Databases

Cap. 4. Indexing. DB Optimization

Textbook: Ramakrishnan, Gehrke, “Database Management Systems”, McGraw Hill, 2003

2020 UPT

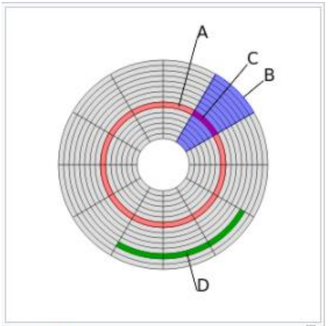
Conf.Dr. Dan Pescaru

Data on External Storage

1. RAM
   - + Efficient processing
   - - Low capacity/expensive
   - - Volatile
2. HDD (SSD)
   - + High (mid) capacity/Average (high) price
   - + Non-volatile
   - - Significant (medium) access time (10^5 x t_{RAM})
3. Tape/DVD
   - + High/medium capacity / Low price
   - - Sequential access only / Slow random access

HDD file system

1. File system on HDD
   - Sectors/blocks organization
   - Slow data seek (due mechanical parts)
   - Reading several consecutive pages is much cheaper than reading them in random order
   - Managed by OS or DBMS (e.g. one table span on several disks)
2. Data reading
   - Sectors reading + buffering
   - Buffer manager: OS+DBMS(why?)



Disk structure:
(A) track
(B) geometrical sector
(C) track sector
(D) cluster

Data file

1. A sequence of records
2. Records are mapped on disks sectors
3. Variable or fixed length records (space optimization v.s. efficient seek op.)
4. Variable length records: more complex organization (e.g. lists)
5. File structure
   - File header (e.g. dBase)
   - DB Catalog (e.g. Oracle, MySQL)

Access optimization

1. Heap files (random order) - suitable when typical access is a file scan retrieving all records
2. Inefficient for record seek based on a logical condition
3. Solutions
   - Record sorting
   - Indexing (B+ trees, hash index, clustering, page virtualization)

File sorting

1. Efficient for
   - Best if records must be retrieved in sorting order
   - Suitable for extracting certain range of records
   - Finding a record based on sorting expression – *log_2*(N) efficiency
2. Issues
   - <u>Very slow</u> data insertion/updating/deleting
   - <u>Only one</u> sorting criteria is possible
   - Changing <u>sorting criteria</u> is highly inefficient

E.g. xBase sorting

```
SORT [rec. domain] TO file
   ON exp1 [/A][/C][/D], exp2 [/A], ...
   [FOR condition]
   [ASCEN/DESC]
```

- Create a new file (space requirement on HDD)
- Parameters
  - /A – ascending order
  - /C – case insensitive
  - /D – descending order
  - ASCE/DESC – global sorting order (for all expressions)

xBase Example

```
USE STUD                 && open the file
LIST                     && list records
SORT TO STUD_S ON NAME
                         && sorting by name
USE STUD_S               && open the sorted file
LIST                     && list records
```

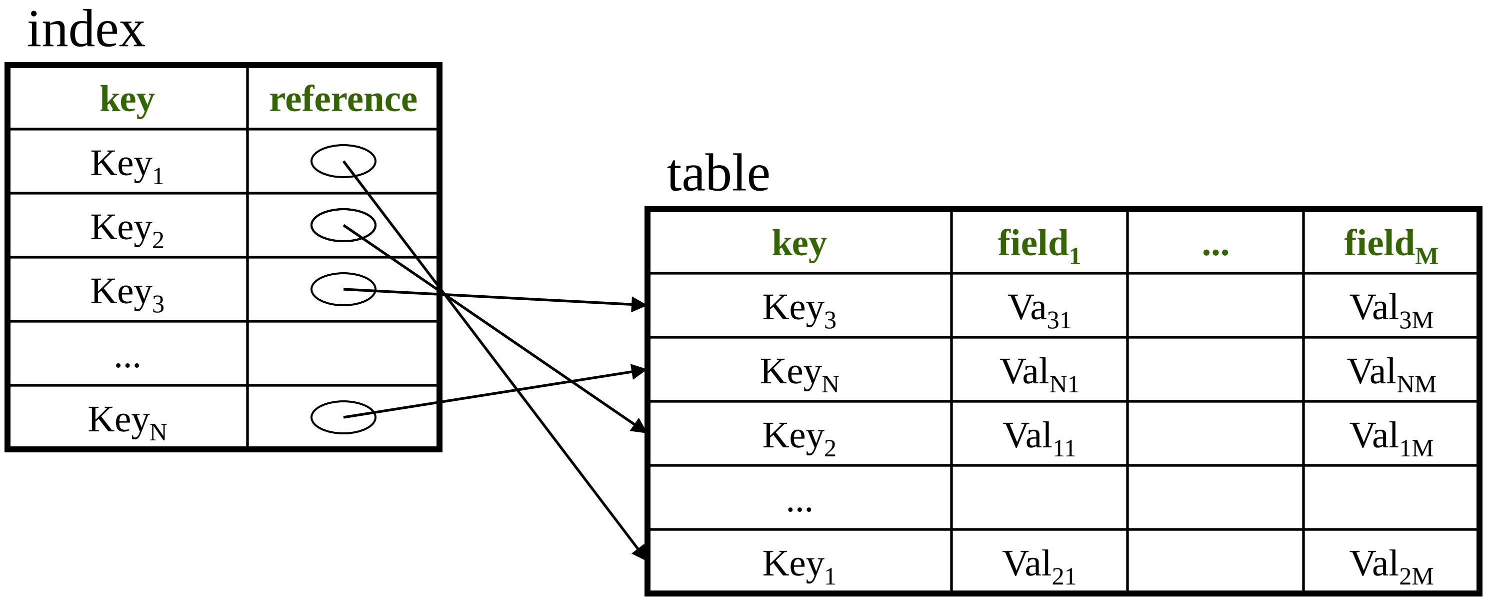
- Space requirement on external storage
  - Initial file (stud.dbf)
  - Sorted file (stud_d.dbf)
  - Temporary file (external sorting algorithm)

Indexing

1. Better alternative to sorting
2. Efficient for
    - Retrieving records in sorting order
    - Range extraction
    - Data seek – efficiency $\log_k(N)$, k>>2
    - More than one index possible => more than one sorting criteria
3. Issues
    - Relatively slow data insertion/updating/deleting – index updating is required
    - Significant space requirements

