Applied Databases

Handout 3. Storage and Indexing.

27 Oct 2010

Storage and Indexing

Reading: R&G Chapters 8, 9 & 10.1

We typically store data in external (secondary) storage. Why? Becuase:

- Secondary storage is cheaper. £100 buys you 1gb of RAM or 100gb of disk (2003 figures!)
- Secondary storage is more stable. It survives power cuts and – with care – system crashes.

[Second point may be contentious – I've seen more disk failures than power cuts!]

Differences between disk and main memory

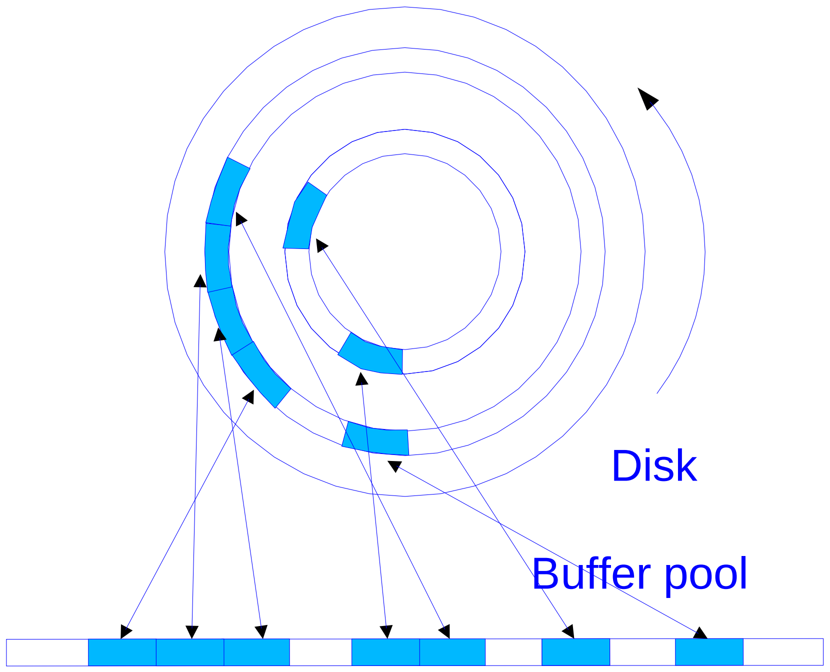
- Smallest retrievable chunk of data: memory $=$ 1 byte, disk $=$ 1 page $=$ 1kbyte (more or less)
- Access time (time to dereference a pointer): memory $< 10^{-8}$ sec, disk $> 10^{-2}$ sec.

However *sequential data*, i.e. data on sequential pages, can be retrieved rapidly from disk.

Communication between disk and main memory

A *buffer pool* keeps images of disk pages in main memory *cache*.

Also needed a table that *maps* between positions on the disk and positions in the cache (frames) (in both directions)



When a page is requested

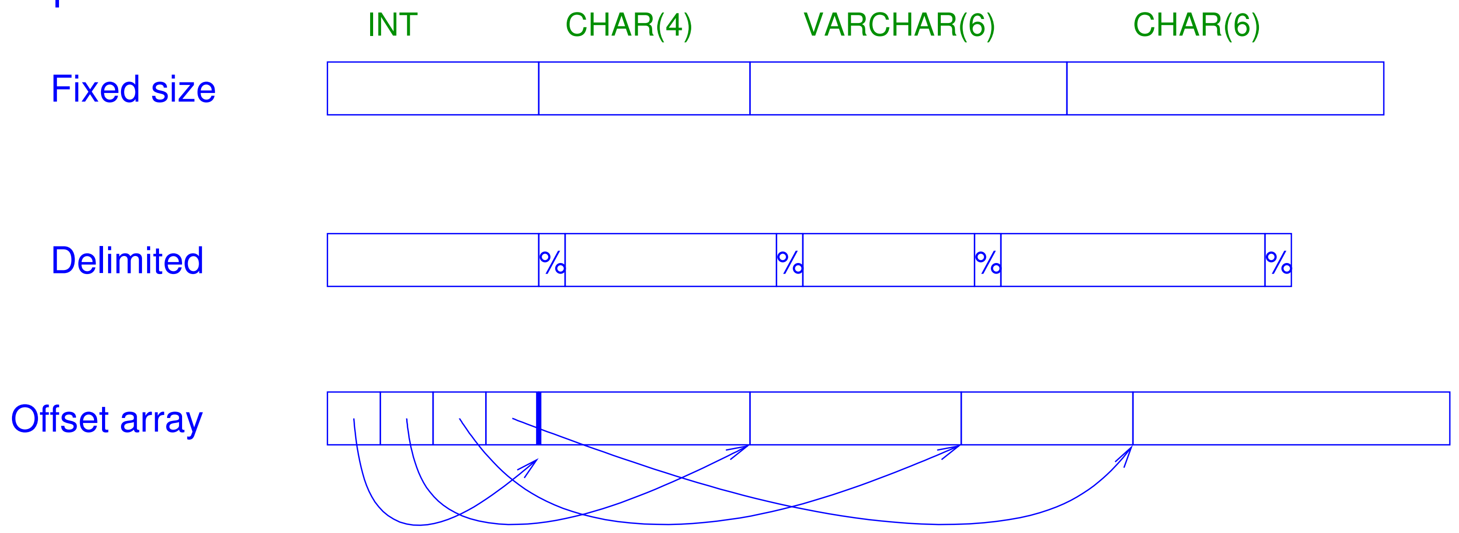
- If page is already in pool present, return address.
- If there is room in the pool, read page in and return address.
- If no room, choose a frame for replacement.
  - if current frame is *dirty* – it has been written to – write frame to disk.
- read page in and return address.

Requesting process may *pin* page. Indicating that it “owns” it.

Page replacement policy: LRU, MRU, random, etc. Pathological examples defeat MRU and LRU.

Storing tuples

Tuples are traditionally stored *contiguoulsy* on disk. Three possible formats (at least) for storing tuples.



