CS3CRT06 : Database Management Systems (Core)
Unit IV : Normalization and Indexing

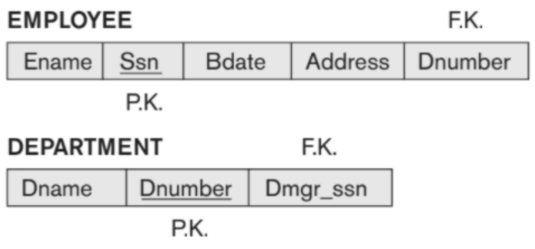
Part 1. Normalization

Informal guidelines for the design of a relation schema.

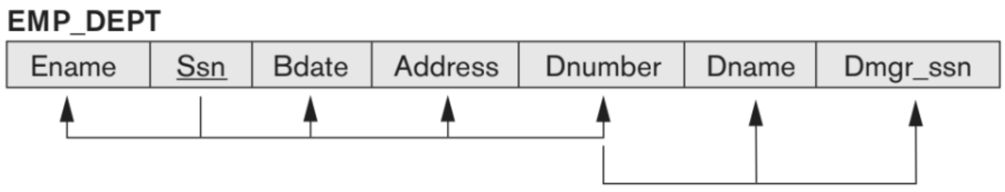
The four Informal Design Guidelines for Relation Schema are :

1. Semantics of the attributes should be clear in the schema
2. Reducing the redundant information in tuples
3. Reducing the NULL values in tuples
4. Disallowing the possibility of generating spurious tuples

1. Clear Semantics to Attributes in Relations:
The easier it is to explain the semantics of a relation, the better the relational schema design will be.
Guideline 1: Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation.
Eg: Tables Employee and Department of the Company Database are having clear semantics.

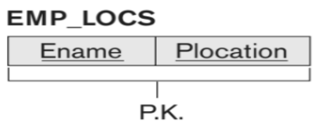


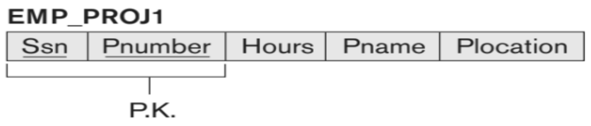
2. Redundant Information in Tuples and Update Anomalies:
To minimize the storage space, grouping attributes of base relations by natural joins leads to an additional problem referred to as **update anomalies**. <u>These can be classified into insertion anomalies, deletion anomalies, and modification anomalies</u>.
Anomalies cause redundant work to be done during insertion and modification of a relation. They may cause accidental loss of information during a deletion from a relation.
Eg: relation suffering from update anomaly



Guideline 2: Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.
3. NULL Values in Tuples:
If many of the attributes of a relation have NULL values, This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes.
Guideline 3: As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL. If NULLs are unavoidable, make sure that they apply in exceptional cases only.
4. Generation of Spurious Tuples:

When we attempt a NATURAL JOIN operation on some relations, the result may produce many more tuples than the original set of tuples. Additional tuples that were not in the base tables are called spurious tuples.
Eg: Natural join of the following tables will generate spurious tuples.





Guideline 4: Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples. Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated.

<u>Functional Dependencies</u>

A functional dependency is a constraint between two sets of attributes from the database. Let A1, A2, ...,An are the n attributes of our relational database schema. Let the whole database is described by the relation schema R = {A1, A2, ... , An}.
A functional dependency, denoted by X →Y, between two sets of attributes X and Y that are subsets of R specifies a constraint. The constraint is that, for any two tuples t1 and t2 in r that have t1[X] = t2[X], they must also have t1[Y] = t2[Y].
X →Y means there is a functional dependency from X to Y, or Y is functionally dependent on X. The abbreviation for functional dependency is FD or f.d. The set of attributes X is called the left-hand side of the FD, and Y is called the right-hand side. Thus, X functionally determines Y in a relation schema R if and only if, whenever two tuples of r(R) agree on their X-value, they must necessarily agree on their Y value.
If X is a candidate key of R, then X →Y for any subset of attributes Y of R .

Relation extensions r(R) that satisfy the functional dependency constraints are called legal relation states of R. <u>Hence, the main use of functional dependencies is to describe a relation schema R by specifying constraints on its attributes that must hold at all times</u>

Example: Consider the relation schema EMP_PROJ. We can see that the following functional dependencies should hold:

a. Ssn→Ename
b. Pnumber →{Pname, Plocation}
c. {Ssn, Pnumber}→Hours

These functional dependencies specify that
(a) the value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename),
(b) the value of a project's number (Pnumber) uniquely determines the project

name (Pname) and location (Plocation), and
(c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours).

Normal Forms Based on Primary Keys

Functional dependencies can be used to specify the semantics of relation schemas. **Normalization of Relations :** The normalization process was first proposed by Codd in 1972. Initially, Codd proposed three normal forms namely first, second, and third normal form. A stronger definition of 3NF is called Boyce-Codd normal form (BCNF). BCNF was proposed later by Boyce and Codd.

Definition: The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

Definition: Normalization of data can be considered as a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of (1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies. If the relation schemas do not meet certain conditions, then **they are** decomposed into smaller relation schemas with desirable properties.

Denormalization: There are higher normal forms such as the 4NF and 5NF. The database design in industry normalizes only up to 3NF, BCNF, or at most 4NF. The database designers need not normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF, for performance reasons,
Definition: Denormalization is the process of keeping the join of higher normal form relations as a base relation, which is in a lower normal form.

