The University of Manchester

H11

File Organization and Indexing

Fundamentals of Databases

Alvaro A A Fernandes, SCS, UoM

[COMP23111 Handout 11 of 12]

Acknowledgements

- These slides are adaptations (mostly minor, but some major) of material authored and made available to instructors by **Thomas Connolly and Carolyn Begg** to accompany their textbook **Database Systems: A Practical Approach to Design, Implementation, and Management, 5th Edition. Addison-Wesley, 2010, 978-0-321-52306-8**
- Copyright remains with them and the publishers, whom I thank.
- Some slides had input from Sandra Sampaio, whom I thank.
- All errors are my responsibility.

In Previous Handouts

- We learned about the relational algebra and SQL, both its DDL and DML capabilities and its querying constructs.
- We also learned how to design databases for implementation in a relational DBMS.
- We explored some advanced features of SQL that can be used to capture more application semantics in the DBMS environment.
- We also explored how a DBMS provides transaction, concurrency and recovery management services.

In This Handout

- We'll look into how DBMSs provide basic file organization and indexing facilities, which are of great importance for efficient query processing.

Basic Concepts

- Databases store data in secondary storage (e.g., magnetic disks).
- Data is organized into one or more files.
- Each file consists of one or more records and each record of one or more fields.
- Typically, a field corresponds to an attribute, a record to an entity, a file to an entity type in the domain.
- When data is requested for access in terms of the logical model of the data (e.g., a tuple), the DBMS maps this to the physical storage model.
- The DBMS retrieves the physical record(s) into the buffers held for that purpose in primary storage (i.e., RAM).

Physical Records

- The physical record is the unit of transfer between secondary and primary storage.
- In general, one physical record contains many logical ones, but the opposite is also possible (i.e., a logical record spanning more than one physical record).
- It is more common for us to refer to physical records as blocks, or pages.
- Pages is, in fact, the most common term.

File Organization

- The order in which records are stored and accessed in a file is referred to as the file organization, i.e., the physical arrangement of data in a file into records and pages in secondary storage.

- The main types of file organization according to the way records are placed in storage are:
  - Heap files contain records in no particular order.
  - Sequential files contain records ordered on a given field.
  - Hashed files contain records at logical addresses defined by a hash function.

- The steps involved in storing and retrieving records from a file is referred to as the access method.

- Because some file organizations require (or else prevent) certain access methods, the terms are often used interchangeably.

Unordered Files (Heap / Pile)

- This is the simplest type of file organization:
  - New records are inserted at the end of the file.
  - Record insertion is quite efficient.
  - But retrieval performs a linear search, i.e., it requires reading and searching half the file records on average
  - This is quite expensive unless the number of records is small, or else the retrieval is of a very large proportion of the records.
  - Deletion requires retrieving the required page, deleting the record and writing the page (with a new blank).
  - Performance therefore deteriorates as deletions accumulate.
  - Periodic reorganization is required to reclaim blank spaces left by deleted records.

Ordered Files (Sequential)

- Records are kept sorted by the values of an ordering field.
- If the ordering field is also a key (i.e., each record has a distinct value for the ordering field), it is called the ordering key.
  - Retrieval can use binary search which is a significant improvement over linear search.
  - Reading the records in order of the ordering field is quite efficient.
  - Insertion and deletion are expensive: we must find the correct position, then find space for insertion.
  - If there is space in the page, we insert the record, reorder the page and write it back.
  - If not, then one or more records must be moved to the next page (and this may cascade).
  - Inserting records near the start increases the risk of cascading.
  - Because of this, a separate unordered overflow file for new records may be used to improve efficiency, which is then periodically merged with the main ordered file.
  - This makes insertion efficient at the expense of retrieval because if binary search on the main file does not succeed we must do linear search on the overflow one.

Hashed Files

- The file records are divided into M equal-sized partitions, called buckets.
- One of the file fields is designated to be the hash key of the file.
- The record with hash key value K is stored in bucket i, where i=h(K), and h is the hash function.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full.
- Records in a hashed file appear to be randomly distributed across the available file space.
- For this reason, hashed files are sometimes called, random, or direct-access files.