Comments on storing tuples

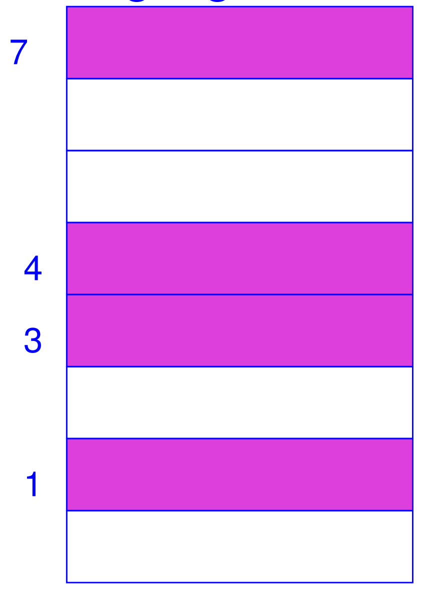
Fixed format appears more efficient. We can “compile in” the offsets. But remember that DB processing is *dominated by i/o*

Delimited can make use of variable length fields (VARCHAR) and simple compression (e.g. deleting trailing blanks)

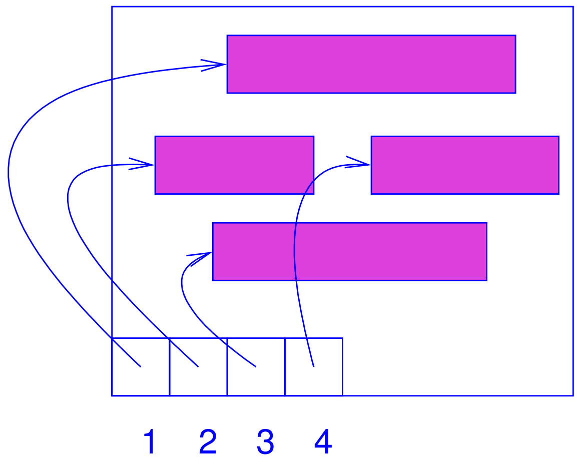
Fixed and delimited formats require *extra* space to represent null values. We get them for free (almost) in the offset array representation.

Placing Records on a Page

We typically want to keep "pointers" or *object identifiers* for tuples. We need them if we are going to build indexes, and we'd like them to be *persistent.*



Array of tuples



Array of pointers

Comments on page layouts

Array of tuples suitable for fixed length records.

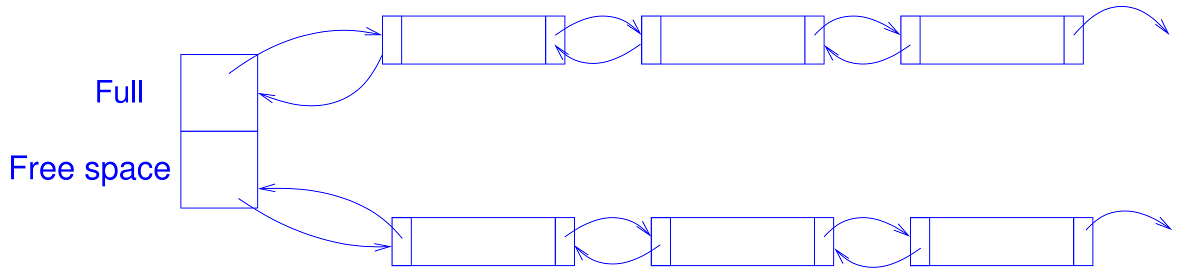
- Object identifier is (page-identifier, index) pair.
- Cannot make use of space economy of variable-length record.

Pointer array is suitable for variable length records

- Object identifier is (page-identifier, pointer-index) pair.
- Can capitalize on variable length records.
- Records may be moved on a page to make way for new (or expanded) records.

File organization – unordered data

Keep two lists: pages with room and pages with no room.



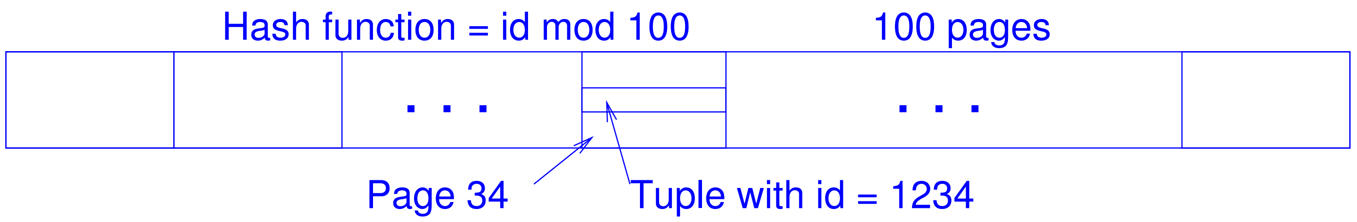
Variations:

- Keep an *array* of pointers.
- Order by amount of free space (for variable length tuples)

These are called *heap* files.

Other organizations

- Sorted files. Records are kept in order of some attribute (e.g. `Id`). Records are assumed to be fixed-length and "packed" onto pages.
  That is, the file can be treated as an array of records.
- Hashed files. Records are kept in an array of pages indexed by some hash function applied to the attribute. Naive example:



I/O Costs

We are primarily interested in the I/O (number of page reads and writes) needed to perform various operations. Assume B pages and that read or write time is D

	Scan	Eq. Search	Range Search	Insert	Delete
Heap	BD	$0.5BD$	BD	2D	*Search* $+D$
Sorted	BD	$D\log_2 B$	$D\log_2 B + m^*$	*Search* $+BD$	*Search* $+BD$
Hashed	$1.25BD$	D	$1.25BD$	$2D$	*Search* $+D$

* m = number of matches
Assumes 80% occupancy of hashed file

Indexing – Introduction

Index is a collection of *data entries* with efficient methods to locate, insert and delete data entries.