Indexing – basic idea

1. Basic index



index

key	reference
Key_1	
Key_2	
Key_3	
...	
Key_N	

table

key	$field_1$	...	$field_M$
Key_3	Va_{31}		Val_{3M}
Key_N	Val_{N1}		Val_{NM}
Key_2	Val_{11}		Val_{1M}
...			
Key_1	Val_{21}		Val_{2M}

Index - organization

- An index on a file speeds up selections on the search key fields for the index
  - Any subset of the fields of a relation can be the search key for an index on the relation
  - Search key is not the same as key (minimal set of fields that uniquely identify a record in a relation)
- An index supports efficient retrieval of all data entries *k** with a given key value *k*

Alternatives for data entry *k** in index (I)

- Three alternatives:
  i. data record with key value *k*
  ii. <*k*, *rid* of data record with search key value *k*>
  iii. <*k*, list of *rid* of data records with search key *k*>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate key value *k*
  - Examples of indexing techniques: B+ trees, hash-based structures, binary indexes etc.
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for data entry (II)

- Alternative 1 (clustering):
  - If this is used, index structure is a file organization for data records (instead of a heap file or sorted file)
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency)
  - If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large

Alternatives for data entry (III)

- Alternatives 2 and 3:
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small
  - Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length

Index classification

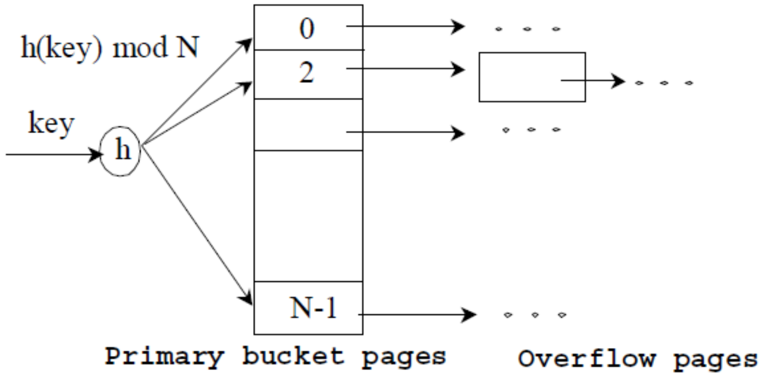
1. **Primary** vs. **secondary**: If search key contains primary key, then called primary index
   - Unique index: Search key contains a candidate key
2. **Clustered** vs. **unclustered**: If order of data records is the same as order of data entries, then called clustered index
   - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare)
   - A file can be clustered on at most one search key
   - Cost of retrieving data records through index varies greatly based on whether index is clustered or not!

Hash-based index

- Good for equality selections
    - Index is a collection of buckets
    - Bucket = primary page plus zero or more overflow pages
    - Hashing function: *h(r)* = bucket in which record *r* belongs.
    - *h* directs the search for indexing key
- If Alternative (1) is used, the buckets contain the data records; otherwise, they contain <key, rid> or <key, rid-list> pairs

Static Hashing (I)

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed
- *h(k)* mod *N* = bucket to which data entry with key k belongs (*N* = # of buckets)



Static Hashing (II)

- Buckets contain data entries
- Hash fct works on search key field of record *r*. Must distribute values over range 0 ... N-1
  - *h(key)* = (*a* * *key* + *b*) usually works well
  - *a* and *b* are constants; multiple choices to tune *h*
- Long overflow chains can develop and degrade performance
  – Extendible and Linear Hashing: Dynamic techniques to fix this problem

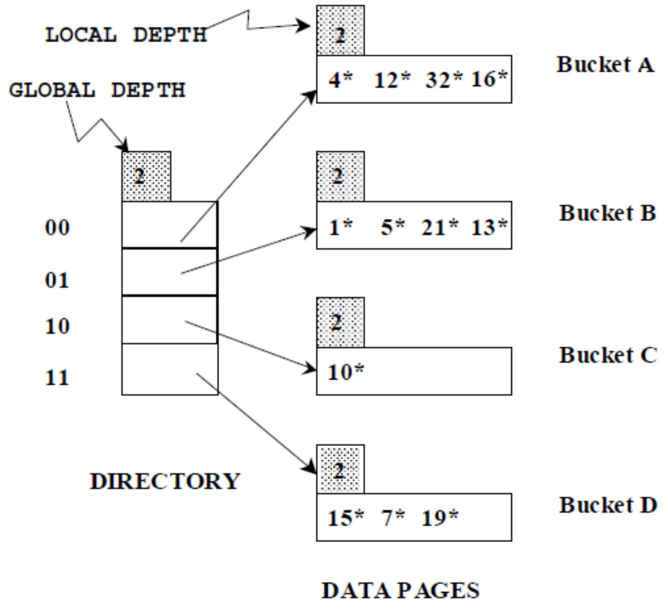
Extensible Hashing (I)

- Situation: Bucket (primary page) becomes full. Why not re-organize file by doubling # of buckets?
  - Reading and writing all pages is expensive!
  - Idea: Use directory of pointers to buckets, double # of buckets by doubling the directory, splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. No overflow page!
  - Trick lies in how hash function is adjusted!

Extensible Hashing (II)

