Database Systems

Luiz Jonatã Pires de Araújo
l.araujo@innopolis.university

4. Indexing Structures for Files and Physical Database Design

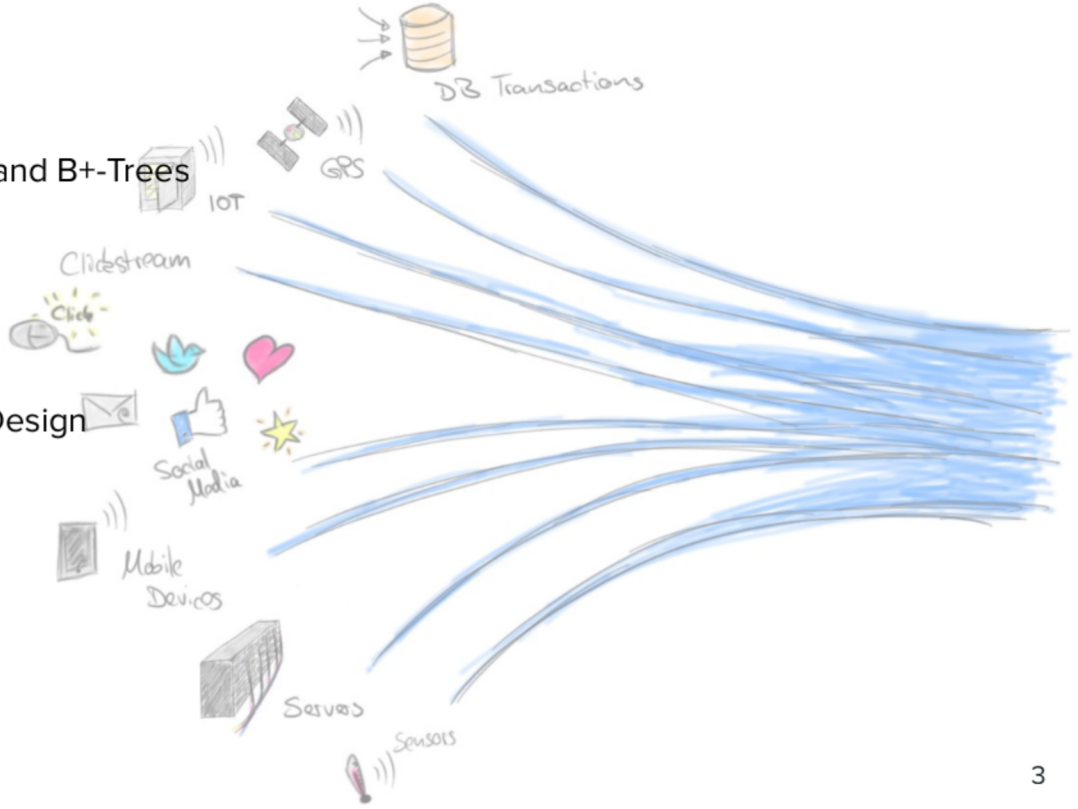
INNOPOLIS UNIVERSITY

Giving credit where it's due:

- This material is based on the 7th edition of 'Fundamentals of Database Systems" by Elmasri and Navathe
- https://www.pearson.com/us/higher-education/program/Elmasri-Fundamentals-of-Database-Systems-7th-Edition/PGM189052.html

Outline

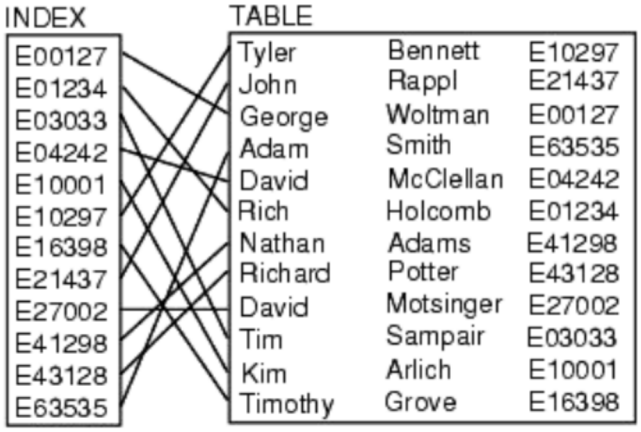
- Types of Single-Level Ordered Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- Indexes on Multiple Keys
- Other Types of Indexes
- Some General Issues Concerning Indexing
- Factors That Influence Physical Database Design