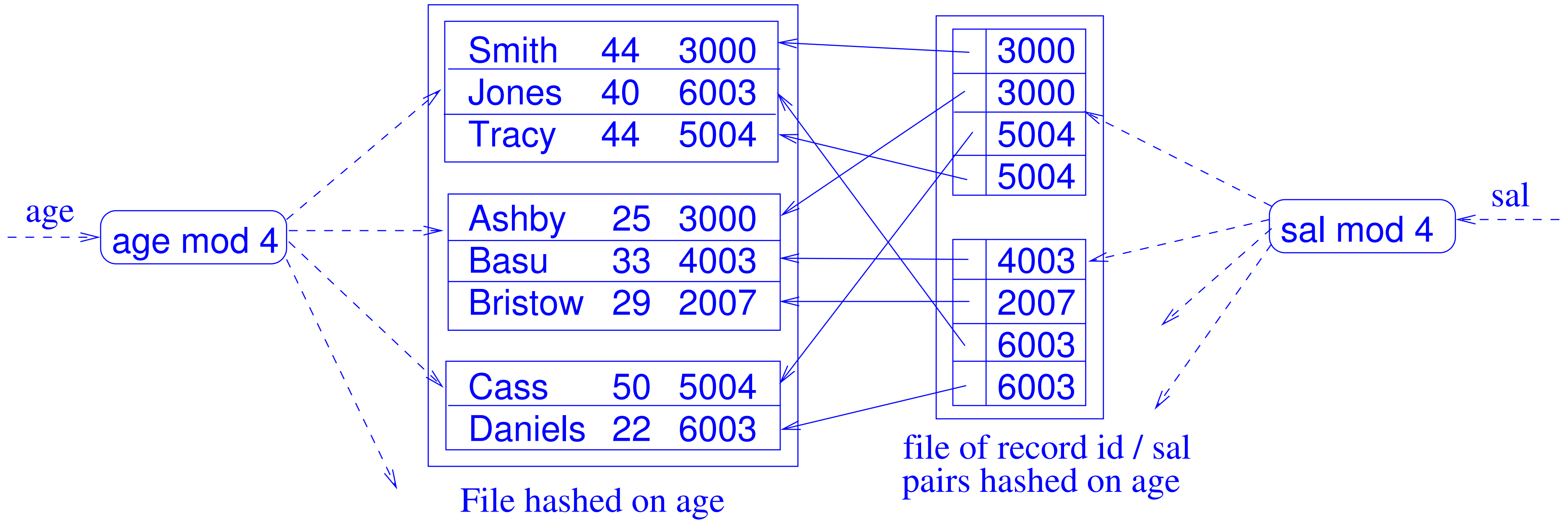
Hashed files and sorted files are simple examples of indexing methods, but they don't do all of these efficiently.

We index on some *key*.

Note the index key is not (necessarily) the "key" in the database design sense of the term.

We can only organize a data file by one key, but we may want indexes on more than one key.

Example. Hash indexes and files



Indexes are needed for optimization

How are these queries helped by the presence of indexes?

```
SELECT   *
FROM     Employee
WHERE    age = 33
```

```
SELECT   *
FROM     Employee
WHERE    age > 33
```

```
SELECT   *
FROM     Employee
WHERE    sal = 3000
```

```
SELECT   *
FROM     Employee
WHERE    sal > 3000
```

What an index can provide

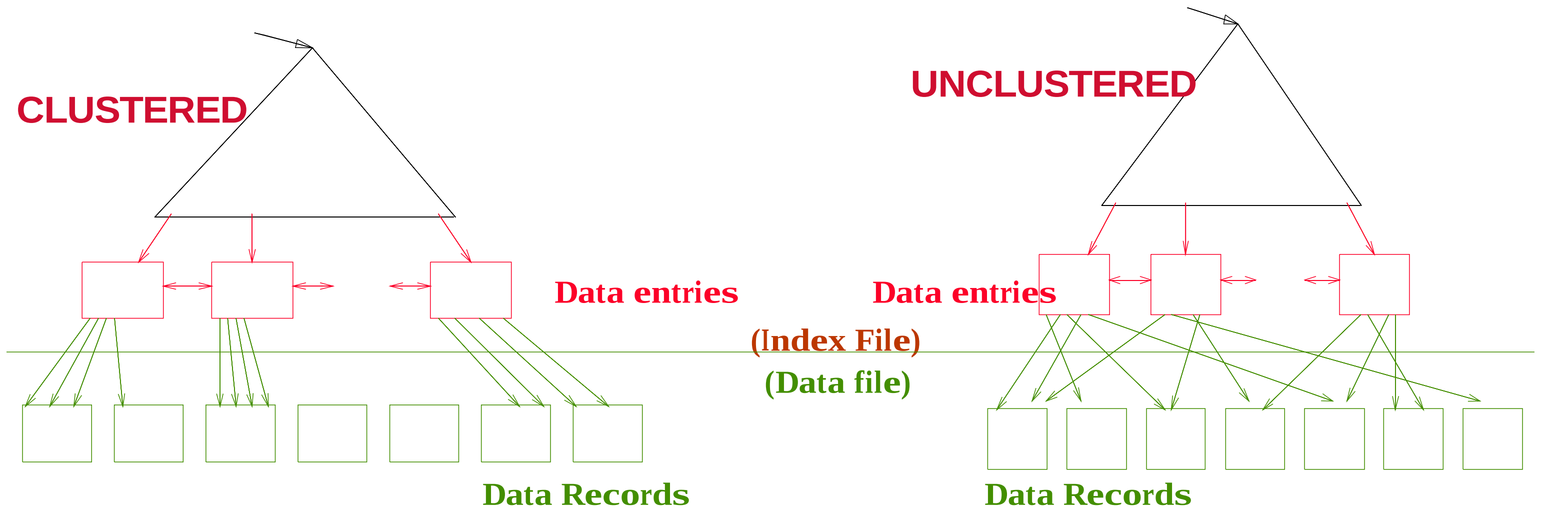
Given a key k, an index returns k^* where k^* is one of three things:

1. A *data record* (the tuple itself) with the search key value k
2. A *pointer* to a record with search key k together with k.
3. A *list of pointers* to records with search key k together with k.