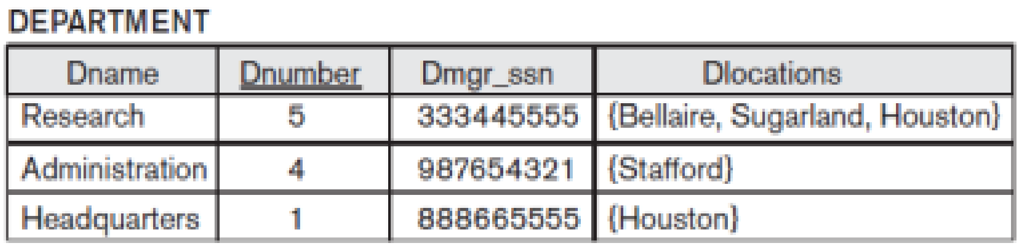
Definitions of Keys:
A **superkey** of a relation schema R = {A1, A2, ... , An} is a set of attributes S belonging to R with the property that no two tuples t1 and t2 in any legal relation state r of R will have t1[S] = t2[S].
A key is a minimal super key. A **key** K is a super key with the additional property that removal of any attribute from K will cause K not to be a super key any more. Eg: {Ssn} is a key for EMPLOYEE, whereas {Ssn}, {Ssn, Ename}, {Ssn, Ename, Bdate}, and any set of attributes that includes Ssn are all superkeys.

If a relation schema has more than one key, each is called a **candidate key**. One of the candidate keys is designated to be the **primary key**, and the others are called secondary keys or alternate keys. In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a super key.

An attribute of relation schema *R* is called a **prime attribute** of *R* if it is a member of *some candidate key* of *R*. An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.
Both Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

First Normal Form

Definition : First normal form (1NF) states that the domain of an attribute must include only atomic (indivisible) values. The value of any attribute in a tuple must be a single value from the domain of that attribute. The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.
1NF disallows multi-valued attributes, composite attributes, and their combinations. 1NF disallows relations within relations or relations as attribute values within tuples.
Eg: The following DEPARTMENT relation schema is not in 1NF.

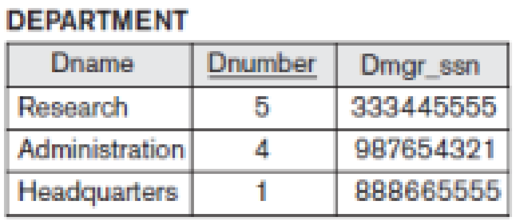


DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

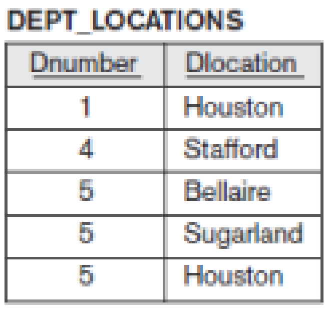
Its primary key is Dnumber. Assume that each department can have *a number of* locations. The DEPARTMENT relation is not in 1NF because Dlocations is not an atomic attribute.

There are three main techniques to achieve first normal form for such a relation:
1. Remove the attribute Dlocations that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key Dnumber of DEPARTMENT. The primary key of this relation is the combination {Dnumber, Dlocation} as shown.



DEPARTMENT

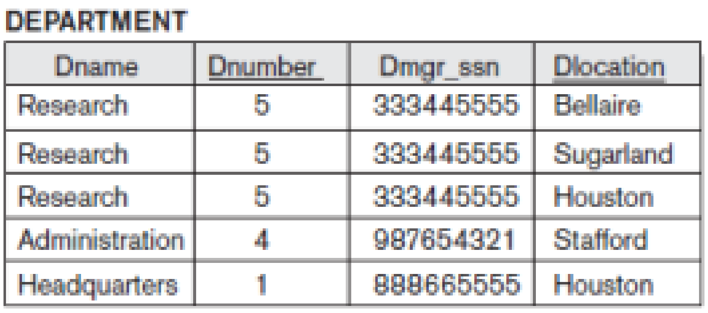
Dname	Dnumber	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555



DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

This decomposes the non-1NF relation into two 1NF relations.
2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT.



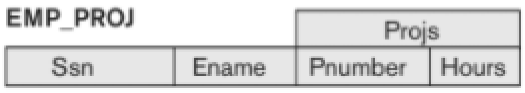
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

3. If a *maximum number of values* is known for the attribute—for example, if it is known that *at most three locations* can exist for a department—replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3. This solution has the disadvantage of introducing *NULL values* if most departments have fewer than three locations.

The first option is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values.

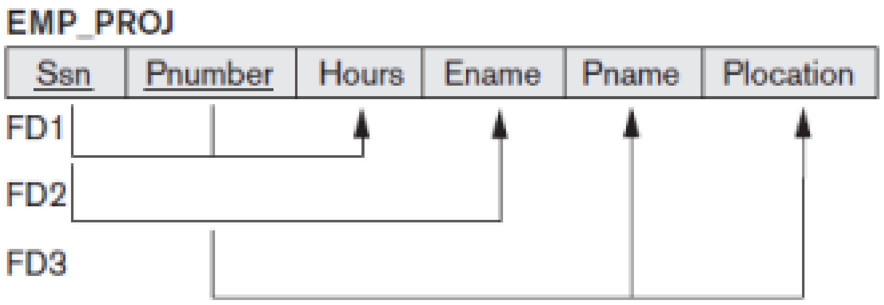
First normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation within it. Figure shows how the EMP_PROJ relation could appear if nesting is allowed.