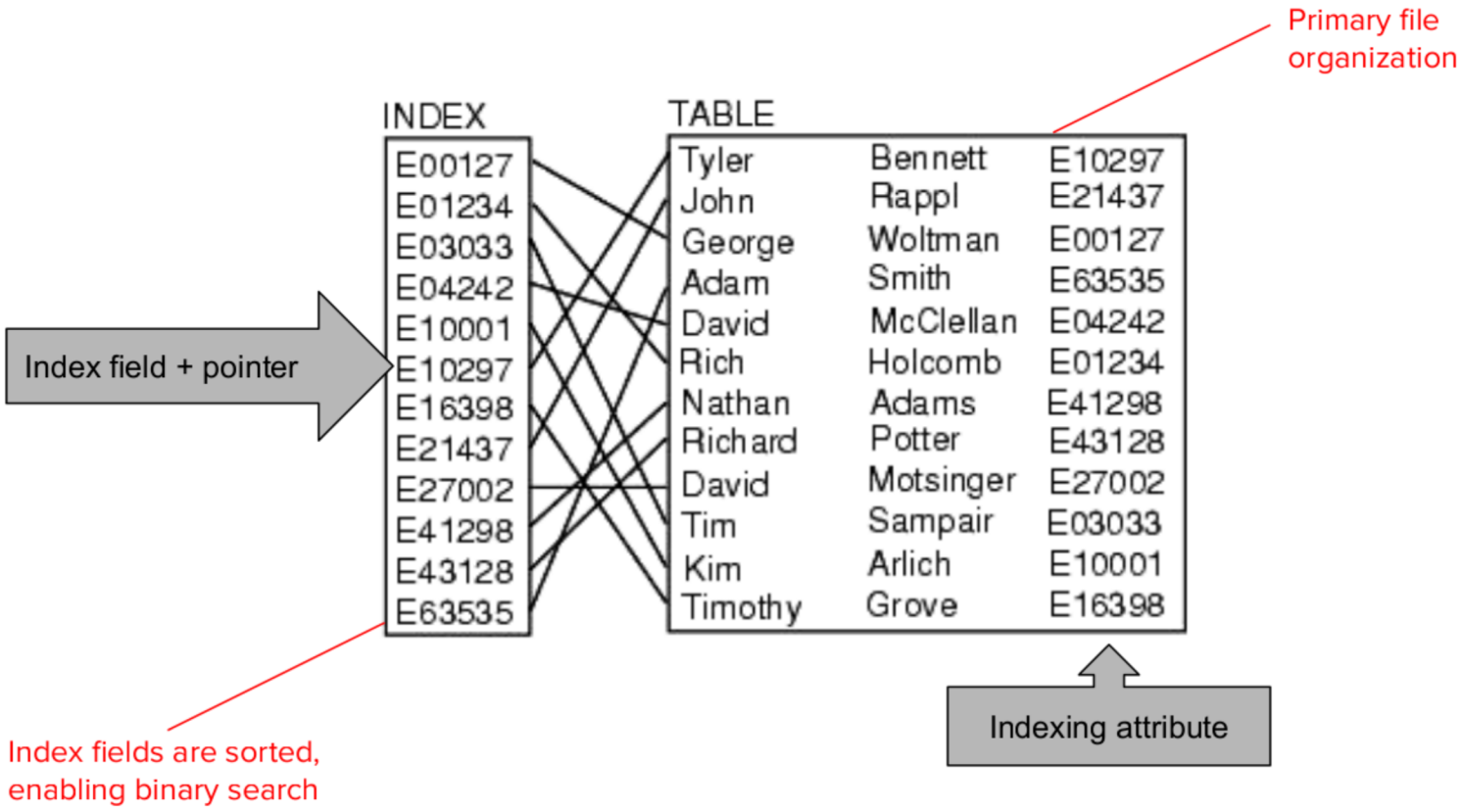


Our fingers did the walking

Today...

- We will describe additional auxiliary access structures called **indexes** used to speed up the retrieval of records in response to certain search conditions
- The **index** structures are additional files on disk that provide secondary access paths, i.e., alternative ways to access the records without affecting the physical placement of records in the primary data file on disk




Primary file organization
INDEX
TABLE
E00127
E01234
E03033
E04242
E10001
E10297
E16398
E21437
E27002
E41298
E43128
E63535
Tyler Bennett E10297
John Rappl E21437
George Woltman E00127
Adam Smith E63535
David McClellan E04242
Rich Holcomb E01234
Nathan Adams E41298
Richard Potter E43128
David Motsinger E27002
Tim Sampair E03033
Kim Arlich E10001
Timothy Grove E16398
Index field + pointer
Indexing attribute
Index fields are sorted, enabling binary search

Today...

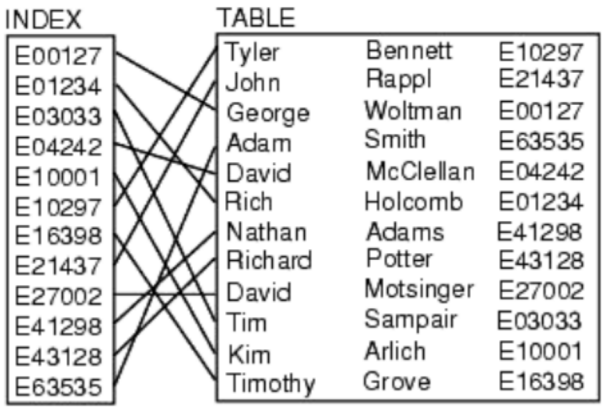
- A variety of indexes are possible; each of them uses a particular data structure
- The most prevalent types of indexes are based on ordered files (single-level indexes) and the use of tree data structures (multilevel indexes, B+-trees) to organize the index
- B+-trees have become a commonly accepted default structure for generating indexes on demand in most relational DBMSs

In PostgreSQL, B-Tree is the **default** that you get when you do CREATE INDEX. Virtually all databases will have some B-tree indexes.

Types of Single-Level Ordered Indexes

Single-Level Ordered Indexes

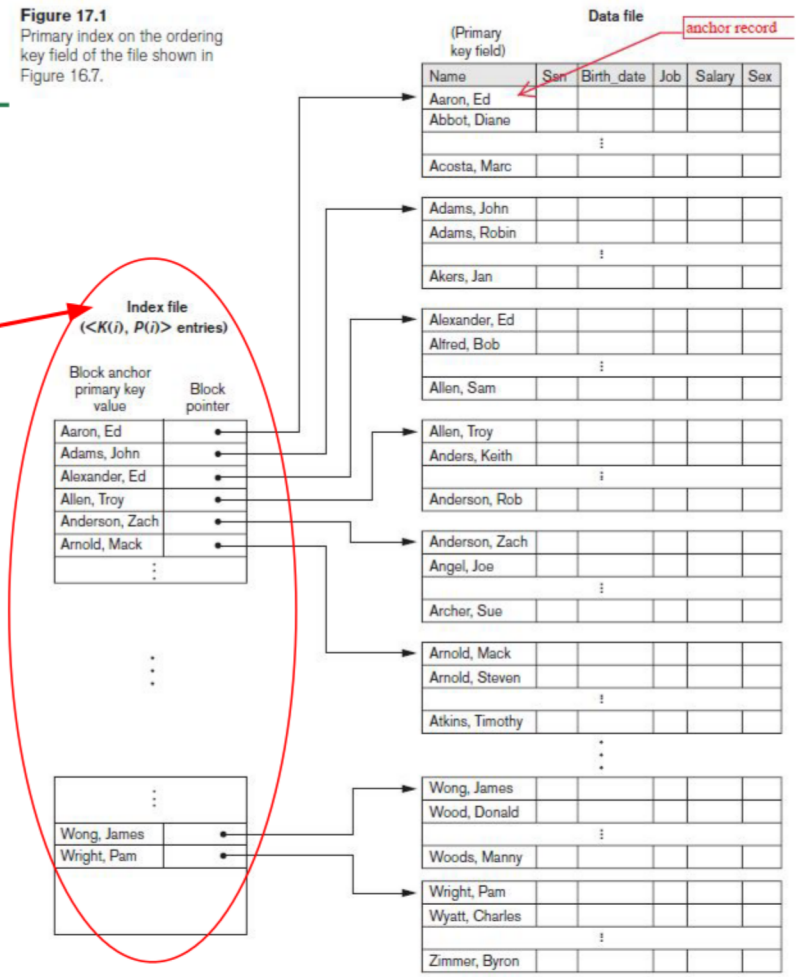
- **Primary index**
  - It is specified on the ordering key field of an ordered file of records
  - An ordering key field is used to physically order the file records on disk
  - Every record has a unique value for that field
- **Clustering index**
  - When the ordering field is not a key field - that is, numerous records in the file can have the same value for the ordering field
  - The data file is called a clustered file
- **Secondary index**
  - It can be specified on any non-ordering field of a file
  - A data file can have several secondary indexes in addition to its primary access method



