

## 10. File Organizations and Indexes | Physical Schema

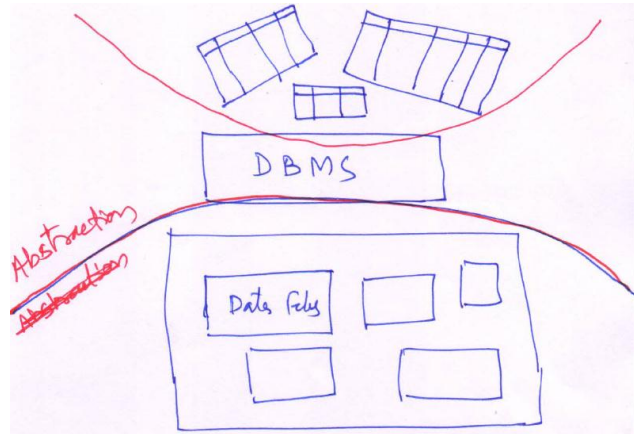
A relation is a logical representation of databases.

DBMS lets us manipulate databases in logical manner like relations. Things like RDBMS implements relational abstraction over data in disk files.

Data are stored in files on disks.

There are certain ways in which relations are stored in disk files –

1. One relation one disk files
2. All relation one file – SQLite, MS-access are such examples
3. One relation many files.



Structure of data on disks is referred as *physical schema*.

Physical schema typically defines the organization of *data records* in file(s), and access paths for finding data records.

In this chapter, we attempt understanding various *file organizations* and *indexes* for efficient access of data records.

### Data Files:

Logically, a *data file* can be thought of collection of *data records*. For storing relations, a record typically represents a *tuple*.

While studying file organizations, we use a concept of *disk block*. It is a smallest storage unit for manipulating data in disk files. A disk block – can be of different sizes on different systems - 4KB, 8KB, 16KB or so. 8KB should a standard on a modern DBMS.

A data file can be chained (or indexed) collection of block clusters.

DBMS typically takes services of Operating system for manipulating disk files. Read/write operations are performed on disk blocks.

A block may store multiple records; though rare, but it is also possible to have records that do not fit in a block, and span to multiple blocks.

Records in a file can be of two types –

1. “fixed length record” and
2. “variable length records” –

These terms mean what they pronounce for.

If storage required for a record is fixed for all records in a file, then it is fixed length record file; otherwise it is a variable length record file.

Length of a record in a file can be variable due to one and more of following reasons-

- If file stores tuples from more than one relations.
- Length of one or more fields is variable.
- A record having multi value fields.

Files with fixed length records are easier to implement and manage. However fix length variable records are less practical and variable length record files are common in practice.

For implementing variable length record files, there are different strategies, like

1. Having record header containing information of length for each field, or
2. We can have field separators, or so.

## Operations on data files

- Insert a record
- Find a record,
- Find a set of records meeting some criteria
- Modify a record
- Delete a record, or a set of records meeting some criteria

There are various techniques for performing these operations on files; however the technique is dependent on “File Organization” used for recording data.

*File Organizations* basically is the way of organizing records in a file. Every file organization defines some methods to locate a record (or a set of records) in a file, referred as *Access Methods*. There can be multiple access methods on file organization but an access method may require a particular file organization.

## File Organizations

File Organization is a method of organizing the records in a disk file. File organization dictates procedure of performing said operations on files.

Each file organization makes certain operations efficient while others expensive.

Many alternative file organizations exist, following are commonly used file organizations-

- Heap (random order) files
- Sorted Files (on some attribute).
- Indexes

### Heap (random order) files:

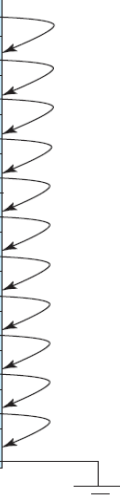
- Records have no order
- Insertions are very efficient; in this case, often, as new records are added at the end of file.
- New records can also be placed at empty spaces anywhere in between of the file.
- Finding a record is expensive, as we can only perform sequential scan – in the order of  $N^2$

### Sorted Files

- Records of file are sorted on some field(s), called as ordering field(s). Ordered field(s) is also called as search key. For example you can store employee relation records sorted on SSN, or Department?
- Figure below from book<sup>1</sup> shows a typical ordered file. Let us say employee records are arranged in ascending order of employee number.