Each tuple represents an employee entity, and a relation PROJS(Pnumber,Hours) within each tuple represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be
represented as follows:
EMP_PROJ(Ssn, Ename, {PROJS(Pnumber, Hours)})

<u>Second Normal Form</u>

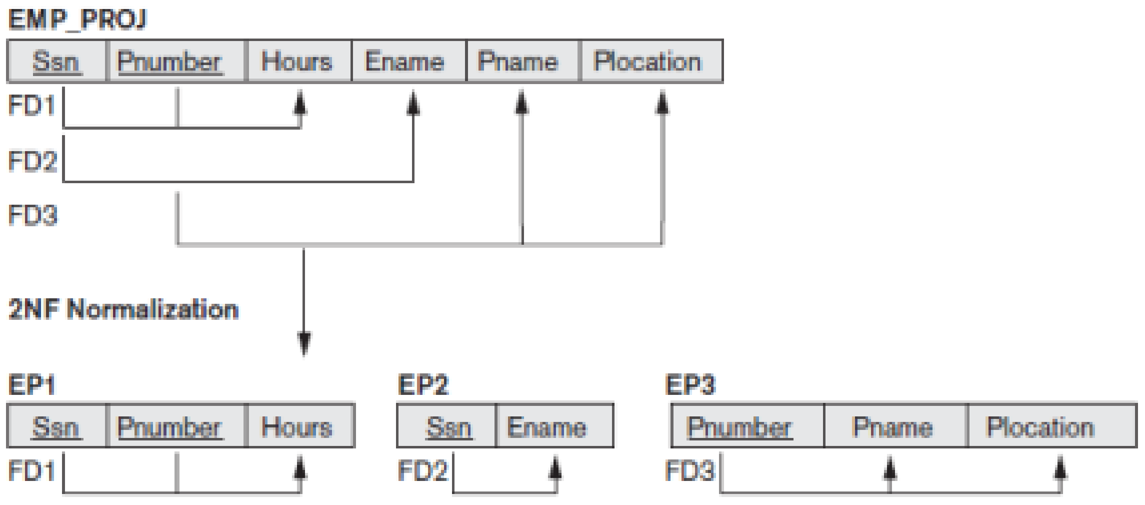
Second normal form (2NF) is based on the concept of full functional dependency.
Definition: A relation schema *R* is in **second normal form (2NF)** <u>if every nonprime attribute *A* in *R* is fully functionally dependent on *any* key of *R*.</u>
A functional dependency X →Y is a **full functional dependency** if removal of any attribute A from X makes the dependency not hold any more. That is, for any attribute A element of X, (X – {A}) does not functionally determine Y.
A functional dependency X→Y is a **partial dependency** if some attribute A element of X can be removed from X and the dependency still holds. That is, for some A element of X, (X – {A}) →Y.



In Figure {Ssn, Pnumber} →Hours is a full dependency (neither Ssn →Hours nor Pnumber→Hours holds). However, the dependency {Ssn, Pnumber}→Ename is partial because Ssn→Ename holds.

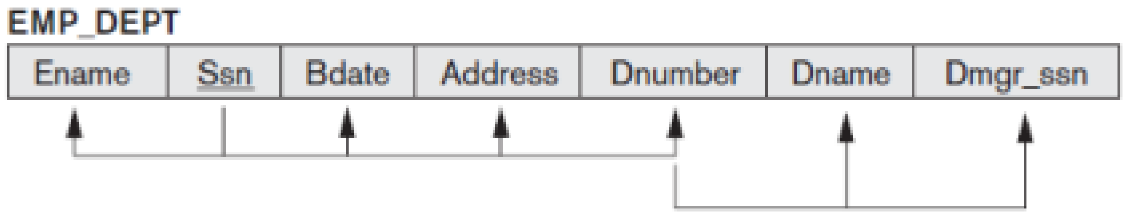
Definition : A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R.
The EMP_PROJ relation in Figure is in 1NF but is not in 2NF. The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent on the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test.
If a relation schema is not in 2NF, it can be second normalized or 2NF normalized into a

number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in figure each of which is in 2NF.

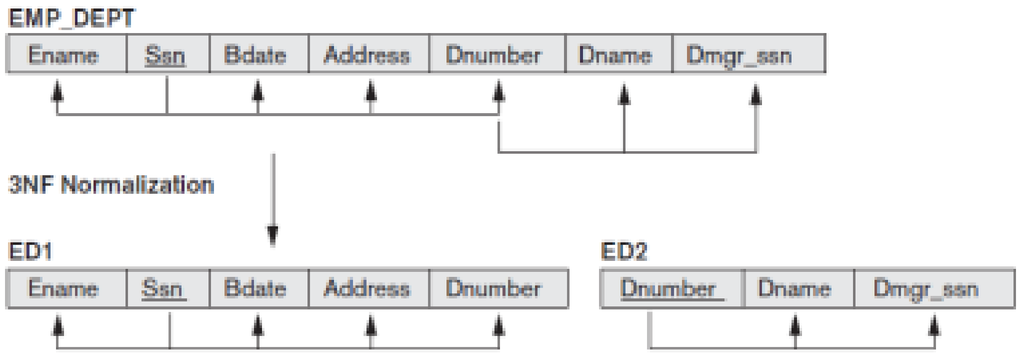


<u>Third Normal Form</u>

Third normal form (3NF) is based on the concept of *transitive dependency*.
Definition. A relation schema *R* is in **3NF** <u>if it satisfies 2NF *and* no nonprime attribute of *R* is transitively dependent on the primary key.</u>
A functional dependency *X*→*Y* in a relation schema *R* is a **transitive dependency** if there exists a set of attributes *Z* in *R* that is neither a candidate key nor a subset of any key of *R*, and both *X*→*Z* and *Z*→*Y* hold.



The dependency Ssn→Dmgr_ssn is transitive through Dnumber in EMP_DEPT because both the dependencies Ssn →Dnumber and Dnumber →Dmgr_ssn hold *and* Dnumber is neither a key itself nor a subset of the key of EMP_DEPT.
The relation schema EMP_DEPT is in 2NF, since no partial dependencies on a key exist. But, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (and also Dname) on Ssn via Dnumber. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure



General Definitions of Second and Third Normal Forms

As a general definition of **prime attribute**, an attribute that is part of *any candidate key* will be considered as prime.