Example

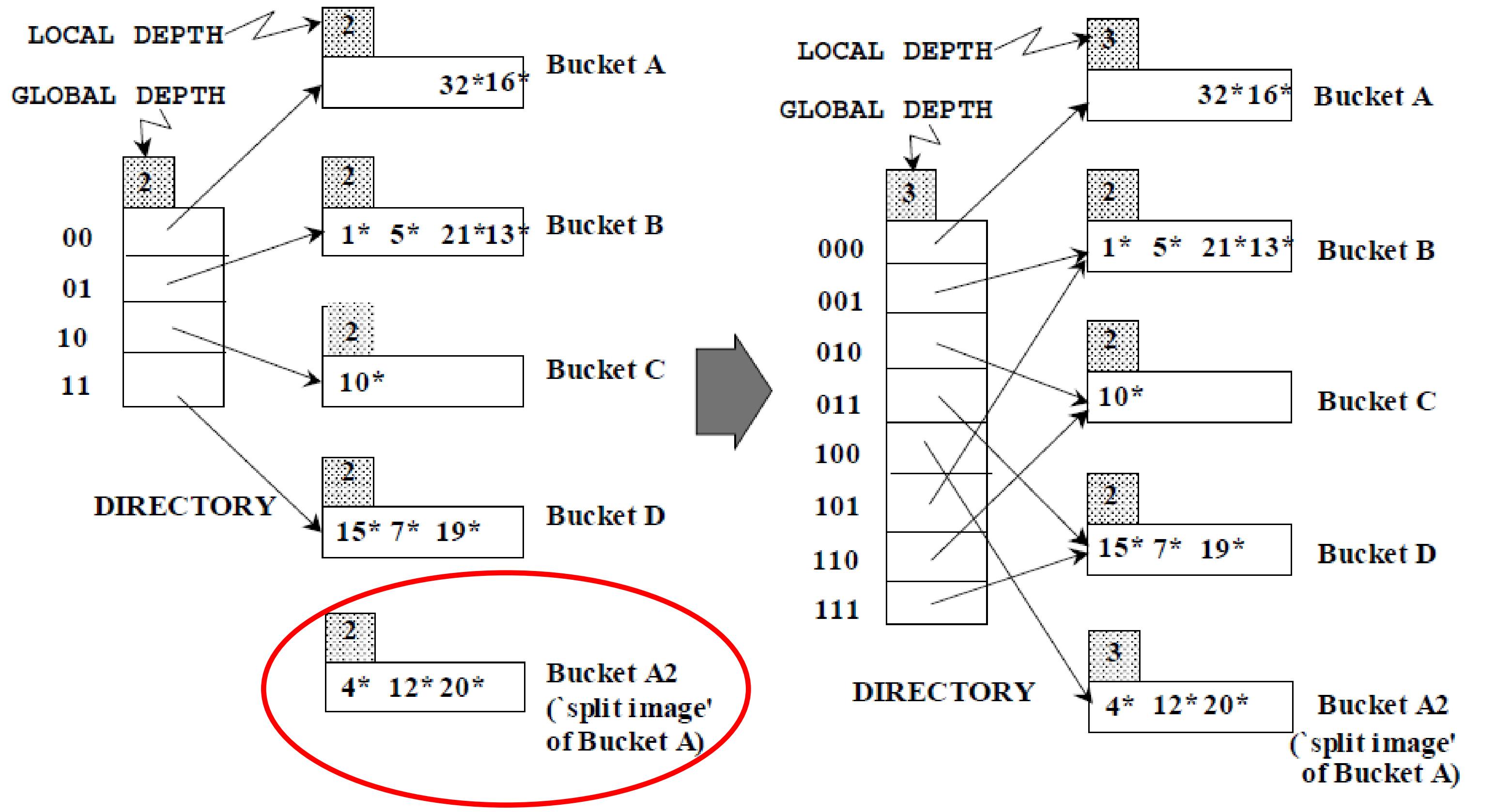
- Directory is array of size 4
- To find bucket for r, take last global depth as # bits of *h(r)*
  - If *h(r)* = 5 (binary 101), it is in bucket pointed by 01



- Insert: If bucket is full, split it (allocate new page and re-distribute)
- If necessary, double the directory (splitting a bucket does not always require doubling; it depends on relation of global depth with local depth for the split bucket)

Extensible Hashing (III)

- Insert h(r)=20 (causes doubling)



Extensible Hashing – splitting mechanism

1. **Global depth** of directory: Max # of bits needed to tell which bucket an entry belongs to
2. **Local depth** of a bucket: # of bits used to determine if an entry belongs to this bucket

- When does bucket split cause directory doubling?
  - Before insert, local depth of bucket = global depth. Insert causes local depth to become > global depth; directory is doubled by copying it over and fixing pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

Extensible Hashing – conclusions

- If directory fits in memory, equality search answered with one disk access; else two
  - 100MB file, 100 bytes/record, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory
  - Directory grows in spurts, and, if the distribution of hash values is skewed, directory can grow large
  - Multiple entries with same hash value cause problems!
- **Delete**: If removal of data entry makes bucket empty, can be merged with split image. If each directory element points to same bucket as its split image, can halve directory

Linear Hashing (I)

- This is another dynamic hashing scheme, an alternative to Extendible Hashing
- LH handles the problem of long overflow chains without using a directory, and also handles duplicates
- Idea: use a family of hash functions h_0, h_1, …
  - $h_i(key) = h(key)$ mod $(2^i N)$; N=initial # buckets
  - h is some hash function (range is not 0 to $N-1$)
  - If $N=2^{d0}$, for some $d0$, h_i consists of applying h and looking at the last di bits, where $di=d0+i$
  - h_i+1 doubles the range of h_i (similar to directory doubling)

Linear Hashing (II)

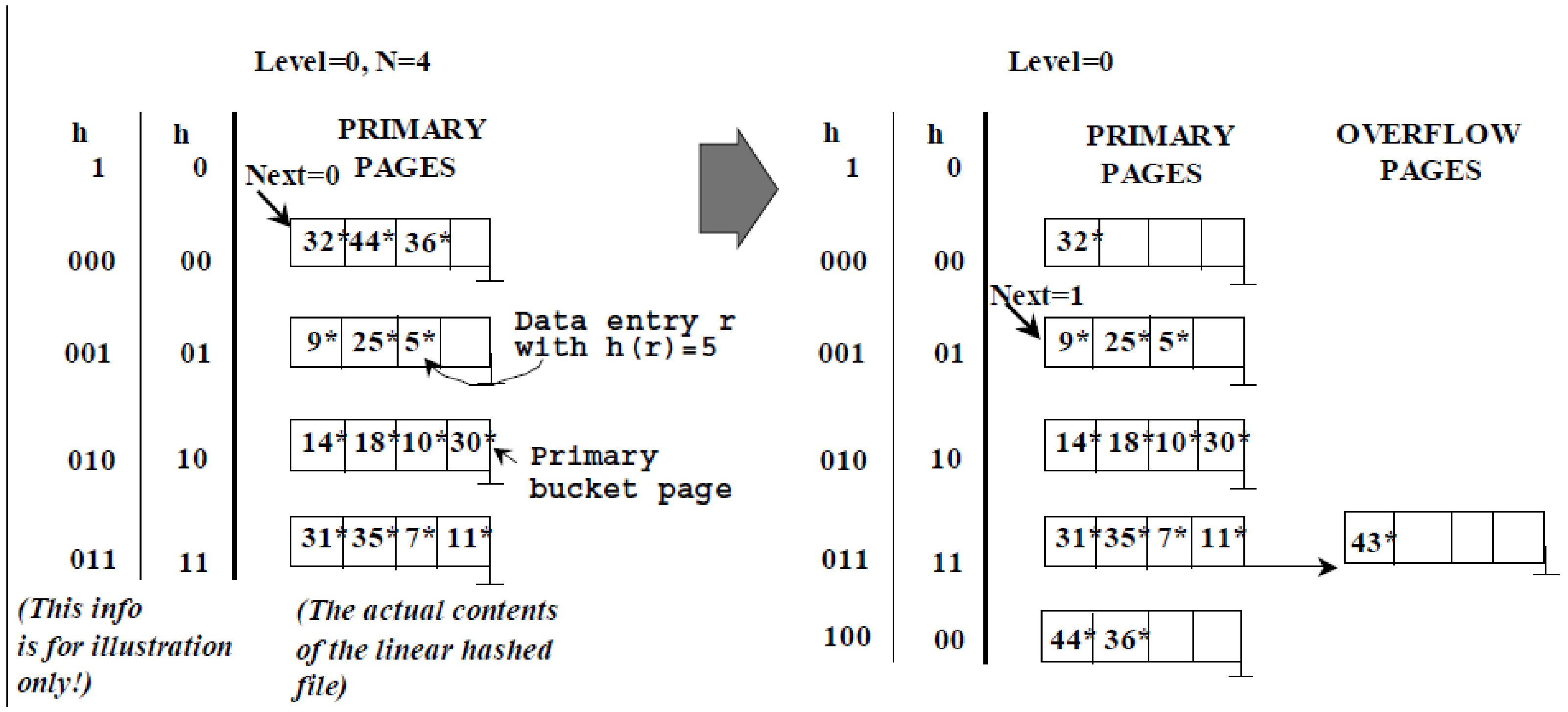
- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin
  - Splitting proceeds in *rounds*. Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next*-1 have been split; *Next* to N_R yet to be split
  - Current round number is *Level*
  - **Search**: To find bucket for data entry *r*, find $h_{Level}(r)$:
    - If $h_{Level}(r)$ in range *Next* to N_R, *r* belongs here
    - Else, *r* could belong to bucket *hLevel(r)* or bucket $h_{Level}(r) + N_R$; must apply $h_{Level+1}(r)$ to find out

Linear Hashing (III)

- **Insert**: Find bucket by applying h_{Level} / $h_{Level+1}$:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (Maybe) Split *Next* bucket and increment *Next*
- Can choose any criterion to *trigger* split
- Since buckets are split round-robin, long overflow chains don't develop!
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased

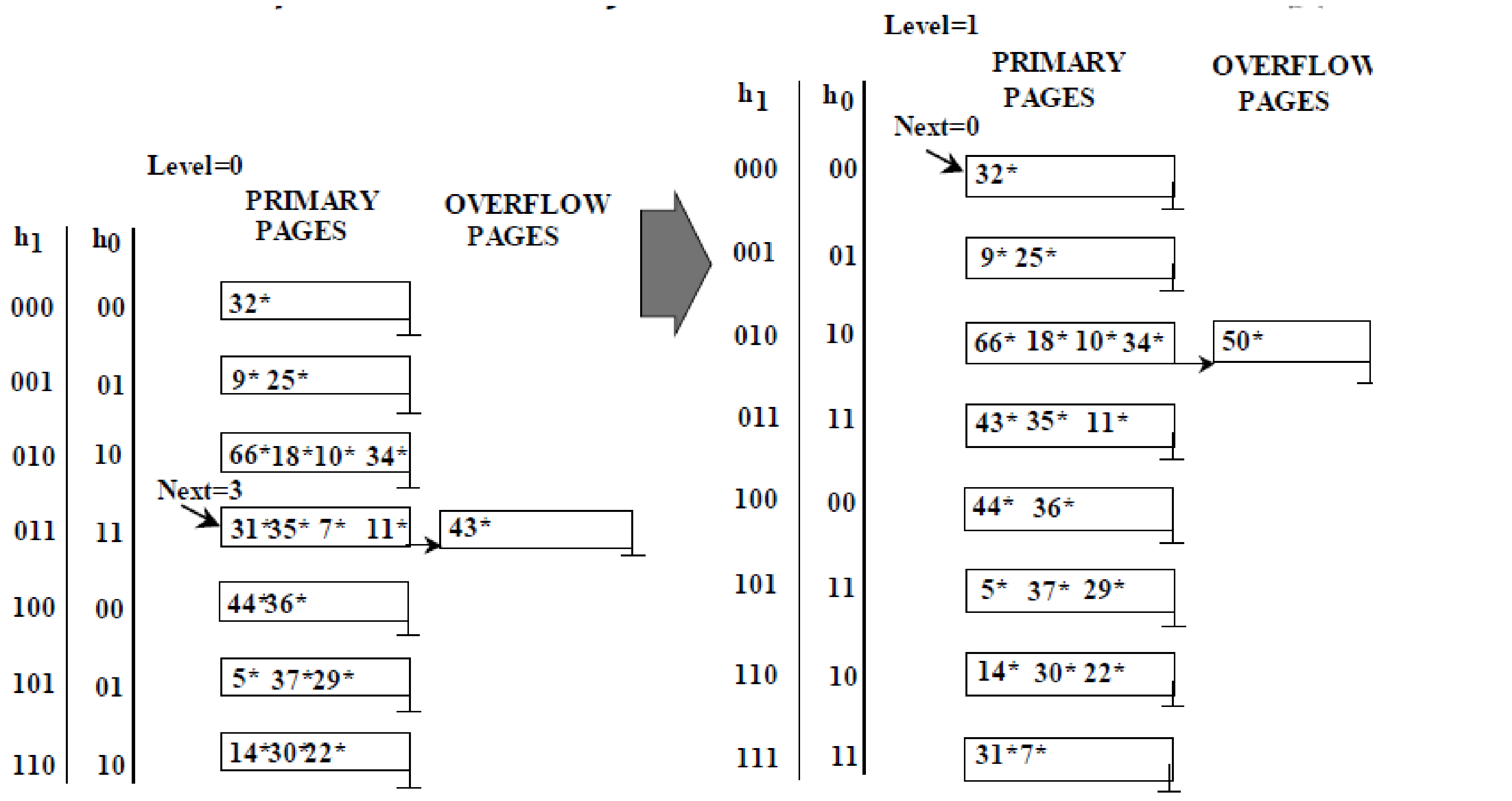
Example of Linear Hashing (I)

- On split, $h_{Level+1}$ is used to re-distribute entries



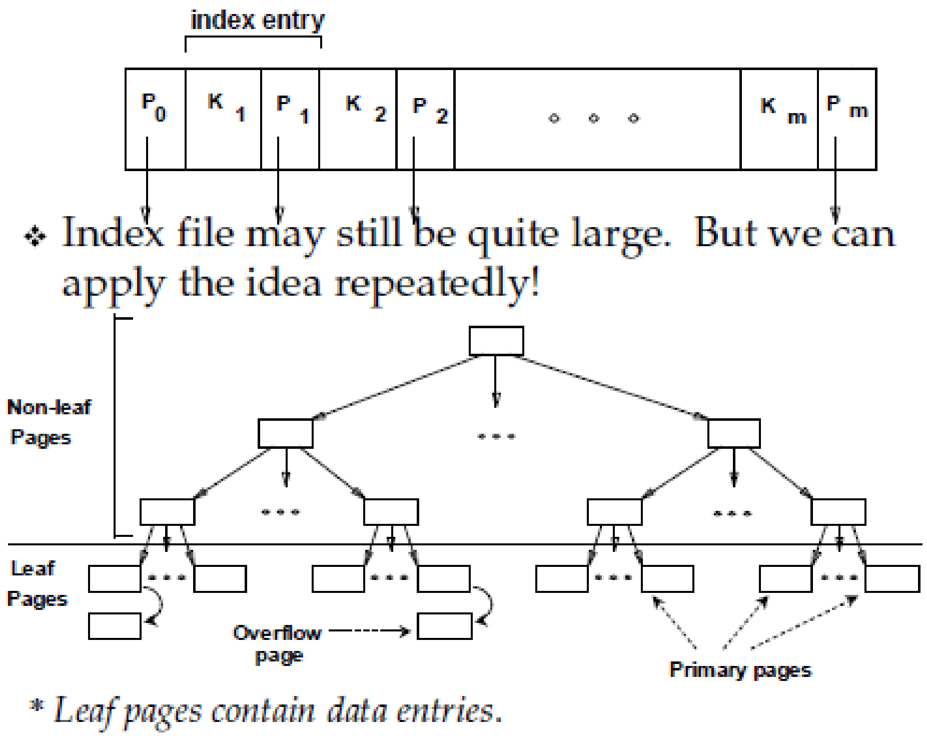
Example of Linear Hashing (II)

- End of round



ISAM tree indexes

- Indexed Sequential Access Method (e.g. Berkeley DB, Paradox, MySQL, Ms. Access, dBase)



❖ Index file may still be quite large. But we can apply the idea repeatedly!

* *Leaf pages contain data entries.*

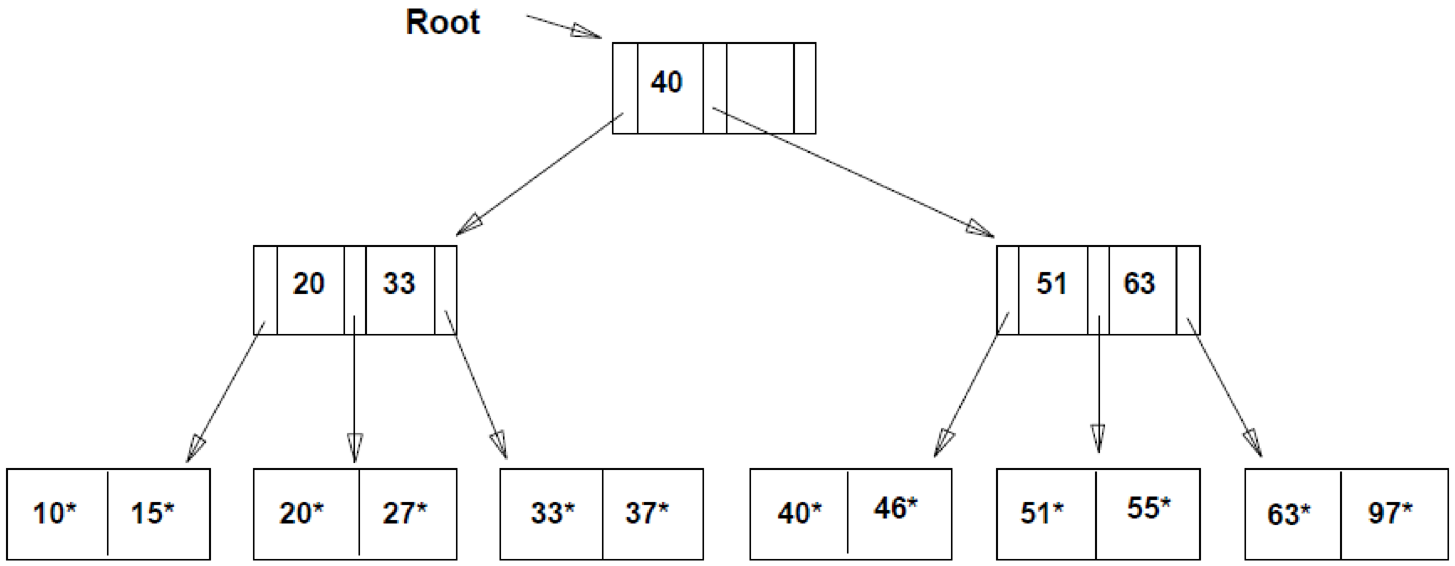
ISAM

1. **File creation**: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages
2. **Index entries**: <*search key value*, *page id*>; they direct search for data entries, which are in leaf pages
3. **Search**: Start at root; use key comparisons to go to leaf. Cost: $\log_F(N)$; F = # entries, N = # leaf pages
4. **Insert**: Find leaf data entry belongs to, and put it there
5. **Delete**: Find and remove from leaf; if empty overflow page, de-allocate

* *Static* tree structure: inserts/deletes affect only leaf pages

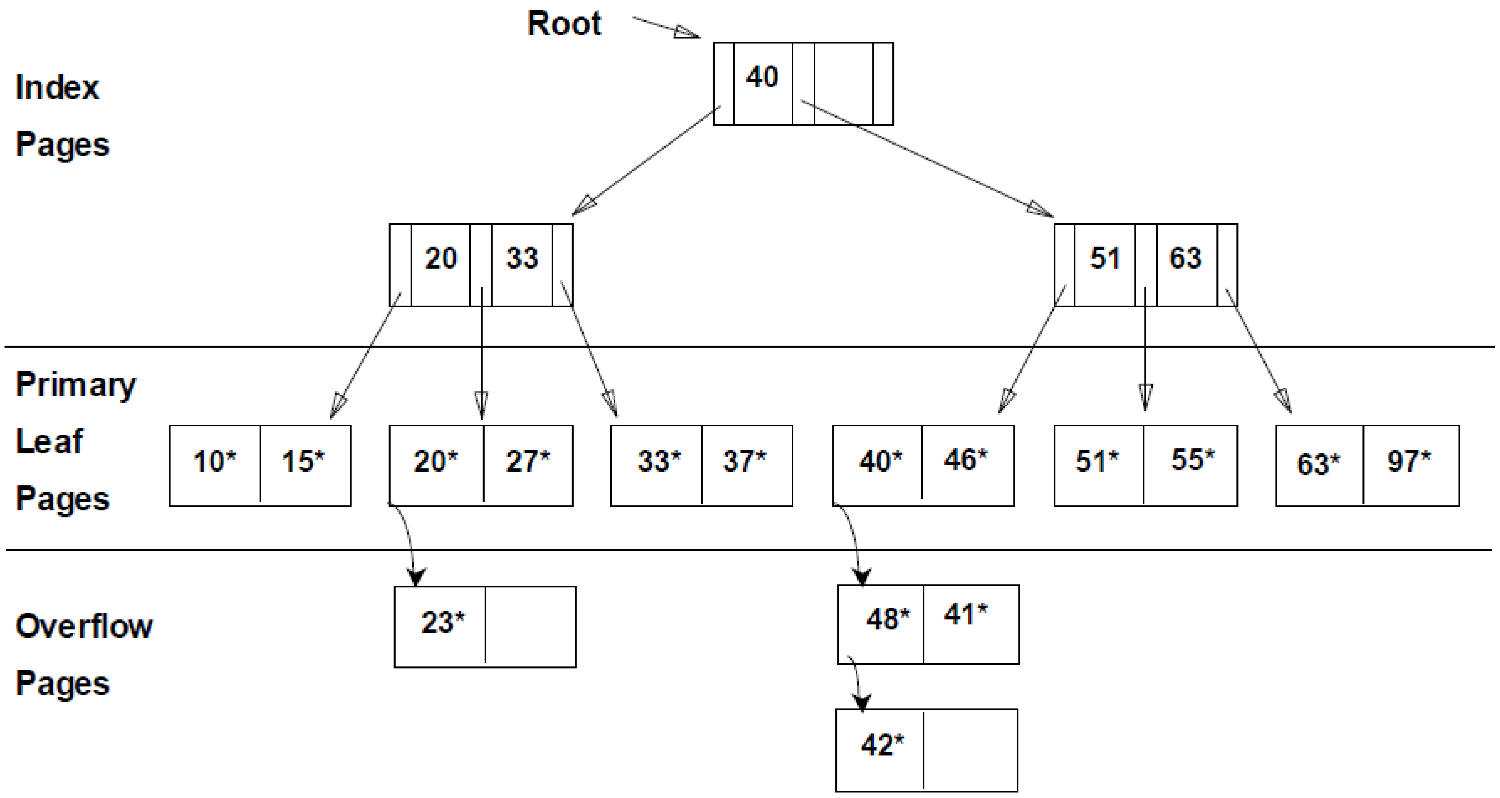
ISAM Example (I)

- Each node can hold two entries



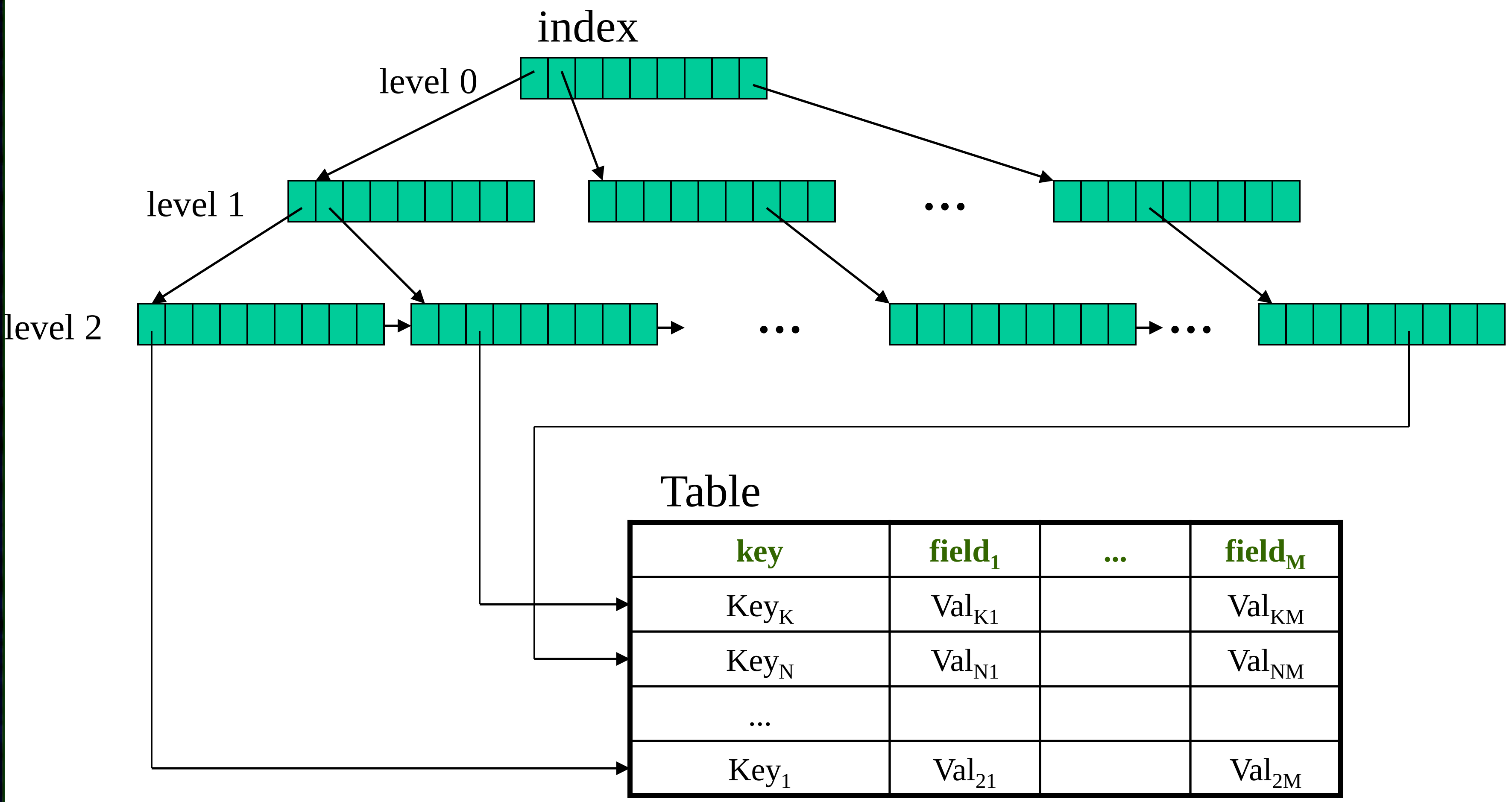
ISAM Example (II)

- After inserting 23*, 48*, 41*, 42*



B+ Tree: most widely used index

- E.g.: three level B+ tree index



key	$field_1$	…	$field_M$
Key_K	Val_{K1}		Val_{KM}
Key_N	Val_{N1}		Val_{NM}
…			
Key_1	Val_{21}		Val_{2M}

B+ Tree index

- Insert/delete at $log_F(N)$ cost; keep tree height balanced
  - (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains $d <= m <= 2 \times d$ entries. The parameter d is called the order of the tree
- Supports equality and range-searches efficiently
- High fanout (F: # of children per node) means depth rarely more than 3 or 4
- Almost always better than maintaining a sorted file

B+ Tree: inserting data

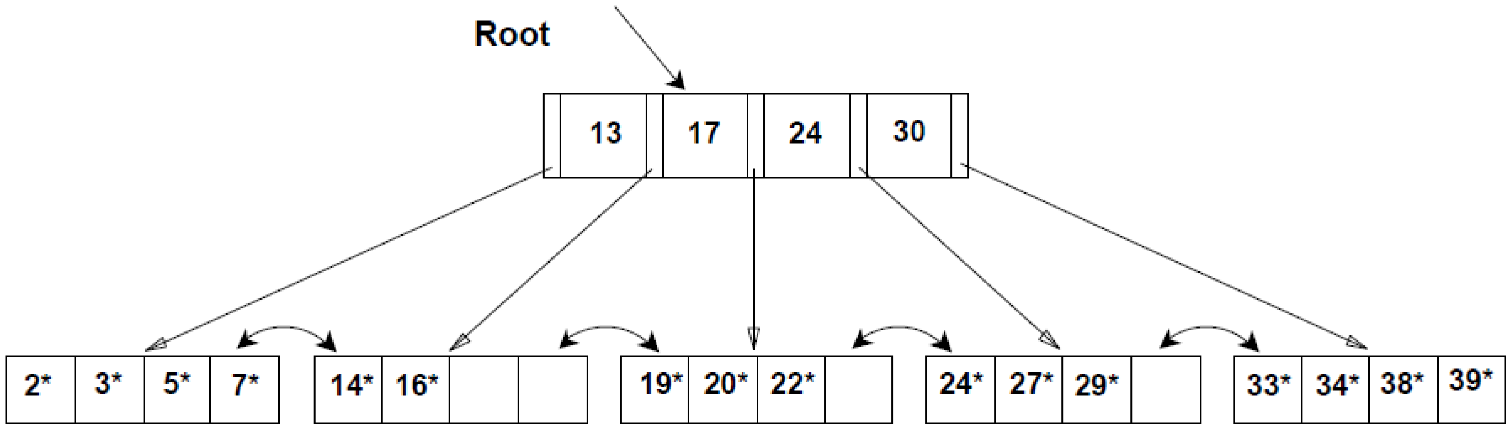
1. Find correct leaf *L*
2. Put data entry onto *L*
   - If L has enough space, done!
   - Else, must split *L* (into *L* and a new node *L2*)
     - Redistribute entries evenly, copy up middle key
     - Insert index entry pointing to *L2* into parent of *L*
3. This can happen recursively
   - To split index node, redistribute entries evenly, but push up middle key (contrast with leaf splits)
4. Splits *grow* the tree - root split increases height
   - Tree growth: gets wider or one level taller at top

B+ Tree: deleting data

1. Start at root, find leaf *L* where entry belongs
2. Remove the entry
   - If *L* is at least half-full, done!
   - If *L* has only d-1 entries,
     - Try to re-distribute, borrowing from sibling (adjacent node with same parent as *L*)
     - If re-distribution fails, merge *L* and sibling
3. If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*
4. Merge could propagate to root, decreasing height

B+ Tree example

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM)
- E.g. search for 5*, 15*, all entries >= 24* ...