Hashed Files (Collision Resolution)

- When collisions occur we need to have strategies to manage it.
- Some of the strategies are:
  - open addressing
  - unchained overflow
  - chained overflow
  - multiple hashing

Hashed Files (Collision Resolution)

- In open addressing, if a collision occurs, the system performs a linear search to find the first available slot for insertion.
- If the last bucket is reached, the system continues at the first.
- In searching, if a blank space is found before the record is found, then the search was unsuccessful.
- (See Figure F.4 in the mandatory readings for an example.)

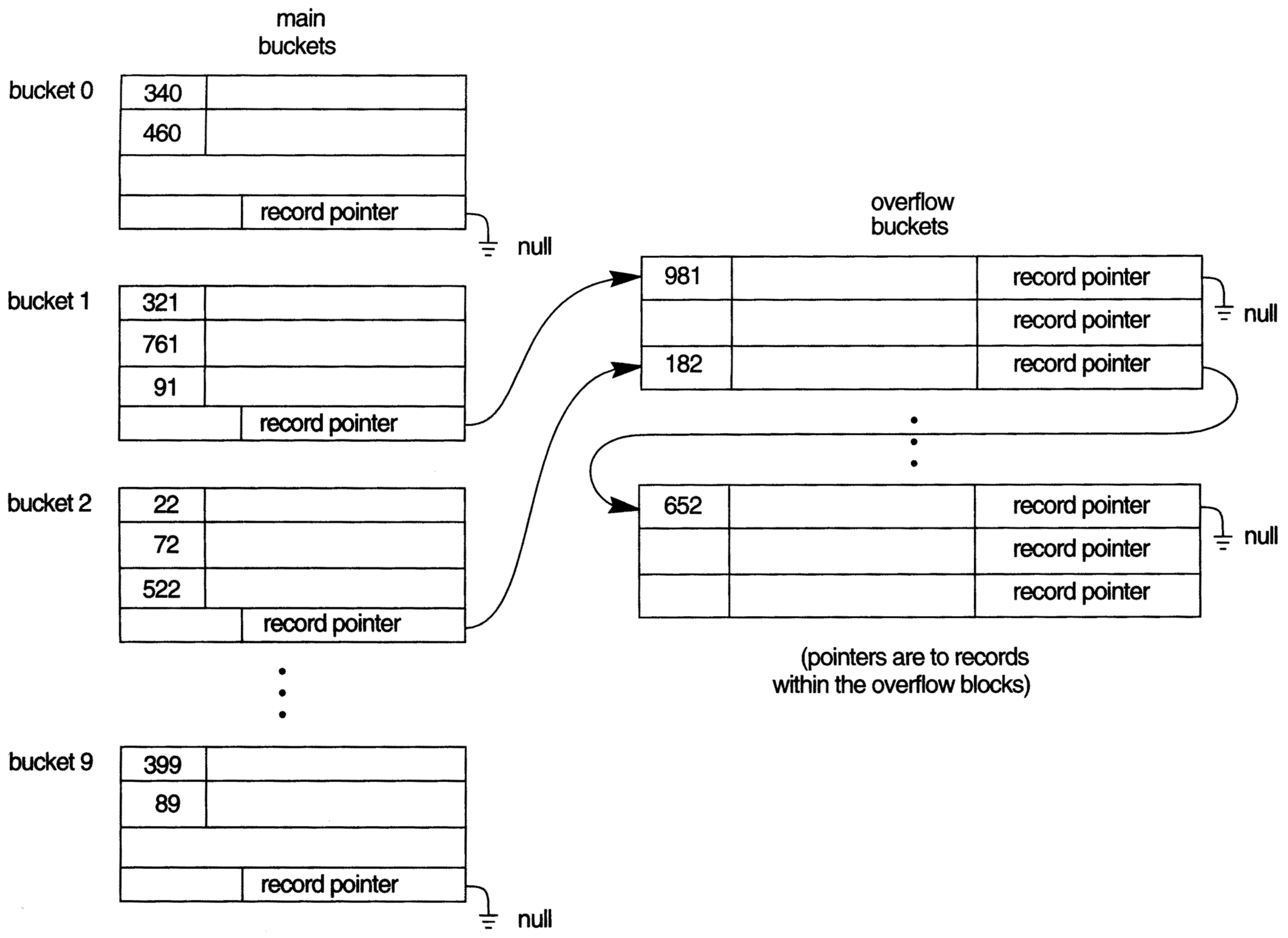
Hashed Files (Collision Resolution)