Clustered vs. Unclustered Indexes

If we use tree indexing (to be described) we can exploit the ordering on a key and make *range queries* efficient.

An index is *clustered* if the data entries that are close in this ordering are stored physically close together (i.e. on the same page).



Tree indexing

Why not use the standard search tree indexing techniques that have been developed for main memory data (variations on binary search trees): AVL trees, 3-3 trees, red-black trees, etc?

Reason: binary search is still slow. 10^6 tuples (common) $\log_2(10^6) = 20$ – order 1 second because “dereferencing” a pointer on disk takes between 0.01 and 0.1 seconds.

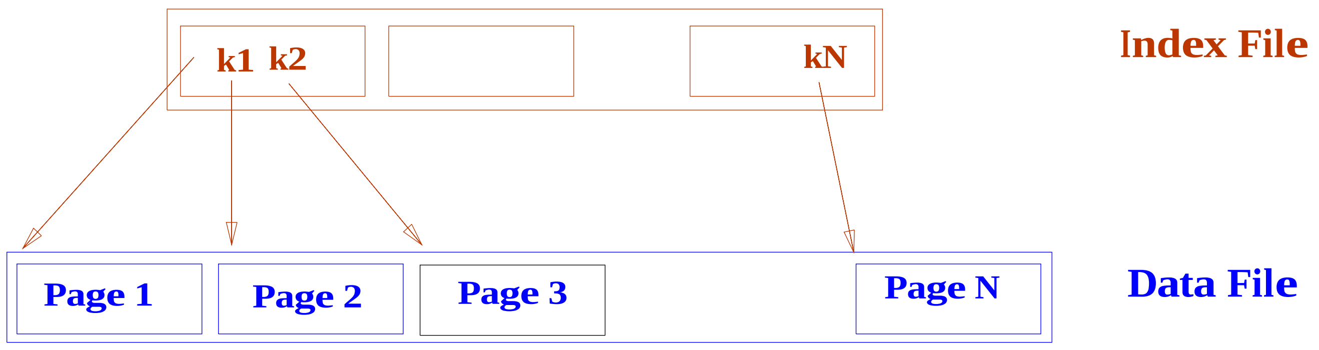
Solution:

1. Use n-ary trees rather than binary.
2. Keep only keys and pointers at internal nodes. Leaves can be data values (either records or record-id/key-value pairs)

Range Search

Example of point (2). We can speed up ordinary binary search on a sorted array by keeping indexes and page pointers in a separate file.

The "index file" will typically fit into cache.



Consider queries such as

```
SELECT   *
FROM     Employee
WHERE    20 < Sal AND Sal < 30
```

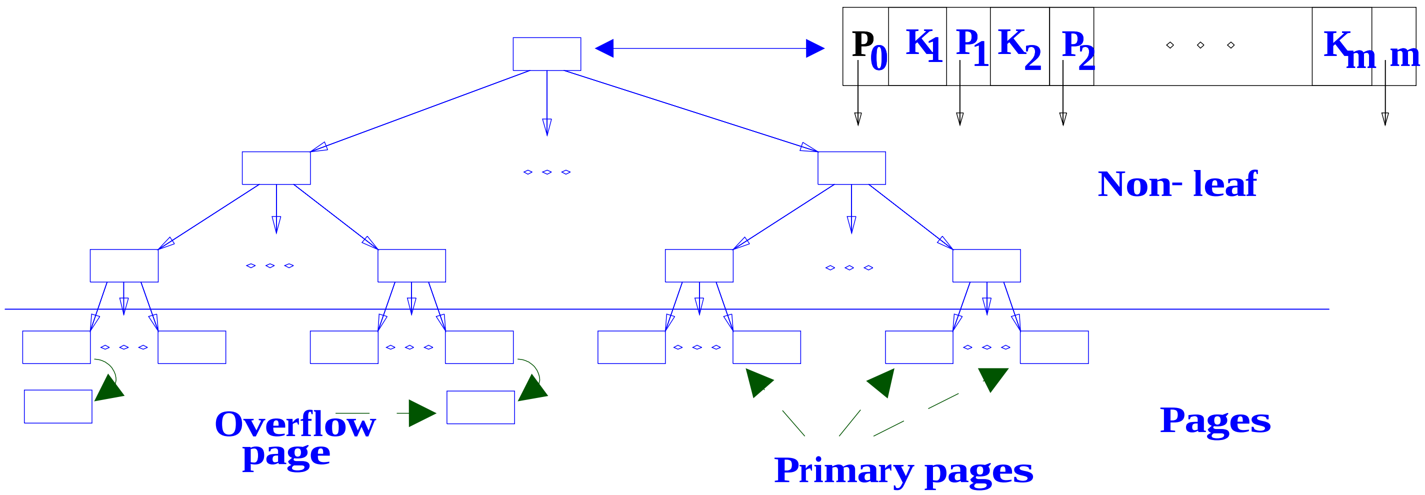
or

“Find all employees whose name begins with ‘Mac’.” (also a range search)

ISAM

ISAM $=$ *Indexed Sequential Access Method*: a search tree whose nodes contain m keys and $m+1$ pointers. m is chosen to "fill out" a page. A "pointer" is a page-id.

The pointer p_i between keys k_{i-1} and k_i points to a subtree whose keys are all in the range $k_{i-1} < k < k_i$.