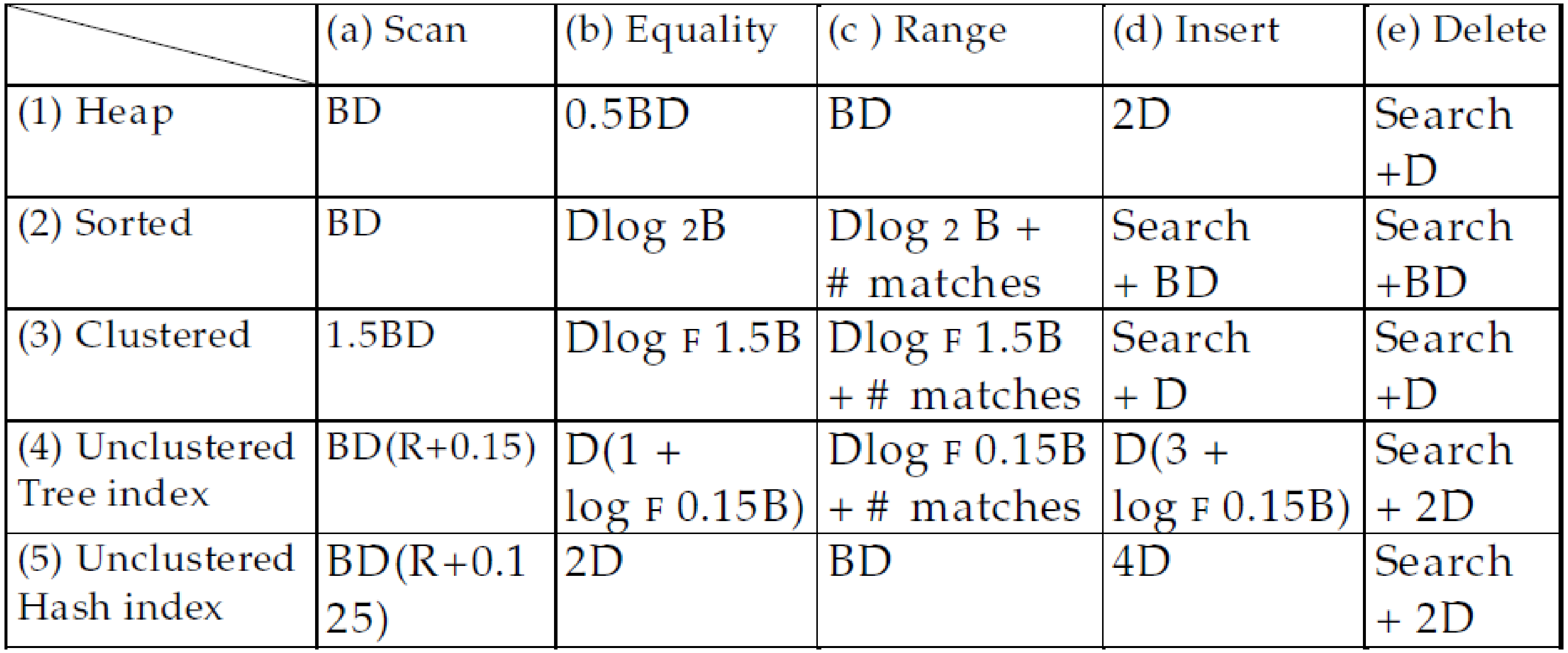
B+ Tree in practice

1. Typical order: 100. Typical fill-factor: 67%. Average fanout = 133
2. Typical capacities:
   - Height 4: 133^4 = 312,900,700 records
   - Height 3: 133^3 = 2,352,637 records
3. 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

Comparing File Organizations

- Considering Employees table:
  1. Heap files (random order; insert at the end)
  2. Sorted files, e.g. sorted on <age, sal>
  3. Clustered B+ tree file, search key <age, sal>
  4. Heap file with an external unclustered B + tree index on search key <age, sal>
  5. Heap file with unclustered hash index on search key <age, sal>

Cost of common operations



	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	$D\log_2 B$	$D\log_2 B$ + # matches	Search + BD	Search +BD
(3) Clustered	1.5BD	$D\log_F 1.5B$	$D\log_F 1.5B$ + # matches	Search + D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	$D(1 + \log_F 0.15B)$	$D\log_F 0.15B$ + # matches	$D(3 + \log_F 0.15B)$	Search + 2D
(5) Unclustered Hash index	BD(R+0.125)	2D	BD	4D	Search + 2D

- B: number of data pages
- D: average time to read a disk page
- R: number of records per page
- Index: 67% occupancy (typical) => file size = 1.5 data size
  Index data-entries plus actual records = BD(R+0.15)
- Hash: 80% page occupancy => file size = 1.25 data size

DB design: choice of indexes (I)

1. What indexes should we create?
   - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
2. For each index, what kind of an index should it be?
   - Clustered? Hash/tree?

DB design: choice of indexes (II)

1. One approach: consider the most important queries in turn. Consider the best query execution plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it
2. Before creating an index, must also consider the impact on updates in the workload!
    - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too

Index selection guide (I)

- Attributes in WHERE clause are candidates for index keys
  - Exact match condition suggests hash index
  - Range query suggests tree index
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates

Index selection guide (II)

- Multi-attribute search keys should be considered when a WHERE clause contains several conditions
  - Order of attributes is important for range queries
  - Such indexes can sometimes enable index-only strategies for important queries
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering

Indexes in xBase

1. Creation
   - INDEX ON key TO index_file[.ndx] [UNIQUE][DESCENDING]
2. Opening
   - USE file INDEX ndx_file_list
   - SET INDEX TO ndx_file
3. Changing active index
   - SET ORDER TO ndx_file
4. Searching
   - SEEK expr, FIND val

Using indexes in Oracle

- Indexes are schema objects that are logically and physically independent of the data in the objects with which they are associated
- Clustering: index-organized table is a table stored in a variation of a B-tree index structure
- Indexes are used automatically by query optimizer module
- PL/SQL statement:

```
CREATE INDEX [UNIQUE] emp_name_dpt_idx
        ON hr.employees(last_name ASC, department_id DESC)
        [COMPUTE STATISTICS]; // statistics are used by query optimizer
DROP INDEX emp_name_dpt_idx;
```

MySQL supported storage engines

1. MyISAM: The default MySQL storage engine. Has no support for clustering, referential integrity (using FK) and transactions
2. InnoDB: A transaction-safe (ACID compliant) storage engine for MySQL that has commit, rollback, and crash-recovery capabilities to protect user data. InnoDB row-level locking (without escalation to coarser granularity locks) and Oracle-style consistent nonlocking reads increase multi-user concurrency and performance. InnoDB stores user data in clustered indexes to reduce I/O for common queries based on primary keys. To maintain data integrity, InnoDB also supports FOREIGN KEY referential-integrity constraints
3. Memory/Heap: Stores all data in RAM for extremely fast access in environments that require quick lookups of reference and other like data
4. NDB: Clustered database engine particularly suited for applications that require the highest possible uptime/availability

Using indexes in MySQL

CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
    [index_type*] ON tbl_name (index_col_name**,...);
    *) index_col_name: col_name [(length)] [ASC|DESC]
    **) index_type: USING {BTREE|HASH}

DROP INDEX emp_name_dpt_idx;

1. UNIQUE index permits multiple NULL values for columns that can contain NULL
2. Full-text indexes can be used only with MyISAM tables and allow: Boolean search for words (~regexp), natural language search and query expression search (for short search phrase – adds noise)
3. Could be created with CREATE TABLE by modifying col. def.:

   {INDEX|KEY} [index_name] [index_type] (index_col_name,...)

1. Supported indexes: MyISAM: BTree, InnoDB: BTree, Memory/Heap: BTree, Hash; NDB: BTree, Hash