[Note that However actual storage may not as simple; there can be some strategy for keeping records ordered within the blocks, and then blocks are chained through linked lists in sorted manner.]

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	



<sup>1</sup> Database System Concepts, 6th ed; Silberschatz, Korth and Sudarshan

- Searching on ordering field is efficient as we can perform things like binary search – locate middle block, see if search key < value in ordering field of middle block, and so on.  
==> Number of disk blocks needs to be processed is in the order of  $\log_2(N)$ .
- Insertion in sorted file is expensive; can be performed as
  - Find appropriate location (block) in file where the record can be inserted (order needs to be maintained)
  - Once neighborhood is found, it may encounter various situations
    - One, the block containing neighborhood has room where the new record can be inserted –if so, new record is placed in that block
    - The block does not have room for new record – then see if next block has the room (and that still maintains order),
    - If still not possible, then an overflow block is added, and the new record is inserted there.
- Deletion is also expensive; done as following
  - Find record, record content is removed from the block; actual action will depend on how records are organized in a block.
  - One of the common strategies for space created by deleting a record is, keep it as blank and we do not shift the remaining records; such empty spaces are recycled using some strategy like “free list” maintenance or so.
- In a record modification operation, if we need to change values of ordering fields; then location of the record will change, in such as modification is performed in two steps-
  - Remove from current location
  - Insert at appropriate new location

### How do you compute cost of an operation?

Disk I/O is considered as most significant component in performing database operations. Total number of disk blocks read and write measures the cost of performing an operation. A technique that processes (reads and writes) larger number of disk block is considered more expensive than the one that processes lesser number of blocks.

Suppose a file has 100000 records; let us say a record uses 350bytes, and store approximately 10 records in a 4KB block. The file will be using 10000 blocks. If file is “Heap File”, in worst case, we need to read all the blocks, i.e. 10000; whereas if file is ordered then we can perform binary search  $\log_2(10000)$ , that is only 14 blocks; therefore, for searching in an “ordered file” is much efficient in comparison to a heap file.

## Indexes

Though sorted files improves the search performance in logarithmic order, it has following issues-

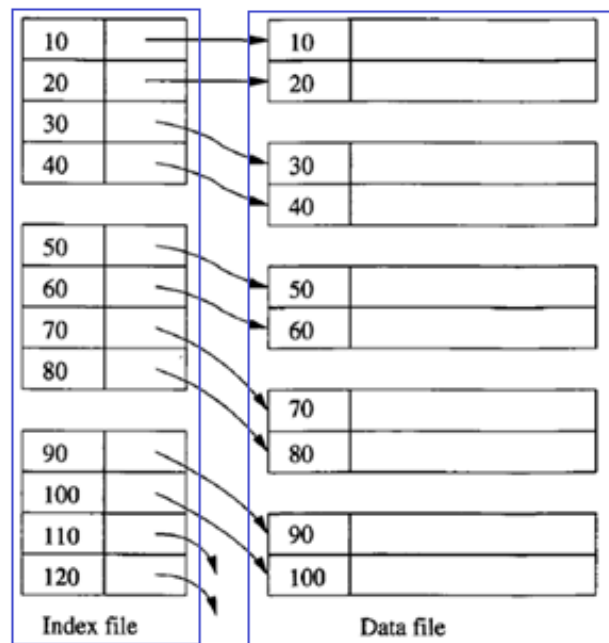
1. A file can be ordered only on one attribute (or attribute set)
2. Complexity of insert/delete/update is higher

Indexes help addressing these issues. Indexes are basically maps data records from a “key-values”. Figure below from book<sup>2</sup> depicts some sample maps.

Index is basically a file that maps a “search-key value” to data record address (typically address of block containing corresponding data record).

