DATABASE TECHNOLOGY - 1MB025

Fall 2005

An introductury course on database systems

http://user.it.uu.se/~udbl/dbt-ht2005/
alt. http://www.it.uu.se/edu/course/homepage/dbastekn/ht05/

Kjell Orsborn
Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden

UPPSALA
UNIVERSITET

Introduction to Physical Database Design

Elmasri/Navathe ch 13 and 14
Padron-McCarthy/Risch ch 21 and 22

Kjell Orsborn
Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden

UPPSALA
UNIVERSITET

Contents - physical database design

- Record and file organization
  - Data structures for physical storage of the database
    - Unsorted files
    - Sorted files
    - Hashing methods
- Indexes and index files
  - a) Simple (one-level) index
    - Primary index
    - Secondary index
    - Cluster index
  - b) Search trees (multi-level index)
  - c) Hash indexes

The physical database

- The physical database is a collection of stored records that have been organized in files on the harddisk.
  - A record consists of a number of data fields.
  - Each field has an elementary data type (integer, real, string, pointer etc.)
- Records are used for physical storage of:
  - Tuples, where each attribute in the tuple is stored as a field.
  - Objects in object-oriented databases.

Block transfer is slow!

- Block – a contiguous sequence of disc sectors from a single track.
  - data is transferred between disk and main memory in blocks
  - sizes range from 512 bytes to several kilobytes
  - block transfer is **slow** (12-60 msec)
    - i.e. position the red/write head of the disk at the right track and at the correct block/sector, and then transfer the block to primary memory.
  - disk-arm–scheduling algorithms order accesses to tracks so that disk arm movement is minimized.
- File organization – optimize block access time by organizing the blocks to correspond to how data will be accessed. Store related information on the same or nearby cylinders.

Storage of records

- A block is usually bigger than a record such that a block consists of one or several records.
- The highest number of records that can be contained in a block is called the **block factor** (here: **bfr**) for the file of records.
- If R is the record size and B the block size:
  bfr = floor[B/R]
  - E.g. assume a block B=512 bytes, record size R=79 bytes.
  - B / R = 512/79 = 6.48
  - Rounded off downwards gives bfr = 6, i.e. we can store 6 records per block.
- A file with *r* no. of records therefore require:
  b = ceiling[r/bfr] blocks (see Elmasri/Navathe Fig 13.6)

Storage access

- A database file is partitioned into fixed-length storage units called blocks. Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- Buffer – portion of main memory available to store copies of disk blocks.
- Buffer manager – subsystem responsible for allocating buffer space in main memory.

Buffer manager

- Programs call on the buffer manager when they need a block from disk
  - The requesting program is given the address of the block in main memory, if it is already present in the buffer.
  - If the block is not in the buffer, the buffer manager allocates space in the buffer for the block, replacing (throwing out) some other block, if required, to make space for the new block.
  - The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
  - Once space is allocated in the buffer, the buffer manager reads in the block from the disk to the buffer, and passes the address of the block in main memory to the requester.

File organization

- The database is stored as a collection of files.Each file is a sequence of records. A record is a sequence of fields.
- Records can have constant (simplest) or variable length
- A file can store records of the same type (simplest) or of different type.
- Specific files can be used to store specific relations (simplest) or the same file can store different relations (maybe even the whole database).

File descriptor

- Contains information that is needed for record access:
  - Block addresses, record format etc.
  - To find records, one or several blocks transferred to (one or several buffers in) primary memory. These blocks can then be searched to find the records that were sought.
  - If the address to the block containing the record is unknown one has to search through all block in the file (so called *linear search*).

Organization of records in files

- **Heap** – a record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function is computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- **Clustering** – records of several different relations can be stored in the same file; related records are stored on the same block

Heap - files with unordered records

- New records are added to the end of the file. Such an organization is callad a **heap** file.
  - Suitable when we don't know how data shall be used.
- *Insert* of a new record is very efficient.
- *Search* after a specific record is expensive (linear to the size).
- *Delete* of a record can be expensive (search - read into - delete - write back).
  - Instead of physically removing a record one can mark the record as deleted. Both methods require a periodically reorganization of the file.
- *Modification* of a record of variable length can be hard.
- *Retrieval* according to a certain *order* requires that the file must be sorted which is expensive.

Sequential - files with ordered records

- The records in the file are ordered according to the value of a certain field (Elmasri/Navathe Figure 13.7)
  - *Ordered retrieval* very fast (no sorting needed).
  - Next record in the order is found on the same block (except for the last record in the block)
  - Search is fast (*binary search* - $\log_2 b$)
  - *Insert* and *delete* are expensive since the file must be kept sorted.
  - Suitable for applications that require sequential processing of the entire file
  - Need to reorganize the file from time to time to restore sequential order

Sequential - files with ordered records cont'd. . .

- To make record insertions cheaper:
  - Create a temporary unsorted file a so called **overflow file** or **transaction file** (the main file is then called "**the master file**")
  - Update the master file periodically in accordance with the transaction file.
  - These measures improve insertion time but search for records beomes more complicated.
- Ordered files are not used that often in databases.
  - Exception: when extra access paths are created, so called primary indexes.

Hashing technique in general

- **Goal**: to make record retrieval faster.
- **How**: find a hash function, *h*, that for a record *p*, *h(f(p))* provides the **address** to the block where *p* shall be stored.
- *h(f(p)) = f(p) mod M* where *f(p)* is called the **hash field** for p and *M* is the table size.
- This means that most of the records will be found in only one block access.
- Observe that we get a collision if: *h(f(p)) = h(f(p'))*

Solving collisions in hashing

There are several method for collision elimination:

- Open addressing:
  - Take next free place *a* in the array $h(f(p)) \leq a \leq M-1$
- Chaining:
  - Choose a larger array of size $M+O$ and use the extra space as overflow places. (E/N Figur 13.8b)
- Multiple hashing:
  - If *h* leads to collision use *h'* in stead. If there is collision again use open addressing or use *h''* and then open addressing if collision occurs.

