Indexes

Kathleen Durant PhD
Northeastern University
CS 3200

Outline for the day

- Index definition
- Types of indexes
  - B+ trees
  - ISAM
  - Hash index
- Choosing indexed fields
- Indexes in InnoDB

Indexes

- A typical file allows us to retrieve records:
  - by specifying a file offset or a *rid*, or
  - by scanning all records sequentially
- Sometimes, we want to **retrieve records by specifying the *values in one or more fields***
  - ***Examples:***
  - Find all students in the "CS" department
  - Find all students with a gpa > 3
- Indexes are file structures that enable us to answer such value-based queries efficiently.

Indexes

- 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 a key in the DB
- An index contains a collection of data entries, and supports efficient retrieval of all data entries with a given key value k.

Why Index?

- Database tables have many records and ..
  - linear search is very slow, complexity is O(n)
  - Keeping a file sorted to apply a binary search is costly
- Indexes improve search performance
  - But add extra cost to INSERT/UPDATE/ DELETE
- Many options for indexes
  - Hash Indexes (MEMORY and NDB)
  - Bitmap Indexes (not available in MySQL)
  - B-Tree Indexes and derivatives (MyISAM, InnoDB)

Index Concept

- Main idea: ***A separate data structure used to locate records***
- Most generally, index is a list of value/address pairs
  - Each pair is an index “entry”
  - Value is the index “key”
  - Address will point to a data record, or to a data page
  - The assumption is that the value/address pair will be much smaller in size than the full record
- If index is small, a copy can be maintained in memory
  - Permanent disk copy is still needed

Indexing Pitfalls

- Index itself is a data store
  - Occupies disk space
  - Must worry about maintenance, consistency, recovery, etc.
- Large indices won't fit in memory
  - May require multiple seeks to locate record entry

Essential for Multilevel Indexes

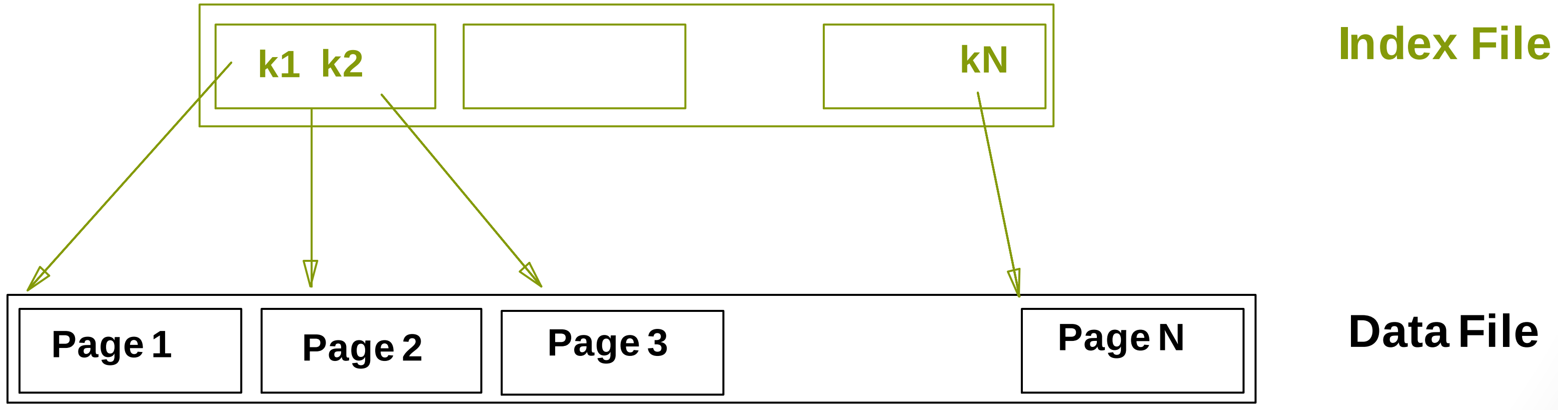
- Should support efficient random access
  - Should also support efficient sequential access, if possible
- Should have low height
  - Implies high fan out: refers to the number of children nodes for an internal node.
- Should be efficiently updatable
- Should be storage-efficient
- Top level(s) should fit in memory

Tree Structured Indexes

- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- Tree structures with search keys on *value-based domains*
  - *ISAM*: static structure
  - *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

Range Searches

- ``*Find all students with gpa > 3.0*''
    - If data is in a sorted file, do binary search to find first such student, then scan to find others.
    - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.



➢ *Can do a binary search on (smaller) index file!*

ISAM

- = Indexed Sequential Access Method
  - IBM terminology
  - “Indexed Sequential” more general term (non-IBM)
  - ISAM as described in textbook is very close to B+ tree
    - simpler versions exist
- Main idea: ***maintain sequential ordered file but give it an index***
  - Sequentiality for efficient “batch” processing
  - Index for random record access

ISAM Technique

- Build a dense index of the pages (1st level index)
  - Sparse from a record viewpoint
- Then build an index of the 1st level index (2nd level index)
- Continue recursively until top level index fits on 1 page
- Some implementations may stop after a fixed # of levels

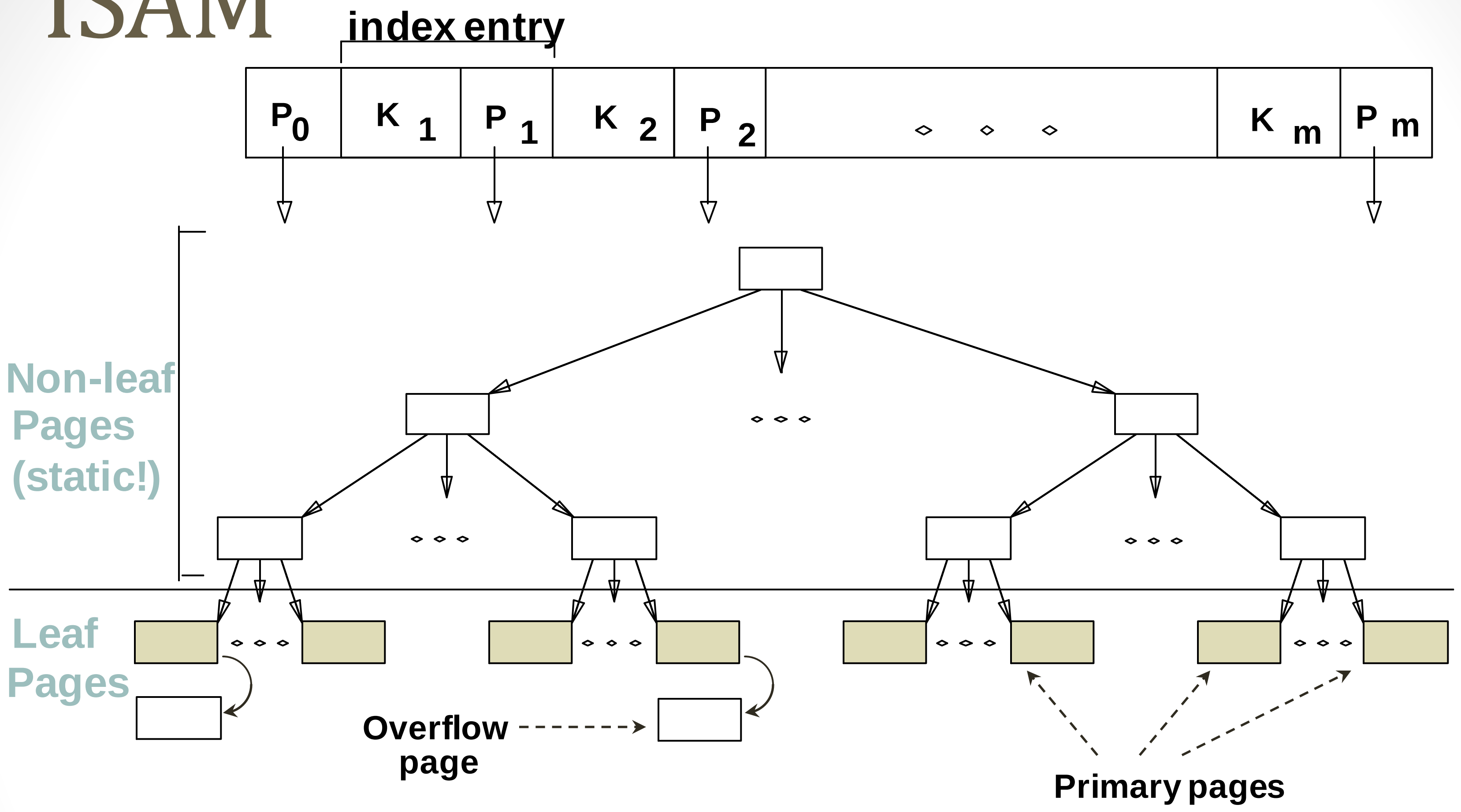
Updating an ISAM File

- Data set must be kept sequential
  - So that it can be processed without the index
  - May have to rewrite entire file to add records
  - Could use overflow pages
    - chained together or in fixed locations (overflow area)
- Index is usually NOT updated as records are added or deleted
- Once in a while the whole thing is "reorganized"
  - Data pages recopied to eliminate overflows
  - Index recreated

ISAM Pros, Cons

- Pro
  - Relatively simple
  - Great for true sequential access
- Cons
  - Not very dynamic
  - Inefficient if lots of overflow pages
  - Can only be one ISAM index per file

ISAM

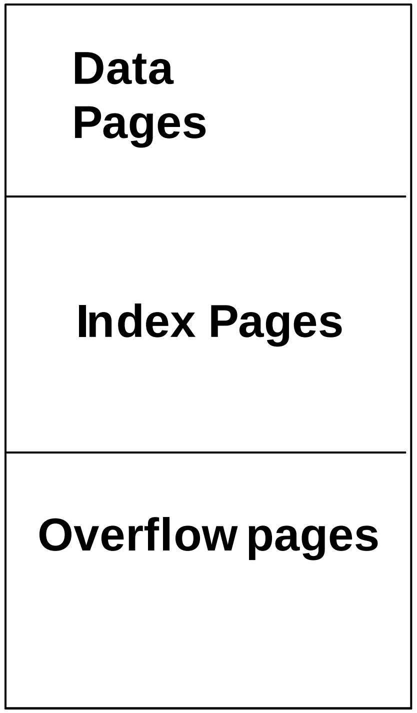


- Leaf pages contain sorted data records
- Non-leaf part directs searches to the data records; static once built
- Inserts/deletes: use overflow pages, bad for frequent inserts.

Comments on ISAM

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

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

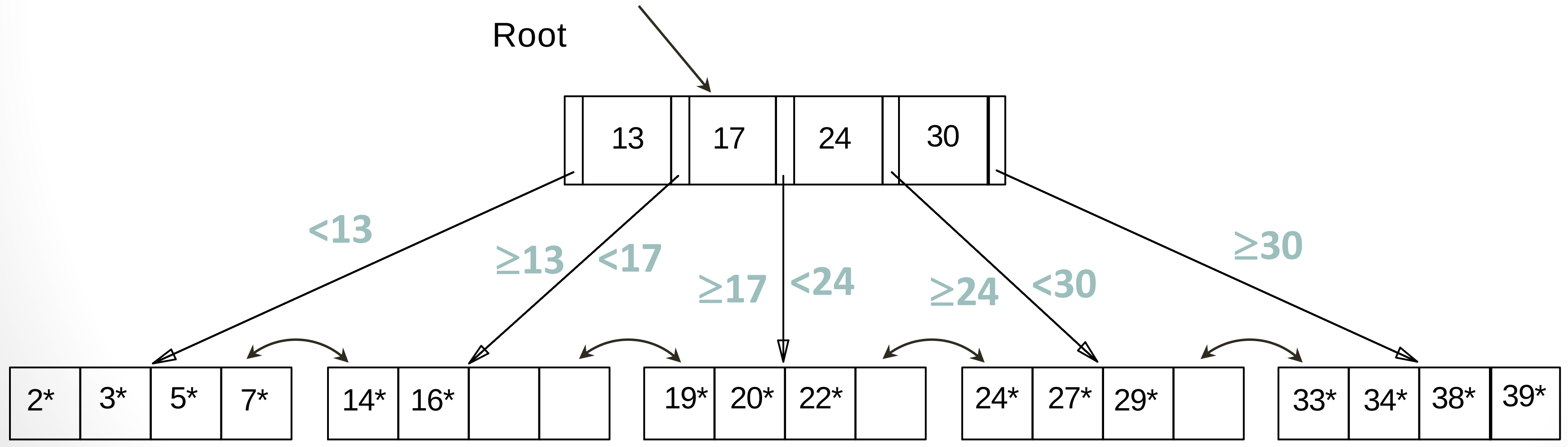


Definition of B+ tree

- A B-tree of order *n* is a height-balanced tree , where each node may have up to *n* children, and in which:
  - All leaves (leaf nodes) are on the same level
  - No node can contain more than *n* children
  - All nodes except the root have at least n/2 children
  - The root is either a leaf node, or it has at least n/2 children
- Ensures that a fixed maximum number of reads would be required to access any data requested, based on the height of the tree

Example B+ Tree

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

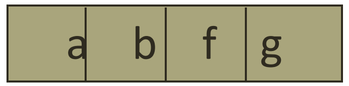


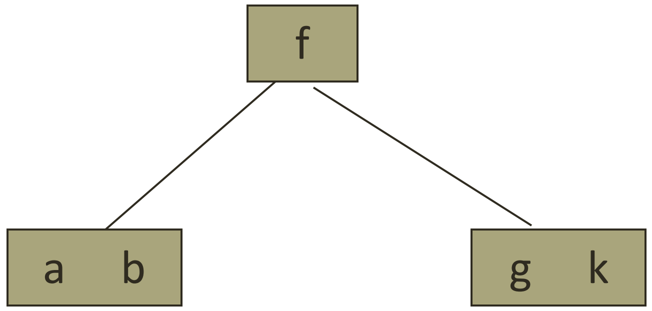
B+ Trees in Practice

- Typical order: 200. Typical fill-factor: 67%.
  - Average fan-out for internal nodes = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 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

Insertion in B-Tree

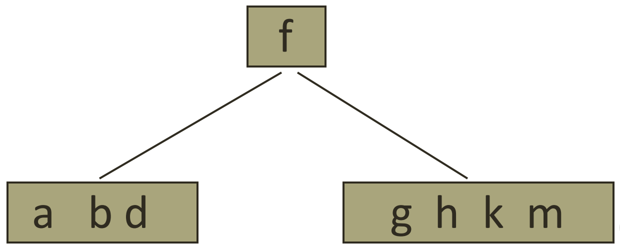
- 1. 2.
- a, g, f,b: k:

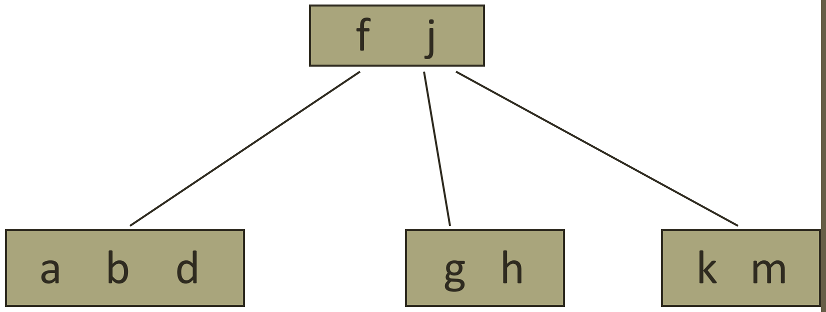




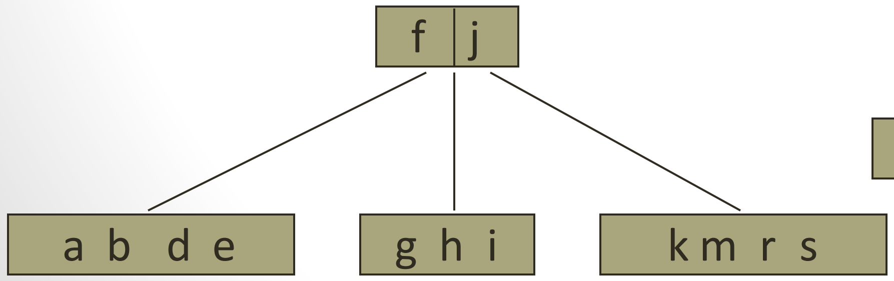
Insertion (cont.)

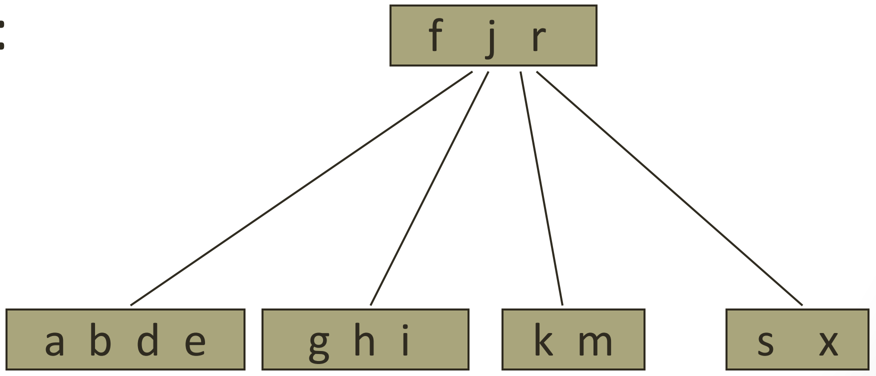
- 3. 4.
- d, h, m: j:





- 5. 6.
- e, s, i, r: x:
-

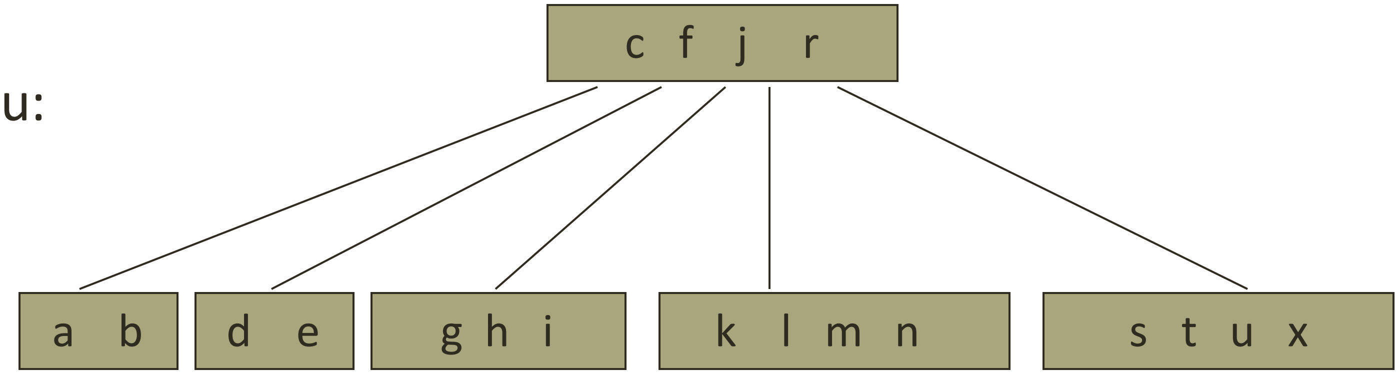




Insertion (cont.)

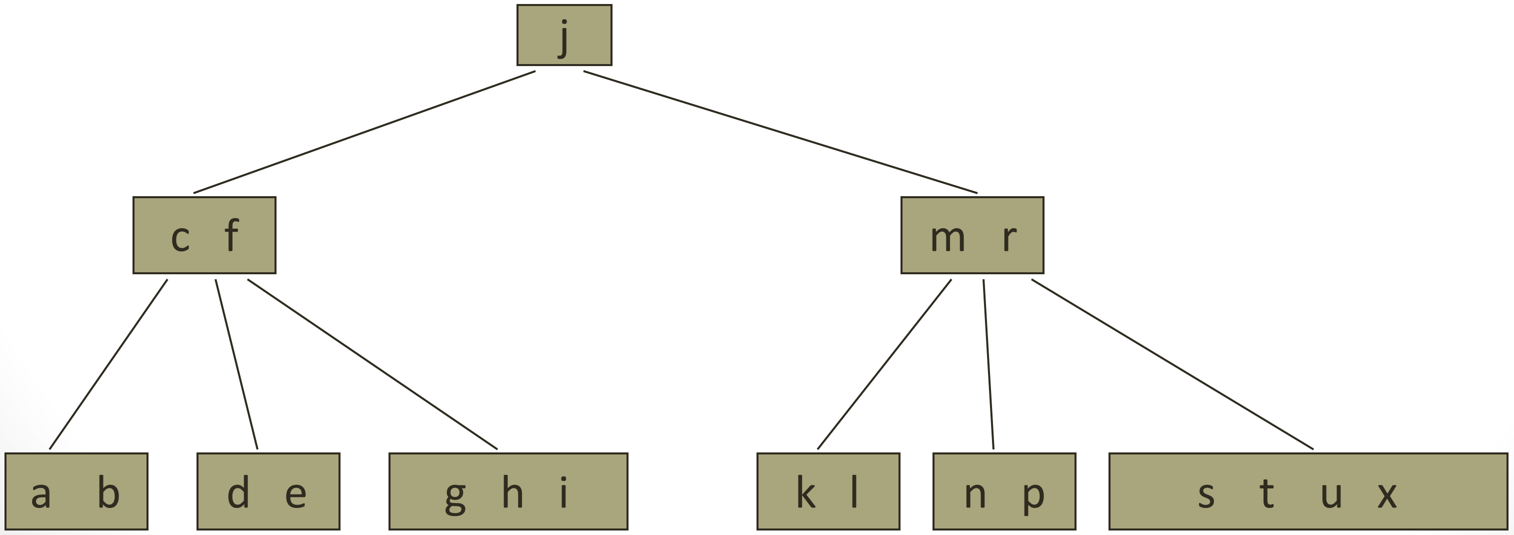
7.

c, l, n, t, u:



8.

p:



Summary: B+ trees

- Typically, 67% occupancy on average.
- Usually preferable to ISAM, modulo *locking* considerations; adjusts to growth gracefully.
- Key compression increases fan-out, reduces height.
- Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.

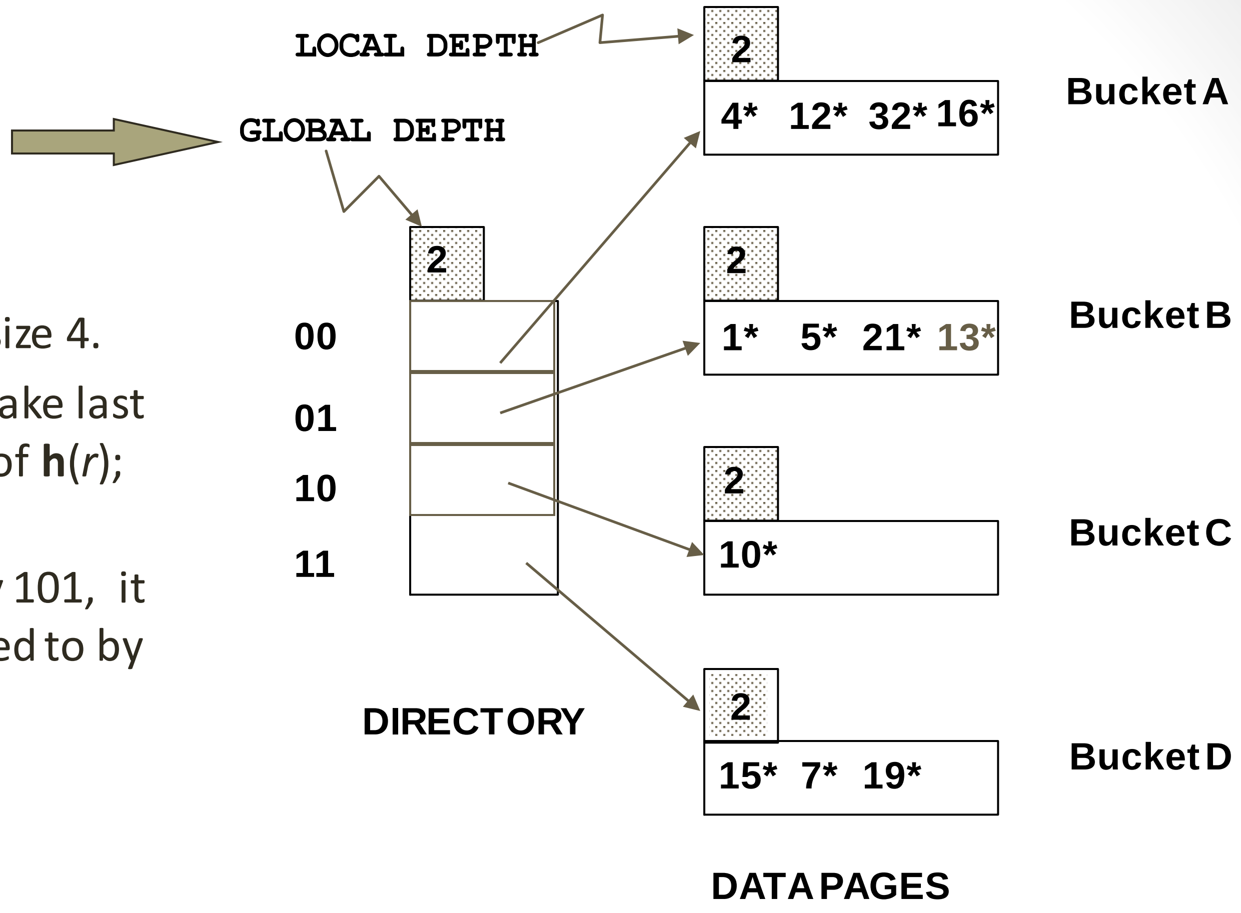
Hashing mechanism

- Your index is a collection of *buckets* (bucket = page)
- Define a hash function, *h*, that maps a key to a bucket.
- Store the corresponding data in that bucket.
- Collisions
  - Multiple keys hash to the same bucket.
  - Store multiple keys in the same bucket.
- What do you do when buckets fill?
  - Chaining: link new pages(overflow pages) off the bucket.

Extendible Hashing

- **Main Idea:** Use a directory of (logical) pointers to bucket pages
- 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!

Example

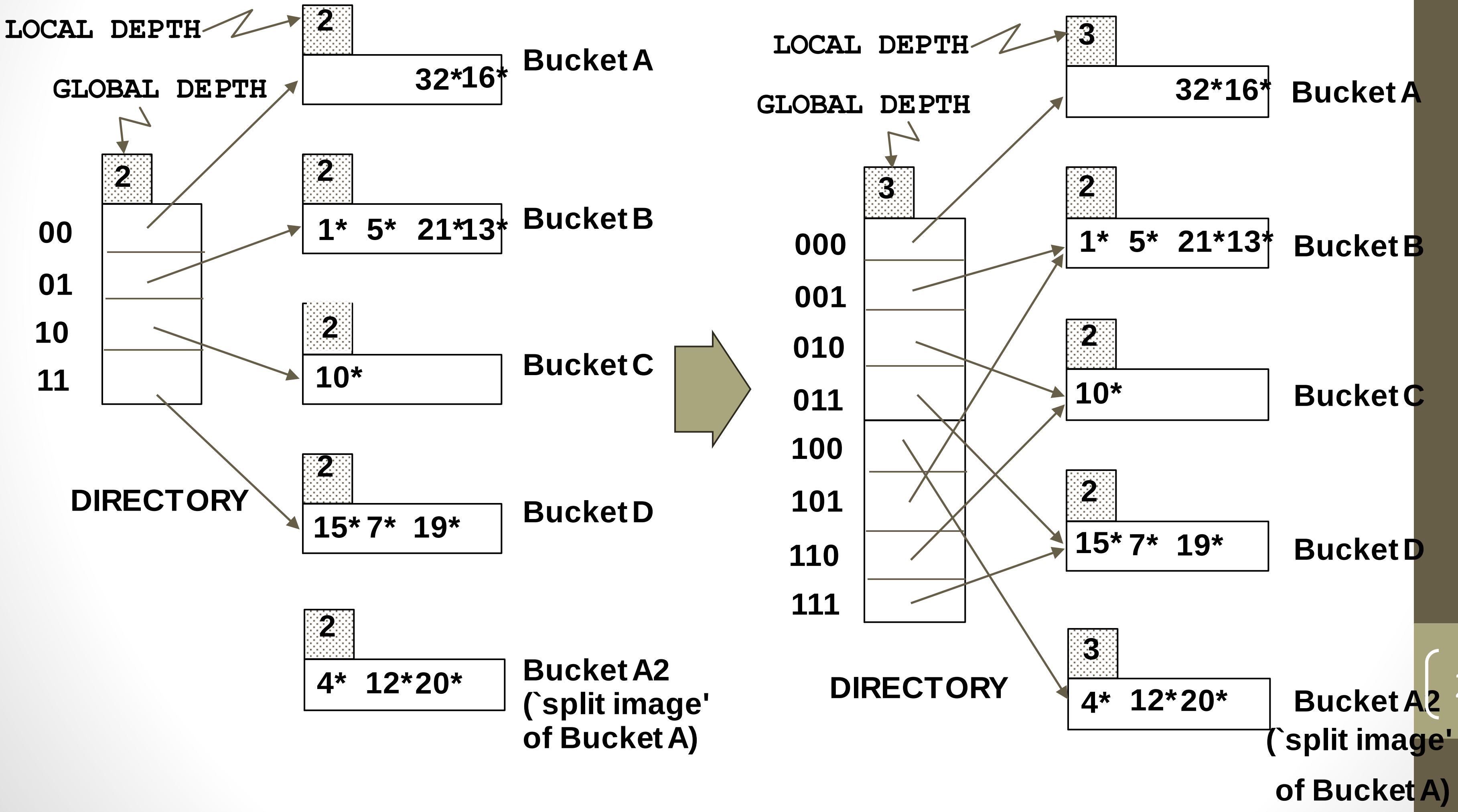


- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).

❖ *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Insert h(r)=20 (Causes Doubling)

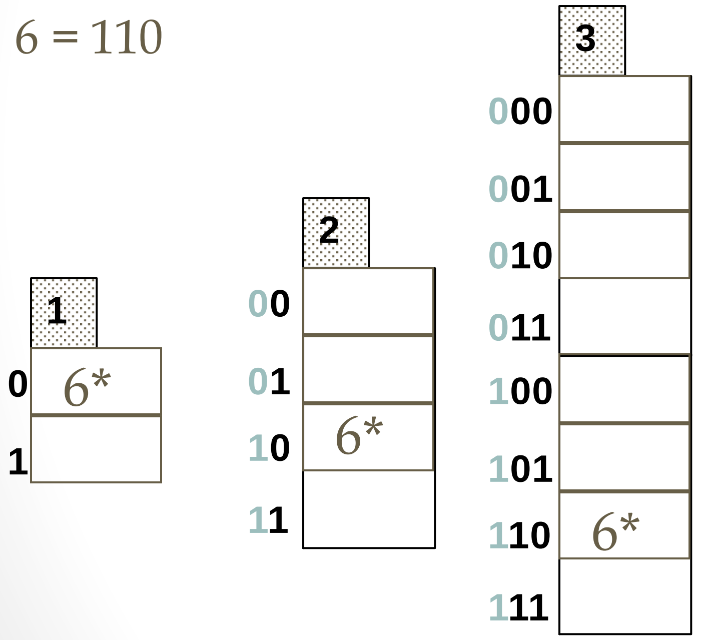


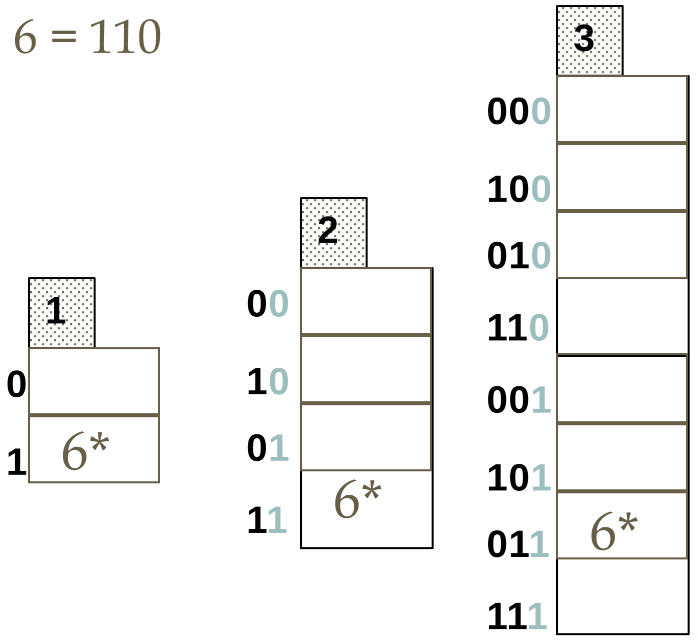
Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **<u>3</u>** bits needed to tell which.
  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *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!)

Directory Doubling

Why use least significant bits in directory?
⇔ Allows for doubling via copying!