Below are some types of index records-

- Index on ssn for employee relation  
<101, adr-1>,  
<103, adr-2>, ....
- Index on DNO for employee relation  
<1, <adr-1, adr-3, adr-7>>,  
<3, <adr-2, adr-5, adr-6>>, ....
- Index on (DNO, Salary)  
<<1, 30000>, <adr-1, adr-7>>,  
<<3, 45000>, <adr-6>>, ....



Indexes speed up searches for a subset of “records”, based on values in certain fields; termed as “search key”.

A “Search keys” for index is -

- Any subset of the attributes of a relation can be the search key for an index on the relation
- Search key may not be same as key of a relation.
- Typically, index has entries as following-  
<search-key, block/record-address>, or <search-key, block/record-address list>

Note that Index helps in reducing number of blocks to be processed for file operations; they are even at advantage over the ordered files.

<sup>2</sup> Garcia-Molina, Hector. *Database systems: the complete book*. Pearson Education India, 2008.

Again consider the example of a file with 11ac records above. Size of index record will be much smaller than actual data record; let us say we can store 100 index records in a block; for 11ac index entries, we need 1000 blocks. Indexes are always sorted, therefore we can always perform binary search. In this case number of blocks to be read are  $\log_2(1000)=10$  blocks. Not much gain, though!

However in practice indexes are not that flat, they are *multi-level*.

### Types of Indexes

- Primary and Secondary
- Dense and sparse
- Clustered and non-clustered

### Structures of indexes

- Tree based
- Hash based

There can be multiple indexes on a data file.

### Clustered and non-clustered Index

- If records on disk file are sorted in index order then it is clustered index otherwise not.
- We can have data clustered only on one index.

### Dense and Sparse Indexes

- If you have index entry for every search key-value in a file (relation), then it is dense index.
- “Clustered indexes” can only be sparse, where you do not store disk page address for each record. Advantage here is you have a smaller index and can load more portion of index in primary memory.
- In this case; search can be done by locating block for a key that is largest less than search-value; and then sequentially scanning sorted data file.

Figure 14.3 above from book<sup>3</sup> depicts concept of sparse index

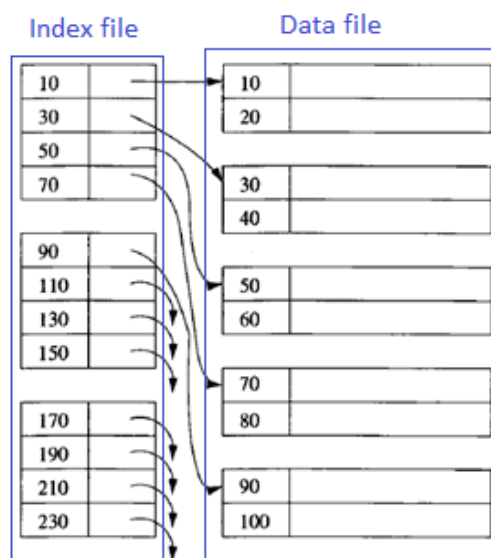


Figure 14.3: A sparse index on a sequential file

<sup>3</sup> Garcia-Molina, Hector. *Database systems: the complete book*. Pearson Education India, 2008.

## Primary Index

- An index based on attributes on which data file is also physically sorted, is referred as *primary index*. In most cases primary index is based on primary key.
- Two data records are said to be duplicate if they have same value for search key field(s).
- If no duplicate exists then it is unique index, typically index on candidate key

## Secondary Index

Indexes other than primary index are secondary indexes, and

- can be on any attributes
- need not to be unique
- However secondary index is always dense
- there can be multiple secondary indexes on a file of records to access records in multiple ways

## Multi-level index

Helps in reducing disk I/O in locating records on disk; figure 14.4 below from book<sup>4</sup> depicts concepts of multiple level indexes.

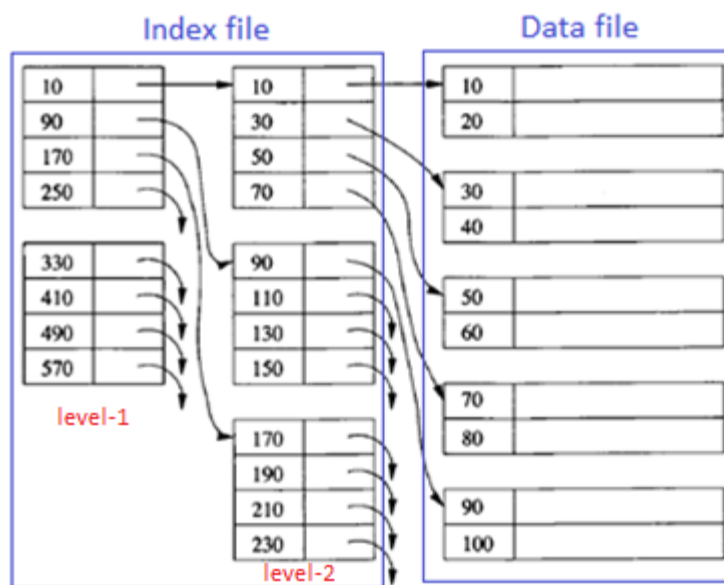


Figure 14.4: Adding a second level of sparse index

To explain motivation of multi-level indexes, let us look at an example-

<sup>4</sup> Garcia-Molina, Hector. *Database systems: the complete book*. Pearson Education India, 2008.

Suppose we build a dense index on a relation with 1 million tuples. Index entries are smaller than data records. Let us assume that 200 index entries fit on an 8 kilobyte block. Thus, our index occupies 5,000 blocks using approximately 40MB.

Let us say that index entries are maintained in ordered file organization.

Searching in such a file using binary search will require reading  $\log_2(5000) = 13$  (approx.) blocks on average case.

Now suppose we create one higher level sparse index that holds one entry for each index block. Let us call this new index as level-1 index while original index as level-2 index, as shown in figure 14.4 above. At index level-1, for storing pointers to 5000 blocks (of level-2), we require  $5000/200 = 25$  blocks.

Therefore size of index at Level-1 is 25 blocks, while it is of size 5000 blocks at level-2. Data record addresses are stored only at lowest level, which is two here.

Now, let us say, searching is performed as following-

1. Perform binary search at level-1, and locate block that contains address of block at level-2 that possibly contains address of data record. Number of blocks read at level-1 are  $\log_2(25) = 5$ .
2. Read the corresponding block at level 2, and find out address of the data record. Number of blocks processed at level-2 is one.
3. Total numbers of blocks read in this approach are six.

==> The gain: 13 blocks (in case of ordered file) vs 6 blocks read (in two level index)!

This way, number of levels in an index can be more than two, and hence called as multi-level index. Number of levels are typically determined such that index at top level (or root level) is small enough to be accommodated in single block.



## B<sup>+</sup>-tree Index

Indexes seen so far are ordered files of index entries, often referred as *sequential index files*. Binary search, in flat index files may become expensive when files are larger. Multi-level indexes provide a significant gain over flat index files. B<sup>+</sup>-trees are a type of multi-level index, and quite popular structure used in database indexing.

Characteristics of B<sup>+</sup>-trees indexes:

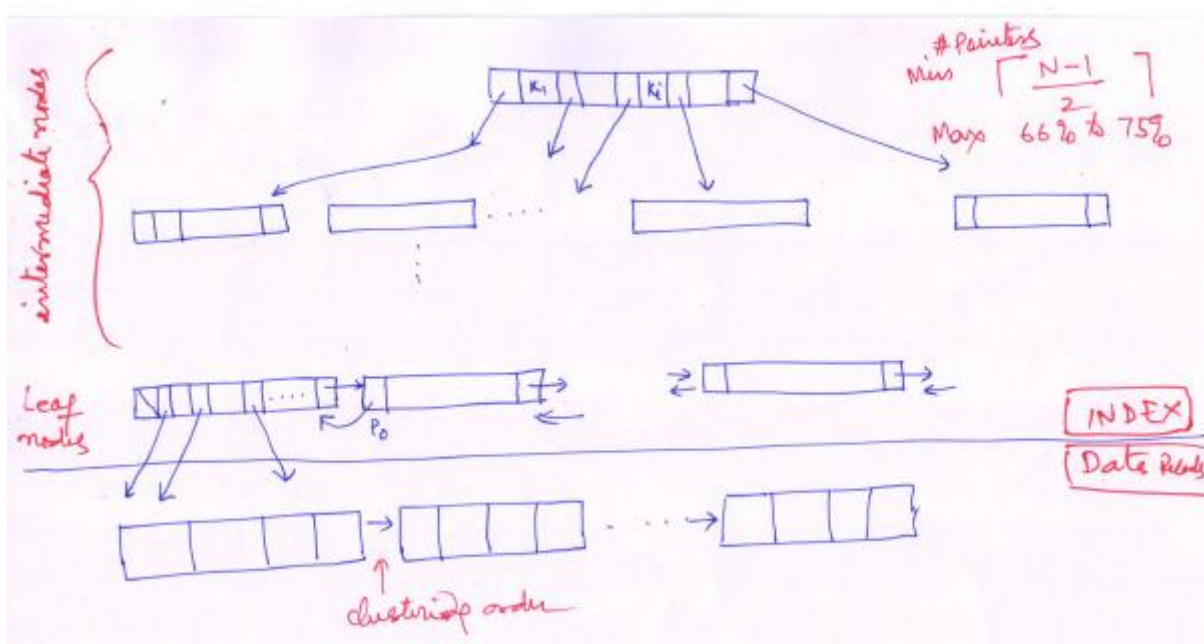
- Primary difference over B-Tree is that only leaf nodes store address for data records. This enables increasing the number of pointers that can be stored in a node. And ultimately reducing the size of tree.
- A node is typically disk block; below is a general B+-tree node with n index entries-

$P_1$	$K_1$	$P_2$	...	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	-----	-----------	-----------	-------

where  $P_1$  through  $P_n$  are pointers, and  $K_1$  through  $K_{n-1}$  are key values.

- Key values in a node are ordered.
- Size of all nodes is same.
- Nodes may not be used for full capacity; some empty space is kept to accommodate efficient inserts. [Min pointers =  $\lceil (N - 1)/2 \rceil$  and max is about 66% to 75%].
- Leaf nodes
  - Leaf nodes only contain pointers to data records
  - A pointer  $P_i$  in a leaf node points to record with search key  $\leq K_i$ .
  - All leaf nodes combined will be in the order of search key.
  - The pointer  $P_n$  (that is last pointer) of a leaf node point to next leaf node, and maintains a linked list of leaf nodes.
  - Leaf node may also be linked doubly; may have a pointer like  $P_0$  that refers to previous node.
  - Leaf node will contain pointer to all data records in data file, if it is dense index, which is usually the case is.
- Non-leaf node
  - Non leaf nodes contain pointers to its children nodes
  - A pointer  $P_i$  in a node points to record with search key  $< K_i$  and  $\geq K_{i-1}$
  - A search value appearing parent node will also appear in child node.

Figure below should be capturing the conceptual layout of a B<sup>+</sup>-tree index. Note that leaf nodes are referring *data records*.



Hopefully you understand how we perform following operations on b-tree; almost same algorithms can be applied on a disk based B<sup>+</sup>-tree of disk blocks containing index entries-

- search (equality and range searches),
- insert,
- delete

We can use tree in figure below to explain.



Below is typical algorithm for searching a value [from book<sup>5</sup>]-

<sup>5</sup> Database System Concepts, 6th ed; Silberschatz, Korth and Sudarshan

```

function find(value V)
/* Returns leaf node C and index i such that C.Pi points to first record
* with search key value V */
  Set C = root node
  while (C is not a leaf node) begin
    Let i = smallest number such that  $V \leq C.K_i$ 
    if there is no such number i then begin
      Let  $P_m$  = last non-null pointer in the node
      Set C = C.Pm
    end
    else if ( $V = C.K_i$ )
      then Set C = C.Pi+1
    else C = C.Pi /*  $V < C.K_i$  */
  end
  /* C is a leaf node */
  Let i be the least value such that  $K_i = V$ 
  if there is such a value i
    then return (C, i)
  else return null ; /* No record with key value V exists */