General Definition of Second Normal Form

Definition. A relation schema *R* is in **second normal form (2NF)** if every nonprime attribute *A* in *R* is not partially dependent on *any* key of *R*.

General Definition of Third Normal Form

Definition. A relation schema *R* is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency *X*→*A* holds in *R*, either (a) *X* is a super key of *R*, or (b) *A* is a prime attribute of *R*.

Alternative Definition. A relation schema *R* is in 3NF if every nonprime attribute of *R* meets both of the following conditions:

- It is fully functionally dependent on every key of *R*.
- It is non-transitively dependent on every key of *R*.

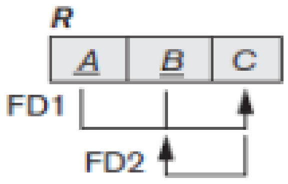
Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; But, a relation in 3NF is not necessarily in BCNF.

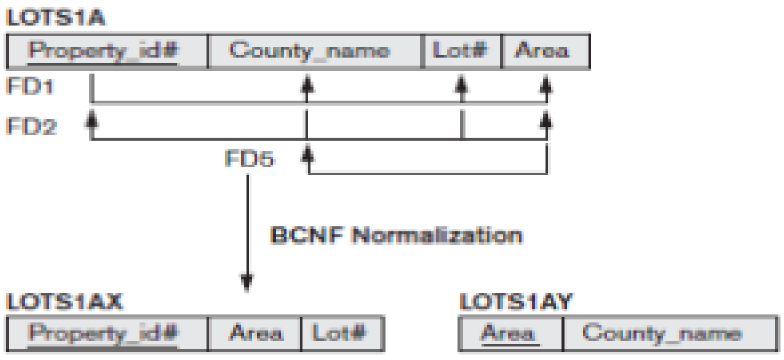
A relation schema *R* is in **BCNF** if whenever a *nontrivial* functional dependency *X*→*A* holds in *R*, then *X* is a super key of *R*.

The difference of BCNF from 3NF: The condition which allows *A* to be prime, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF.

Most relation schemas that are in 3NF are also in BCNF. Only if *X*→*A* holds in a relation schema *R* with *X* not being a superkey *and A* being a prime attribute will *R* be in 3NF but not in BCNF.



The relation schema *R* shown in the figure illustrates the general case of such a relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema.



Consider the relation schema LOTS1A shown in Figure. It describes lands for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}. That is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.
Based on the two candidate keys Property_id# and {County_name, Lot#}, the functional dependencies FD1 and FD2 hold. We choose Property_id# as the primary key, so it is underlined. The area of a lot that determines the county, is specified by FD5,
In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. Note that FD5 satisfies 3NF in LOTS1A because County_name is a prime attribute (condition b), but this condition does not exist in the definition of BCNF. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure. This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

Note: The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized. The process of normalization must also confirm the existence of the following properties that the relational schemas, should possess. 1. The **non-additive join or lossless join property**, which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.
2. The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

Update Anomalies

Grouping attributes of two base relations into one relation may increase the use of storage space. But we need to minimize the storage space.
For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT with that for an EMP_DEPT base relation, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT.

Fig1: Employee-Department Database Design 1

EMPLOYEE

Ename	Ssn	Bdate	Address	Dnumber
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1

DEPARTMENT

Dname	Dnumber	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

Fig2: Employee-Department Database Design 2

Redundancy

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

In EMP_DEPT relation, the attribute values of department's information (Dnumber, Dname, Dmgr_ssn) are repeated for every employee who works for that department. In contrast, each department's information appears only once in the DEPARTMENT relation.
Storing natural joins of base relations leads to an additional problem referred to as **update anomalies.** These can be classified into insertion anomalies, deletion anomalies, and modification anomalies. These anomalies were identified by Codd to justify the need for normalization of relations.

Insertion Anomalies: Consider the EMP_DEPT relation:
1. To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs . But in the Employee relation, we do not have to worry about this consistency problem because we enter only the department number in the employee tuple; all other attribute values of department 5 are recorded only once in the DEPARTMENT relation.
2. It is difficult to insert a new department that has no employees in the EMP_DEPT relation.
These anomalies violates the entity integrity for EMP_DEPT.

Deletion Anomalies: The problem of deletion anomalies is related to the second insertion anomaly situation. If we delete from EMP_DEPT an employee who is the only employee working for a particular department, the information concerning that department is lost from the database. This problem does not occur in DEPARTMENT relation because tuples are stored separately.

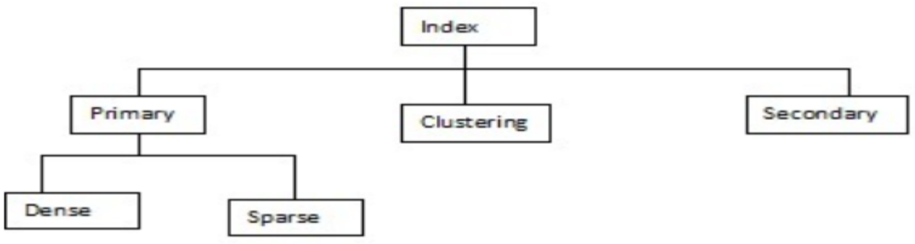
Modification Anomalies: In EMP_DEPT, if we change the value of one of the attributes of a particular department (say, the manager of department 5), we must update the tuples of all employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.
It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data. So we need normalization process in database design.