- In the unchained overflow case, instead of searching for a blank space, an overflow area is maintained for collisions.
- The overflow area is also structured in terms of buckets.
- The goal is to avoid increasing the number of collisions so as to keep small number of records over which linear search is unavoidable.
- Overflow areas reduce the likelihood of collisions compared to open addressing.
- This is because, in open addressing, using an address which previously pointed to black space, increases the likelihood of future collisions, which in turn increases the number of records over which linear search would be done.
- (See Figure F.5 in the mandatory readings for an example.)

Hashed Files (Collision Resolution)

- In the chained overflow case, an overflow area is also maintained for collisions.
- But, in this case, each bucket has an additional field, sometimes called, a synonym pointer, that indicates whether a collision has occurred.
- If the pointer is zero, no collision has occurred, otherwise the value points to the overflow page used.

Hashed Files - Chained Overflow Handling



Hashed Files (Collision Resolution)

- In multiple hashing, the system applies a second hash function if the first resulted in a collision.
- The aims is to produce a new address that avoids collision.
- In general, the second hash function is used to place records in an overflow area.
- If another collision results, the system can use open addressing or apply a third hash function and then uses open addressing if necessary, and so on.

Hashed Files (cont.)

- With hashing, a record can be located efficiently by applying the hash function.
- If a collision occurs, the strategies above can be used.
- To update a hashed record, the record must first be located.
- If the field to be updated is not the hash key, the update goes ahead and the record is written back.
- If the updated field is the key, the hash function must be applied to the new value, and, if a new address results, a deletion on the old page is followed by an insertion in the new page.
- Ordered access on the hash key is quite inefficient, since it requires sorting the records.

Hashed Files (cont.)

- So far, we have assumed that the hash address space is fixed when the file is created.
- This is referred to as static hashing.
- When space becomes scarce, the address space becomes saturated.
- The main disadvantage of static hashing is that saturation of the address space implies
  - creating a new file with more space
  - adjusting (or choosing another) hash function
  - mapping the old file to the new one
- An alternative is dynamic hashing, which allows the file size to grow or shrink.
- The basic principle is is to manipulate the number generated by the hash function as a bit sequence and to allocate records to buckets based on the progressive digitization of this sequence.
- (See Sections F.4.1-2 for more details.)