Important

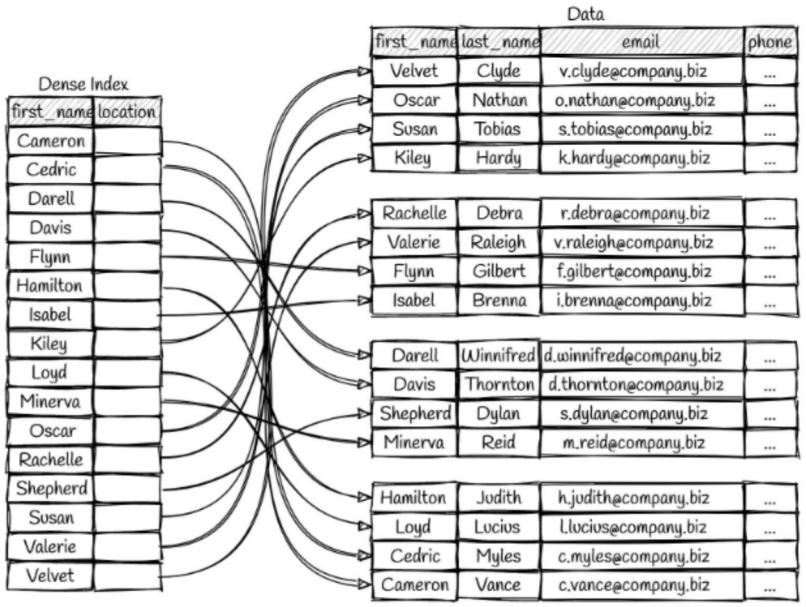
Primary Indexes

- It acts like an access structure to efficiently search for and access the data records in a data 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 (primary key) of the data file
  - 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 *i* has the value of the primary key field for the first record in a block - K(i); and
  - A pointer to that block - P(i)



Dense vs. sparse indexes

- 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 nondense) index**, on the other hand, has index entries for only some of the search values
  - A sparse index has fewer entries than the number of records in the file
- A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record



Primary Indexes

- The index file for a primary index occupies a much smaller space than does the data file, for two reasons:
  - 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, both of which tend to be short in size
- Consequently, more index entries than data records can fit in one block
- Therefore, a binary search on the index file requires fewer block accesses than a binary search on the data file saving in block accesses that is attainable when a primary index is used to search for a record

Primary Indexes

Example 1. Suppose that we have an ordered file with $r = 300{,}000$ records stored on a disk with block size $B = 4{,}096$ bytes.[5] File records are of fixed size and are unspanned, with record length $R = 100$ bytes. The blocking factor for the file would be $bfr = \lfloor (B/R) \rfloor = \lfloor (4{,}096/100) \rfloor = 40$ records per block. The number of blocks needed for the file is $b = \lceil (r/bfr) \rceil = \lceil (300{,}000/40) \rceil = 7{,}500$ blocks. A binary search on the data file would need approximately $\lceil \log_2 b \rceil = \lceil (\log_2 7{,}500) \rceil = 13$ block accesses.

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. The size of each index entry is $R_i = (9 + 6) = 15$ bytes, so the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/15) \rfloor = 273$ entries per block. The total number of index entries r_i is equal to the number of blocks in the data file, which is 7,500. The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (7{,}500/273) \rceil = 28$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 28) \rceil = 5$ block accesses. To search for a record using the index, we need one additional block access to the data file for a total of $5 + 1 = 6$ block accesses—an improvement over binary search on the data file, which required 13 disk block accesses. Note that the index with 7,500 entries of 15 bytes each is rather small (112,500 or 112.5 Kbytes) and would typically be kept in main memory thus requiring negligible time to search with binary search. In that case we simply make one block access to retrieve the record.

Primary Indexes

- 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 must 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
- Solutions
  - Use a linked list of overflow records for each block in the data file
  - Records within each block and its overflow linked list can be sorted to improve retrieval time
  - Record deletion is handled using deletion markers

Clustering Indexes

- The file records are physically ordered on a nonkey 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
- 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

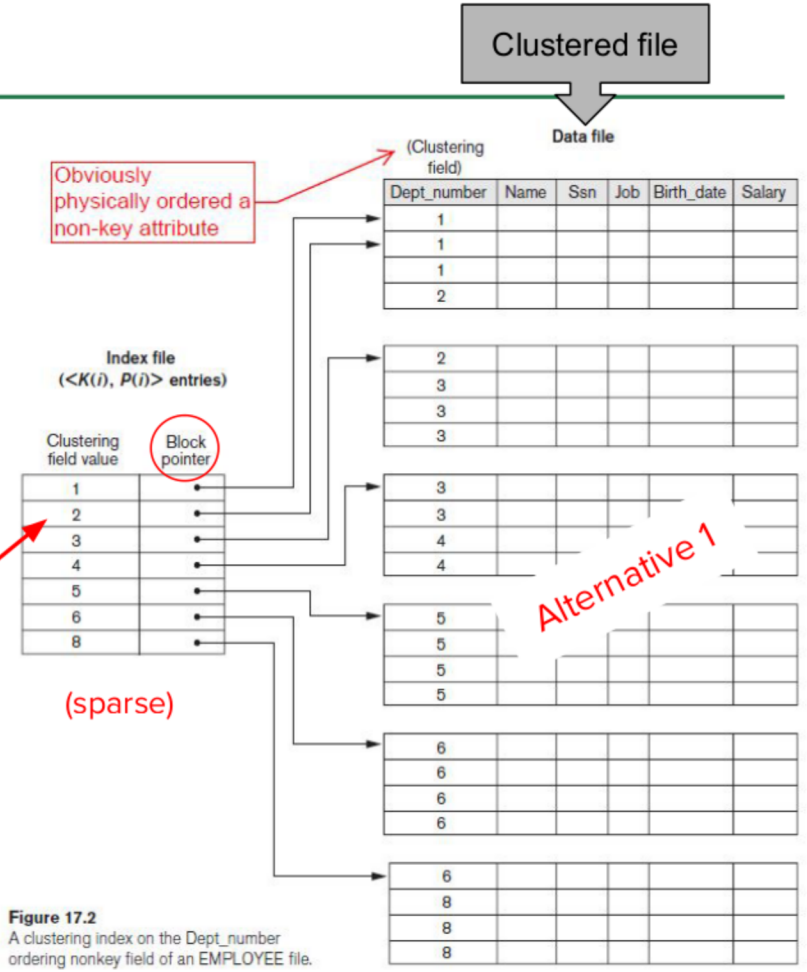
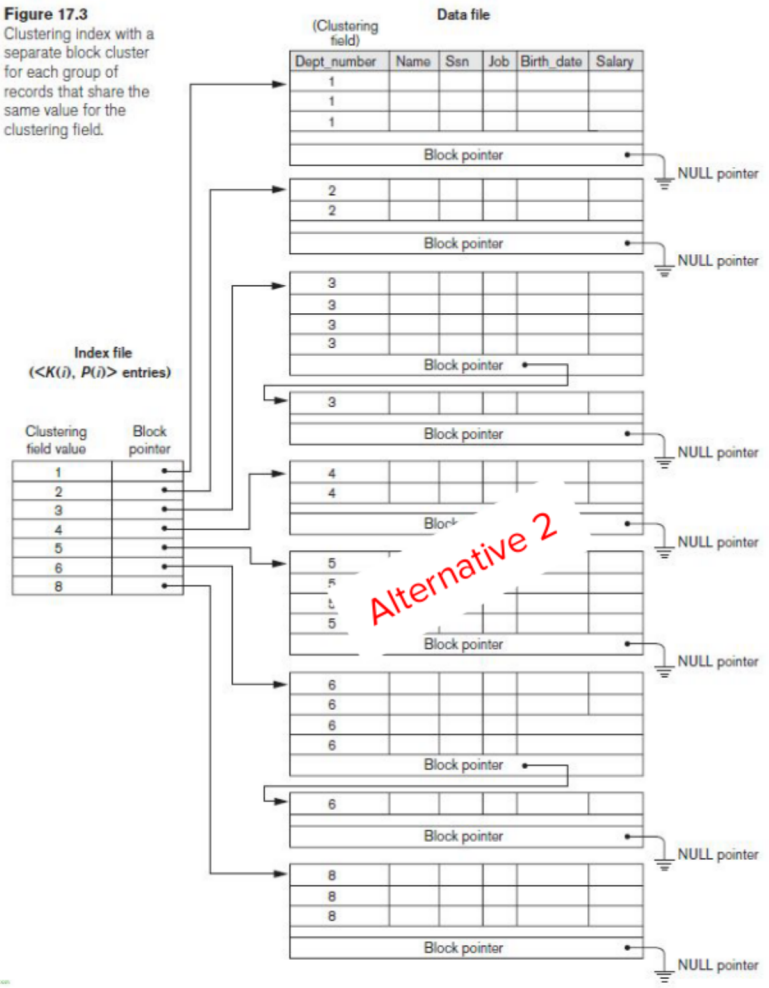


