Advances in Data Management
Principles of Database Systems - 2
A.Poulovassilis

1 Storing data on disk

The traditional **storage hierarchy** for DBMSs is:

1. main memory (primary storage) for data currently being accessed
2. disk (secondary storage) for the entire database
3. tertiary storage (tapes, CDs) for older versions of the database

The data handled by DBMSs needs to persist on disk between programme executions.

Data on disk is stored and retrieved in units called **blocks** or **pages** (of typical size 1K - 10K).

Transfers between disk and main memory occur in multiples of pages. The cost of such transfers is the cost of a **page access** or **I/O operation** i.e. the reading/writing of a single page.

Logical records, e.g. rows in relational databases or objects in OO databases, are implemented as **physical records** — called just records from now on.

A (physical) record is a sequence of one or more data values, $d_1, d_2, \ldots d_n$, termed **fields**.

A sequence of field names and their types, $f_1 : type_1, f_2 : type_2, \ldots f_n : type_n$ is termed a **record format**.

A record conforms to a record format if it has the same number of items, n, and if each data value d_i represents an instance of field name f_i and is of type $type_i$.

A **file** is a set of records all conforming to the same record format, stored in one or more blocks.

A record is uniquely identified by a pair (b,r) where b identifies the block containing the record and r identifies the position of the record within the block. Such a pair is called a **pointer to a record**.

An **index** consists of extra information that provides a fast access path to portions of a file, based on some **search key**.

A **clustered index** stores the actual data records as part of the index structure.

An **unclustered index** stores pointers to records (rather then the records themselves) within the index structure.

At most one index on a file will be a clustered one (unless records are duplicated).

A **primary index** uses the primary key field(s) of the records in the file as the search key and locates a record given the value of its primary key.

A **secondary index** uses some other set of record fields and locates the set of records with a given secondary key value.

Indexes should be chosen so as to maximise database performance for the expected **workload** i.e. the expected types and frequencies of updates and queries. This is part of **database tuning**[1].

2 B-trees

A B-tree is a balanced tree index.

The tree is visualised up-side down, with a single root node at the top and branching out towards the bottom.

Each node in the tree contains a sequence of keys and pointers $p_0, k_1, p_1, k_2, p_2, \ldots p_{n-1}, k_n, p_n$ (where the ks are keys and the ps are pointers).

The keys within each node, and within the tree overall, are ordered according to the numerical or lexicographical ordering of the underlying domain to which the key values belong.

The B-tree is a generalisation of the **binary tree** in that more than two pointers can originate from each node.

A **B-tree of order** d contains in each node at most

$$2d \text{ keys and } 2d+1 \text{ pointers}$$

and (apart from possibly the root node) at least

$$d \text{ keys and } d+1 \text{ pointers}$$

Figure 1 illustrates a B-tree node for a B-tree. The order d of this B-tree must satisfy $2 \leq d \leq 4$ (why ?).

Note that the storage utilisation of each node (apart from perhaps the root node) is always better than 50%.

Insertions/deletions leave a B-tree balanced, with a path length of about $log_d n$ from root to leaves, where n is the number of record keys.

3 B+ trees

A B+ tree is a variant of a B-tree, and is a standard organisation for indexes in DBMSs.

In a B+ tree, the keys of the non-leaf nodes are **search keys** while the keys of the leaf nodes are actual **record keys**.

Leaf nodes are linked together by pointers left-to-right for efficient sequential access of record keys.

Optionally, there are also right-to-left pointers between the leaf nodes to support efficient traversal of records in descending order of their key values.

To find a record key, we now need to search all the way to a leaf node, even if that value is found in an inner node of the tree (since this will only be a search key).

[1] Throughout these notes I am assuming *dense* indexes i.e. there is a key value in the index for every key value in the data records. It is also possible to have *sparse* indexes which store only a selection of the key values from the data records.