Indexes as Access Paths

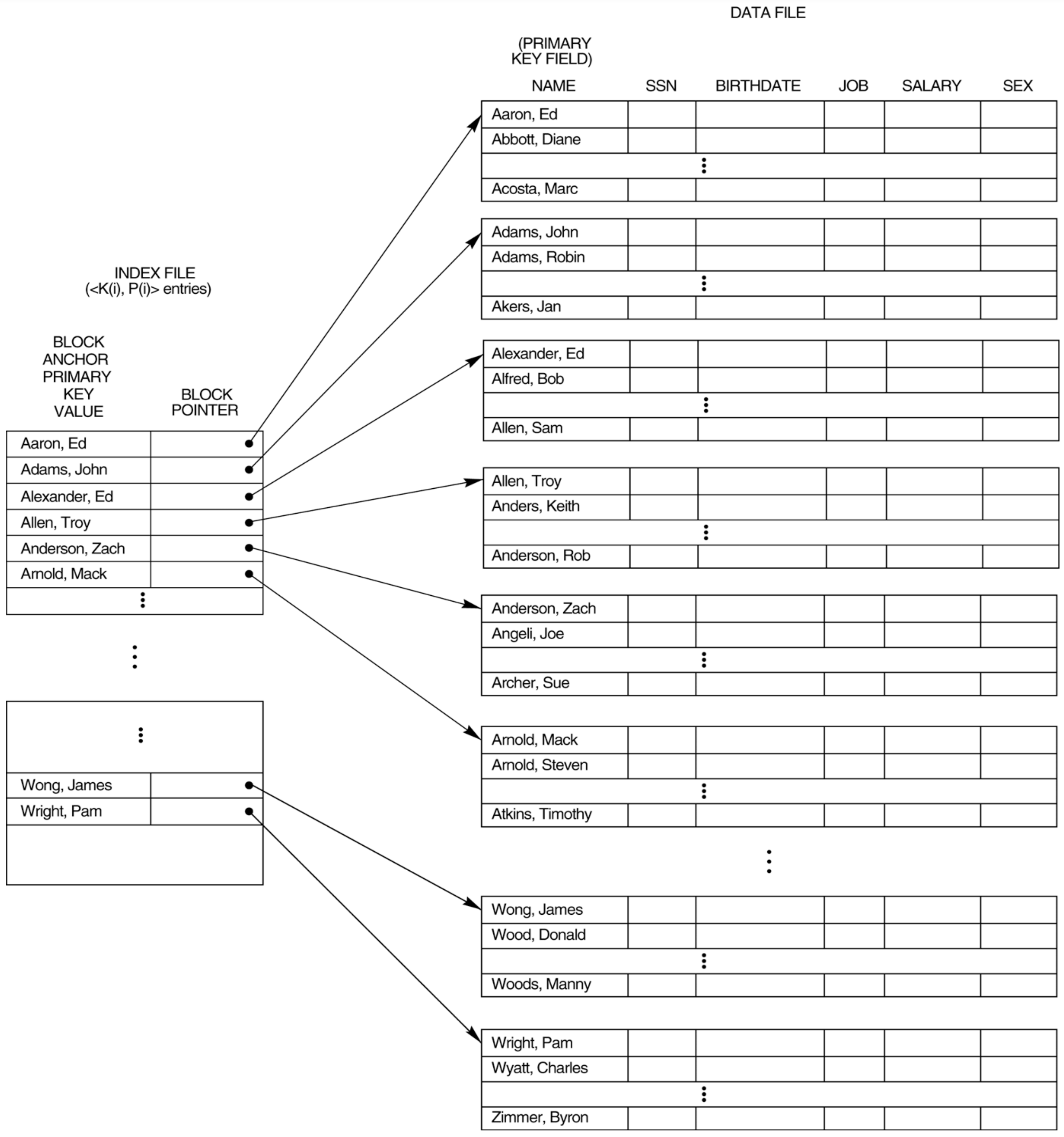
- An index is a data structure (stored as a file) that allows the DBMS to locate particular records in a file more quickly and thereby speed up the response to user queries.
- It is similar to an index in a book, in which, given a name or concept, we can find the pages where the term that we want occurs.
- An index file removes the need for us to scan sequentially through the data file.
- As with a book index, in a DBMS an index file is ordered and each index entry contains the item required (e.g., the value of a field, referred to as the search key) and the locations (i.e., the record identifiers or addresses) where the item can be found in the data file.
- An index file is usually specified on one field of the data file, i.e., it is associated with a particular search key.
- One form of an index is a file of entries <field value, pointer to record>, ordered by field value.
- An index is referred to as an access path on the field and can be dense or sparse, defined as follows:
    - A dense index has an index entry for every search key value (and hence every record) in the data file.
    - A sparse index, on the other hand, has index entries for only some of the search values.
- The search key can consist of more than one field.

Types of Index

- The main types of index are:
  - Primary indices
  - Clustering indices
  - Secondary indices
- A file can have at most one primary index or one clustering index, but may have many secondary indices.

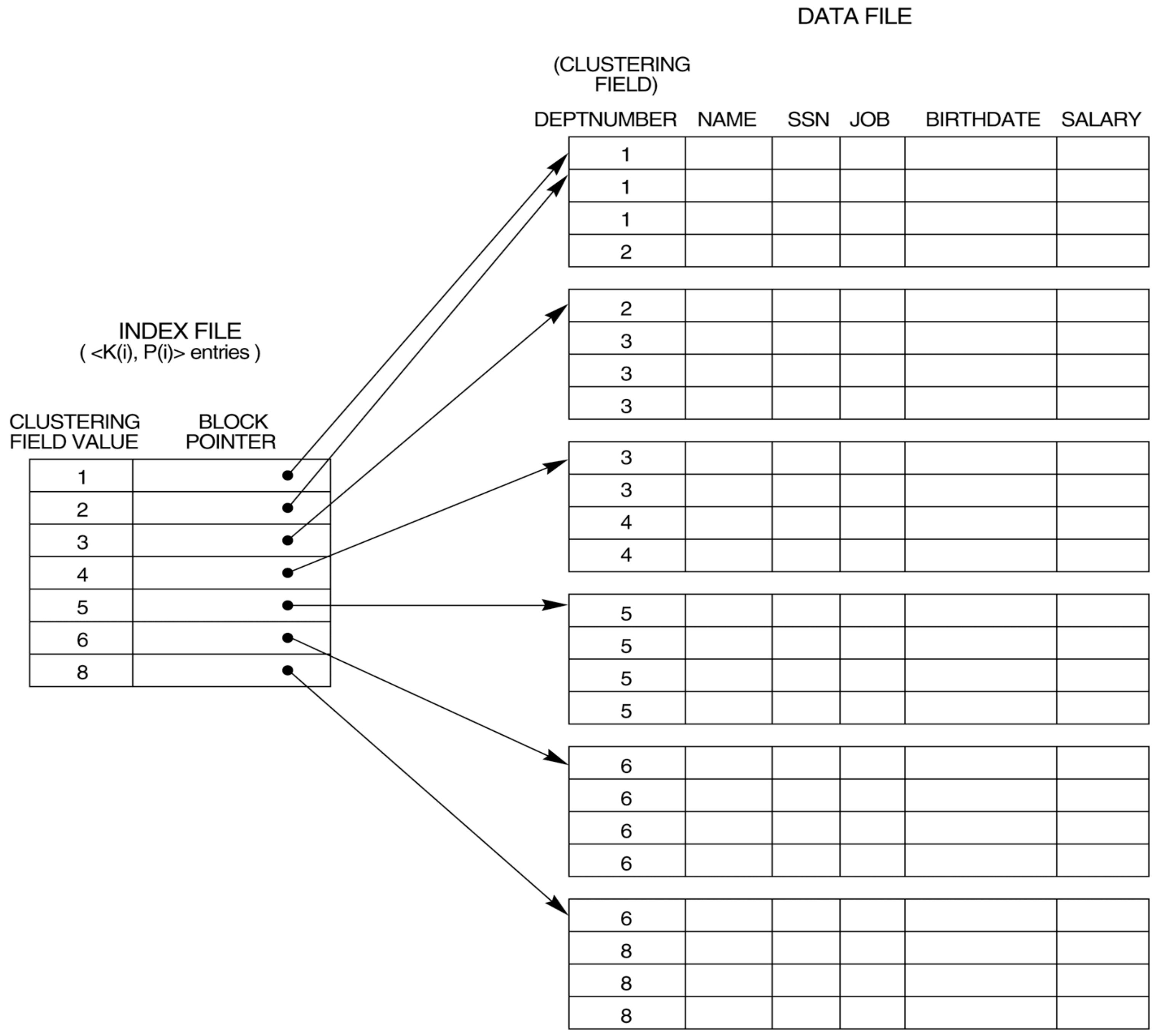
Types of Single-Level Indexes (1)

- Primary Index
  - Defined on an ordered data file
  - The data file is ordered on a key field
  - A primary index is a non-dense (sparse) index, since it includes an entry for each disk block of the data file and the key of its anchor record rather than for every search value.



Types of Single-Level Indexes (2)

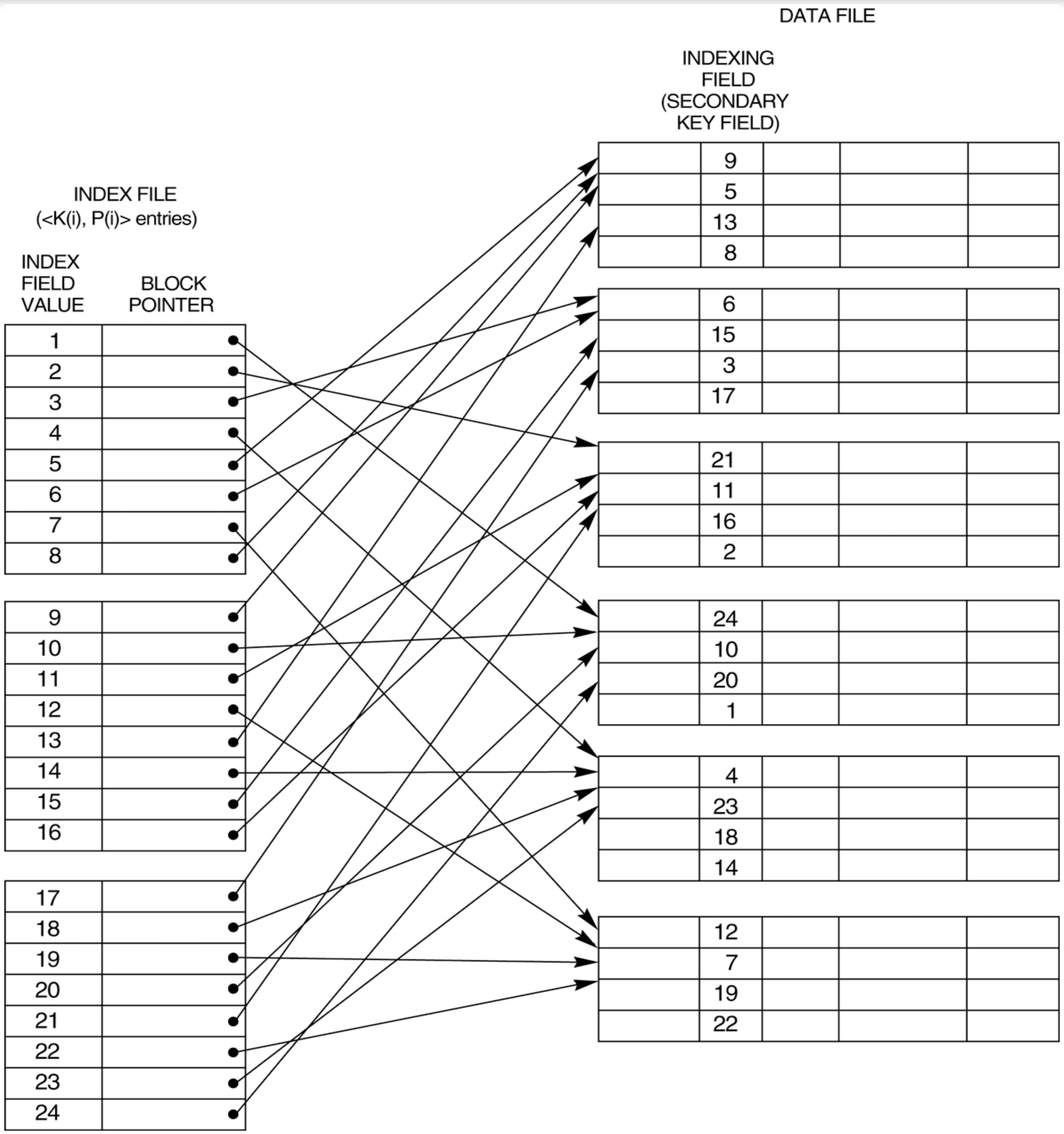
- Clustering Index
  - The data file is ordered on a non-key field (also called clustering field), so that there can be more than one record corresponding to a single value of the indexing field.
  - It includes one index entry for each distinct value of the field; the index entry points to the first data block that contains records with that field value.
  - It is another example of non-dense (sparse) index.



Types of Single-Level Indexes (3)

- A secondary index is defined on a non-ordering field of the data file.
- It 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 non-key with duplicate values.
- The index is 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.
  - There can be many secondary indexes (and hence, indexing fields) for the same file.
  - It includes one entry for each record in the data file; hence, it is a dense index.

A Dense Secondary Index

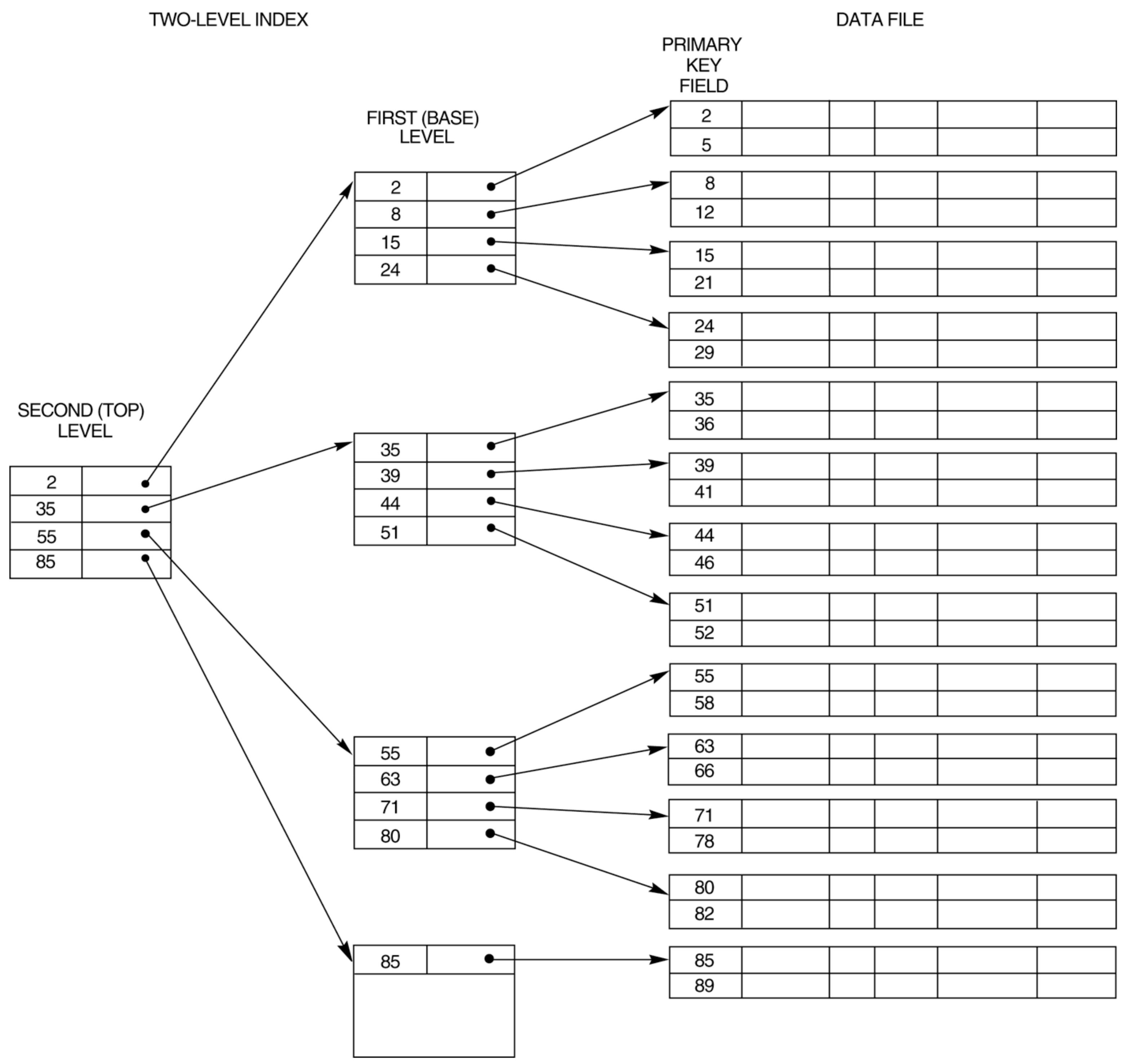


Multi-Level Indexes

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

A Two-Level Primary Index

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

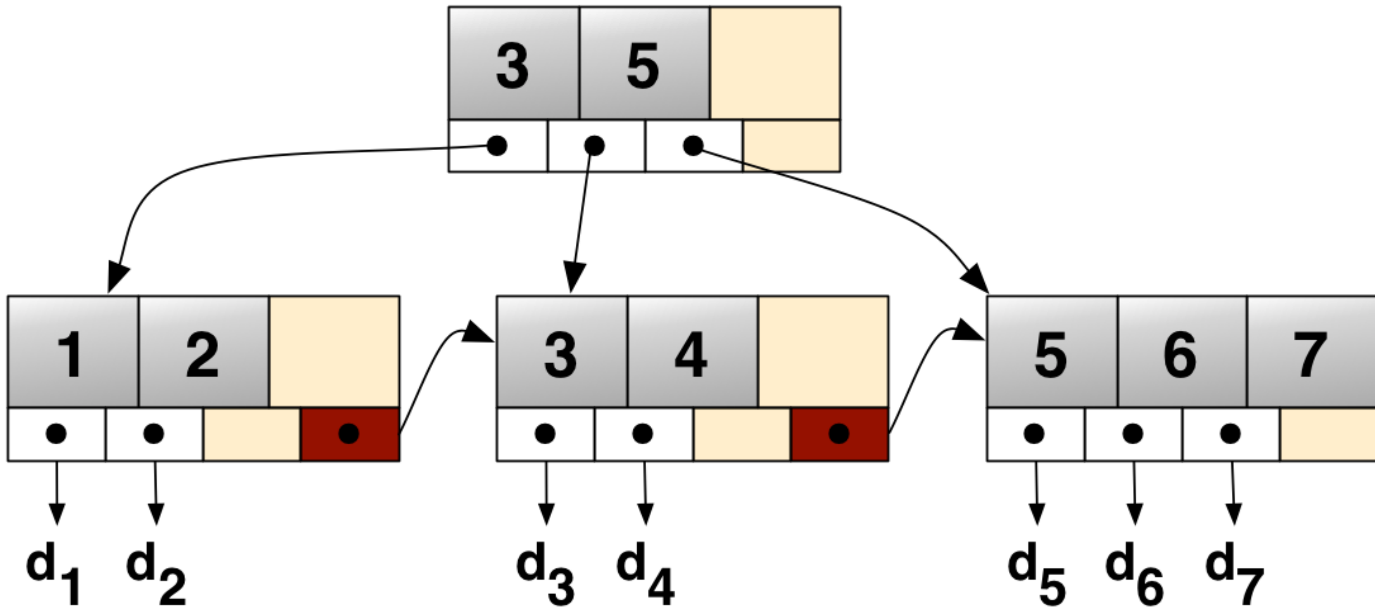


Balanced Trees

- If a tree-structure index has the same depth (i.e., if the length of the path from root to leaf is constant), it is called a balanced tree (or a B-tree).
- The degree, or order, or branching factor, of a tree, is the maximum number of children per parent.
- Large degrees create broader, shallower trees.
- Access time in a tree structure depends more often on depth than breadth, it is advantageous to have shallower (called bushier) trees.

B+-Trees

- A B+-tree is a B-tree in which each node contains only keys (rather than key-value pairs), with an additional leaf level for the values where leaves are linked.
- The B+ tree to the right links the keys 1–7 to data values d1-d7.
- The linked list at the leaves allows rapid in-order traversal.
- For a b-order B+-tree with n records, insertion, retrieval and deletion (of a previously retrieved record) requires $O(\log_b n)$ operations, which is very efficient.



Some Guidelines for Selecting File Organizations

- Heap is good
  - when bulk loading
  - if the relation requires only few pages
  - every tuple is almost always retrieved
  - the relation has an additional access method, such as an index

Some Guidelines for Selecting File Organizations

- Hashed is good