Figure 17.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Clustering Indexes

- Notice that 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)

Figure 17.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



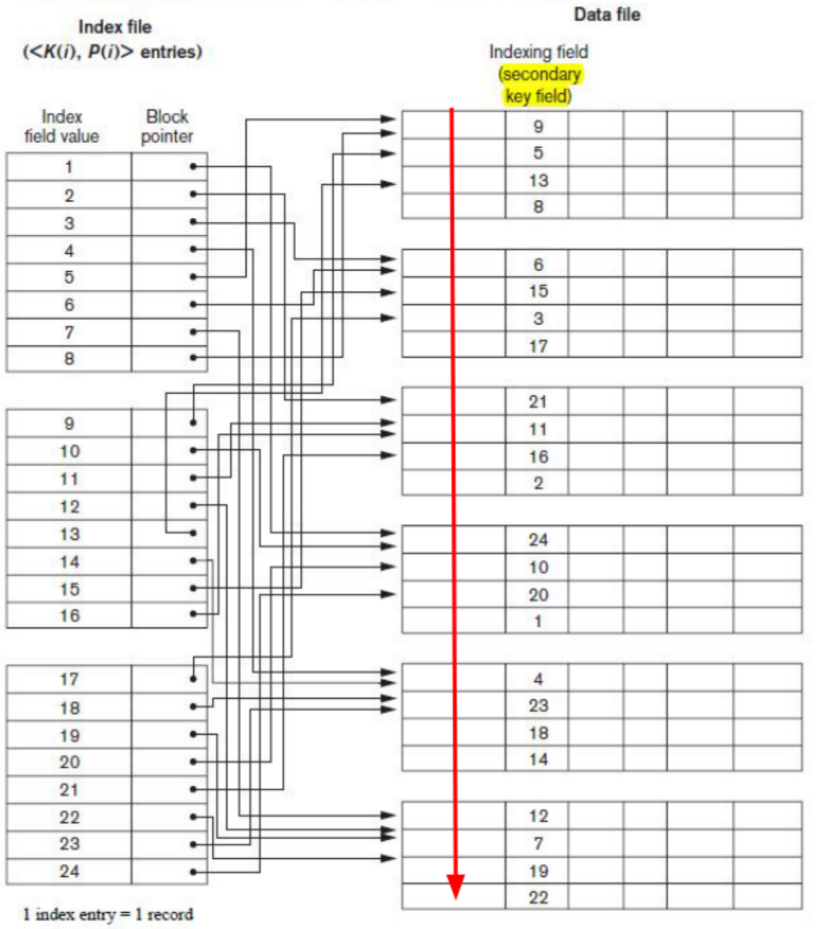
Clustering Indexes

Example 2. Suppose that we consider the same ordered file with $r = 300{,}000$ records stored on a disk with block size $B = 4{,}096$ bytes. Imagine that it is ordered by the attribute Zipcode and there are 1,000 zip codes in the file (with an average 300 records per zip code, assuming even distribution across zip codes.) The index in this case has 1,000 index entries of 11 bytes each (5-byte Zipcode and 6-byte block pointer) with a blocking factor $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4{,}096/11) \rfloor = 372$ index entries per block. The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (1{,}000/372) \rceil = 3$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 3) \rceil = 2$ block accesses. Again, this index would typically be loaded in main memory (occupies 11,000 or 11 Kbytes) and takes negligible time to search in memory. One block access to the data file would lead to the first record with a given zip code.

Secondary Indexes

- A secondary index provides a secondary means of accessing a data file for which some primary access already exists
- Secondary index = <Indexing field, block or record pointer>
- In this case there is one index entry for each record in the data file → dense index

Figure 17.4
A dense secondary index (with block pointers) on a nonordering key field of a file.