```

Update on data file requires updating indexes as well

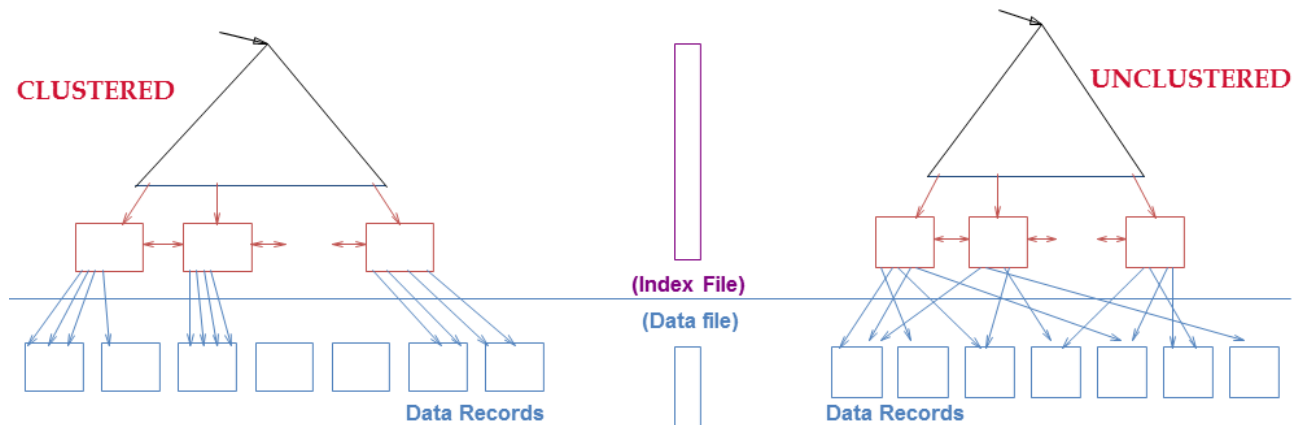
Below are some observation related to updating B+-tree based indexes-

- When we update data files (insert/update/delete), we need to updates all indexes (as said, there can be multiple indexes on a data file)
- To insert a record in data file, we also need to make appropriate index entry in B+-tree node; this new entry may require we splitting the node, and may further require balancing of tree, etc.
- While deleting a record from data file, we also need to remove corresponding entry from B+-tree node; if the node becomes empty more than (let us say  $k/2$ ), we require merging it with some other node,
- And so forth

Clustered vs. Un-clustered Indexes

- If order of data records is the same as, or “close to”; order of data entries in index, then index is called clustered index.
- A file can be clustered on at most one search key.
- To build clustered index, first sort the Heap file (with some free space on each block for future inserts).
  - Overflow blocks may be needed for inserts. (Thus, order of data recs is ‘close to’, but not identical to, the sort order.)

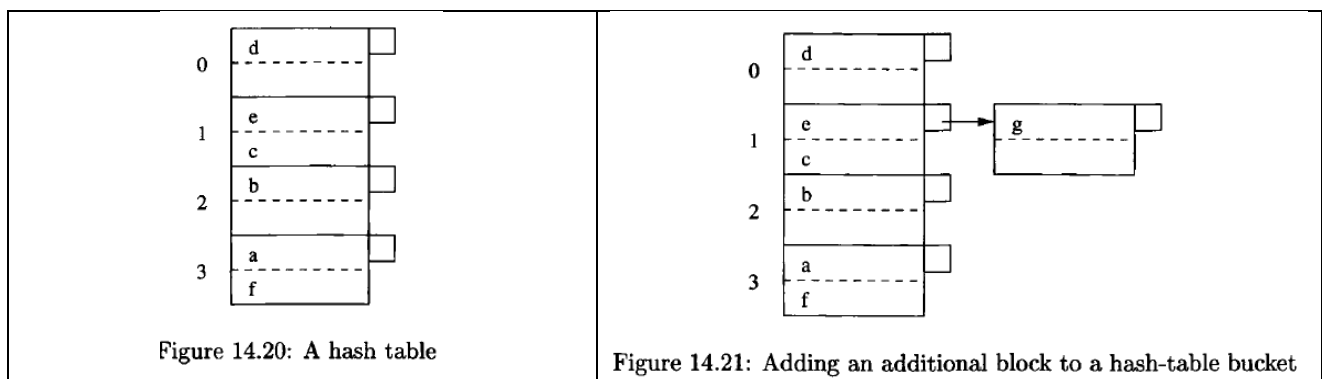
Figure below depicts the logical layout of clustered and un-clustered indexes (from book<sup>6</sup>)