Part 2 : Indexing Structures for Files

The common file organization methods are ordered, unordered, and hashed file organizations. **Indexes** are additional **access structures** used to retrieve records quickly. The index structures are additional files on disk. The indexes provide **secondary access paths**.
The index file is much smaller than the data file. An index record is a (key value, address) pair. To find a record in the data file based on a search condition on an indexing field, the index is searched. The corresponding pointer leads to the disk block in the data file where the record is located.

The additional indexes on single-level ordered indexes leads to **multilevel indexes.**

Single-Level Ordered Indexes :
An ordered index is similar to the index used at the end of a textbook (Indexes in a book lists important terms in alphabetical order with page numbers).
An index structure is usually defined on a single field of a file, called an **indexing field.** The index file stores each value of the index field with a list of pointers to disk blocks that contain records with that field value. In **Single-Level Ordered Indexes,** the values in the index are *ordered.* We can do a *binary search* on this index.
The different types of ordered indexes are:
1. Primary
2. Secondary
3. Clustering



A **primary index** can be specified on the key field of an **ordered file** of records. Every record in it has a *unique value* for that field. If the ordering field is not a key field, another type of index, called a **clustering index**, can be used. A **secondary index** can be specified on any *non-ordering* field of a file. A data file can have several secondary indexes in addition to its primary index.

1. Primary Indexes:

Primary Index is an ordered file whose records are of fixed length with two fields. The first field of the index contains the primary key of the data file in an ordered manner, and the second field of the ordered file contains a pointer that points to the data-block where a record containing the key is available. A **primary index** is used to efficiently search and access the data records in a data file, ordered by a primary key. There is one **index record** in the index file for each *block* in the data file. Each index record has the value of the primary key field of the *first* record in a block and a pointer to that block.

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

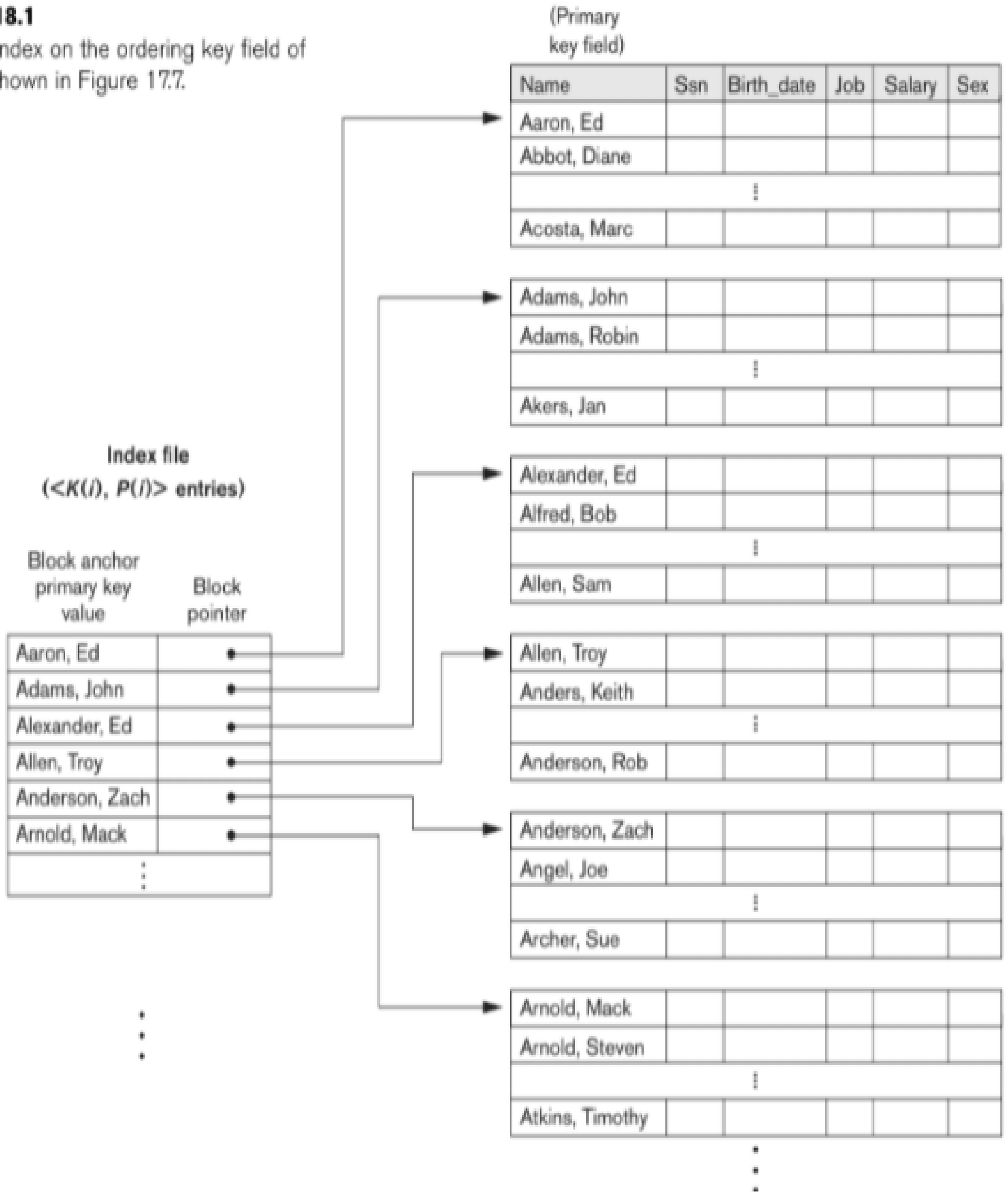


Figure illustrates the primary index. The total number of entries in the index is the same as the *number of disk blocks* in the ordered data file. The first record in each block of the data file is called the **anchor record** (or **block anchor**).
Indexes can also be classified as dense or sparse. A **dense index** has an index entry for *every search key value* in the data file. A **sparse** (or **nondense**) **index**, has index entries for only some of the search values. A sparse index has fewer entries than the number of records in the file. Thus, a primary index is a nondense (sparse) index.
The index file for a primary index occupies a much smaller space than the data file. The two reasons are: First, there are *fewer index entries* than the number of records in the data file. Second, each index entry is *smaller in size* than a data record because it has only two fields. So more index entries than data records can fit in one block. A binary search on the index file requires fewer block accesses than a binary search on the data file.