Secondary Indexes

- **Case 1.** Secondary index access structure on a key (unique) field that has a distinct value for every record
  - Because the records of the data file are not physically ordered by values of the secondary key field, we cannot use block anchors
  - A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries
- **Case 2.** Secondary index on a nonkey, non-ordering field of a file
  - In this case, numerous records in the data file can have the same value for the indexing field
  - Implementations
    - Include duplicate index entries with the same K(i) value - one for each record. This would be a dense index.
    - The most 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

Previous slide

Next slide

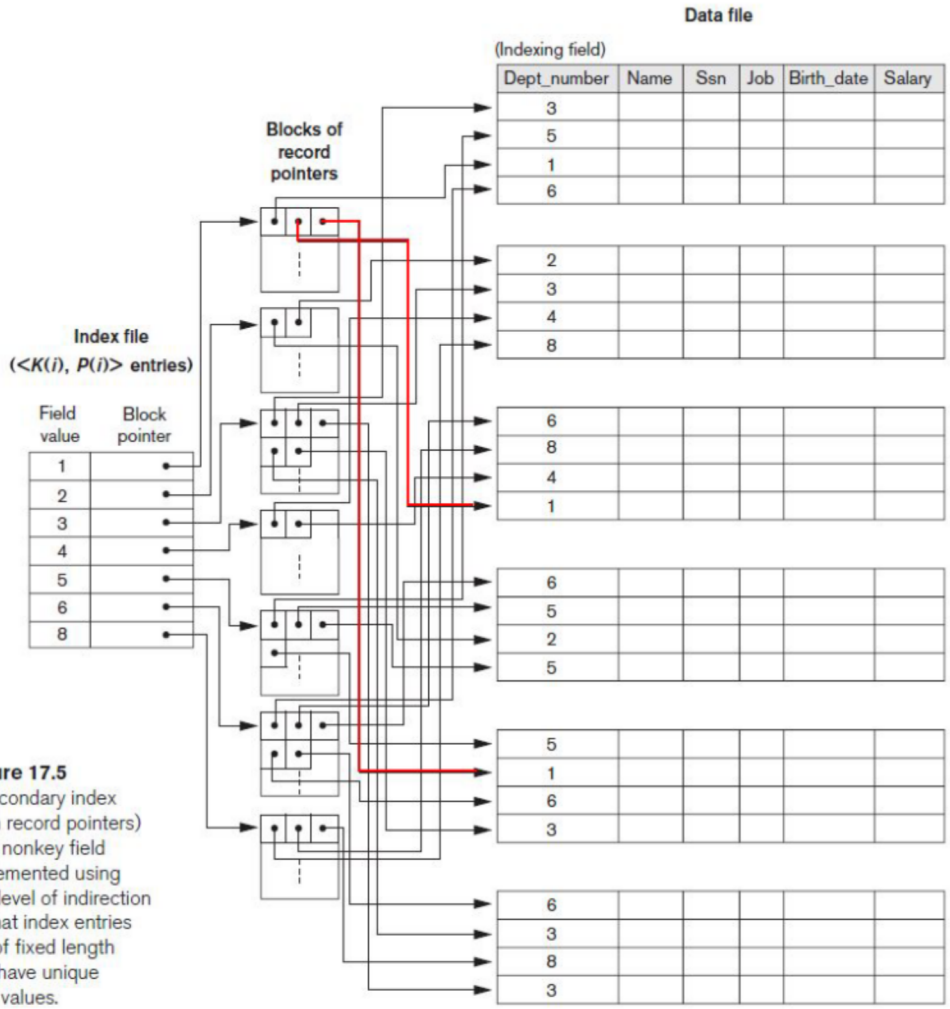
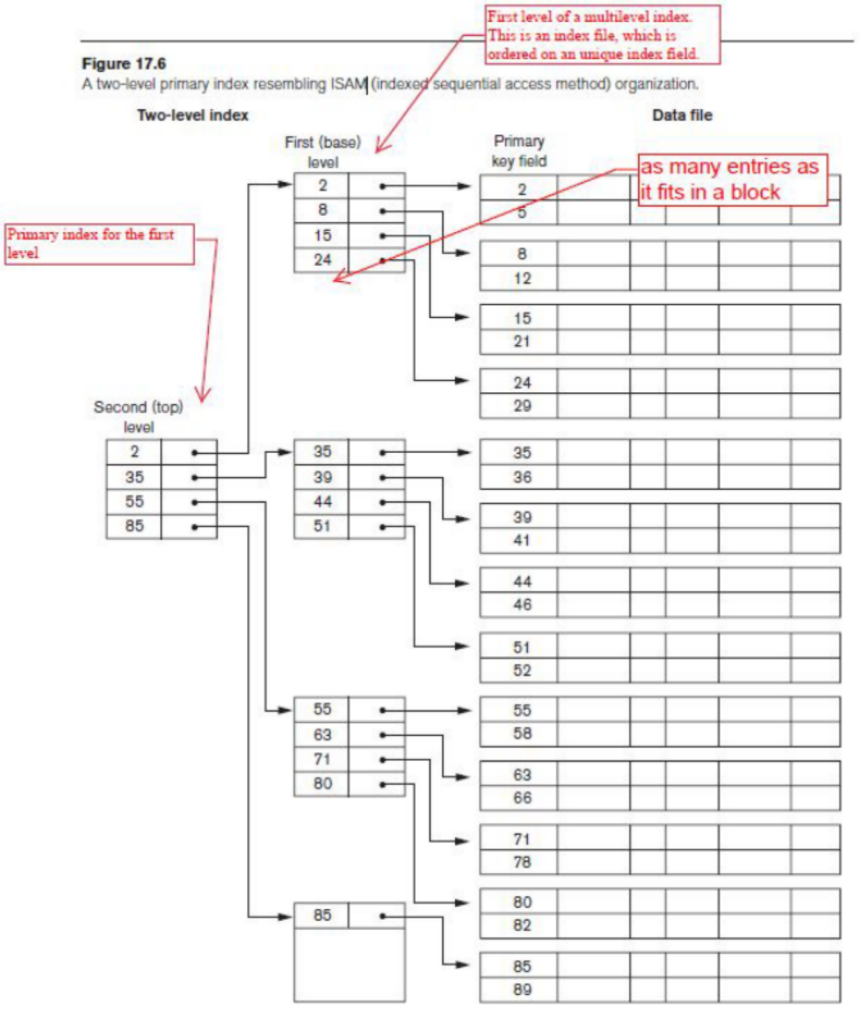


Figure 17.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Multilevel Indexes