Also note how range search will get affected if data records are also in the order of index

## Hash-Based Indexes

- Index is a collection of *buckets*. [also known as Hash Tables]
  - Bucket = *primary bucket* plus zero or more *overflow buckets*. (for collision handling)
  - Buckets contain index records.
- Hashing function h*:  $h(k)$  = bucket in which index entry for key  $k$ .



==> Hash based indexes are only good for equality selections.

Extensible Hashing [figures from book<sup>7</sup>]

<sup>6</sup> Database Management Systems 3ed, Ramakrishnan and Gehrke

<sup>7</sup> Garcia-Molina, Hector. Database systems: the complete book. Pearson Education India, 2008.

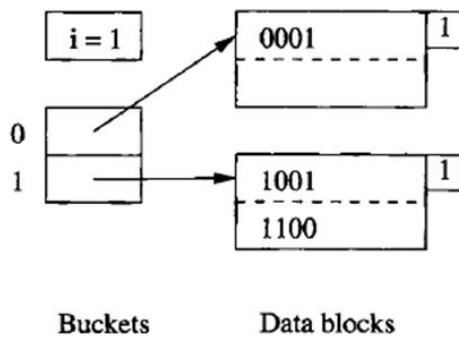
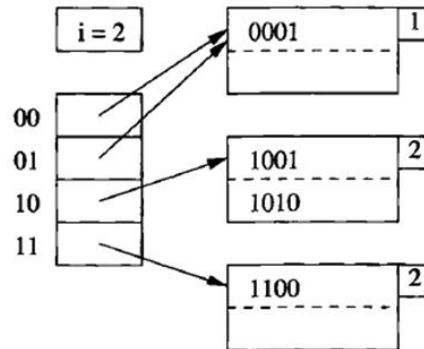


Figure 14.23: An extensible hash table



Now, two bits of the hash function are used

## Maintenance of Indexes

- Since indexes are used as access path for data records on the disk, we need to update index too when
  - New record in the data file is added
  - “search-key” value of a record is getting modified
  - A record is getting deleted

## Indexes – Benefit/Cost

- Indexes consume extra storage space – sometimes may be as large as database itself
- Still indexes are used, but a tradeoff is used for deciding number of indexes or so
- If selection queries are frequent, sorting the file or building an *index* is important.
- Index Maintenance: needs to be in sync with the data files and that slows down update (INSERT/Modify/Delete) operations.
- Reduces “concurrency”
- If table size is small – you may not need index, and if table is huge and rarely updated we may have index on almost all attributes that are frequently used in retrieval queries

## Indexing support in PostgreSQL

- Has CREATE INDEX command. Though not part of standard SQL!!  
Short form Syntax (postgre-SQL) -  
**CREATE [ UNIQUE ] INDEX *index-name* ON *table* [ USING *method* ] (*col-name*)**
- Method can be btree, hash, and some more. The default method is btree.
- Postgres's hash method hardly has any advantage over btree for equality check
- In most RDBMS, indexing on Primary Key is automatically created. For example, in relation, we get index named "employee\_pkey" automatically created.

**CREATE INDEX EMP\_DNO\_IDX ON EMPLOYEE(DNO);**

**CREATE INDEX EMP\_DNO\_IDX ON EMPLOYEE USING BTREE(DNO);**

- We also ALTER INDEX and DROP INDEX commands

**DROP INDEX EMP\_DNO\_IDX;**

- We can cluster the database based on some index

**CLUSTER employee USING emp\_dno\_idx;**

**CLUSTER employee USING employee\_pkey;**

### Rebuilding indexes

- Indexes may go "Corrupted" or "Bloated", and then we require rebuilding indexes. For the same we have re-index command
- Syntax: **REINDEX { INDEX | TABLE | DATABASE | SYSTEM } name**