How ISAM works

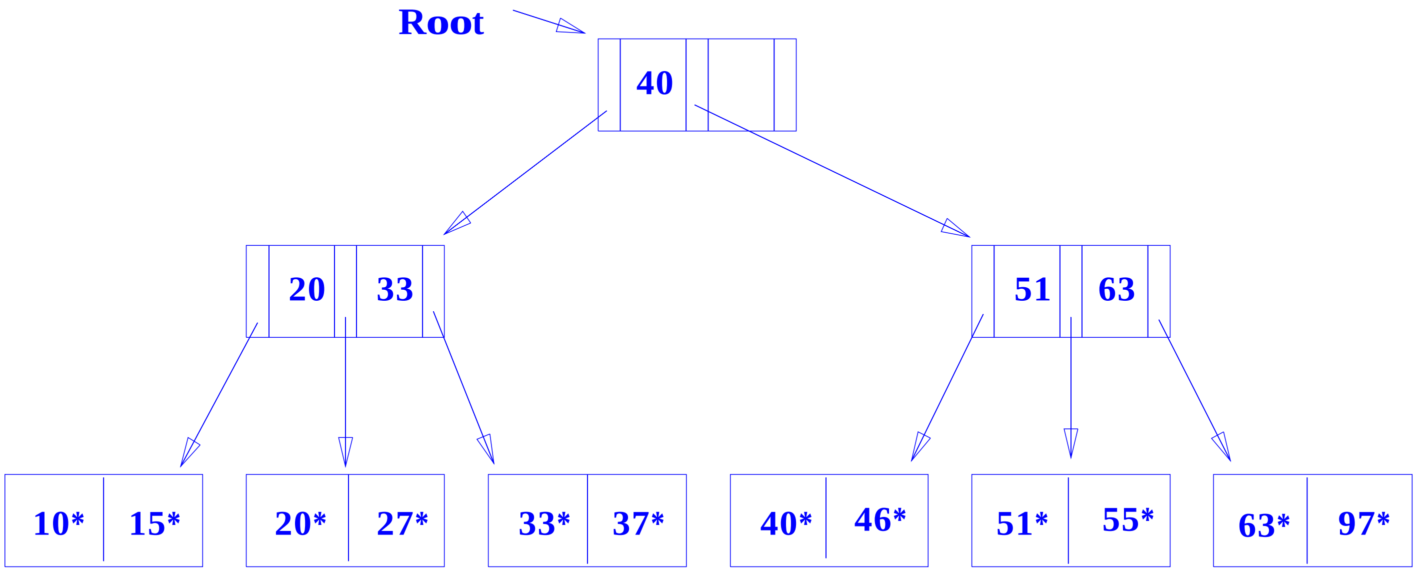
- Create file(s): Data entries are sorted. Leaf data pages are allocated sequentially. Index is constructed. Space for overflow pages is allocated.
- Find an entry (search). Obvious generalisation of method for binary search tree.
  - If pages are large, we can also do binary search on a page, but this may not be worth the effort. I/o costs dominate!
- Insert an item. Find leaf data page (search) and put it there. Create overflow page if needed.
- Delete and item. Find leaf data page (search) and remove it. Maybe discard overflow page.

Note. In ISAM, the index remains *fixed* after creation. It is easy to construct pathological examples which make ISAM behave badly.

A simple ISAM example

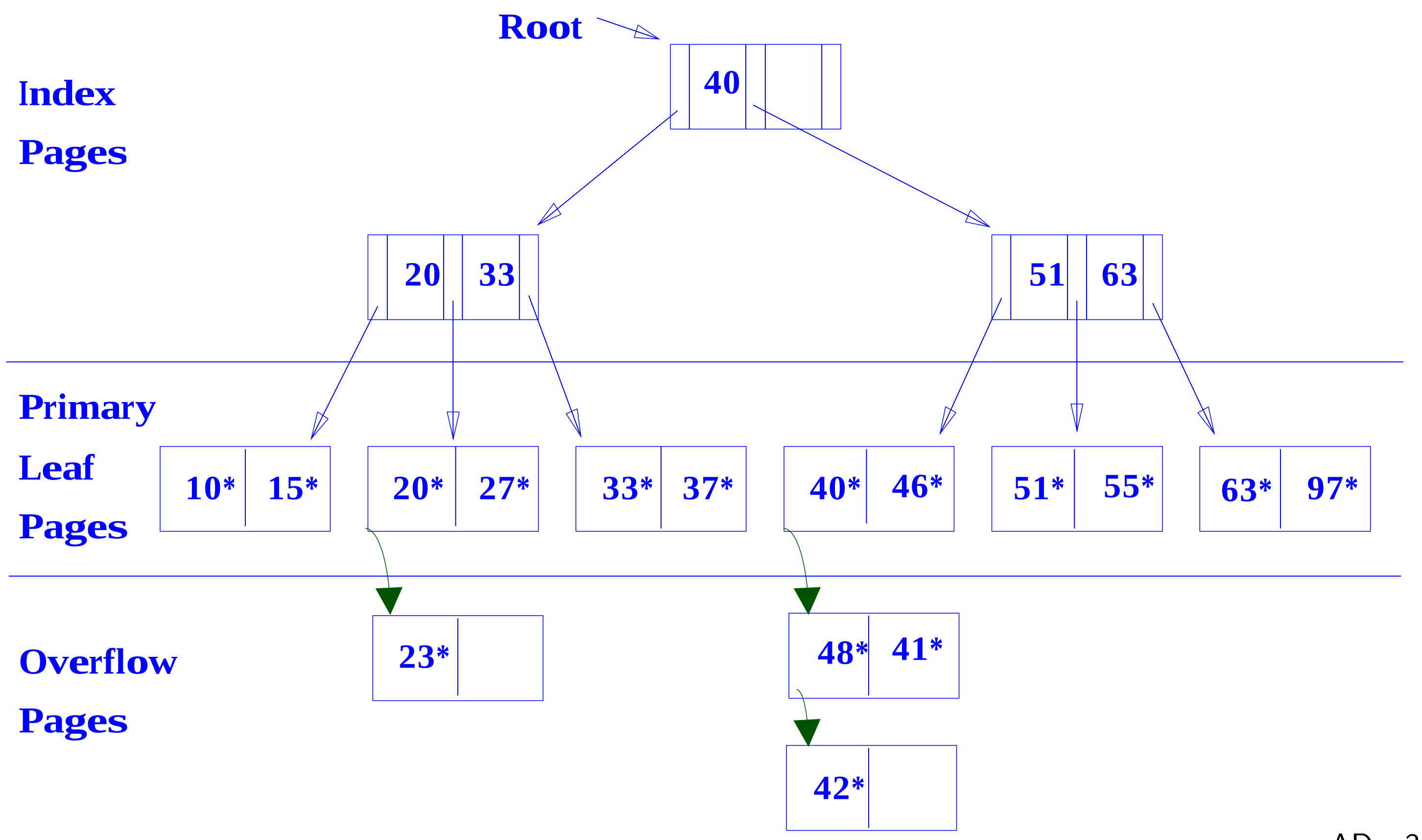
This is not realistic. The example is only a 3-ary tree. In practice one might have 100-way branching.

Note that we can perform an ordered traversal of the data entries by a traversal of the index.



ISAM after inserts

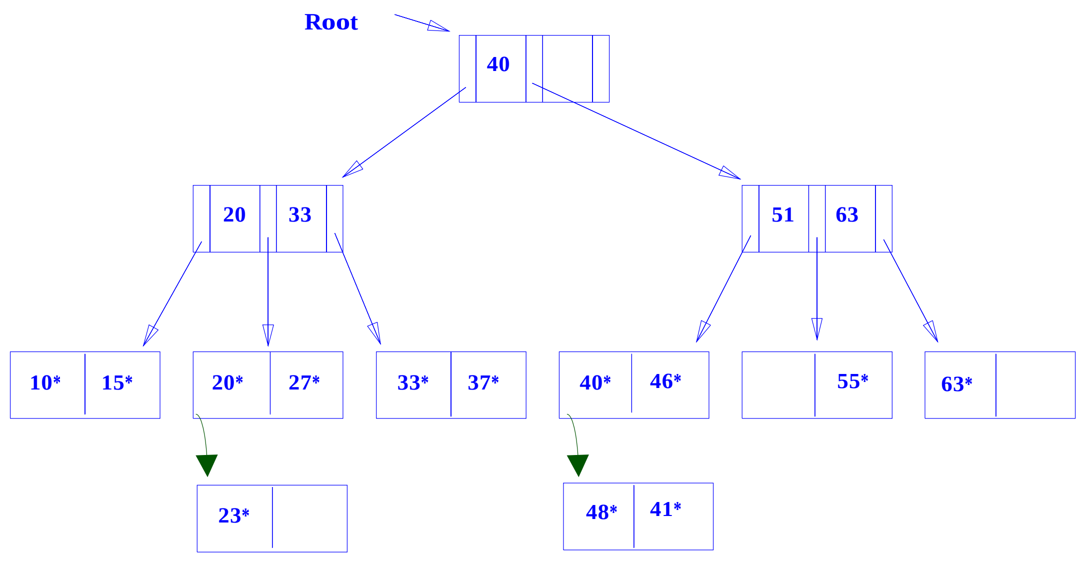
Inserting $23^*, 48^*, 41^*, 42^*$



ISAM after deletes

Deleting $42^*, 51^*$

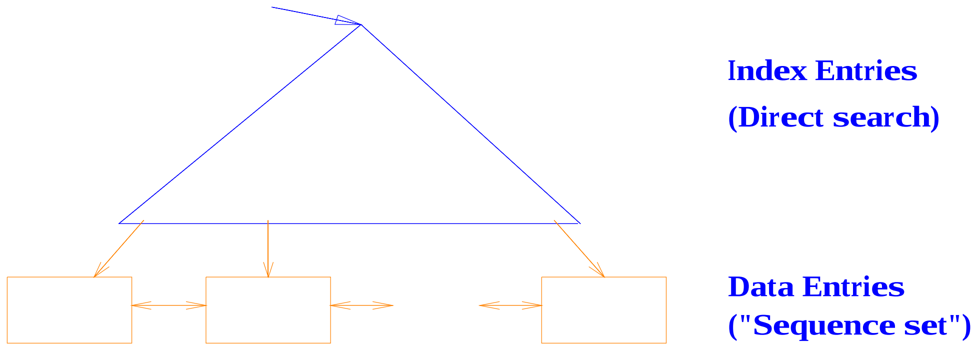
Note that 51 appears as a key but no longer as a leaf value.



Main problem with ISAM: *index is fixed*. Can become worse than useless after a long series of inserts and deletes.

B+ Tree. The Standard Index

- Each node (page) other than the root contains between d and $2d$ entries. d is called the *order* of the tree. Pages are not always full.
- Suitable for both equality and range searches.
- Lookup (equality), insertion and deletion all take approx. $\log_k(N)$ page accesses where $d \leq k \leq 2d$.
- Tree remains perfectly balanced! All the leaf nodes are the same distance from the root.
- In practice B-trees are never more than 5 levels deep, and the top two levels are typically cached.

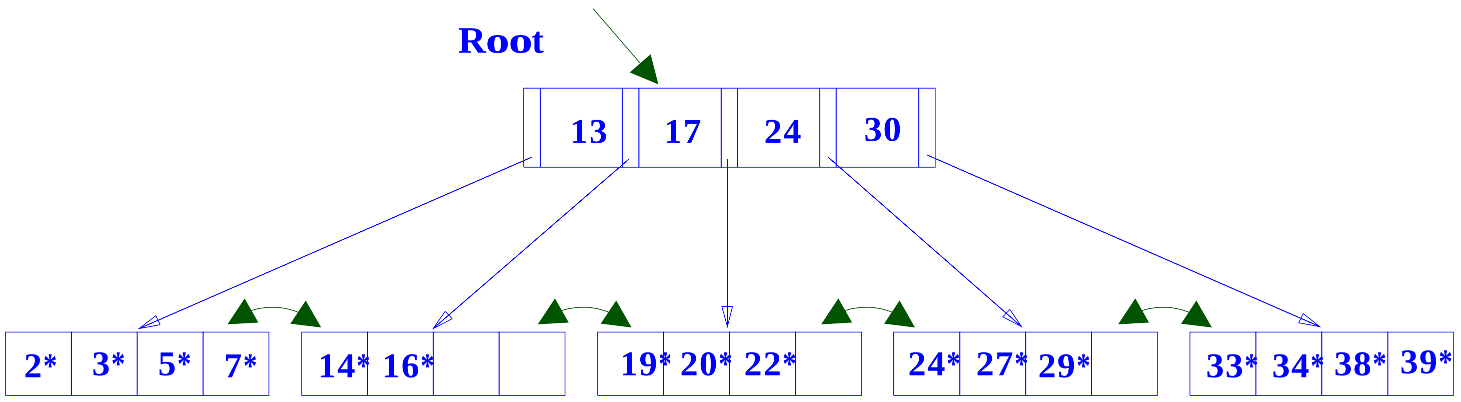


Example B+ Tree

Search is again the obvious generalization of binary tree search. Could do binary search on nodes.