Multilevel Indexes

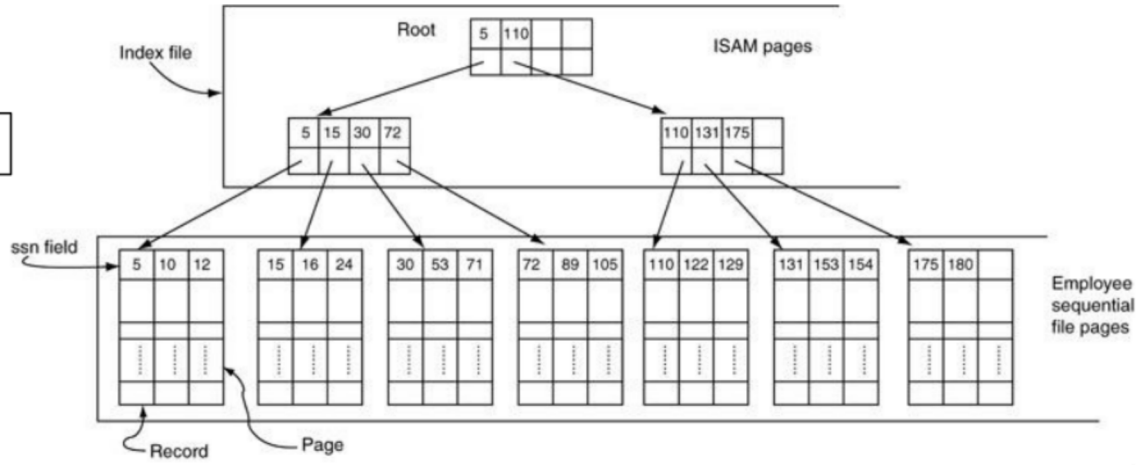
- The indexing schemes we have described thus far involve an ordered index file
  - A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file having a specific index field value
  - A binary search requires approximately $(\log_2 b_i)$ block accesses for an index with b_i blocks
- The idea behind a multilevel index is to reduce the part of the index that we continue to search by bfr_i, the blocking factor for the index, which is larger than 2



Multilevel Indexes

- 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
- Dynamic multilevel index leaves some space in each of its blocks for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks when the data file grows and shrinks
- It is often implemented by using data structures called B-trees and B+-trees

Important



Dynamic Multilevel Indexes Using B-Trees and B+-Trees

Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- B-trees are balanced search trees designed to work well on disks or other directaccess secondary storage devices. B-trees are good at minimizing disk I/O operations thanks to its branching factor
- Properties of B-trees
  - Every node has the following attributes: number of keys currently stored in node x, the keys themselves stored in nondecreasing order, a boolean whether it is a leaf node (otherwise it is an internal node)
  - Each internal node x also contains x.n+1 to its children
  - The keys separate the ranges of keys stored in each subtree
  - All leaves have the same depth, which is the tree's height

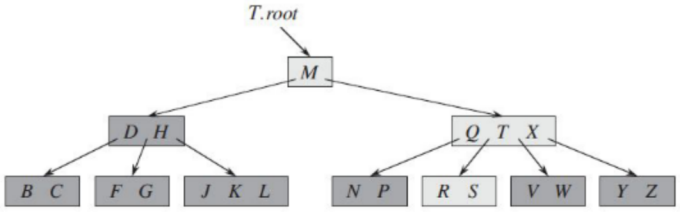


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- Properties of B-trees
  - Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer t >=2 called the minimum degree

Theorem 18.1

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2} .$$

The B-TREE-SEARCH procedure is $O(\log_t n)$
The B-TREE-INSERT and DELETE are $O(t \log_t n)$

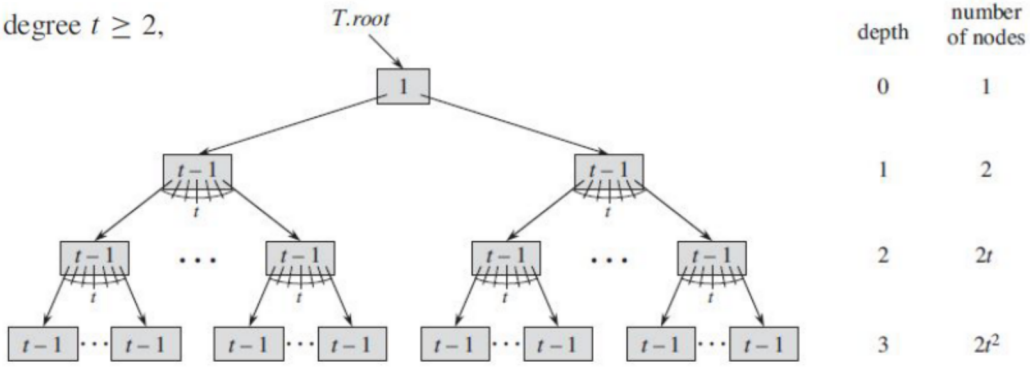


Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $x.n$.

Comparing B-trees and B+-trees

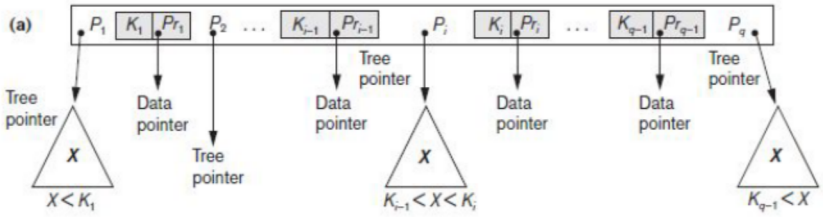


Figure 17.10
B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

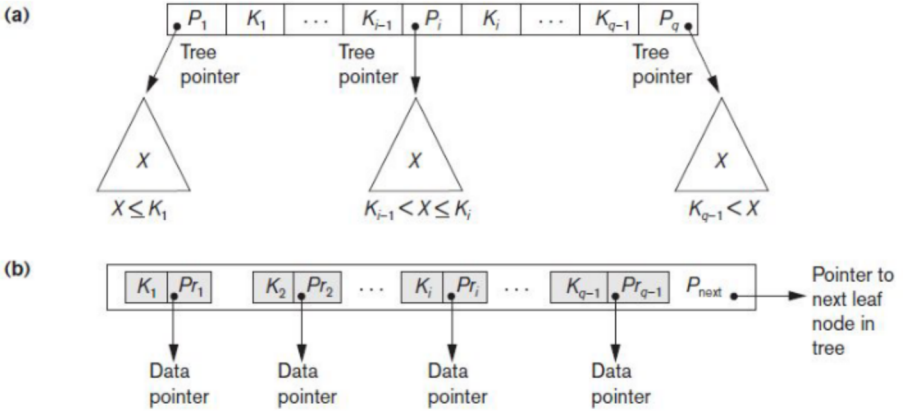


Figure 17.11
The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.

Using B-Trees and B+-Trees

- Insertion and deletion are relatively efficient
- B-trees are sometimes used as primary file organizations. In this case, whole records are stored within the B-tree nodes rather than just the <search key, record pointer> entries
- Because entries in the internal nodes of a B+-tree include search values and tree pointers without any data pointers, more entries can be packed into an internal node of a B+-tree than for a similar B-tree. Thus, for the same block (node) size, the order p will be larger for the B+-tree than for the B-tree. This can lead to fewer B+-tree levels, improving search time.

