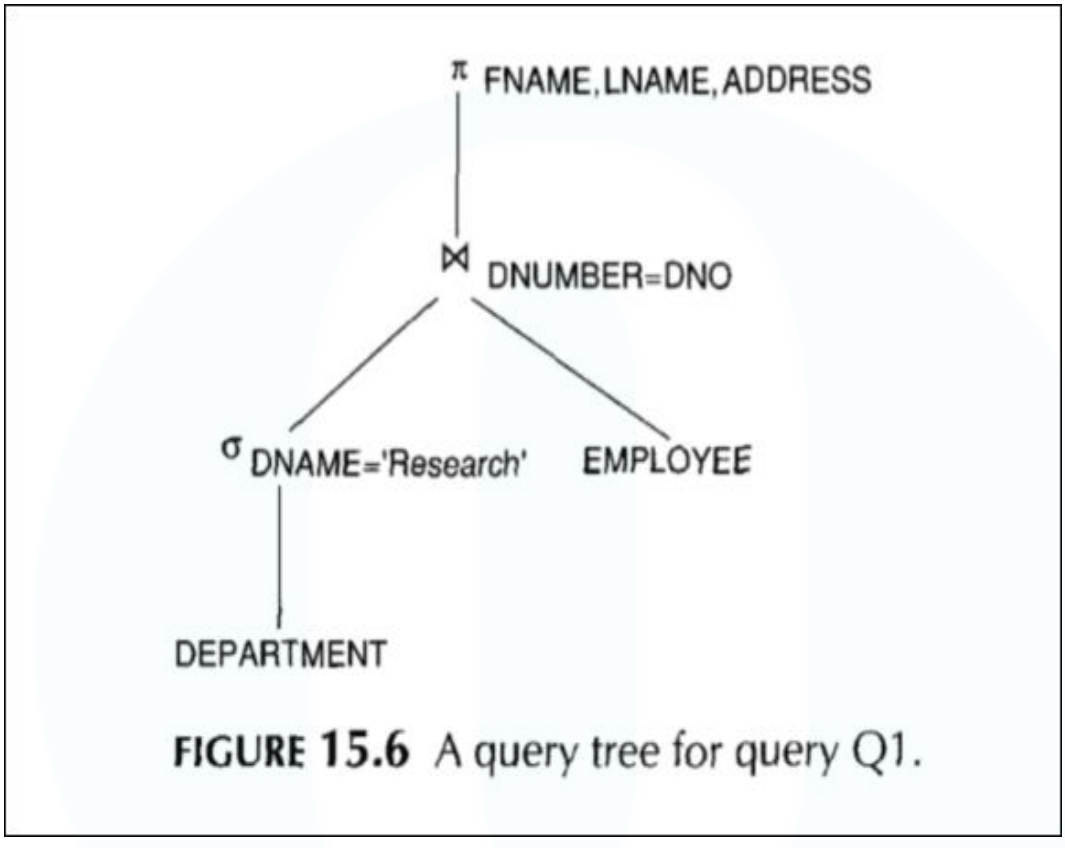


FIGURE 15.6 A query tree for query Q1.

To convert this query tree into an **execution plan**, the optimizer might choose an **index search** for the SELECT operation (assuming one exists), a **table scan** as access method for EMPLOYEE, a **nested-loop join algorithm** for the join, and a scan of the JOIN result for the PROJECT operator. In addition, the approach taken for executing the query may specify a **materialized or a pipelined evaluation.**

- materialized evaluation

the result of an operation is stored as a temporary relation

For instance, the join operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation

- pipelined evaluation

the resulting tuples of an operation are produced, they are forwarded directly to the next operation in the query sequence

For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm.