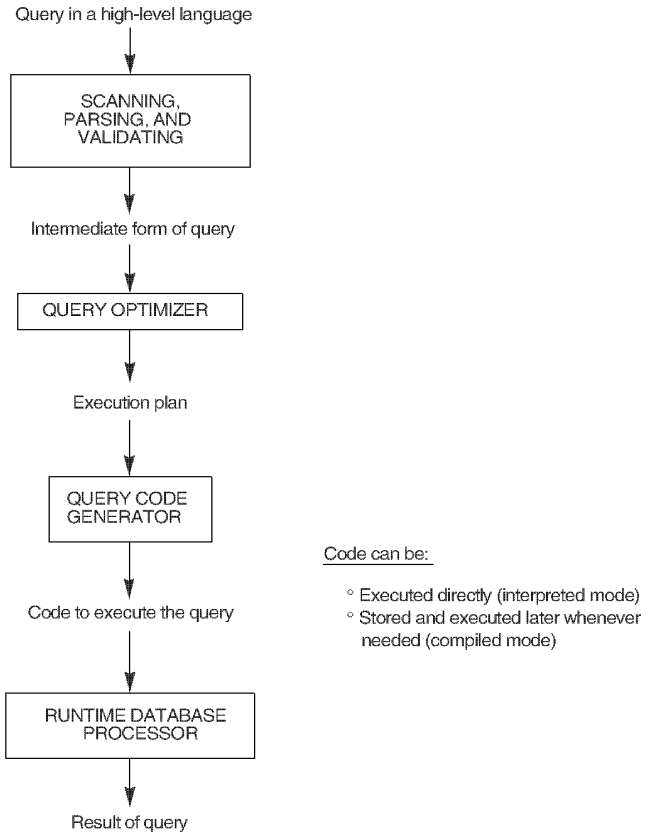
Module-5
Query Processing, Indexing & Tuning Physical Database Design

Dr. Parimala M

SCORE

Introduction to Query Processing (2)

Figure 18.1 Typical steps when processing a high-level query.



1. Translating SQL Queries into Relational Algebra

- **Query block:** the basic unit that can be translated into the algebraic operators and optimized.
- A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clause if these are part of the block.
- **Nested queries** within a query are identified as separate query blocks.
- Aggregate operators in SQL must be included in the extended algebra.

Translating SQL Queries into Relational Algebra

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

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

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

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

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

Using Heuristics in Query Optimization

- **Process for heuristics optimization**

1. The parser of a high-level query generates an *initial internal representation*;
2. Apply heuristics rules to optimize the internal representation.
3. A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

- The **main heuristic** is to apply first the operations that reduce the size of intermediate results.

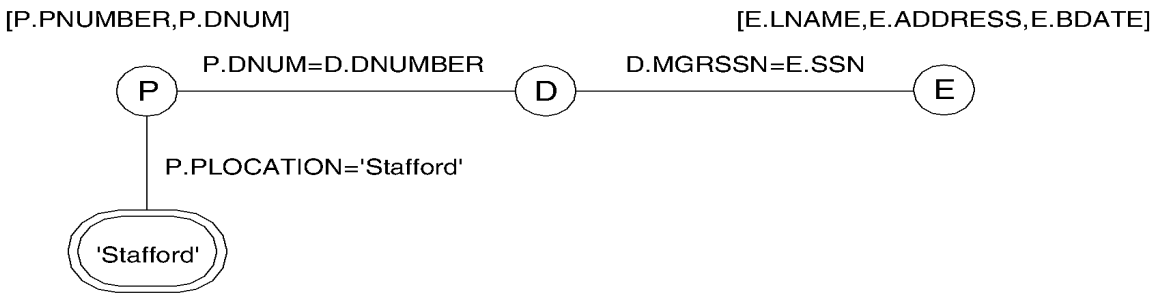
  E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Using Heuristics in Query Optimization

- **Query tree**: a tree data structure that corresponds to a relational algebra expression. It represents the input relations 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.
- **Query graph**: a graph data structure that corresponds to a relational calculus expression. It does **not** indicate an order on which operations to perform first. There is only a **single** graph corresponding to each query.

Query graph

Figure 18.4 (c) Query graph for Q2.



Using Heuristics in Query Optimization

Heuristic Optimization of Query Trees:

- The same query could correspond to many different relational algebra expressions — and hence many different query trees.

- The task of heuristic optimization of query trees is to find a **final query tree** that is efficient to execute.

- **Example**:

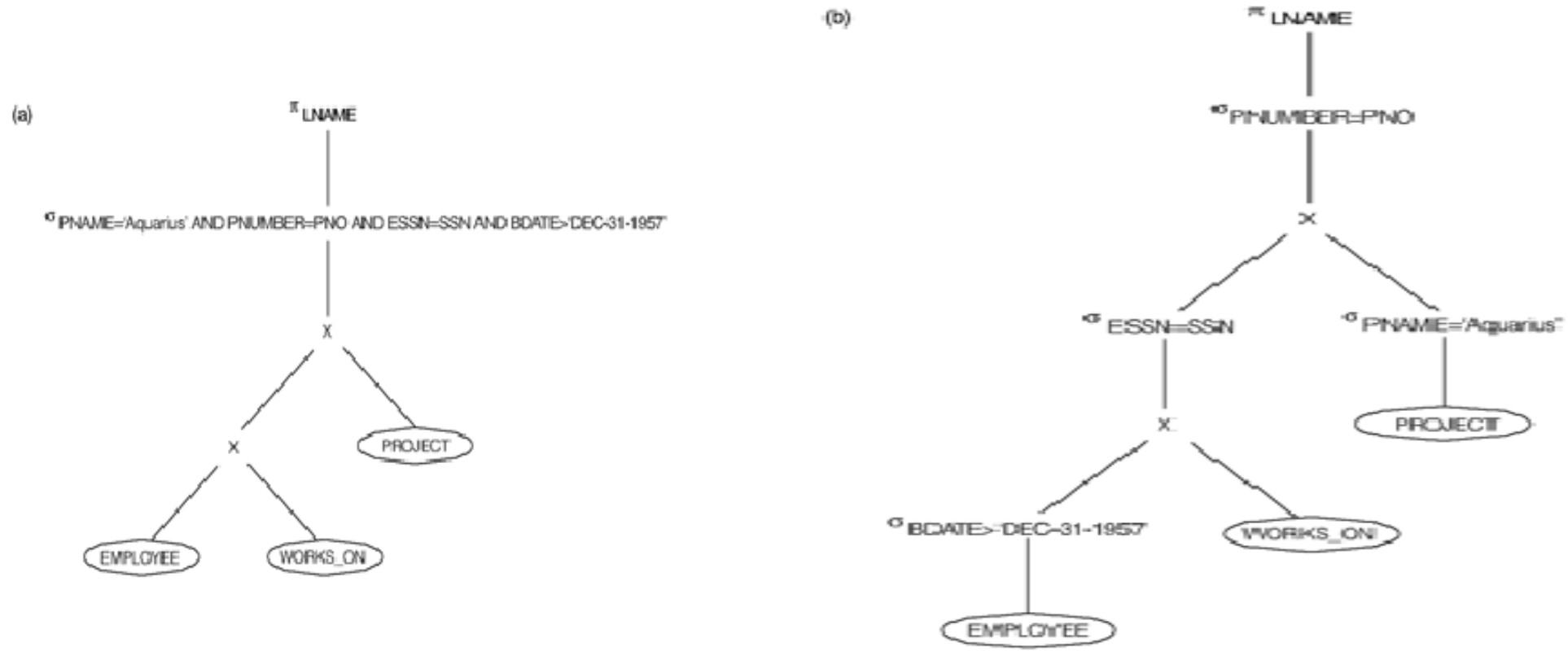
  Q: SELECT LNAME

  FROM EMPLOYEE, WORKS_ON, PROJECT

  WHERE PNAME = 'AQUARIUS' AND PNMUBER=PNO

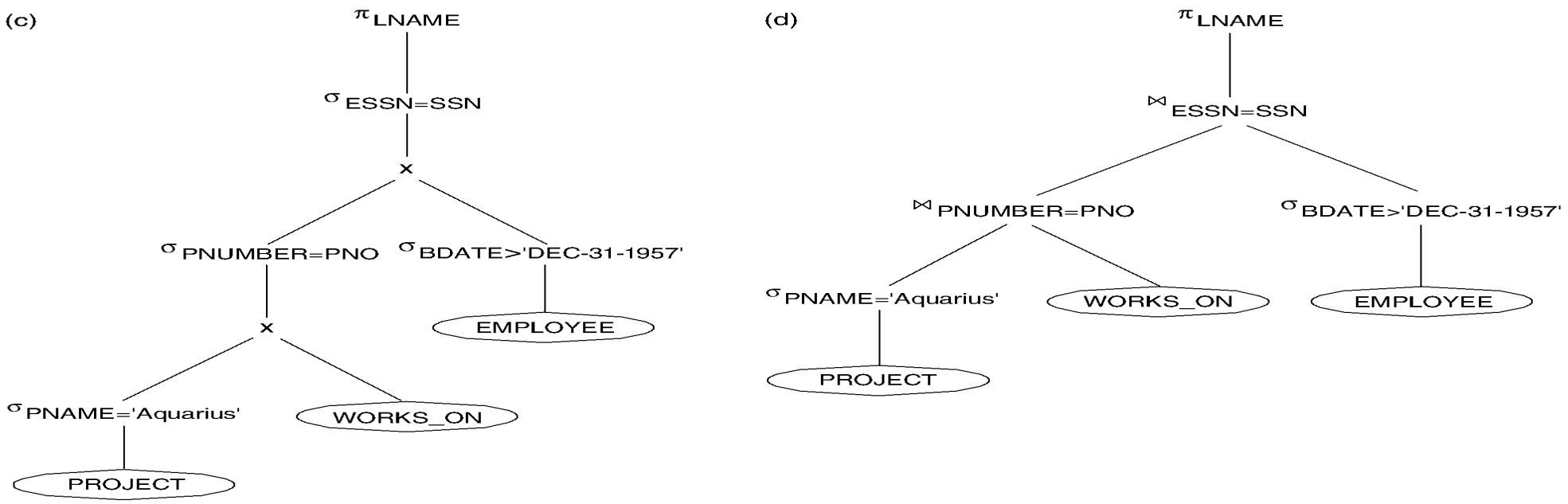
  AND ESSN=SSN AND BDATE > '1957-12-31';

Figure 18.5 Steps in converting a query tree during heuristic optimization. (a) Initial (canonical) query tree for SQL query Q. (b) Moving SELECT operations down the query tree.



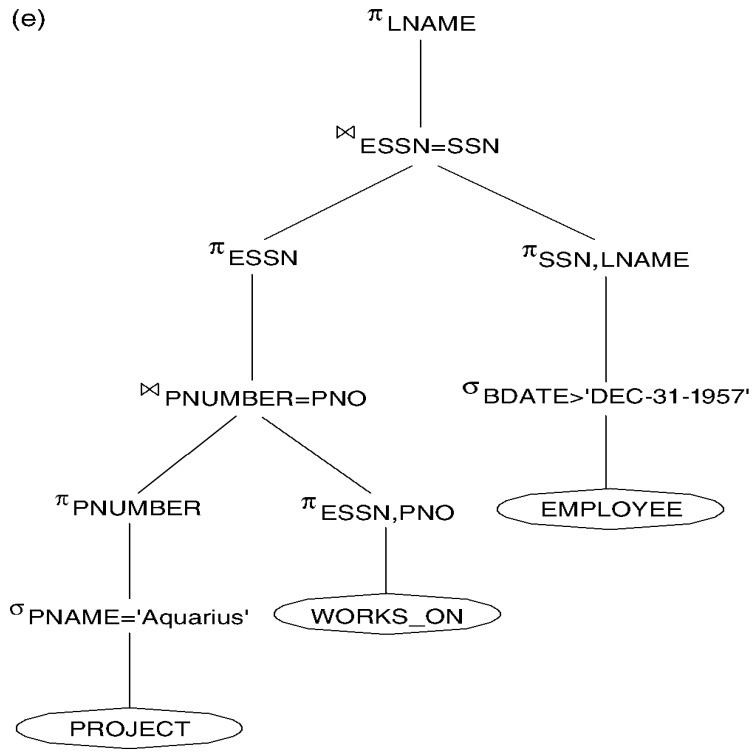
Using Heuristics in Query Optimization

Figure 18.5 Steps in converting a query tree during heuristic optimization. (c) Applying the more restrictive SELECT operation first. (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.



Using Heuristics in Query Optimization

Figure 18.5 Steps in converting a query tree during heuristic optimization. (e) Moving PROJECT operations down the query tree.



Indexing

An index is an access structure that allows us to find the location of records with given values in certain fields which are called index field.

An index may also be defined as an auxiliary file that facilitates fast access to database file.

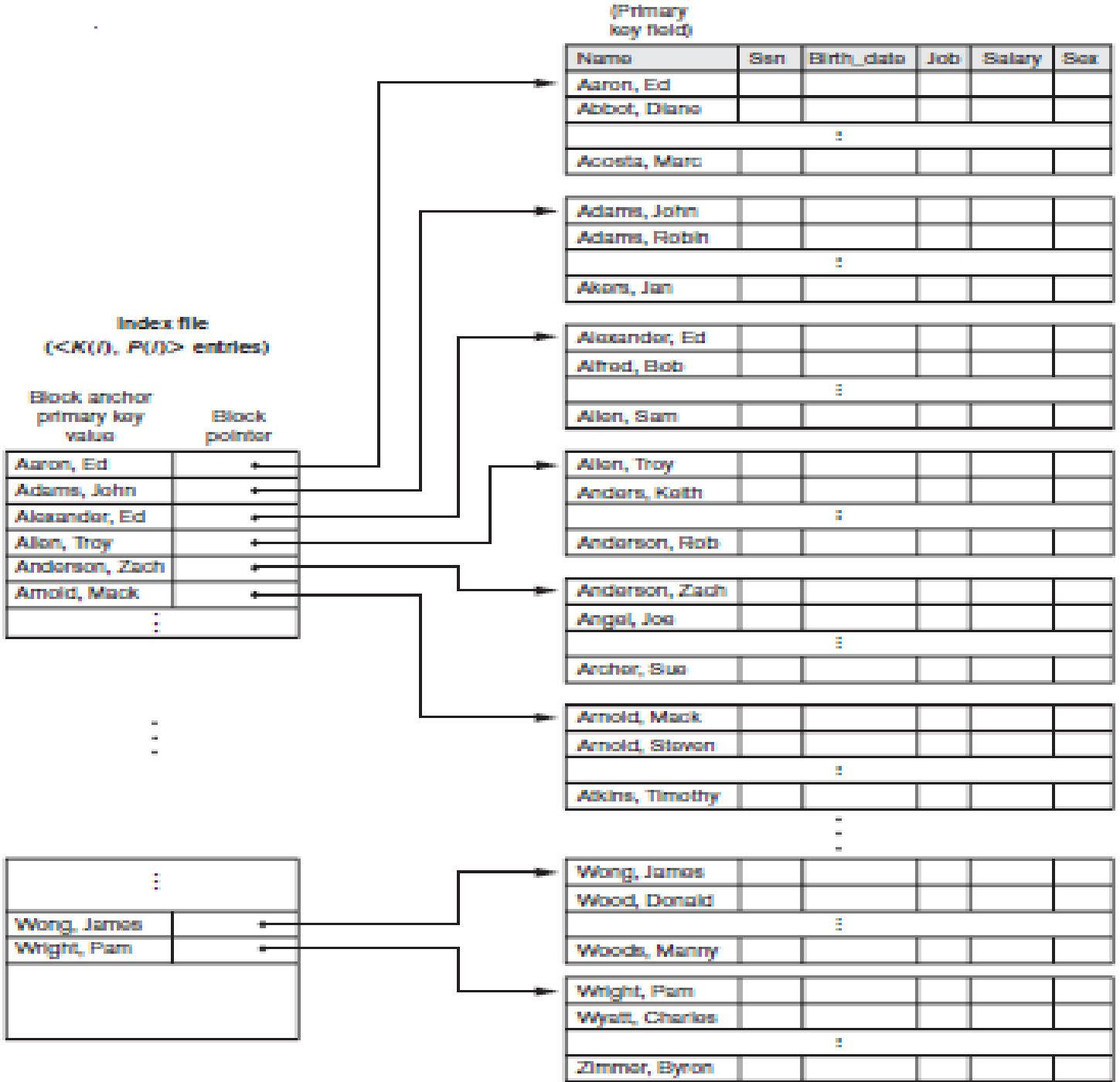
Primary index

- A **primary index** is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access the data records in a data file.

- An index defined on the ordering key field of a data file is called primary index.

- 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)>.

Primary index on ordering key field



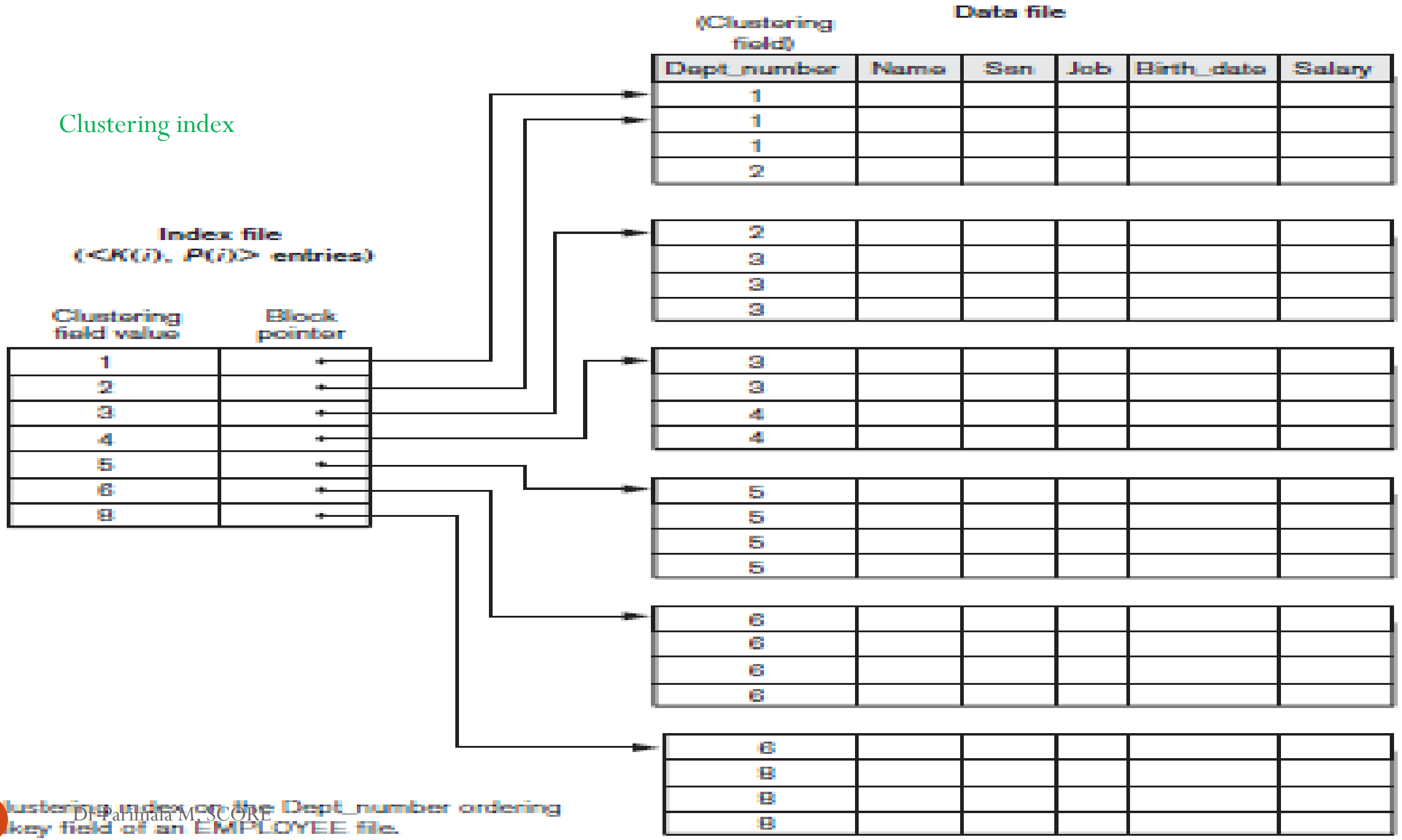
Dense index & Sparse index

- 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** has index entries for only some of the search key values.
- A sparse index has fewer entries than the number of records in the file.
- Thus, a primary index is a sparse index, since it includes an entry for each disk block of the data file.

Clustering index

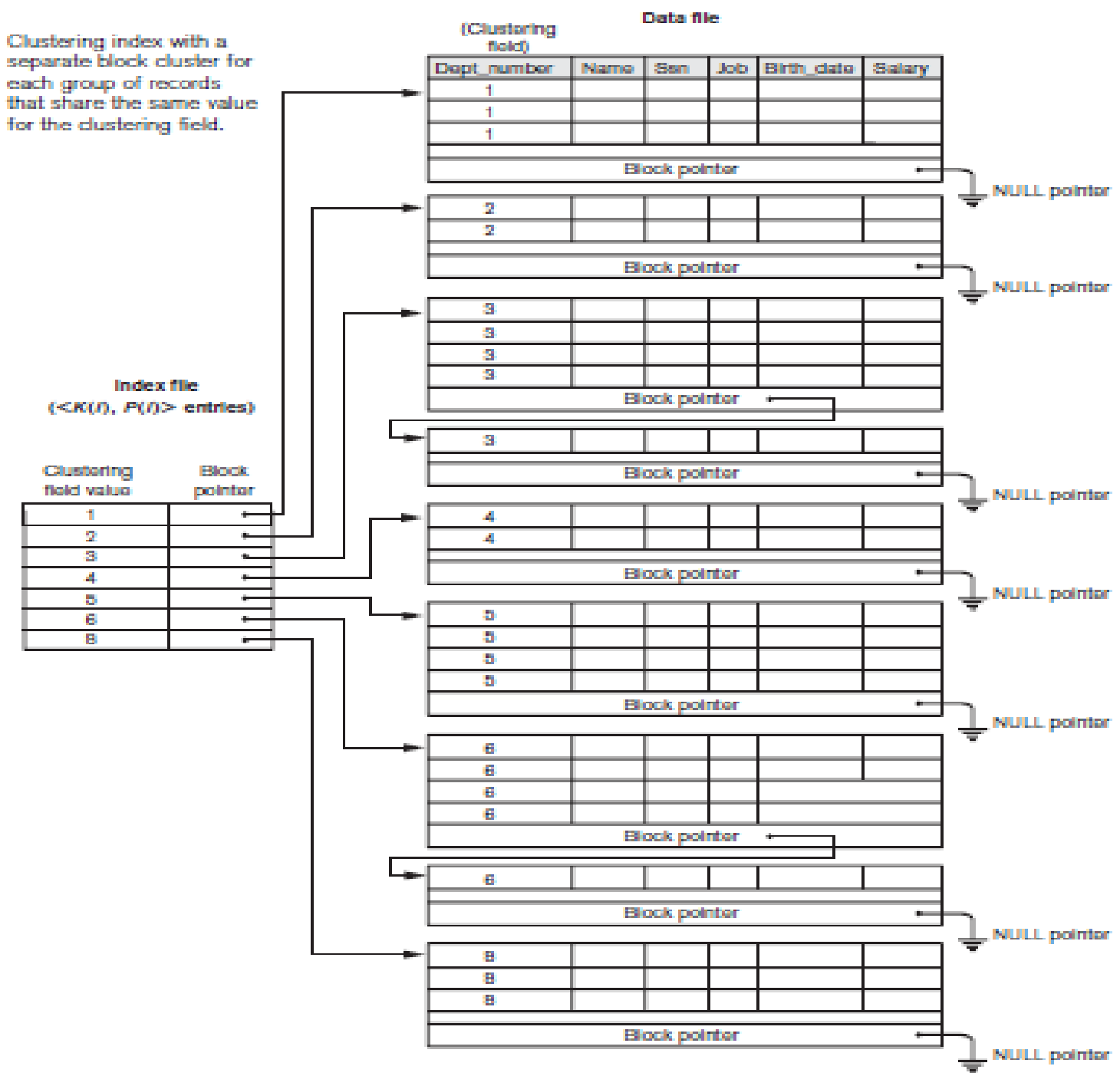
- If file records are physically ordered on a non-key field then this field is called the **clustering field** and index created using clustering field is called clustering index.
- A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a disk block pointer.
- There is one entry in the clustering index for each *distinct value* of the clustering field, and it contains the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field.
- An index defined on the ordering non-key field of data file is called clustering index.

Clustering index



Clustering index on the Dept_number ordering key field of an EMPLOYEE file.

Clustering index with separate block (or a group of blocks) for each value of clustering field.