Let the index entry i of the index file be $<K(i), P(i)>$. A record whose primary key value is K lies in the block whose address is $P(i)$, where $K(i) \leq K < K(i + 1)$. To retrieve a record with primary key value K, we do a binary search on the index file to find the appropriate index entry i, and then retrieve the data file block whose address is $P(i)$.

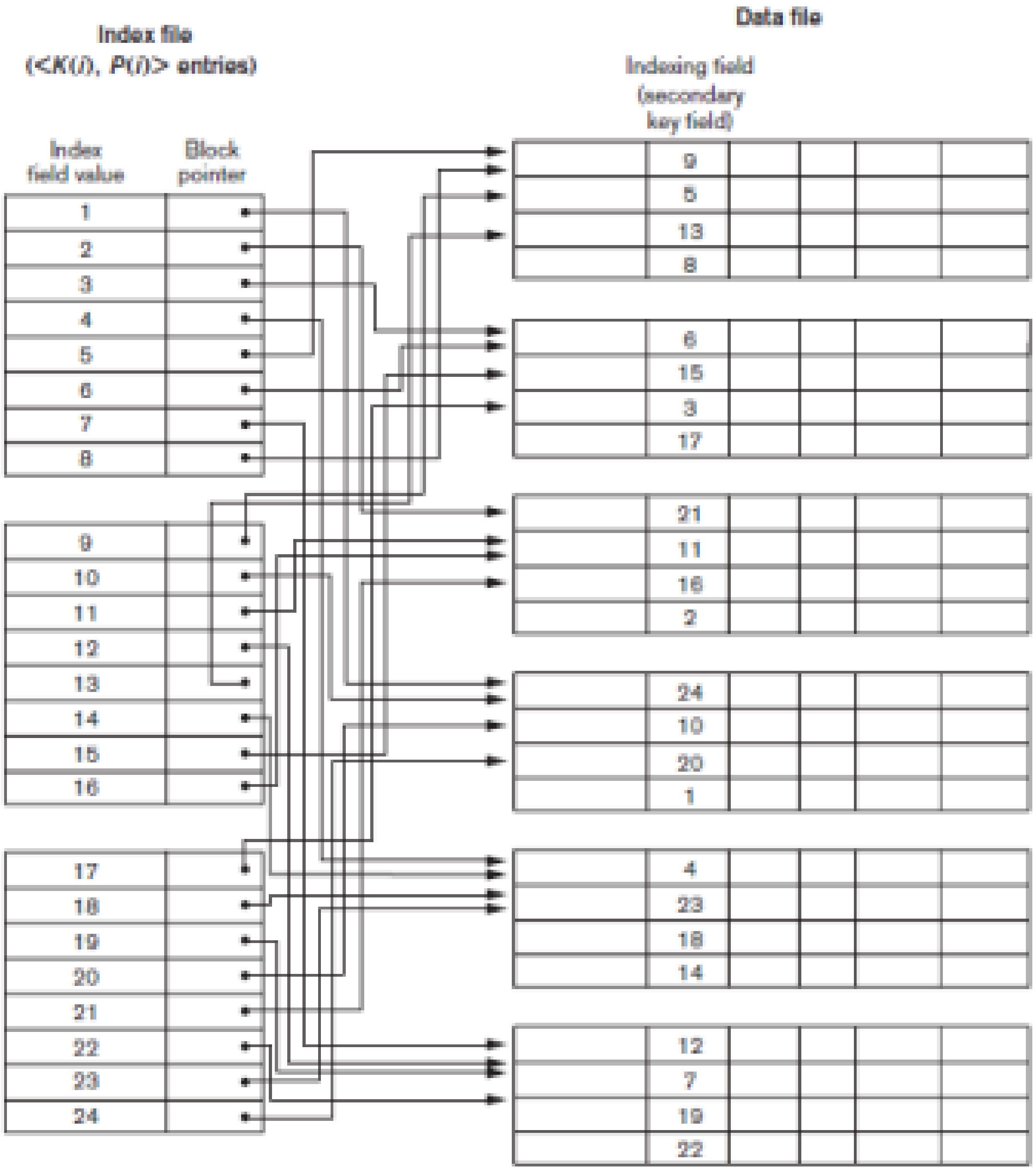
The Example illustrates the saving in block accesses that is attainable when a primary index is used to search for a record.
A major problem with a primary index is insertion and deletion of records. To insert a record in the data file, we have to move records to make space for the new record and also change some index entries (since moving records will change the *anchor records* of some blocks.) Using an unordered overflow file can reduce this problem. Another possibility is to use a linked list of overflow records for each block in the data file.

Example 1. 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. The blocking factor for the file would be $bfr = (B/R) = (1024/100) = 10$ records per block. The number of blocks needed for the file is $b = (r/bfr) = (30000/10) = 3000$ blocks. A binary search on the data file would need approximately $\log_2 b = (\log_2 3000) = 12$ block accesses. Now suppose that the length of the index key field is 9 bytes, and the length of the pointer is 6 bytes long. The size of each index entry is $Ri = (9 + 6) = 15$ bytes, so the blocking factor for the index is $bfri = (B/Ri) = (1024/15) = 68$ entries per block. The total number of index entries r_i is equal to the number of blocks in the data file, which is 3000. The number of index blocks is hence $bi = (ri/bfri) = (3000/68) = 45$ blocks. To perform a binary search on the index file we need $(\log_2 bi) = (\log_2 45) = 6$ block accesses. 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. *It is* an improvement over binary search on the data file, which required 12 disk block accesses.