Hashing - properties

- Different hashing methods need different insertion, delete and retrieval algorithms.
- What is a good hashing method?
  - Records should be distributed uniformaly in the hash table such that collision and unused positions are minimized.
- Principles - rules of thumb:
  - 70-90% of the hash table must be utilized.
  - If *r* records shall be stored use a hash table of the size between *r/0.9* and *r/0.7*
  - Preferably chose a prime number size of the sizeof the hash table - this will give a more uniform distribution.

External hashing

- Hashing methods for files on disc:
  - Records are hashed according to the bucket method.
  - A bucket = one block.
  - The bucket address is used as the address for a record.
  - Info. regarding the block address for each bucket is stored in the beginning of the file.

(Se E/N Figur 13.9)

- Since several records can be stored in the same block, the number of collisions decreases.
- When a bucket is filled one can use a chaining method where a number of buckets are used as overflow buckets.

(Se E/N Figure 13.10)

Pros and cons with external hashing

Pros:

- Retrieval of a random record is fast.

Cons:

- Retrieval of records in an order according to the value of the hash field.
- Searching for records with regard to another data field than the hash field is very costly.
- Files will get a predetermined size. However, there are various techniques to provide dynamic file sizes - e.g. *linear hashing* techniques

Clustering file organization

- Simple file structure stores each relation in a separate file
- Can instead store several relations in one file using a clustering file organization
- E.g., clustering organization of customer and depositor:

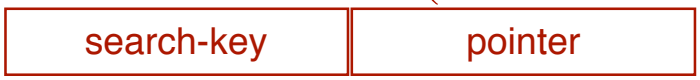
Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Hayes	A-305	

  - good for queries involving depositor customer, and for queries involving one single customer and his accounts
  - bad for queries involving only customer
  - results in variable size records

UPPSALA UNIVERSITET

Indexes - index files (ch. 14)

- An **index** (or index file) is an extra file structure that is used to make the retrieval of records faster.
  - **Search key** (or *index field*)– attribute or set of attributes (*data fields*) used to look up records in a file.
- An **index file** consists of records (called index entries) of the form:



  - These entries determine the physical address for records having a certain value in their *index field*.
  - Index files are typically much smaller than the original file
  - The file that should be indexed is called the *data file*.
- Two basic kinds of indexes:
  - Ordered indexes: search keys are stored in sorted order
  - Hash indexes: search keys are distributed uniformly across "buckets" using a "hash function".

Index evaluation metrics

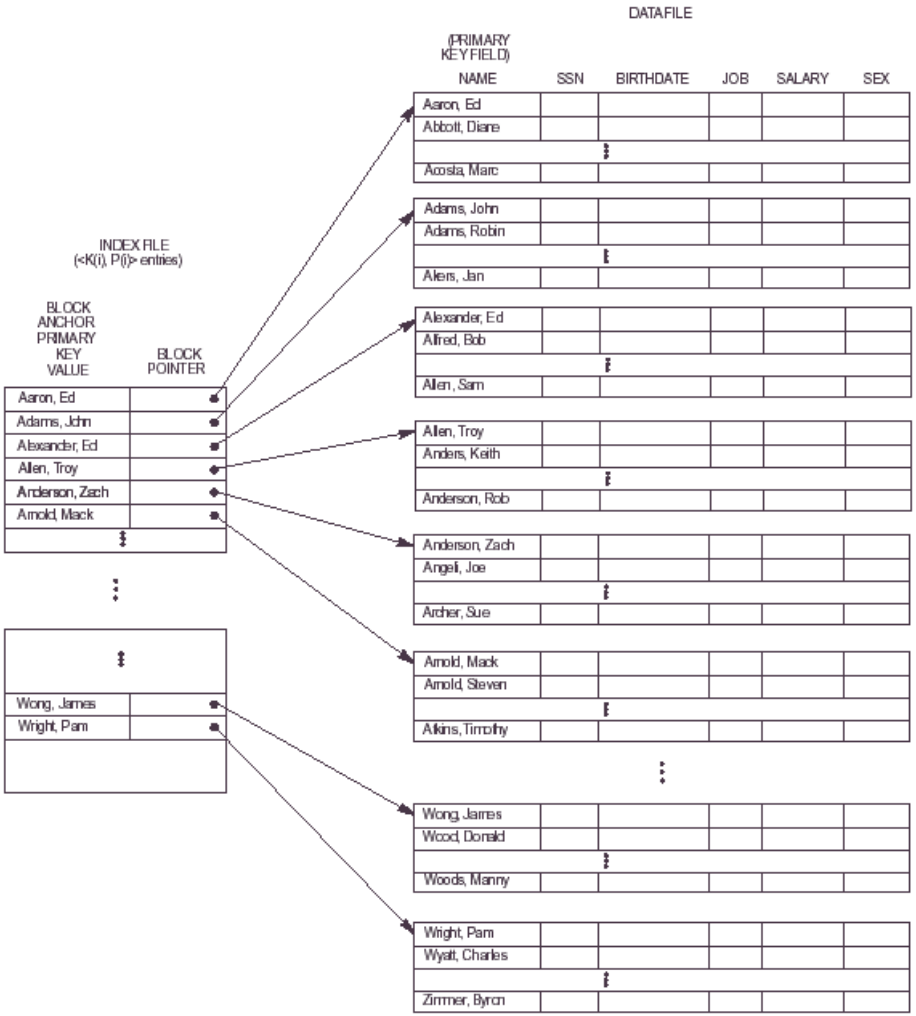
Indexing techniques evaluated on basis of:

- Access types supported efficiently. E.g.,
  - records with a specified value in an attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Primary index

- A primary index is a file that consists of records with two fields. The first field is of the same type as the ordering field (index field) for the data file and the second field is a pointer to a block (*block pointer*).
- A primary index has one index record for each block in the data file, and therefore is called a **sparse index** (or "**nondense index**")
- A **dense index** consists of one record for each record in the data file.
- The first record in each block is called the *anchor record* of the block. (see Elmasri/Navathe Fig 14.1)

Example - primary index (Fig 14.1)



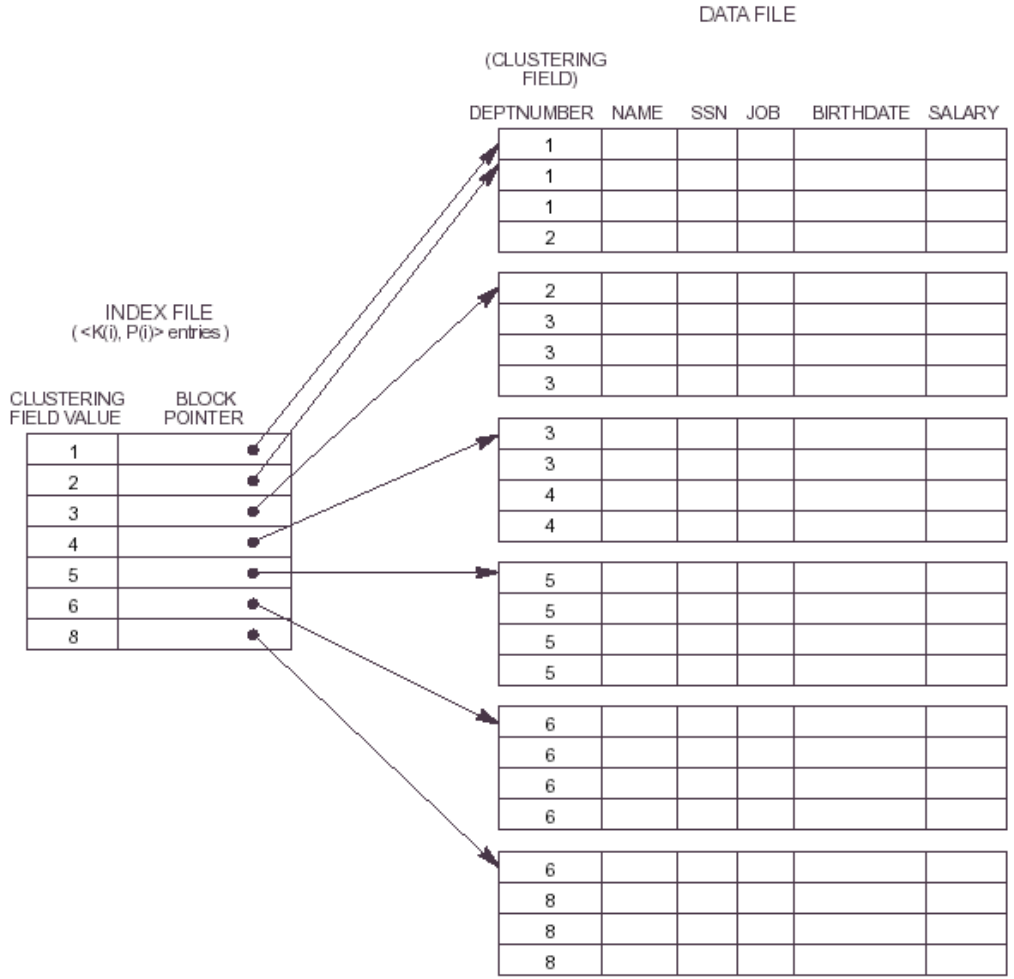
Primary index - pros and cons

- Require much less space than the data file.
  - a) There is much fewer index records than records in the data file.
  - b) Every index record need less space (⇒ fewer memory blocks).
- Problem with insertion and deletion of records.
  - If anchor records are changed the index file must be updated.

Cluster index

- Cluster index is defined for files that are ordered according to a non-key field (the *cluster field*), i.e. several records in the data file can have the same value for the cluster field.
- A cluster index is a file consisting of records with two fields. The first field is of the same type as the cluster field for the data file and the second field is a pointer to a block of records (*block pointer*) in the data file. (see Elmasri/Navathe Fig 14.2)
- Insertion and deletion of records is problematic. However, if each block only can contain records with the same cluster value the insertion problem is solved. (see Elmasri/Navathe Fig 14.3)

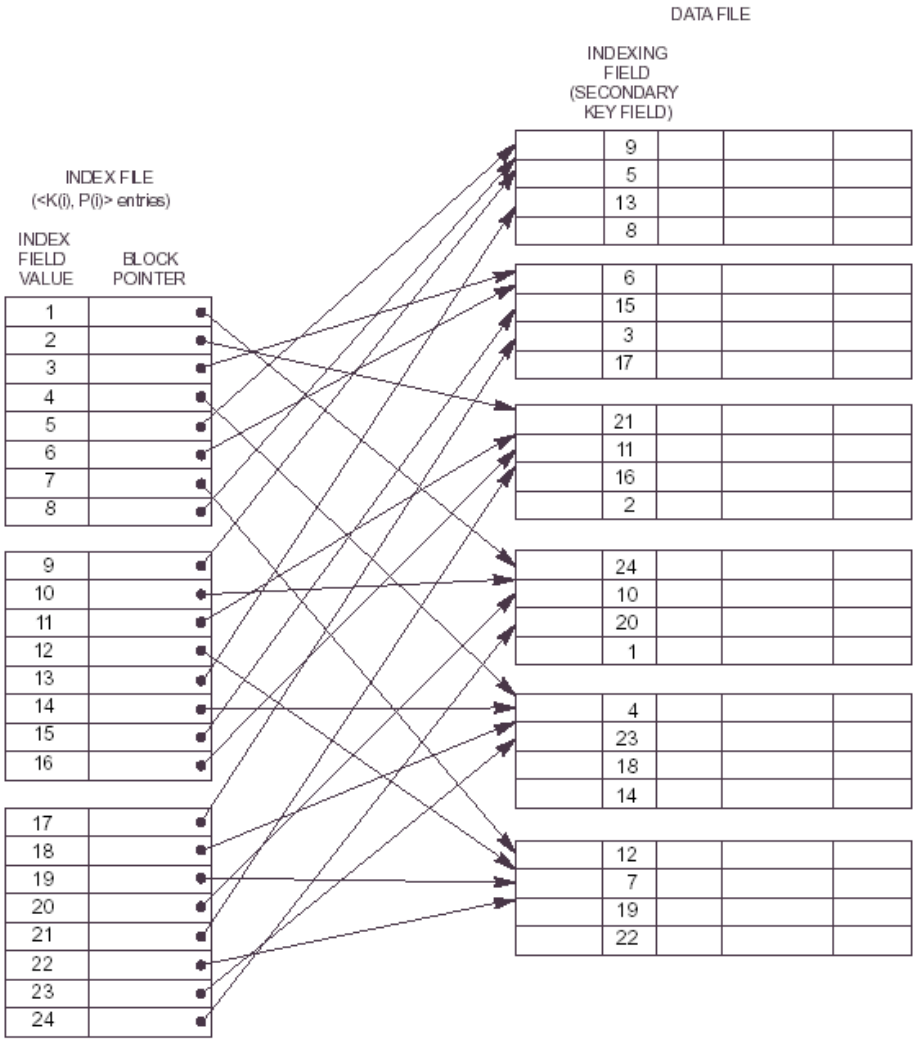
Example - cluster index (Fig 14.2)



Secondary index

- A secondary index is an ordered file that consists of records with two fields.
- The first field is of the same type as the indexing field (any field in the data file) and the second field is a block pointer.
- The data file is not sorted according to the *index field*.
- There are two different cases:
  1. The index field has unique values for each record (see Elmasri/Navathe Fig 14.4).
  2. Several records in the data file can have the same values for the index field.

Example - secondary index (Fig 14.4)



Secondary index ...

- Based on *non-key* fields.
- Several records in the data file can have the same values for the index field. How to implement the index file?
  a) Have several index records with the same value on the index field (dense index).
  b) Allow index records of varying size with multiple pointers. Each pointer gives the address to a block contaning a record with the same value for the index field.
  c) Let the pointer in the index record point to a block of pointers where each pointer gives the address to a record. (see Elmasri/Navathe Fig 14.5)

Comp. different indexes

	No. of index records (1st level)	Dense / sparse	Anchor block
primary	block in the data file	sparse	yes
cluster	unique index field values	sparse	yes/no*
secondary (key field)	records in the data file	dense	no
secondary (nonkey field option 1)**	records in the data file	dense	no
secondary (nonkey field option 2,3)**	unique index field values	sparse	no

* Yes, if every distinct index field begins with a new block, else no.

** Implementation dependent - see Elmasri/Navathe p. 111.

Primary and secondary indexes

- Indexes offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated. Updating indexes imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive (each record access may fetch a new block from disk.

Search trees

- A search tree of order p is a tree such that every node contain at most $p-1$ search values and p number of pointers as follows:
  $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$
- where $q \leq p$, P_i is a pointer to a child node (null if no childs) and K_i is a search value that is part in some ordered set of values..
- Search trees can be used to search for records in a file.
- Values in the search tree can be values of a specific field in the file (the search field).
- Each value in the tree is associated with a pointer that either points to the record in the data file that has the value for the search field or the data block that contains the record.

Requirements on search trees

- A search tree must always fulfil two conditions:

  1. For every node should hold that $K_1 < K_2 < ... < K_{q-1}$

  2. For all values X_i in a subtree identified by P_i the following should hold:

  - $K_{i-1} < X < K_i$ $(1<i<q)$
  - $X < K_i$ $(i=1)$ (see E/N Fig 14.8)
  - $K_{i-1} < X$ $(i=q)$

- Condition 2 is important for choosing the correct subtree when searching for a specific value X.

Problems with general search trees

- Insertion/deletion of data file records result in changes to the structure of the search tree. There are algorithms that guaranties that the search-tree conditions continue to hold also after modifications.
- After an update of a search tree one can get the following problems:
  1. inbalance in the search tree that usually result in slower search.
  2. empty nodes (after deletions)
- To avoid these types of problem one can use some type of so called B-trees (balanced trees) that fulfil stricter requirements than general serach trees.

B^+-tree index files

- B^+-tree indexes are an alternative to indexed-sequential files.
- Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B^+-trees: extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages, and they are used extensively.

B^+-Tree index files ...

A B^+-tree of order n is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length.
- Each node that is not a root or a leaf has between $\lceil n / 2 \rceil$ and n children.
- A leaf node has between $\lceil (n - 1) / 2 \rceil$ and n - 1 values.
- Special cases: if the root is not a leaf, it has at least 2 children.
- If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n - 1).

B^+-tree node structure

- A typical node

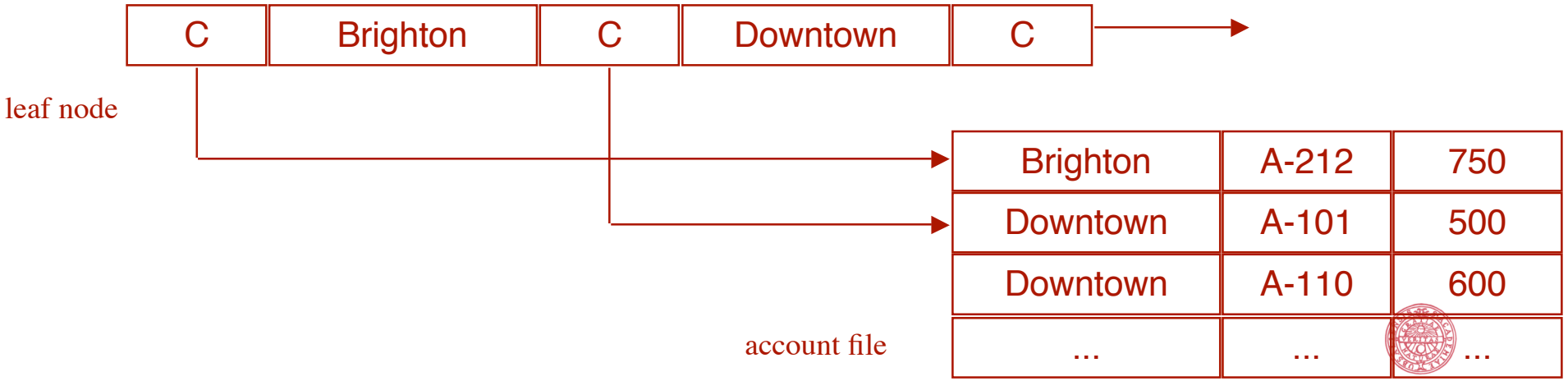
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

  $K_1 < K_2 < K_3 < ... < K_{n-1}$

Leaf nodes in B+-trees

Properties of a leaf node:

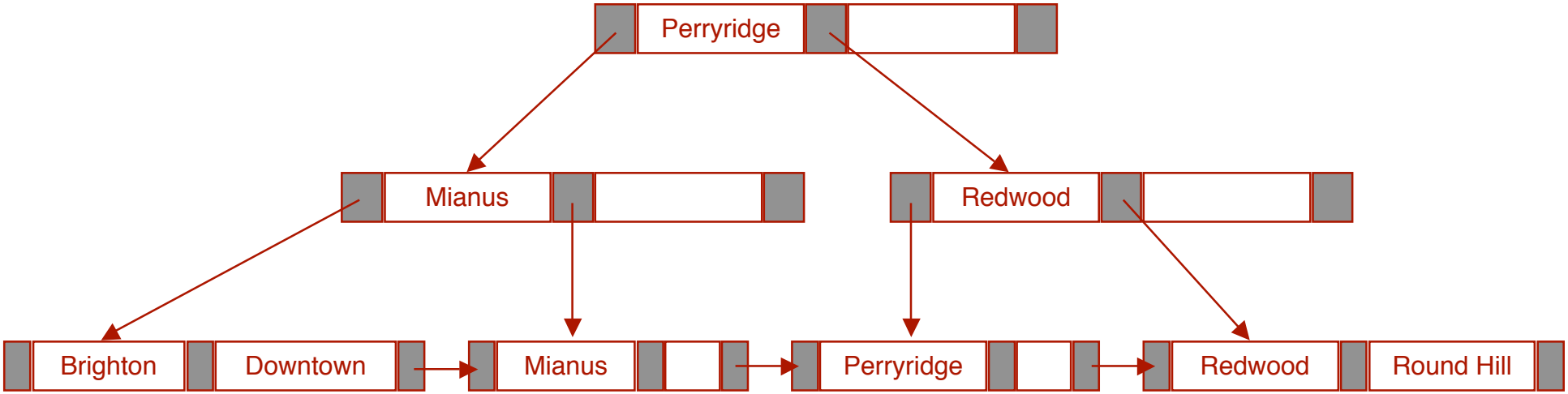
- For i = 1, 2, ..., n-1, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and i < j, L_i 's search-key values are less than L_j's search-key values
- P_n points to next leaf node in search-key order



Non-leaf nodes in B$^+$-trees

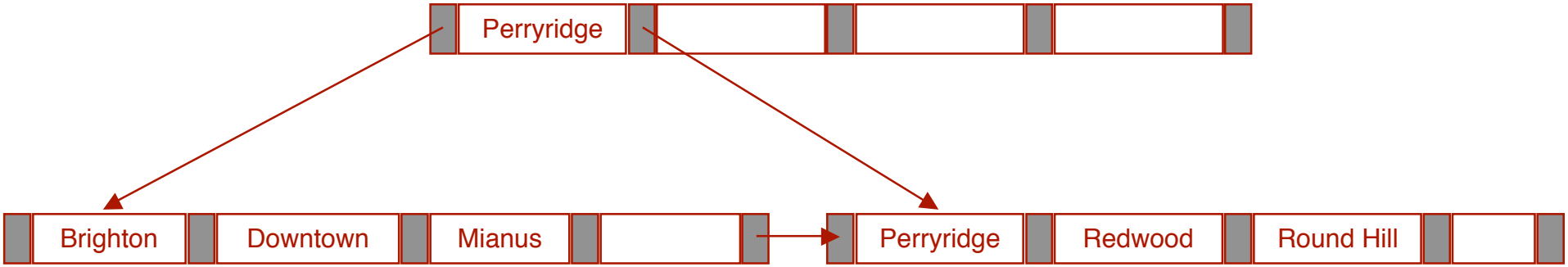
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n-1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than P_i
  - All the search-keys in the subtree to which P_m points are greater than or equal to K_{m-1}

Example of a B+-tree



- B^+-tree for *account* file (n = 3)

Example of a B+-tree



- B^+-tree for account file (n = 5)
- Leaf nodes must have between 2 and 4 values ($\lceil (n - 1) / 2 \rceil$ and n - 1, with n = 5).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil (n / 2 \rceil$ and n with n = 5).
- Root must have at least 2 children.

Observations about B$^+$-trees

- Since the inter-node connections are done by pointers, there is no assumption that in the B$^+$-tree, the "logically" close blocks are "physically" close.
- The non-leaf levels of the B$^+$-tree form a hierarchy of sparse indexes.
- The B$^+$-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

Queries on B$^+$-Trees

- Find all records with a search-key value of k.
  - Start with the root node.
  - Examine the node for the smallest search-key value $> k$.
  - If such a value exists, assume it is K_i. Then follow P_i to the child node.
  - Otherwise $k \geq K_{m-1}$, where there are m pointers in the node. Then follow P_m to the child node.
  - If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
  - Eventually reach a leaf node. If key $K_i = k$, follow pointer P_i to the desired record or bucket. Else no record with search-key value k exists.

Queries on B$^+$-Trees ...

- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes, and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and n = 100, at most $\log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup.
  - above difference is significant since every node access may need a disk I/O, costing around 30 millisecond!

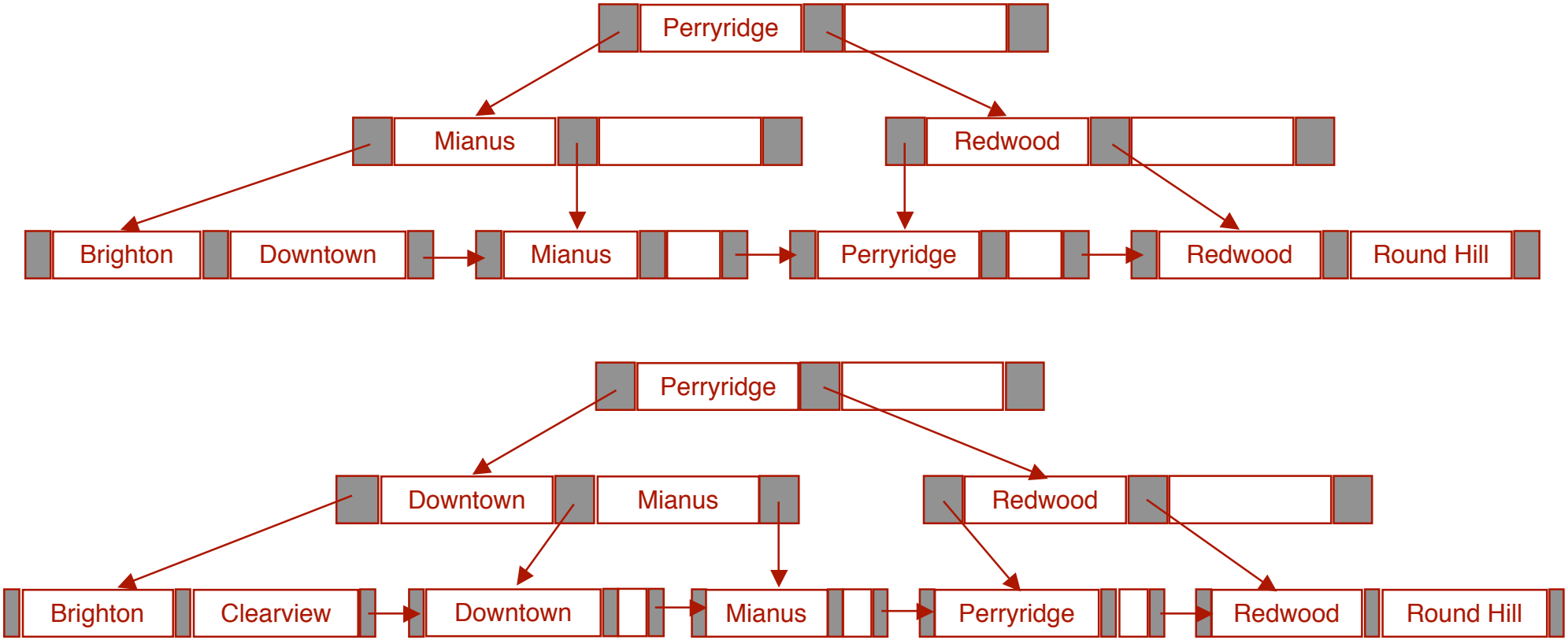
Updates on B+-Trees: insertion

- Find the leaf node in which the search-key value would appear.
- If the search-key value is already there in the leaf node, record is added to file and if necessary pointer is inserted into bucket.
- If the search-key value is not there, then add the record to the main file and create bucket if necessary. Then:
  - if there is room in the leaf node, insert (search-key value, record/bucket pointer) pair into leaf node at appropriate position.
  - if there is no room in the leaf node, split it and insert (search-key value, record/bucket pointer) pair as discussed in the next slide.

Updates on B$^+$-Trees: insertion ...

- Splitting a node:
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be p, and let k be the least key value in p.
- Insert (k,p) in the parent of the node being split. If the parent is full, split it and propagate the split further up.
- The splitting of nodes proceeds upwards till a node that is not full is found. In the worst case the root node may be split increasing the height of the tree by 1.

Updates on B+-Trees: insertion ...



- B+-Tree before and after insertion of “Clearview”

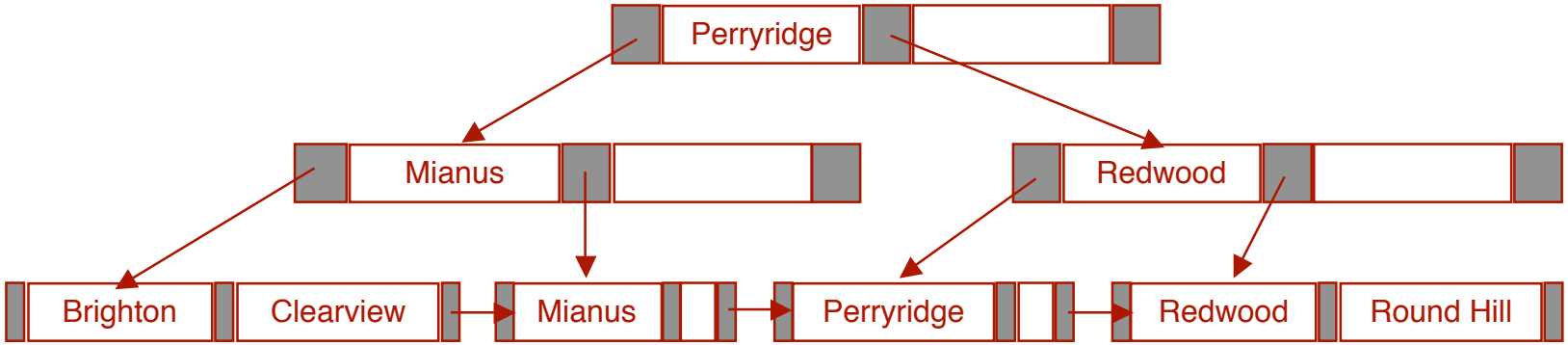
Updates on B$^+$-Trees: deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present).
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty.
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1} , P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

Updates on B+-Trees: deletion

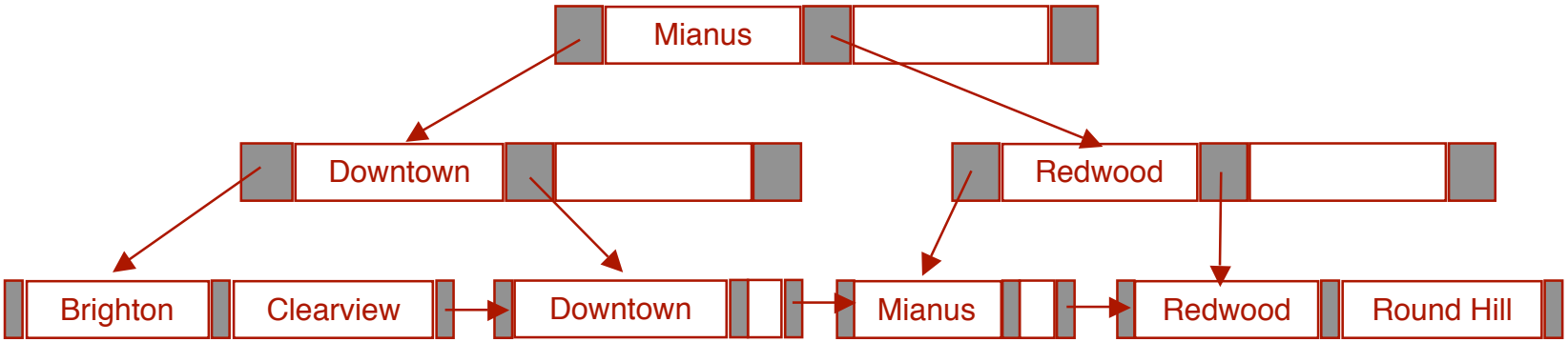
- Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling do not fit into a single node, then
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has ⌈ n/2⌉ or more pointers is found. If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Examples of B+-Tree deletion



- Result after deleting “Downtown” from account
- The removal of the leaf node containing “Downtown” did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node’s parent.

Examples of B+-Tree deletion ...



- Deletion of "Perryridge" instead of "Downtown"
- The deleted "Perryridge" node's parent became too small, but its sibling did not have space to accept one more pointer. So redistribution is performed. Observe that the root node's search-key value changes as a result.

B-Tree index files

- Similar to B^+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node:

- Nonleaf node in B-trees where extra pointers B_i are the bucket or file record pointers.

B-Tree index files ...

- Advantages of B-Tree indexes:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indexes:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.

Static hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B*.
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of hash file organization

- Hash file organization of *account* file, using *branch-name* as key
- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i*.
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Perryridge) = 5
    h(Round Hill) = 3
    h(Brighton) = 3

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Hash functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.

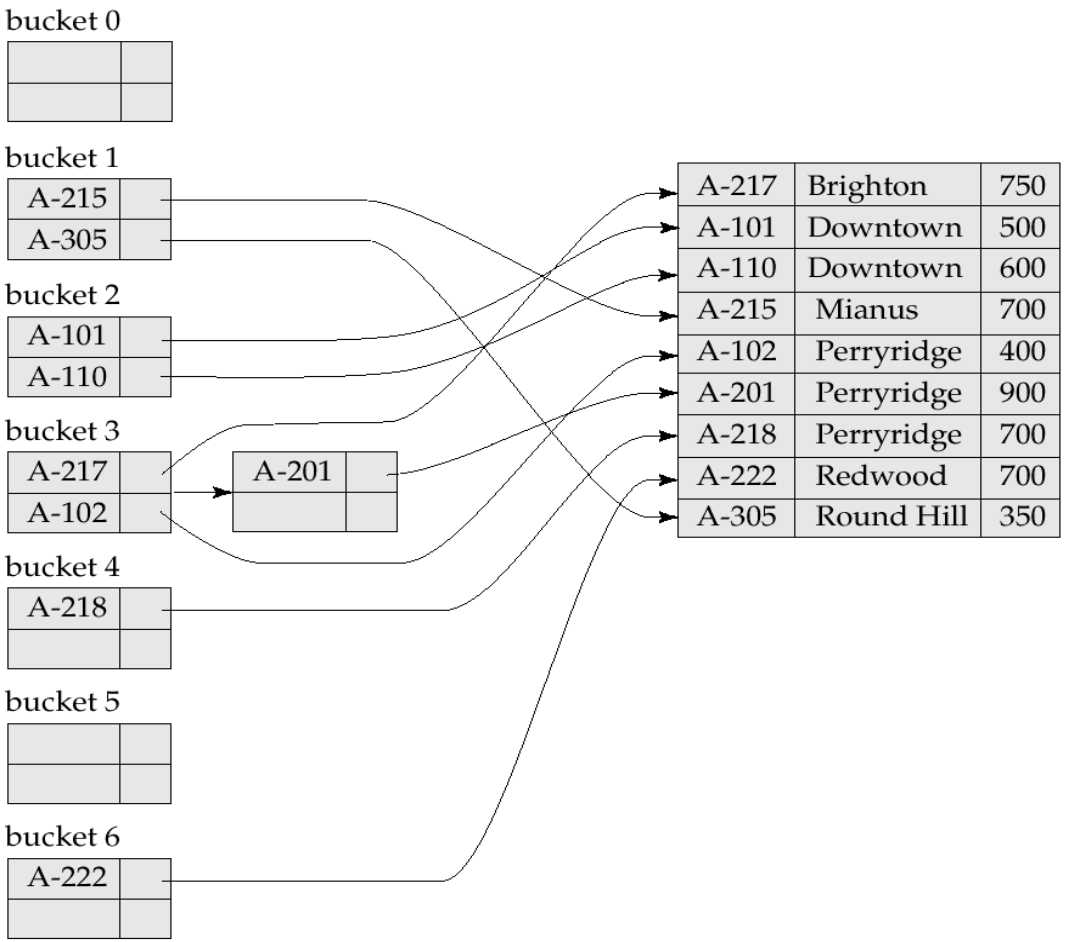
Handling of bucket overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets*.
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called *closed hashing*.
  - An alternative, called *open hashing*, which does not use overflow buckets, is not suitable for database applications.