Key values need not appear in any data entries.

Data pages are linked (we'll see why shortly)


Root
13
17
24
30
2*
3*
5*
7*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*

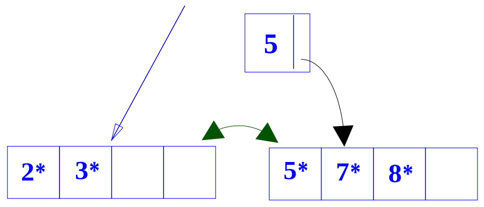
Inserting into a B+ Tree

- Find leaf page that should hold k^*
- If page is not full, we can insert k^* and we are done.
- If not, split the leaf page into two, putting "half" he entries on each page and leaving a middle key $k'$to be inserted in the node above.
- Recursively, try to insert k' in the parent node.
- Continue splitting until *either* we find a node with space, *or* we split the root. The root need only contain two children.

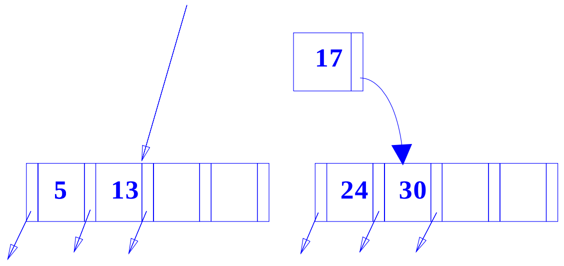
Either the tree grows fatter or (very occasionally) it grows deeper by splitting the root.

Example Insertion

Inserting 8^*. The page is full so the leaf page splits and 5, the middle key, is pushed up. (5^* remains in the leaf page.)

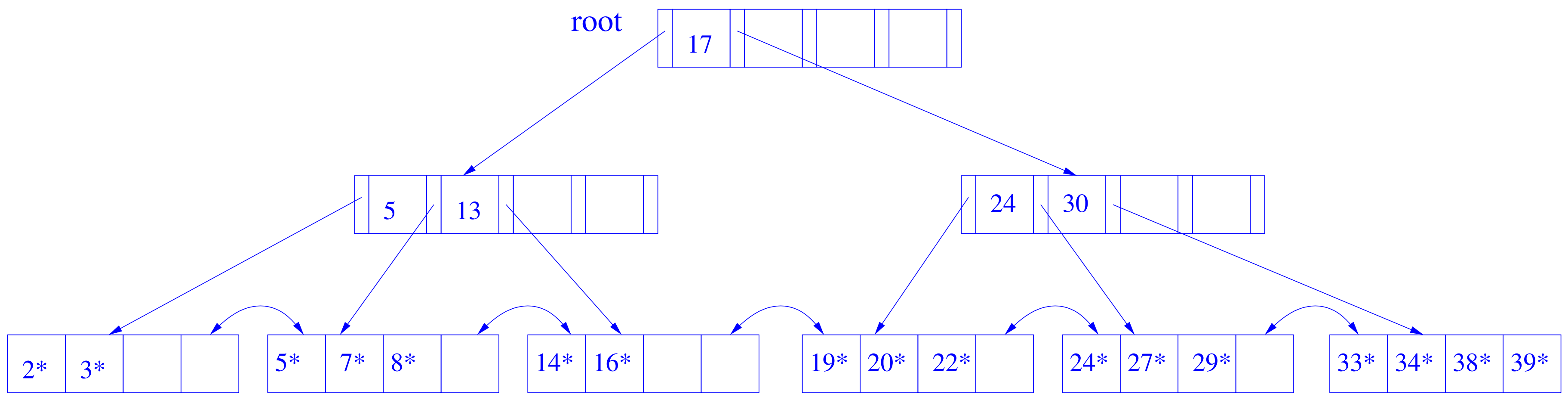


The parent page is also full so this page splits and 17 is pushed up. (This is not in a data entry, so it does not remain on one of the child pages.)



The new root contains just 17

The Resulting Tree



We could have avoided the root split by sideways redistribution of data on the leaf pages. But how far sideways does one look?

More properties of B+ trees

- Efficient deletion
- Bulk insertion from ordered data

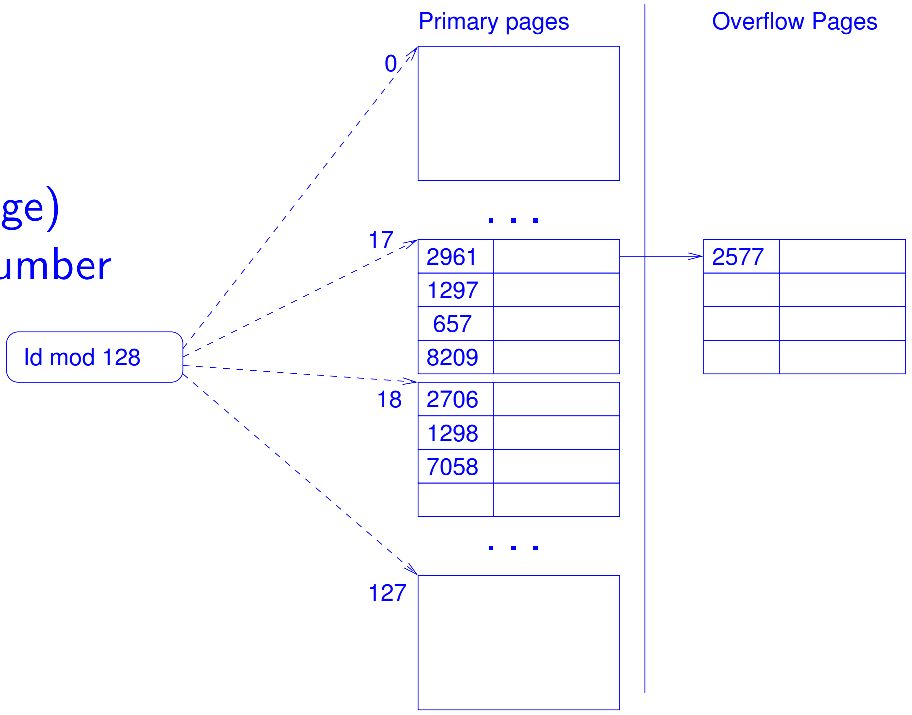
B+ Trees in practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 1334 = 312,900,700 records
  - Height 3: 1333 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Hashing