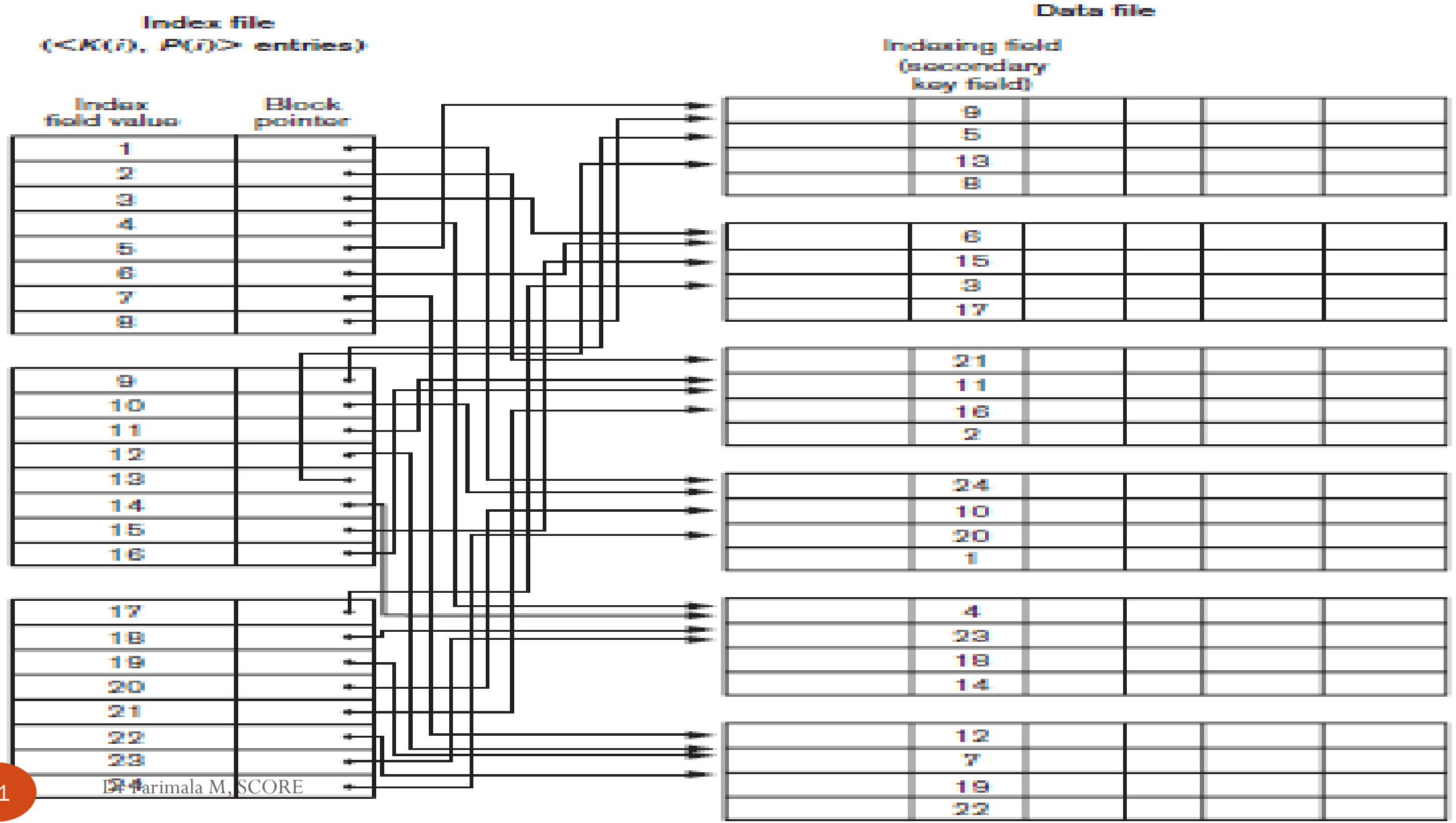
Secondary index

- The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a non-key field with duplicate values.
- The index is again an ordered file with two fields. The first field is of the same data type as some *non-ordering field* of the data file that is an **indexing field**. The second field is either a *block* pointer or a *record* pointer.
- *Many* secondary indexes (and hence, indexing fields) can be created for the same file — each represents an additional means of accessing that file based on some specific field.

Secondary index

- A secondary index on a key (unique) field 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 field 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.**

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



Secondary index with non-key & non-ordering indexing field

- We can also create a secondary index on a *non-key, non-ordering field* of a file.
- In this case, numerous records in the data file can have the same value for the indexing field.
- An index defined on non-ordering field of data file is called secondary index.

There are several options for implementing such an index:

- Option 1: Include duplicate index entries with the same $K(i)$ value—one for each record. This would be a dense index.

Secondary index with non-key & non-ordering indexing field

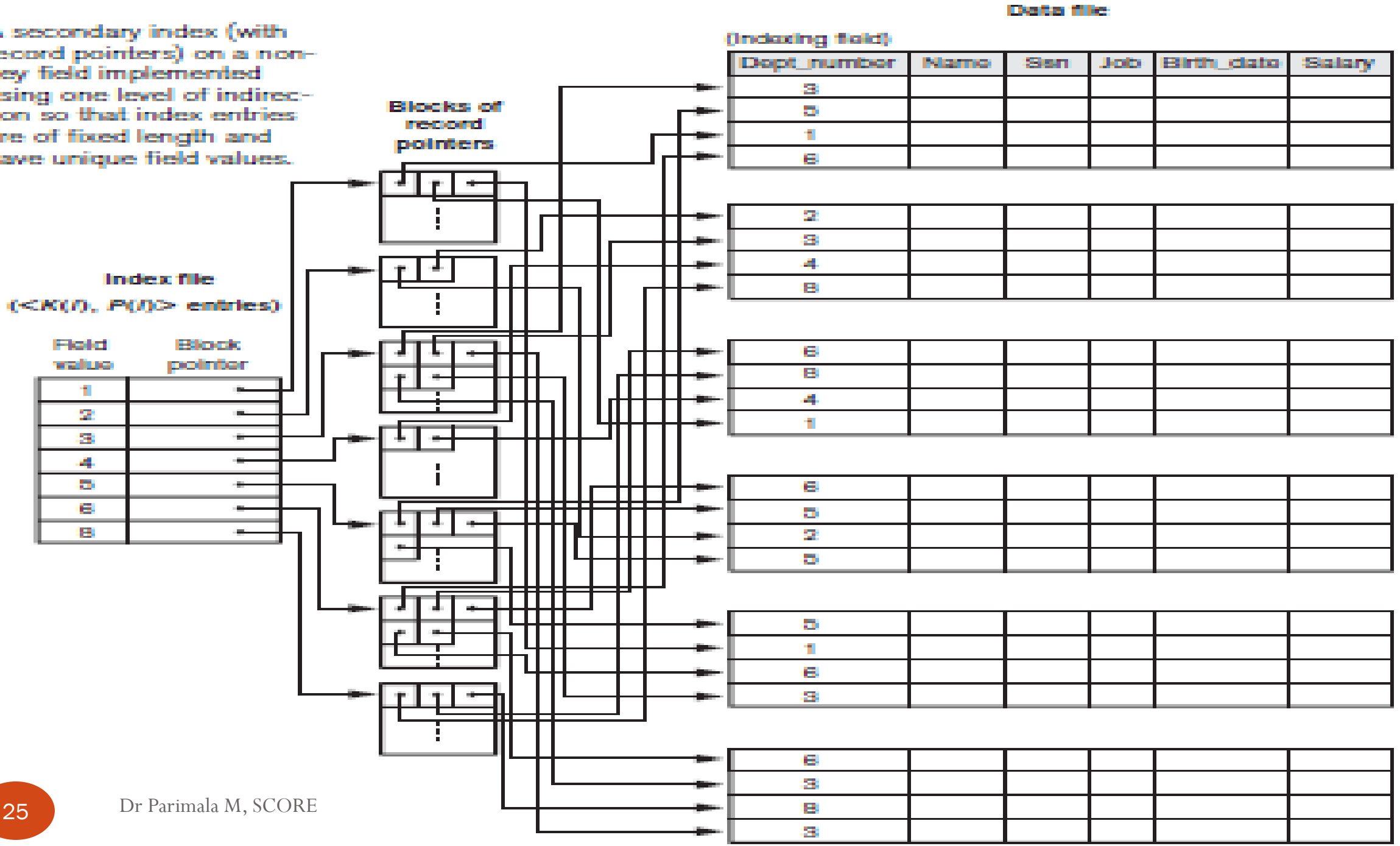
- Option 2: Use variable-length records for the index entries, with a repeating field for the pointer. Index entries in this case are of the form <K(i), P(i, 1), P(i, 2), … P(i, m)>. (*The pointers are block pointers*)

- Option 3: (more commonly used): Keep the index entries themselves at a fixed length and have a single entry for each *index field value*, but to create *an extra level of indirection* to handle the multiple pointers. The index is a sparse index.