Hash indexes

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Hash indexes are always secondary indexes
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - however, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of a hash index



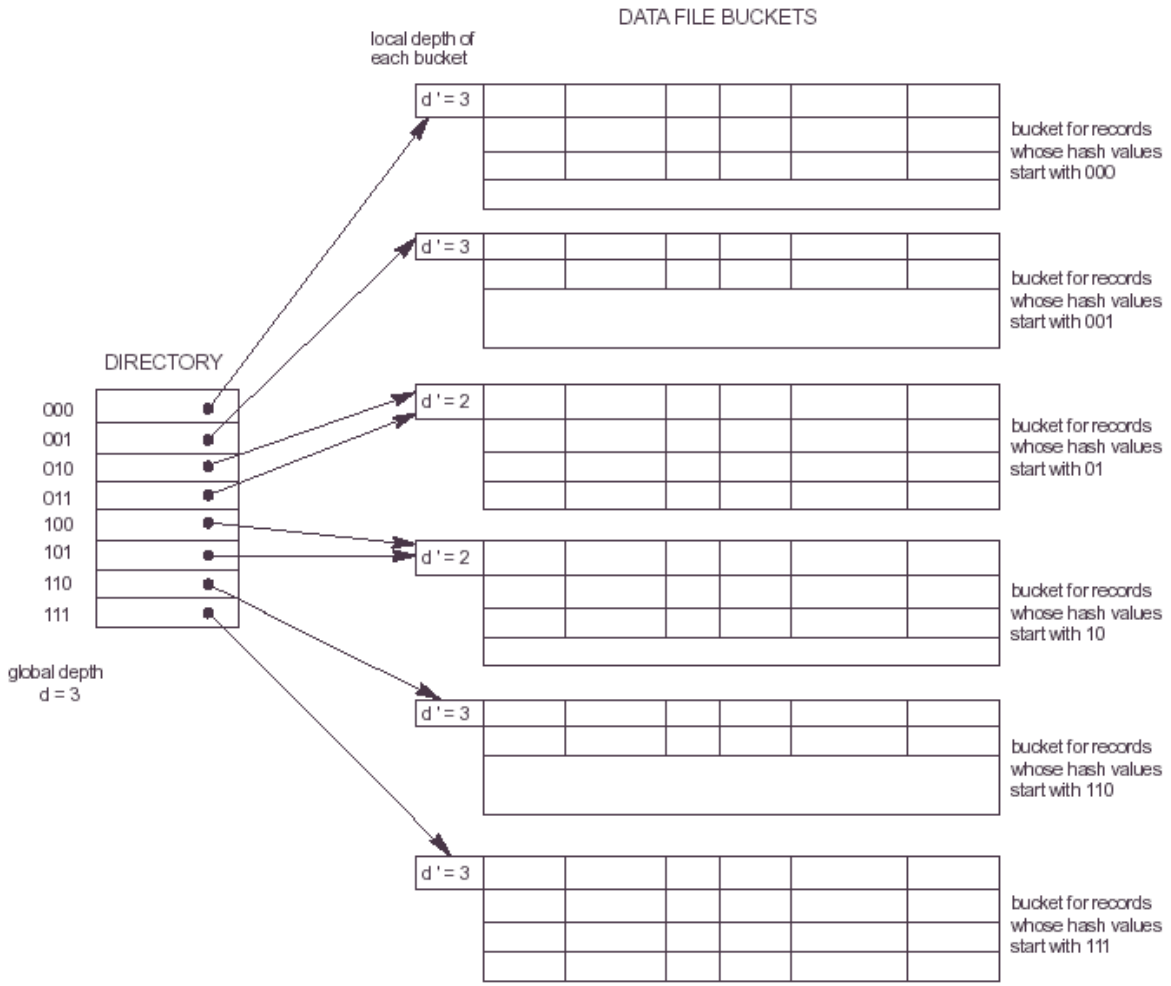
Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using *dynamic hashing* techniques that allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - A table with the size 2^d is used as **directory** or index where d is called the **global depth** of the directory.
  - The first d bits in the hash field is used to find the correct place in the directory and then the address in that position is used to find the right bucket.
  - Several directory element s with the same first d' bits in their hash values can contain the same bucket address (if all records fits in one single bucket).
  - The directory size is changed by increasing (or decreasing) d.
    - $d \rightarrow d+1$
    - $2^d \rightarrow 2^{d+1}$

Extendable hashing



Extendable hashing vs. other schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Need a tree structure to locate desired record in the structure!
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows

Linear hashing

- No directory is created
- One starts with M buckets [0,1,…,M-1],
  and the hash function h_0=k mod 2^0M,
  and the number of bucket divisions n = 0
- If the **file load factor** becomes to high a new bucket is created M+n and bucket number n is split up between bucket n and M+n via a new hash function h_1=k mod 2^1M
- Thus we can keep the file load factor, l, within wanted interval by letting it trig possible splits and combinations.
- l = r / bfr * N l = file load factor
  r = no of records
  N = no of buckets

UPPSALA UNIVERSITET

Comparison of ordered indexing and hashing

Issues to consider:

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred

Index definition in SQL

- Create an index
  - **create index** <index-name> **on** <relation-name> (<attribute-list>)
  - E.g.: **create index** *b-index* **on** *branch(branch-name)*
- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>