  - when tuples are retrieved based on an exact match of the hash field value
- Hashed is <u>not</u> good
  - when tuples are retrieved based on a pattern match (e.g., ‘starts with 103’) of the hash field value
  - or on a range of hash field values
  - or on a field other than the hash field
  - or only on part of the hash field
  - or when the hash field is frequently updated

Some Guidelines for Selecting File Organizations

- B+-trees are more versatile than hashed files.
- It supports retrieval based on exact or pattern match, as well as ranges and part-of-keys.
- B+-trees are inherently dynamic so performance does not particularly deteriorate as more updates are made.
- It also allows for efficient retrieval in order of access key.

Summary

- There are 3 main types of file organisation:
  - Heap or unordered files: records are placed on disk in no particular order.
  - Sequential or ordered files: records are ordered by the value of a specified field.
  - Hash files: records are placed on disk according to a hash function.
- Indexes represent a technique for making the retrieval of data more efficient/faster.
- Indexes can be sparse or dense, can have one or more levels, and can be primary secondary or clustering, etc.
- File organisation as well as indexes have a significant impact on the performance with which data is retrieved from and written to disk.

In the Next Handout

- We'll have a closer look at the internal architecture of a DBMS.

- We'll also look at the various ways in which DBMSs can be deployed in very large scales.