Least Significant vs. Most Significant

Comments on Extendible Hashing

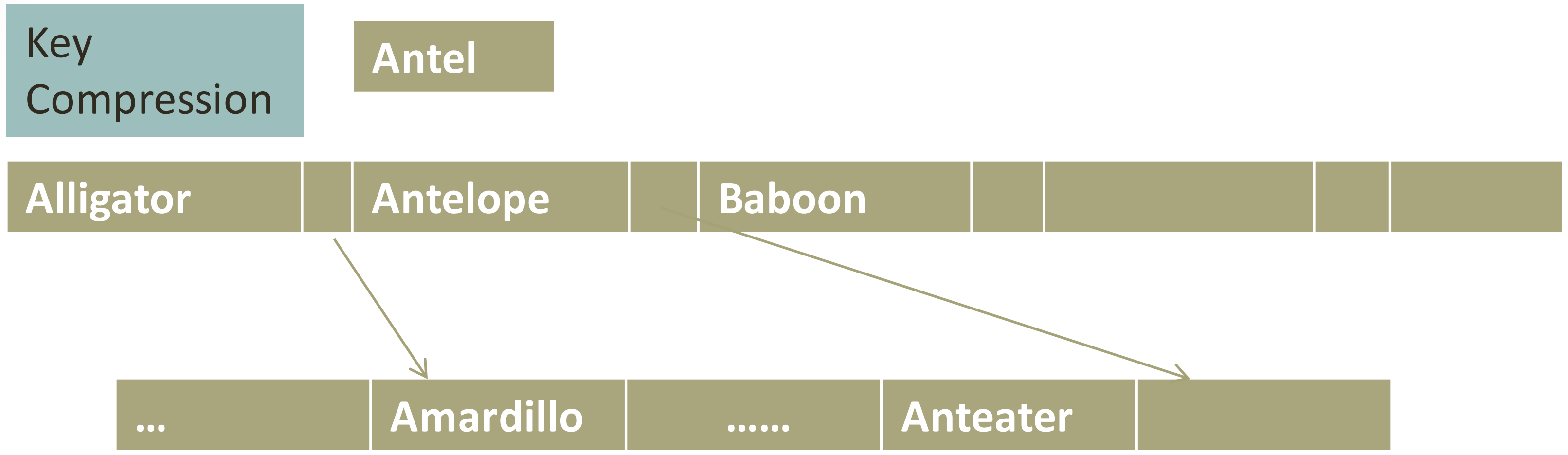
- If directory fits in memory, equality search answered with one disk access; else two.
  - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems
    - Need a decent hash function
- **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.

Prefix Key Compression

- Height of a B+ tree depends on the number of data entries and the size of index entries
  - Size of index entries determines the number of index entries that will fit on a page – and therefore the fan-out of the tree.
- Key Compression can increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., adjacent index entries with search key values

    [*Dave Jones*, *David Smith* and *Devarakonda Murthy*]
  - Can we abbreviate *David Smith* to *Dav*?
    - Not correct! Can only compress *David Smith* to *Davi*.
    - In general, while compressing, must <u>leave each index entry greater than every key value (in any subtree) to its left</u>.
- Insert/delete must be suitably modified.

Prefix Key compression



Creating Secondary Indexes

- You can create other indexes on your data
  - These secondary indexes do not need to have a UNIQUE values for each record
  - Typically created on foreign keys to speed up JOIN operation
- Add a secondary index after the table has been populated when the data and the primary index exists within the database
  - (ALTER TABLE … ADD INDEX)
  - Process can use the primary index to locate the records to be indexed by the secondary index

Process: Choice of indexes

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

Index selection guideline

- 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.
- 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: when only indexed attributes are needed.
    - For index-only strategies, clustering is not important.
- Try to choose indexes that benefit many queries.
  - Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

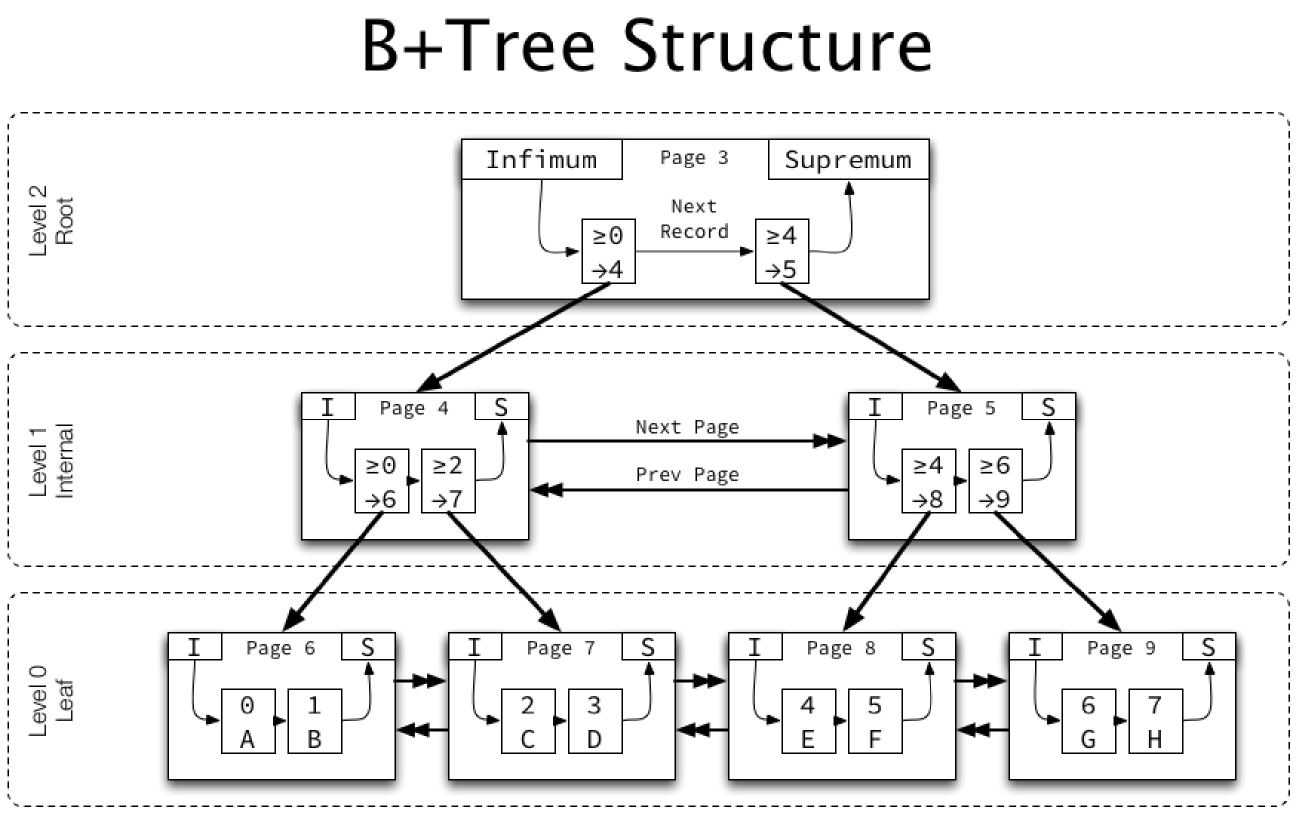
Indexes in InnoDB

- Every InnoDB table has a special index called the clustered index (by default built on the primary key)
- Record locks always lock index records
  - Even if a table is defined with no indexes
  - InnoDB creates a hidden clustered index and uses this index for record locking
- Accessing a row through the clustered index is fast because the index search leads directly to the page with all the row data.
- All InnoDB indexes are B+ trees where the index records are stored in the leaf pages of the tree
- You can configure the page size for all InnoDB tablespaces in a MySQL instance with the variable innodb_page_size
  - default size of an index page is 16KB

InnoDB Index structure

- Root page is allocated when the INDEX is created and is stored in the data dictionary
  - It can never be relocated or removed
- All pages at each level are double-linked to each other
- All pages have anchors for the beginning and the end of the linked list of records
  - Statically defined: Infimum – lowest key, supremum – highest key
- Within a page, records are singly-linked in ascending order
  - Records not stored in ascending order
- Non-leaf pages contain page addresses to a child node
- Leaf pages contain the actual data record (non-key data)

Tree Levels in InnoDB



Levels are numbered starting from 0 at the leaf pages, incrementing up the tree.
Pages on each level are doubly-linked with previous and next pointers in ascending order by key.
Records within a page are singly-linked with a next pointer in ascending order by key.
Infimum represents a value lower than any key on the page, and is always the first record in the singly-linked list of records.
Supremum represents a value higher than any key on the page, and is always the last record in the singly-linked list of records.
Non-leaf pages contain the minimum key of the child page and the child page number, called a "node pointer".

http://blog.jcole.us/2013/01/10/btree-index-structures-in-innodb/

InnoDB: Secondary Index

- You can create multiple indexes on a table
  - These additional indexes that are not on the primary key are secondary indexes
- Each record in a secondary index contains the primary key columns for the row, as well as the columns specified for the secondary index
- InnoDB uses this primary key value to search for the row in the clustered index

Index Optimizations in InnoDB

- The change buffer is a special data structure that caches changes to secondary index pages when affected pages are not in the buffer pool
- The buffered changes are merged later when the pages are loaded into the buffer pool by other read operations.
- secondary indexes are usually non-unique, and inserts into secondary indexes happen in a relatively random order.
- Merging cached changes at a later time, when affected pages are read into the buffer pool by other operations, avoids substantial random access I/O that would be required to read-in secondary index pages from disk
- Periodically, the purge operation that runs when the system is mostly idle, writes the updated index pages to disk
- The purge operation can write disk blocks for a series of index values more efficiently than if each value were written to disk immediately

InnoDB locks

- My SQL sets record locks on every index record that is scanned in the processing of a SQL statement
- Types of object locks
  - Record lock: This is a lock on an index record.
  - Gap lock: This is a lock on a gap between index records, or a lock on the gap before the first or after the last index record.
  - Next-key lock: This is a combination of a record lock on the index record and a gap lock on the gap before the index record.
- InnoDB uses next-key locks for searches and index scans
- If one session has a shared or exclusive lock on record R in an index, another session cannot insert a new index record in the gap immediately before R in the index order

InnoDB hash indexes

- Based on the observed pattern of searches, MySQL builds a hash index using a prefix of the index key.
  - Hash indexes are built on demand for those pages of the index that are often accessed.
- The prefix of the key can be any length, and it may be that only some of the values in the B+tree appear in the hash index.
- If a table fits almost entirely in main memory, a hash index can speed up queries by enabling direct lookup of any element, turning the index value into a sort of in memory pointer.

Hash index limitations

- They are used only for equality comparisons
  - They cannot be used for comparison operators such as < that find a range of values.
- The optimizer cannot use a hash index to speed up ORDER BY operations. (This type of index cannot be used to search for the next entry in order.)
- MySQL cannot determine approximately how many rows there are between two values (this is used by the range optimizer to decide which index to use).
- Only whole keys can be used to search for a row. (With a B+-tree index, any leftmost prefix of the key can be used to find rows.)

Summary: Tree-based Index

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure.
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
  - High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.
- InnoDB provides many optimizations to speed up the access to a record