Secondary index with non-key & non-ordering indexing field

- The pointer $P(i)$ in index entry $<K(i), P(i)>$ points to a disk block, which contains a *set of record pointers*;
- each record pointer in that disk block points to one of the data file records with value $K(i)$ for the indexing field.
- If some value $K(i)$ occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used.
- This technique is illustrated in Figure on the next slide.

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



Remarks

- A file can have *at most* one primary index or one clustering index, and in addition can have several secondary indexes.

Table 17.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Table 17.2 Properties of Index Types

Type of Index	Number of (First-Level) Index Entries	Dense or Nondense (Sparse)	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

B-tree vs. Hash index

- A B-tree index can be used to evaluate a point query as well as a range query.
- A hash index can be used to evaluate a point query.
- A query is called a range query if at least one selection condition is a range condition.
- E.g. select * from employee where salary > 30000 is a range query.
- If there exists a B-tree index with salary as indexing field then this index can be used to evaluate the query.

B-tree vs. Hash index

- A query is called a point query if there is no selection condition that involves inequality.
- E.g. select * from employee where dno = 5 is a point query.
- If there exists a hash index with dno as indexing field then this index can be used to evaluate the query.
- But a B-tree index on dno field could as well be used to evaluate this query if such an index exists. But hash index is more efficient than B-tree index, if such an index exists.

Factors that Influence Physical Database Design

A. Analyzing the Database Queries and Transactions

For each **retrieval query**, the following information about the query would be needed:

1. The files (relations) that will be accessed by the query

2. The attributes on which any selection conditions for the query are specified

3. Whether the selection condition is an equality, inequality, or a range condition

4. The attributes on which any join conditions or conditions to link multiple tables or objects for the query are specified

5. The attributes whose values will be retrieved by the query

Factors That Influence Physical Database Design

A. For each **update operation** or **update transaction**, the following information would be needed:

1. The files that will be updated

2. The type of operation on each file (insert, update, or delete)

3. The attributes on which selection conditions for a delete or update are specified

4. The attributes whose values will be changed by an update operation

B. Analyzing the Expected Frequency of Invocation of Queries and Transactions

Frequency information, along with the attribute information collected on each query and transaction, is used to compile a cumulative list of the expected frequency of use for all queries and transactions.

This is expressed as the expected frequency of using each attribute in each file as a selection attribute or a join attribute, over all the queries and transactions.

Generally, for large volumes of processing, the informal *80–20 rule* can be used: approximately 80% of the processing is accounted for by only 20% of the queries and transactions.

C. Analyzing the Time Constraints of Queries and Transactions

Some queries and transactions may have stringent performance constraints.

For example, a transaction may have the constraint that it should terminate within 5 seconds on 95% of the occasions when it is invoked, and that it should never take more than 20 seconds. Such timing constraints place further priorities on the attributes that are candidates for access paths.

The selection attributes used by queries and transactions with time constraints become higher-priority candidates for primary access structures for the files, because the primary access structures are generally the most efficient for locating records in a file.

D. Analyzing the Expected Frequencies of Update Operations.

A minimum number of access paths should be specified for a file that is frequently updated, because updating the access paths themselves slows down the update operations.

For example, if a file that has frequent record insertions has 10 indexes on 10 different attributes, each of these indexes must be updated whenever a new record is inserted. The overhead for updating 10 indexes can slow down the insert operations.

E. Analyzing the Uniqueness Constraints on Attributes.

Access paths should be specified on all *candidate key* attributes—or sets of attributes—that are either the primary key of a file or unique attributes.

The existence of an index (or other access path) makes it sufficient to search only the index when checking this uniqueness constraint, since all values of the attribute will exist in the leaf nodes of the index.

For example, when inserting a new record, if a key attribute value of the new record *already exists in the index*, the insertion of the new record should be rejected, since it would violate the uniqueness constraint on the attribute.

Design Decisions about Indexing

- The performance of queries largely depends upon what indexes or hashing schemes exist to expedite the processing of selections and joins.

- On the other hand, during insert, delete, or update operations, the existence of indexes adds to the overhead. This overhead must be justified in terms of the gain in efficiency by expediting queries and transactions.

(i) Whether to index an attribute. The general rules for creating an index on an attribute are that the attribute must either be a key (unique), or there must be some query that uses that attribute either in a selection condition (equality or range of values) or in a join condition

(ii) What attribute or attributes to index on. An index can be constructed on a single attribute, or on more than one attribute if it is a composite index.

(iii) Whether to set up a clustered index. At most, one index per table can be a primary or clustering index, because this implies that the file be physically ordered on that attribute. If the attribute is a *key*, *a primary index* is created, whereas a *clustering index* is created if the attribute is *not a key*

(iv) Whether to use a hash index over a tree index.

B+-trees support both equality and range queries on the attribute used as the search key. Hash indexes work well with equality conditions, particularly during joins to find a matching record(s), but they do not support range queries.