An alternative to tree indexing

- Lookup, insert and update usually take 1 read *except when restructuring is needed.*
- No good for range searches.
- Typically hash to bucket (page) $h(x)$ mod m where m is number of pages.



Expanding Hash Tables

If the hash table becomes "full", which can happen because

- we don't have overlow pages and a page becomes full, or
- the number (ratio) of overflow pages exceeds some threshold,

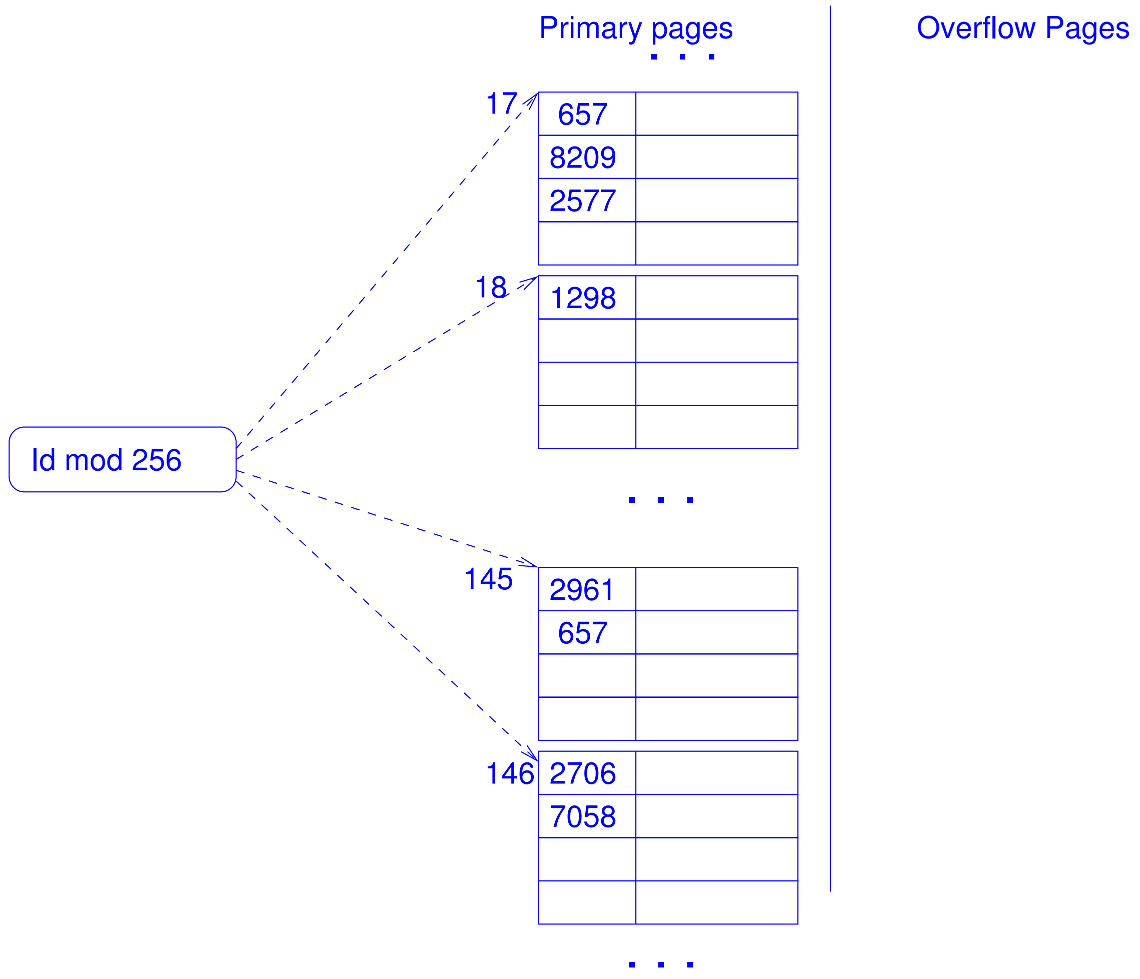
we need to *restructure* the hash table.

We can double the number of buckets and split each one.

E.g. $1297 \bmod 128 = 17 = 1297 \bmod 256$. So 1297 stays on page 17.

However $2961 \bmod 128 = 17$ but $2961 \bmod 256 = 145$, so 2961 gets moved from page 17 to page 145.

Doubling the table size



Alternative Hashing Techniques

Reconfiguring a whole hash table because can be very time consuming. Need methods of "amortizing" the work.

- *Extendible hashing* Keep an index vector of n pointers. Double the index when needed, but only split pages when they become full.
- *Linear Hashing*. Clever hash function that splits pages one at a time. This does not avoid overflow pages, but we add a new page when the number of records reaches a certain threshold. (80-90%) of maximum capacity.

Storage and indexing – Review

- Physical properties of disks and seek times for random and sequential access.
- File organization
- Representation and placement of tuples
- Clustering
- ISAM files
- B+ trees
- Hash tables
- Relative merits of B+ trees and hashing