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

$$(p * P) + ((p - 1) * V) \leq B$$
$$(p * 6) + ((p - 1) * 9) \leq 512$$
$$(15 * p) \leq 512$$

We can choose p to be the largest value satisfying the above inequality, which gives $p = 34$. This is larger than the value of 23 for the B-tree (it is left to the reader to compute the order of the B-tree assuming same size pointers), resulting in a larger fan-out and more entries in each internal node of a B^+-tree than in the corresponding B-tree. The leaf nodes of the B^+-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the order p_{leaf} for the leaf nodes can be calculated as follows:

$$(p_{leaf} * (Pr + V)) + P \leq B$$
$$(p_{leaf} * (7 + 9)) + 6 \leq 512$$
$$(16 * p_{leaf}) \leq 506$$

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

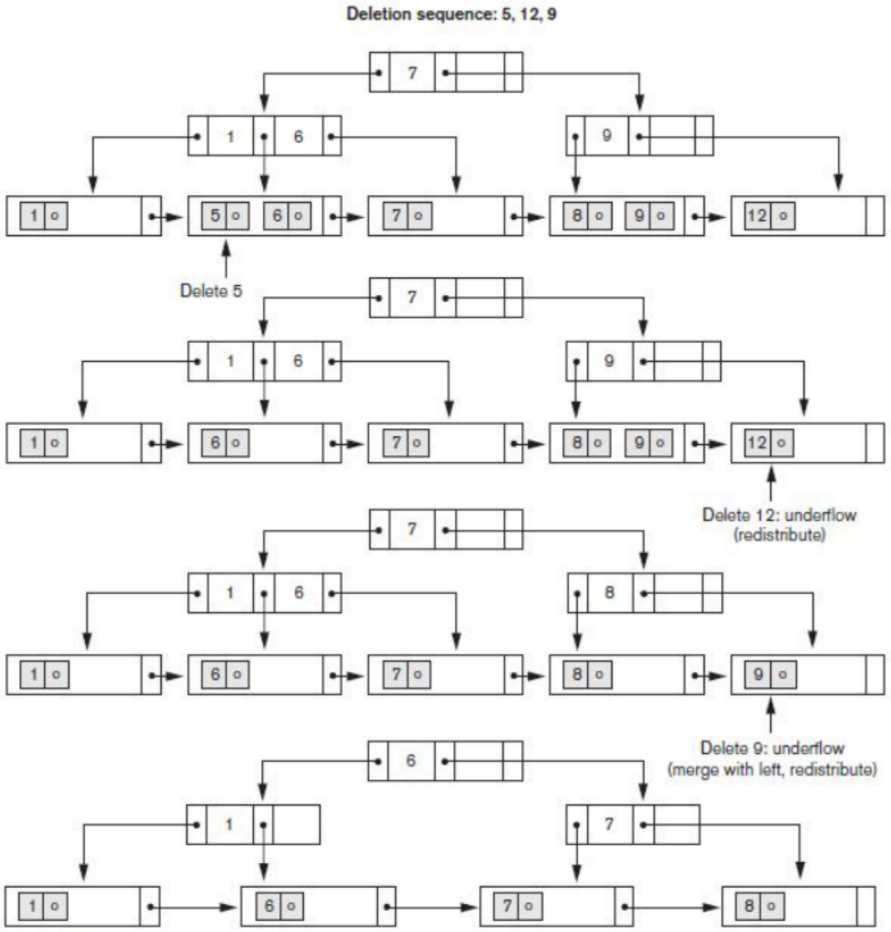


Figure 17.13
An example of deletion from a B^+-tree.

```
CREATE INDEX index_name ON table_name USING btree
(
   column_name [ASC | DESC] [NULLS {FIRST | LAST }],
   ...
);
```

Indexes on Multiple Keys

Indexes on Multiple Keys

- In many retrieval and update requests, multiple attributes are involved
  - If a certain combination of attributes is used frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes
- Say and you frequently issue queries like:

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

then it might be appropriate to define an index on the columns major and minor together, e.g.:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

- In PostgreSQL, only the B-tree, GiST, GIN, and BRIN index types support multicolumn indexes. Up to 32 columns can be specified.

Implementation

- Ordered Index on Multiple Attributes
  - A lexicographic ordering of these tuple values establishes an order on this composite search key. For our example, all of the department keys for department number 3 precede those for department number 4. Thus <3, n> precedes <4, m> for any values of m and n
- Partitioned Hashing
  - It is suitable only for equality comparisons; range queries are not supported
  - For a key consisting of n components, the hash function is designed to produce a result with n separate hash addresses. The bucket address is a concatenation of these n addresses
  - For example, consider the composite search key <Dno, Age>. If Dno and Age are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that Dno = 4 has a hash address ‘100’ and Age = 59 has hash address ‘10101’. Then to search for the combined search value, Dno = 4 and Age = 59, one goes to bucket address 100 10101

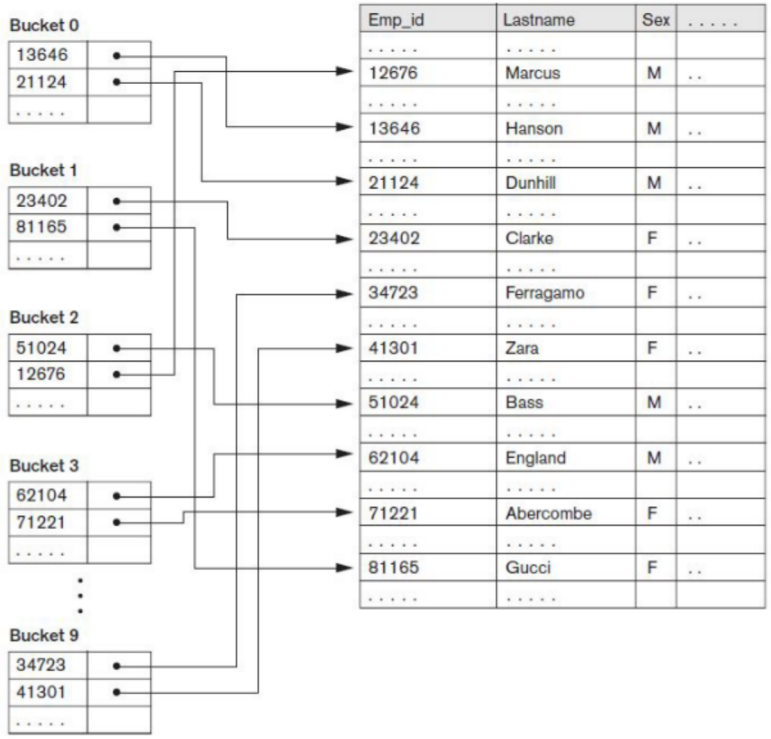
Other Types of Indexes

Hash Indexes

- The hash index is a secondary structure to access the file by using hashing on a search key other than the one used for the primary data file organization
- The index entries are of the type <key, pointer to the block or record>

Figure 17.15
Hash-based indexing.