2. Secondary Indexes
A **secondary index** provides a secondary means of accessing a data file for which some primary access already exists. The data file records could be ordered, unordered, or hashed. 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. Secondary index is also 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 can be

created for the same file. Each represents an additional means of accessing that file based on some specific field.
Consider a secondary index access structure on a key (unique) field that has a distinct value for every record. Such a field is sometimes called a **secondary** key. In the relational model, this would correspond to any UNIQUE key attribute or to the primary key attribute of a table. 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 called **dense index.**

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



We 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.
Figure illustrates a secondary index in which the pointers P(i) in the index entries are block pointers, not record pointers. Once the appropriate disk block is transferred to a main memory buffer, 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 a primary index, because of its larger number of entries. But, the improvement 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.

Example .

Consider the file of Example 1 with *r* = 30,000 fixed-length records of size *R* = 100 bytes stored on a disk with block size *B* = 1024 bytes. The file has *b* = 3000 blocks. Suppose we want to search for a record with a specific value for the secondary key—a nonordering key field of the file that is *V* = 9 bytes long. Without the secondary index, to do a linear search on the file would require *b*/2 = 3000/2 = 1500 block accesses on the average. Suppose that we construct

a secondary index on that *nonordering key* field of the file. Let a block pointer is *P* = 6 bytes long, so each index entry is *Ri* = (9 + 6) = 15 bytes, and the blocking factor for the index is bfr_i =(*B*/*Ri*)=(1024/15)= 68 entries per block. 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 *bi* = (*ri* /*bfri*)= (3000/68)= 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 accesses. It is a vast improvement over the 1500 block accesses needed on the average for a linear search, but slightly worse than the 7 block accesses required for the primary index. This difference arose because the primary index was nondense and hence shorter, with only 45 blocks in length.

We can also create a secondary index on a *nonkey, nonordering field* of a file. In this case, numerous records in the data file can have the same value for the indexing field. There are several options for implementing such an index:

■ Option 1 is to include duplicate index entries with the same *K*(*i*) value—one for each record. This would be a dense index.

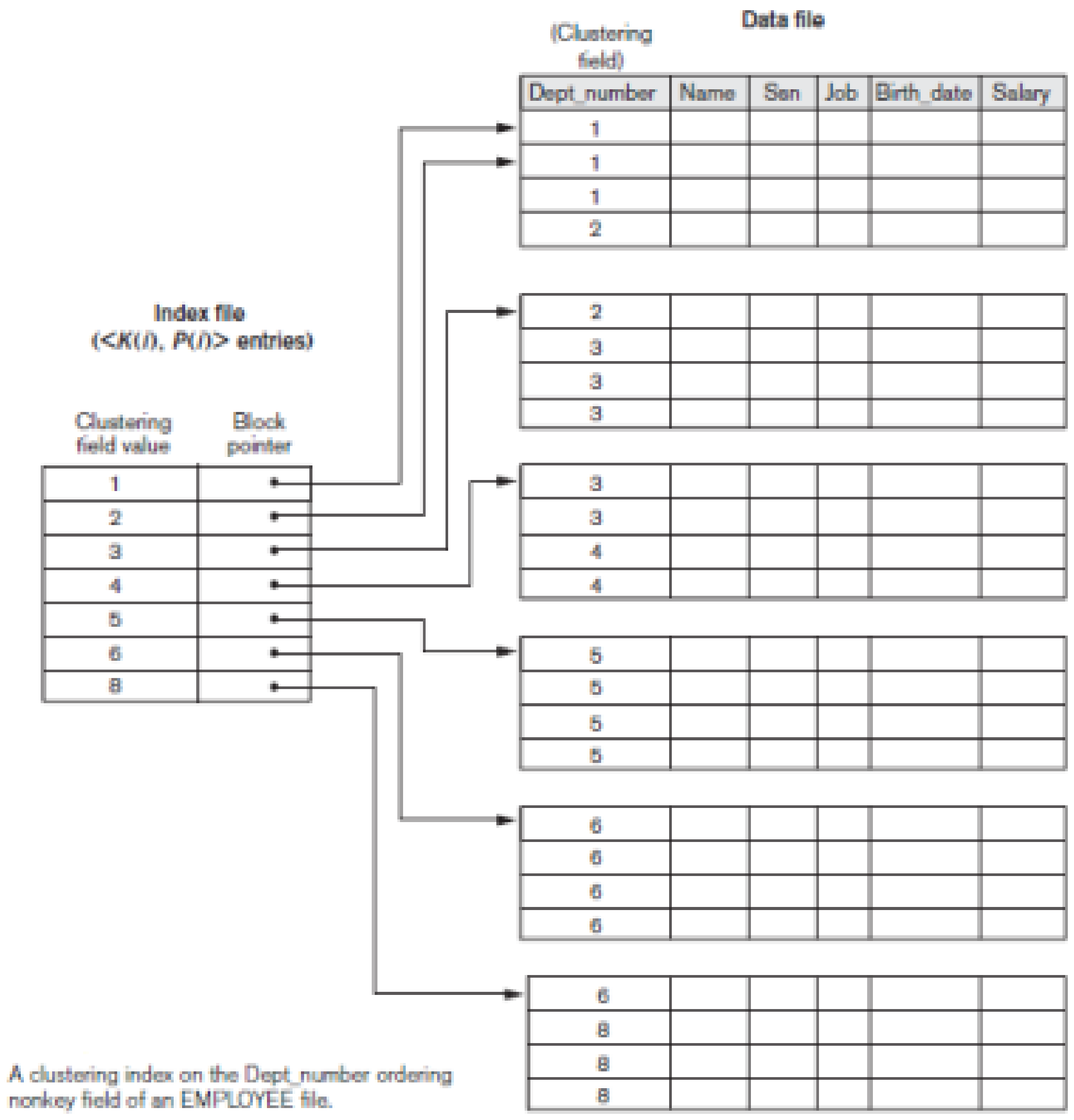
■ Option 2 is to have variable-length records for the index entries, with a repeating field for the pointer.We keep a list of pointers <*P*(*i*, 1), ..., *P*(*i*, *k*)> in the index entry for *K*(*i*)—one pointer to each block that contains a record whose indexing field value equals *K*(*i*). In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately to account for a variable number of index entries per index key value.