(v) Whether to use dynamic hashing for the file. For files that are very volatile that is, those that grow and shrink continuously—one of the dynamic hashing schemes can be used

Database Tuning

- Conceptual
- Physical
  - Indexing
  - Queries

Physical Database Tuning

Design Decisions about Indexing

- **Whether to index an attribute?**
- **What attribute or attributes to index on?**
- **Whether to set up a clustered index?**
- **Whether to use a hash index over a tree index?**
- **Whether to use dynamic hashing for the file?**

- Certain queries may take too long to run for lack of an index.
- Certain indexes may not get utilized at all.
- Certain indexes may undergo too much updating because the index is on an attribute that undergoes frequent changes.

Query Tuning

- There are mainly two indications that suggest that

  query tuning may be needed:

1. A query issues too many disk accesses (for example, an exact match query scans an entire table).
2. The query plan shows that relevant indexes are not being used.

Situations for requirement of query tuning

- Many query optimizers do not use indexes in the presence of arithmetic

expressions (such as Salary/365 > 10.50), numerical comparisons of attributes of different sizes and precision (such as Aqty = Bqty where Aqty is of type INTEGER and Bqty is of type SMALLINTEGER),

NULL comparisons (such as Bdate IS NULL), and

substring comparisons (such as Lname LIKE '%mann')

- Indexes are often not used for nested queries using IN;

SELECT Ssn **FROM** EMPLOYEE

WHERE Dno **IN** (**SELECT** Dnumber **FROM** DEPARTMENT

WHERE Mgr_ssn = '333445555');

- Some DISTINCTs may be redundant and can be avoided without changing the result. A DISTINCT often causes a sort operation and must be avoided as much as possible.
- Unnecessary use of temporary result tables can be avoided by collapsing multiple queries into a single query *unless* the temporary relation is needed for some intermediate processing.
- In some situations involving the use of correlated queries, temporaries are useful. Consider the following query, which retrieves the highest paid employee in each department:

```
SELECT Ssn FROM EMPLOYEE E
WHERE Salary = SELECT MAX (Salary)
 FROM EMPLOYEE AS M WHERE M.Dno = E.Dno;
```

This has the potential danger of searching all of the inner EMPLOYEE table M for *each* tuple from the outer EMPLOYEE table E.

- To make the execution more efficient, the process can be broken into two queries, where the first query just computes the maximum salary in each department as follows:
- **SELECT** MAX (Salary) **AS** High_salary, Dno **INTO** TEMP
  FROM EMPLOYEE
  GROUP BY Dno;
- **SELECT** EMPLOYEE.Ssn
  FROM EMPLOYEE, TEMP
  WHERE EMPLOYEE.Salary = TEMP.High_salary
  AND EMPLOYEE.Dno = TEMP.Dno;

- If multiple options for a join condition are possible, choose one that uses a clustering index and avoid those that contain string comparisons. For example, assuming that the Name attribute is a candidate key in EMPLOYEE and STUDENT, it is better to use EMPLOYEE.Ssn = STUDENT.Ssn as a join condition

  rather than EMPLOYEE.Name = STUDENT.Name if Ssn has a clustering index in one or both tables.

- One idiosyncrasy with some query optimizers is that the order of tables in the FROM-clause may affect the join processing. If that is the case, one may have to switch this order so that the smaller of the two relations is scanned and the larger relation is used with an appropriate index.
- Some query optimizers perform worse on nested queries compared to their equivalent unnested counterparts. There are four types of nested queries:
  - Uncorrelated subqueries with aggregates in an inner query.
  - Uncorrelated subqueries without aggregates.
  - Correlated subqueries with aggregates in an inner query.
  - Correlated subqueries without aggregates.

Of the four types above, the first one typically presents no problem, since most query optimizers evaluate the inner query once. However, for a query of the second type, most query optimizers may not use an index on Dno in EMPLOYEE. However, the same optimizers may do so if the query is written as an unnested query. Transformation of correlated subqueries may involve setting temporary tables.

- Finally, many applications are based on views that define the data of interest to those applications. Sometimes, these views become overkill, because a query may be posed directly against a base table, rather than going through a view that is defined by a JOIN.

Numerical problems on indexing

Suppose that we have an ordered file with r = 30,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed size and are unspanned with record length R = 100 bytes.

(i) The blocking factor for the file would be bfr =B/R = 1024/100 = 10 records per block.

(ii) The number of blocks needed for the file is b = (r/bfr) = r (30,000/1O) = 3000 blocks.

(iii) A binary search on the data file would need approximately $\log_2 b = \log_2 3000 = 12$ block accesses.

Primary index

Now suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file.

(i) The size of each index entry is $R_i = (9 + 6) = 15$ bytes,

(ii) Blocking factor for the index is $bfr_i = (B/R_i) = (1024/15) = 68$ entries per block.

(iii) The total number of index entries ri is equal to the number of blocks in the data file, which is 3000. The number of index blocks is hence $b_i = (r_i/bfr_i) = (3000/68) = 45$ blocks.

(iv) To perform a binary search on the index file would need $(\log_2 b_i) = (\log_2 45) = 6$ block accesses.

(v) To search for a record using the index, we need one additional block access to the data file for a total of 6 + 1 = 7 block accesses--an improvement over binary search on the data file, which required 12 block accesses.

Secondary index

From the values taken from the previous example of primary index, we can calculate for secondary index as follows,

- In a dense secondary index such as this, the total number of index entries r_i is equal to the *number of records* in the data file, which is 30,000.
- The number of blocks needed for the index is hence $b_i = (r_i / bfr_i) = (30{,}000/68) = 442$ blocks.
- A binary search on this secondary index needs $(\log_2 bi) = (\log_2 442) = 9$ block accesses.
- To search for a record using the index, we need an additional block access to the data file for a total of 9 + 1 = 10 block access

Clustering Index

- Entries in index table depends on the number of unique values present in the non-key field. Remaining calculation are same as for primary and secondary.