1+3+6+4+6 mod 10 =
2+1+1+2+4 mod 10



Hash Indexes

```
CREATE TABLE b_table (id VARCHAR(256));
CREATE TABLE h_table (id VARCHAR(256));

COPY b_table FROM '/PATH/data.txt';
COPY h_table FROM '/PATH/data.txt';
```

```
CREATE INDEX b_table_id_index ON b_table USING btree (id);

CREATE INDEX
Time: 3800872.035 ms

CREATE INDEX h_table_id_index ON h_table USING hash (id);

CREATE INDEX
Time: 290529.397 ms
```

Size of the indexes

```
SELECT relname, pg_size_pretty(pg_relation_size(C.oid))
FROM pg_class C
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE nspname NOT IN ('pg_catalog', 'information_schema', 'pg_toast') AND relkind = 'i';

     relname      | pg_size_pretty
------------------+----------------
 h_table_id_index | 4096 MB
 b_table_id_index | 3873 MB
(2 rows)
```

```
EXPLAIN ANALYSE SELECT id FROM b_table WHERE id = '00000000000000000001';
                                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Index Only Scan using b_table_id_index on b_table  (cost=0.57..8.59 rows=1 width=21) (actual time=3.913..3.913 rows=0 loops=1)
   Index Cond: (id = '123123123'::text)
   Heap Fetches: 0



EXPLAIN ANALYSE SELECT id FROM h_table WHERE id = '00000000000000000001';
                                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Index Scan using h_table_id_index on h_table  (cost=0.00..8.02 rows=1 width=21) (actual time=1.526..1.529 rows=1 loops=1)
   Index Cond: ((id)::text = '00000000000000000001'::text)
```

Compare with this instruction before:
SET enable_indexonlyscan = off;

```
Warming up --------------------------------------
               btree   283.000  i/100ms
                hash   363.000  i/100ms
Calculating --------------------------------------
               btree      2.912k (±15.5%) i/s -     14.150k
                hash      3.967k (± 6.1%) i/s -     19.965k


Comparison:
                hash:     3966.7 i/s
               btree:     2911.6 i/s - 1.36x slower
```

Bitmap Indexes

- Bitmap indexes facilitate querying on multiple keys
- Bitmap indexing is used for relations that contain a large number of rows
- It creates an index for one or more columns, and each value or value range in those columns is indexed
- Typically, a bitmap index is created for those columns that contain a fairly small number of unique values

EMPLOYEE

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Bitmap index for Sex

M	F
10100110	01011001

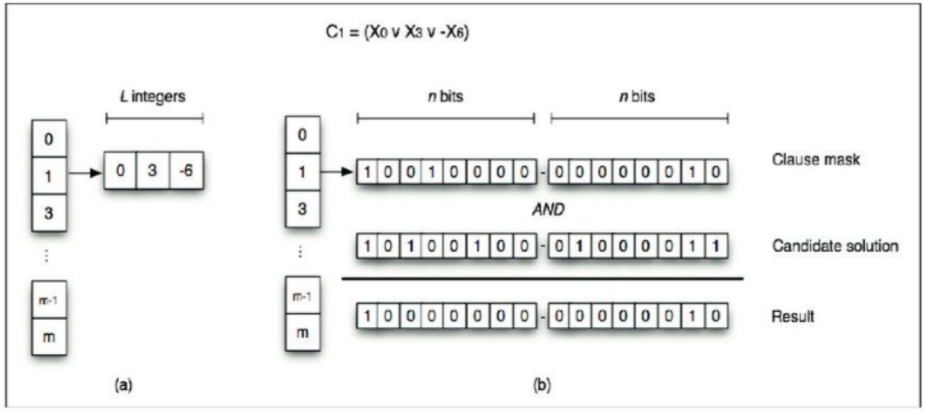
Bitmap index for Zipcode

Zipcode 19046	Zipcode 30022	Zipcode 94040
00101100	01010010	10000001

Figure 17.16
Bitmap indexes for Sex and Zipcode.

Bitmap Indexes

- Bitmap indexes are efficient in terms of the storage space that they need
- Example: If we consider a file of 1 million rows (records) with record size of 100 bytes per row, each bitmap index would take up only one bit per row and hence would use 1 million bits or 125 Kbytes
- Suppose this relation is for 1 million residents of a state, and they are spread over 200 ZIP Codes; the 200 bitmaps over Zipcodes contribute 200 bits (or 25 bytes) worth of space per row; hence, the 200 bitmaps occupy only 25% as much space as the data file
- They allow an exact retrieval of all residents who live in a given ZIP Code by yielding their Row_ids



Bitmap Indexes

- When records are deleted, renumbering rows and shifting bits in bitmaps becomes expensive
- Another bitmap, called the existence bitmap, can be used to avoid this expense
  - This bitmap has a 0 bit for the rows that have been deleted but are still physically present and a 1 bit for rows that actually exist
- Whenever a row is inserted in the relation, an entry must be made in all the bitmaps of all the columns that have a bitmap index
  - Rows typically are appended to the relation or may replace deleted rows to minimize the impact on the reorganization of the bitmaps
  - This process still constitutes an indexing overhead

Some General Issues Concerning Indexing

Issues Concerning Indexing

- Index creation
  - Insertion of a large number of entries into the index is done by a process called bulk loading the index
  - We must go through all records in the file to create the entries at the leaf level of the tree
  - These entries are then sorted and filled according to the specified fill factor; simultaneously, the other index levels are created. It is more expensive and much harder to create primary indexes and clustering indexes dynamically, because the records of the data file must be physically sorted on disk in order of the indexing field
- Tuning Indexes
  - 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

Factors That Influence Physical Database Design

Factors That Influence Physical Database Design

1. **Analyzing the Database Queries and Transactions**: we must have a good idea of the intended use of the database by defining in a high-level form the queries and transactions that are expected to run on the database
    - The files (relations) that will be accessed by the query
    - The attributes on which any selection conditions for the query are specified
    - Whether the selection condition is an equality, inequality, or a range condition
    - The attributes on which any join conditions or conditions to link multiple tables or objects for the query are specified
    - The attributes whose values will be retrieved by the query

2. **Analyzing the Expected Frequency of Invocation of Queries and Transactions**
    - 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

Factors That Influence Physical Database Design

3. **Analyzing the Time Constraints of Queries and Transactions**

   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

4. **Analyzing the Expected Frequencies of Update Operations**

   The overhead for updating indexes can slow down the insert operations

5. **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