■ Option 3, which is more commonly used, is to 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 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.

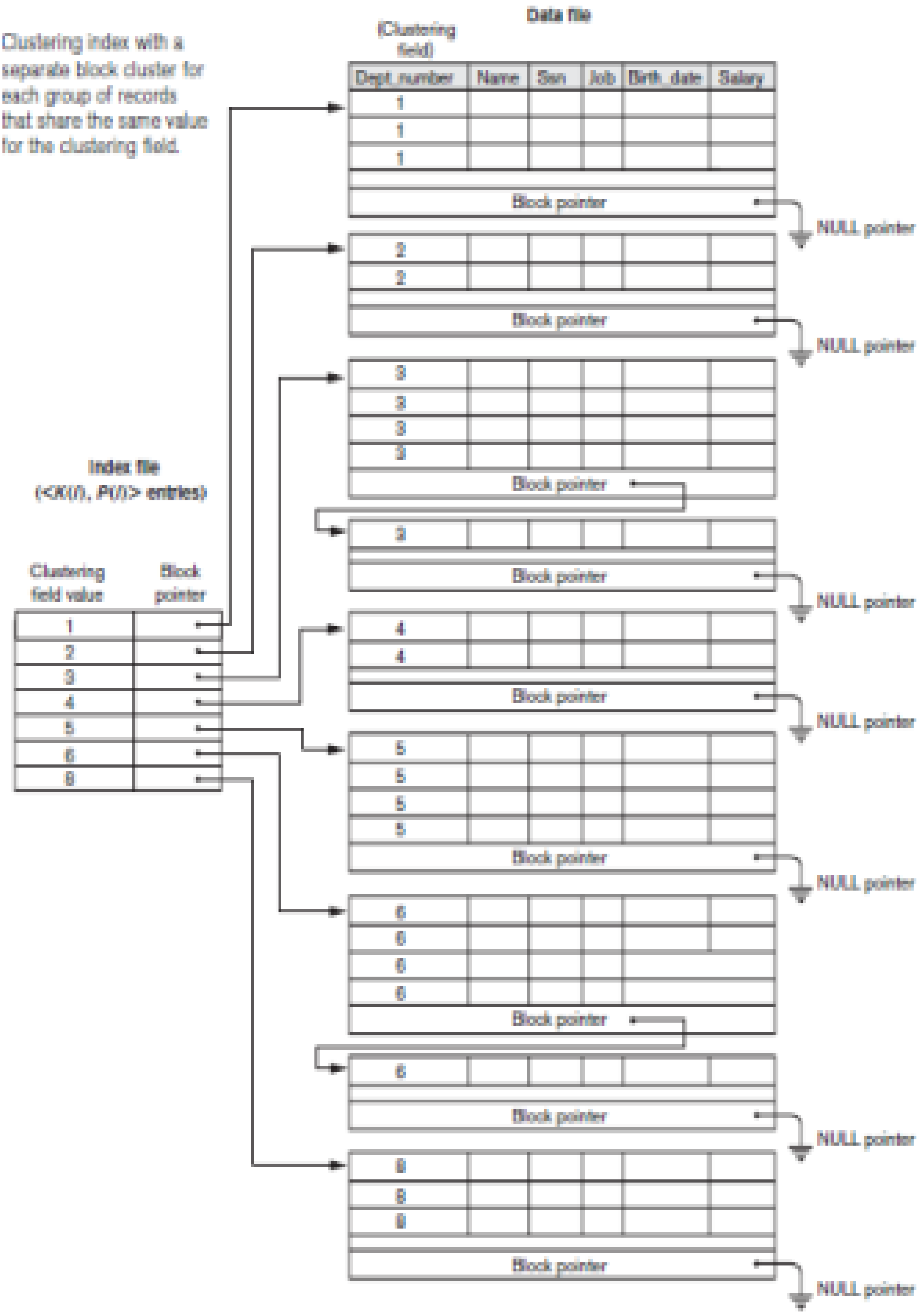
Retrieval via the index requires one or more additional block accesses because of the extra level, but the algorithms for searching the index and for inserting of new records in the data file are straightforward

A secondary index provides a **logical ordering** on the records by the indexing field. If we access the records in order of the entries in the secondary index, we get them in order of the indexing field.

Clustering Indexes
If data file records are physically ordered on a non-key field—which *does not* have a distinct value for each record—that field is called the **clustering field** and the data file is called a **clustered file.** We can create a different type of index, called a **clustering index**, to speed up retrieval of all the 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 has 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 disk block pointer. There is one entry in the clustering index for each distinct value of the clustering field.
Record insertion and deletion still cause problems because the data records are physically ordered. To avoid 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. Figure shows this scheme.


Data file
(Clustering field)
Dept_number
Name
Ssn
Job
Birth_date
Salary
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.
Block pointer
NULL pointer
Index file (<K(i), P(i)> entries)
Clustering field value
Block pointer
1
2
3
4
5
6
8

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 a unique value for every record in the file.