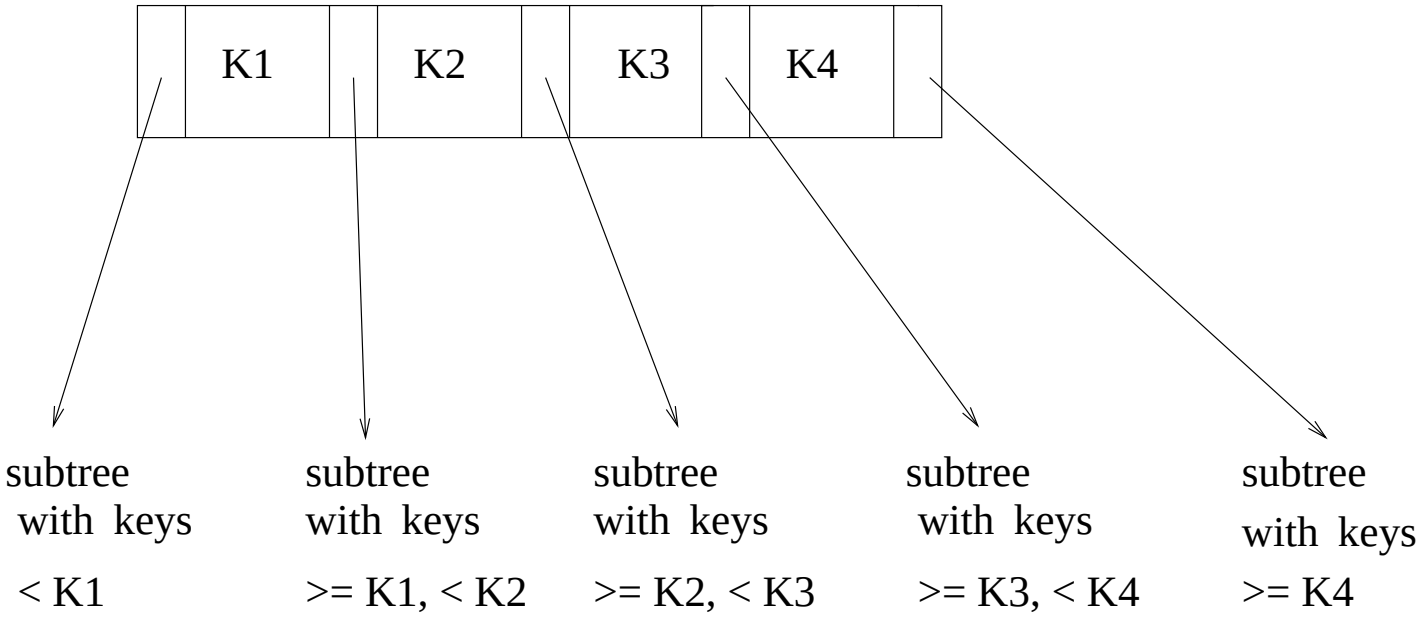

Figure 1: A B-tree node

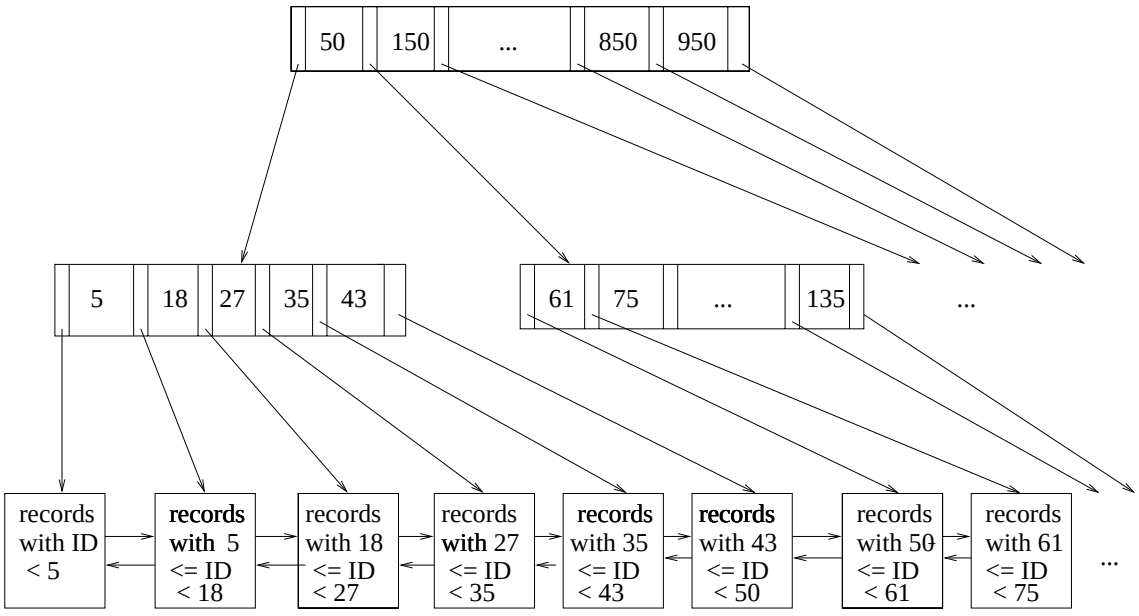

Figure 2: An example B+tree

Insertions/deletions again leave a B+ tree balanced i.e. all leaf nodes are always at the same depth of about $log_d n$, where n is the number of record keys.

For example, Figure 2 shows part of a B+ tree index on the ID attribute of the Employee table.

Insertion of a record key

(i) Find the appropriate leaf node, N, for the new key and slot the key into the appropriate position.

(ii) If N is overfull, **split** its $2d + 1$ keys into:

- the lowest d keys, which remain in N
- the highest $d + 1$ keys, which are transferred to a new sibling node N^{new} of N

Add a copy of the lowest key in N^{new} to their parent node.

(iii) If the parent node P is now overfull, **split** its $2d + 1$ keys into:

- the lowest d keys, which remain in P
- the middle key, which is transferred to the parent node
- the highest d keys, which are transferred to a new sibling node P^{new} of P

Repeat step (iii) for P's parent node.

Deletion of a record key

(i) Search for the leaf node N containing the key. If it is not present in the leaves, do nothing; otherwise delete the key.

(ii) If N now has less than d keys left:

- **redistribute** the keys of N and those of one of its siblings if the total no. of keys they contain $\geq 2d$, replacing their separator key in their parent node by the lowest key in the right sibling; otherwise
- **concatenate** the keys of N with those of one of its siblings and delete the separator key from their parent node.

(iii) If the parent node P now has less than d keys left:

- **redistribute** the keys of P and those of one of its siblings if the total no. of keys they contain $\geq 2d$, moving the separator key from their parent node into P and moving the adjacent key from P's sibling into the vacated slot in the parent; otherwise
- **concatenate** the keys of P with those of one of its siblings, deleting the separator key from their parent node, and moving it into the new merged node.

Repeat step (iii) for P's parent node.

B+ tree indexes may be either clustered (whole data records are stored in the leaves) or unclustered (record keys and pointers to records are stored in the leaves). For example, the index illustrated is a clustered index.

The search key for a B+ tree may be a **composite key**, in which case the index can be used to support efficiently an equality or a range search on any **prefix** of the composite key.

For example, for the table
Enrolled(sid: VARCHAR(10), cid: CHAR(6), examMark:NUMERIC(3))
if there is a B+ tree index on (cid,sid), then this index can be used to support efficiently a query that specifies a cid value or range of values; but it cannot be used to support efficiently a query that specifies a sid value or range of values;

The minimum and average storage utilisation of B+ trees is 50% and 66%, respectively.

B+ trees in main-memory DBs vs disk-resident DBs:

In conventional disk-resident DBs, each B+ tree node is stored as a block on disk, and the pointers from parent to child nodes reference the child block identifiers on disk.

However, B+ trees can also be used as a main-memory data structure, in which case the pointers from parent to child nodes nodes refer to the child node's address in memory (as do pointers to individual records).

4 Multi-media indexes

Multimedia databases extend conventional relational or object-based databases with facilities for storing and manipulating more complex data, such as spatial, temporal, text, image, audio and video data.

Such data typically need to be indexed and accessed using a combination of "dimensions" e.g.

- spatial attributes for spatial data (e.g. for representing lines, sequences of line segments, polygons, polyhedra, wireframes);
- temporal attributes for temporal data;
- occurrences of keywords from a keywords list within text data;
- content features such as colour, texture, location, scene and object information for image data;
- spatio-temporal attributes and content features for video and audio data.

Applications requiring the storage, retrieval and manipulation of large volumes of multimedia data include: temporal database applications that need to handle past as well as current information, geographic information systems, digital libraries, medical databases, and Computer Aided Design (CAD) systems.

Using conventional unidimensional indexing schemes such as B trees and hashing is insufficient for such applications and new indexing methods have been developed.

Multi-dimensional indexing schemes allow fast retrieval of multimedia data by multidimensional range search or point search queries, e.g. indexing schemes such as k-d trees, quad trees, R-trees.

R-trees (or variants such as R* and R+ trees) are commonly used in DBMSs, so we look at these here.

4.1 R-trees

An R-tree is a height-balanced tree similar to a B-tree, with index records in its leaf nodes containing pointers to data objects. Each node of the tree is stored on one disk page. The index is designed so that a spatial search requires visiting only a small number of nodes.

R-trees can be used to efficiently support queries such as: find all objects that contain a given multidimensional point; find the n nearest objects to a given point; find all objects that intersect with/contain/are contained by a multi-dimensional search box.

Leaf nodes in an R-tree contain records of the form

$$(I, object\_identifier)$$

where $object\_identifier$ is a pointer to a spatial object stored elsewhere on disk and I is an n-dimensional rectangle which is a 'bounding box' for this spatial object i.e.

$$I = (I_1, \ldots, I_n)$$

where each I_j is an interval in the j^{th} dimension. Moreover, I is the *smallest* n-dimensional rectangle that spatially contains the n-dimensional object represented by this record.

Non-leaf nodes contain entries of the form

$$(I, child\_pointer)$$

where $child\_pointer$ is the address of a node at the next level down in the R-tree and the n-dimensional rectangle I spatially contains all the rectangles within this child node's entries. Moreover, I is the *smallest* n-dimensional rectangle that spatially contains all the rectangles in the child node.

Let m / M be the minimum / maximum number of entries allowed in a node (these numbers are chosen according to physical page sizes). Then, an R-tree also satisfies the following properties:

- Every leaf node contains between m and M index records, unless it is the root (which may have less than m records).
- Every non-leaf node has between m and M children unless it is the root (which may have less than m children)
- All leaves appear on the same level i.e. have the same number of ancestor
- The greatest height of the tree is $O(log_m N)$ where N is the number of index records.
- The worst-case space utilisation of all nodes (except the root) is m/M.

  nodes.

R-tree indexes can be used to support both point-based and region-based queries, as described in the **Search** algorithm below.

In the following, we denote the n-dimensional rectangle part of an index entry E by $E.I$ and the object-identifier or child-pointer part by $E.p$:

Search Algorithm. Given an R-tree whose root is T, this finds all index records whose rectangles overlap a *search rectangle* S (S might just be a single point in n-dimensional space):

1: [Search subtrees] If T is not a leaf, check each entry E in T to determine whether $E.I$ overlaps S. For all overlapping entries, invoke **Search** on the subtree whose root node is pointed to by $E.p$.

N.B. Note this difference compared to a B+ tree: a search can lead down several paths in an R-tree whereas only one path is ever followed in a B+ tree search.

2: [Search leaf node] If T is a leaf, check all its records E to determine whether $E.I$ overlaps S. If so, E is a qualifying record. The actual spatial object is then retrieved and inspected to see if it indeed overlaps S.

4.2 R-trees: Insert and Delete Operations (optional)

To insert a new index record (I, o), start at the root and traverse a single path to a leaf (in contrast to searching). At each level, pick the child node whose bounding box needs the least enlargement (in terms of volume) to accomodate I. In the case of a tie, pick the child node with the smallest bounding box (in terms of volume).

At the leaf level, (I, o) is inserted, enlarging if necessary the bounding box of the leaf's parent. If such an enlargement happens at a node, then this must be propagated upwards to its parent.

If the leaf is at maximum occupancy (M), then it must be split into two, and the existing entries plus (I, o) need to be redistributed between the two siblings. The bounding boxes for the two siblings are computed (the redistribution aims to minimise the overall volume of the two bounding boxes), and inserted into their parent node. This increases by 1 the number of entries in the parent; so splits and readjustments of bounding boxes may propagate upwards up the tree (in the worst case, up to the root itself, resulting in the creation of a new root and an increase by 1 in the height of the tree).

Deletion of index records begins by a Search, potentially traversing several paths in order to find the record(s) to be deleted. When an index record is removed from a leaf node this may require the bounding box in the parent node to be adjusted; and such adjustments may potentially propagate upwards up the tree. If a leaf node becomes underfull, then redistribution of its index records with a sibling node, or merging with a sibling node, is required (c.f. B+ tree deletion). Alternatively, an underfull page can be removed altogether and all its entries be reinserted into the tree.

For full details (optional reading) see *R-Trees: A Dynamic Index Structure for Spatial Searching*, A.Guttman, ACM SIGMOD, 1984.

For more discussion (optional reading) see : Chapter 28 of Ramakrishnan and Gehrke; Chapter 25 of Sibserschatz, Korth and Sudarshan.

5 Hash Files

Stating Hashing

This divides the main file into a number of **buckets**. A bucket consists of a single page or a linked list of pages.

A **hash table** of fixed size, n, is maintained. Each entry in this table is a pointer to a bucket in the main file.

A **hash function**, h, is used to map record keys to entries in the hash table i.e.

$$h : DOM(key) \rightarrow \{0, 1, \ldots, n-1\}$$

Records are stored in the bucket that the hash table entry points to.

h should randomise the assignment of keys over the entire address space $0 \ldots n-1$.

A possible h is

$$h(k) = k \ mod \ n$$

e.g. for the ID attribute of the Employees table above, we might have the following hash function h which partitions the Employees table into 50 buckets using their ID value:

$$\begin{aligned} h &: \{0, \ldots, 999\} \rightarrow \{0, \ldots, 49\} \\ h(ID) &= k \ \ modulo \ \ 50 \end{aligned}$$

This hash function works well for near-uniform distributions of key values. More sophisticated hash functions break down the input key value into pieces, apply a different function to each piece, and then combine the results to give the final output value.

If more than one record maps to the same bucket, this is termed a **collision**. Generally, many records will fit into a bucket so such collisions will not be a problem.

However, if a new record causes a bucket to **overflow** this must be dealt with e.g. by puting the record into a new **overflow bucket** and chaining this to the main bucket.

There is serious drawback with static hashing in that the size of the hash table n must be estimated before hand:

- If too many buckets are allocated, there is wastage of space.
- If too few buckets are allocated, bucket overflows occur, overflow chains grow longer, and retrievals become costlier.

  Eventually, a file **reorganisation** is required with a new hash function and a larger address space.

These problems have led to the delopment of dynamic hashing schemes — see below.

Note it is also possible to use a hash index as an unclustered index. In this case, the buckets consist of pointers to the data records stored in the main file, rather than the records themselves.

Dynamic Hashing

This alleviates the drawback of static hashing by allowing the number of buckets and the hashing scheme to vary **dynamically** as the file grows/shrinks.

We will look here at one approach to dynamic hashing, called **linear hashing**.

Linear hashing uses a series of different hash functions as the number of buckets expands:

$$\begin{aligned} h_1 &: DOM(key) \rightarrow \{0,1\} \\ h_2 &: DOM(key) \rightarrow \{0,1,2,3\} \\ &\dots \\ h_d &: DOM(key) \rightarrow \{0,\dots,2^d-1\} \end{aligned}$$

These successive hash functions satisfy the following property for any k and $i > 1$:

$$h_i(k) = h_{i-1}(k) \quad or \quad h_i(k) = h_{i-1}(k) + 2^{i-1}$$

For example, given some hash function

$$H : DOM(key) \rightarrow \{0, ..., 2^m - 1\}$$

for some large m, we can obtain a family of hash functions h_1, h_2, ... that satisfy the above property by defining $h_i(k)$ as returning the last (least significant) i bits of $H(k)$.

With linear hashing, the main file is stored contiguously as a sequence of pages on disk.

The h_i map directly to page addresses i.e. to relative offsets from the start of the file.

There is a gradual transition from one using one hash function to using the next one in the sequence, as the file expands. In particular, suppose there are currently 2^d pages in the file and we are currently using h_d (for some $d \geq 1$).

If the i^{th} page now overflows as a result of a new record being inserted, that record is put into an overflow page which is chained to the i^{th} page.

In addition, a new page is appended to the file which is the **buddy** of the 1^{st} page i.e. it is the $(2^d + 1)^{th}$ page. The records currently in the 1^{st} page are redistributed into the 1^{st} and $(2^d + 1)^{th}$ pages according to h_{d+1}.

The next time that an overflow page is created, the $(2^d + 2)^{th}$ page is appended to the file which is the buddy of the 2^{nd} page, and the records currently in the 2^{nd} page are redistributed into the 2^{nd} and $(2^d + 2)^{th}$ pages according to h_{d+1}.

Eventually, the $(2^d + i)^{th}$ page will be appended to the file, which is the buddy of the i^{th} page (that had overflowed earlier). The records in the i^{th} page and in its overflow page(s) will be reallocated to the i^{th} and $(2^d + i)^{th}$ pages according to h_{d+1}.

The current and next hash functions h_d and h_{d+1} continue to be both used until the file grows to 2×2^d pages. At that point, the current hash function then becomes h_{d+1} and the expansion process repeats from the start of the file.

Deletion is the reverse of this: Suppose the current hash function being used is h_{d+1} When the last page in the file is empty it is deleted, and a combination of the hash functions h_d and h_{d+1} is now

used. If all the pages after the 2^dth page have been deleted, the file consists of 2^d pages and the current hash function becomes h_d.

Linear Hashing vs. B trees

Cost of lookup is $O(1)$ with the former and $O(log_d n)$ with the latter. So hash-based indexes are better for equality-based searches.

However hashing does not support the "next" operation whereas B+ trees do. So these are better for range searches.

Homework 2

1. Consider a B+ tree of order 2 which consists of a root node containing keys 40,50,60,70 and five leaf nodes:
   32,34,36,38 42,44,46,48 52,54 62,64 72,74

   Trace what would happen to this B+ tree during the following series of operations: insert 39, insert 49, delete 60, delete 54, delete 42, delete 39.

2. Suppose we have a hash file organised according to the linear hashing scheme. Suppose the file currently consists of 4 pages i.e. we are currently using the hash function h2. Suppose also that every page of this file is currently full. Describe what would happen to the file after each of the following insertions:

   insert record r1, assuming that h2(r1)=1, h3(r1)=5
   insert record r2, assuming that h2(r2)=2, h3(r2)=6
   insert record r3, assuming that h2(r3)=2, h3(r3)=2
   insert record r4, assuming that h2(r4)=0, h3(r4)=4
   insert record r5, assuming that h2(r5)=3, h3(r5)=7

3. Suppose we have a hash file organised according to the linear hashing scheme. Suppose the file currently consists of 8 pages i.e. we are currently using the hash function h3. Suppose also that every page of this file is currently full. Describe what would happen to the file after each of the following insertions:

   insert record r1, assuming that h3(r1)=3, h4(r1)=3
   insert record r2, assuming that h3(r2)=1, h4(r2)=9
   insert record r3, assuming that h3(r3)=0, h4(r3)=8
   insert record r4, assuming that h3(r4)=3, h4(r4)=11