Comp 5311 Database Management Systems

9. File Structures and Indexing

Storage Hierarchy

- Typical storage hierarchy:
  - Main memory (RAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).
- Why Not Store Everything in Main Memory?
- *Costs too much*.
- *Main memory is volatile*. We want data to be saved between runs. (Obviously!)

Storage Access

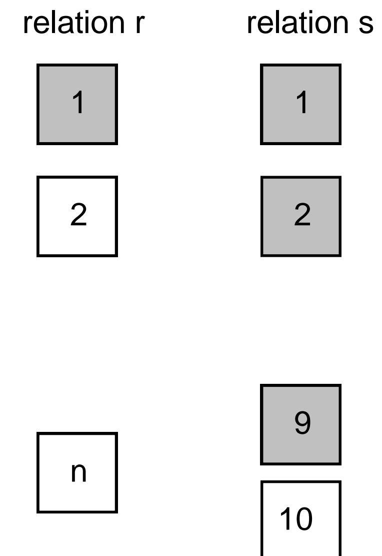
- A database file is partitioned into fixed-length storage units called **blocks (or pages)**. Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping 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.
  - If the block is already in the buffer, the requesting program is given the address of the block in main memory
  - 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 the block from the disk to the buffer, and passes the address of the block in main memory to requester.

Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
- LRU can be a bad strategy for certain access patterns involving repeated scans of data, e.g. when computing the join of 2 relations r and s by a nested loops
- Best for this example: most recently used (MRU) strategy – replace the most recently used block.
- The DBMS usually has its own buffer manager that uses statistical information regarding the probability that a request will reference a particular relation



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.
- Most common approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations
  - This case is easiest to implement; will consider variable length records later.

Fixed-Length Records

- Simple approach:
  - In each page/block, store record *i* starting from byte $n * (i - 1)$, where *n* is the size of each record.
  - Record access is simple but records may cross blocks. Normally, do not allow records to cross block boundaries (there is some empty space at the end of the page)
- Deletion of record *I:*
- Shift up subsequent records
- Moving records inside a page not good when records are pointed by:

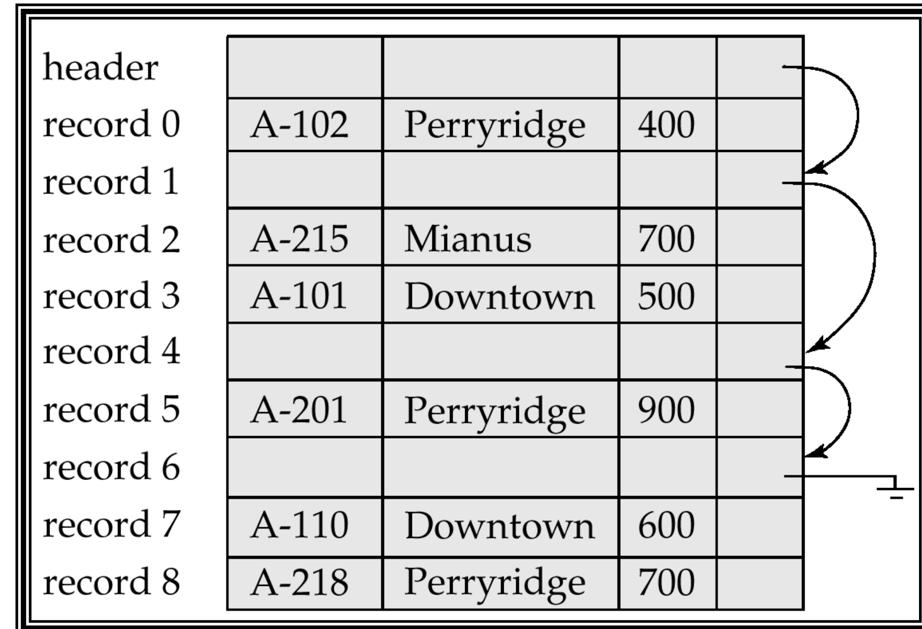
  1] other records (foreign keys)

  2] index entries

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Fixed-Length Records - Free Lists

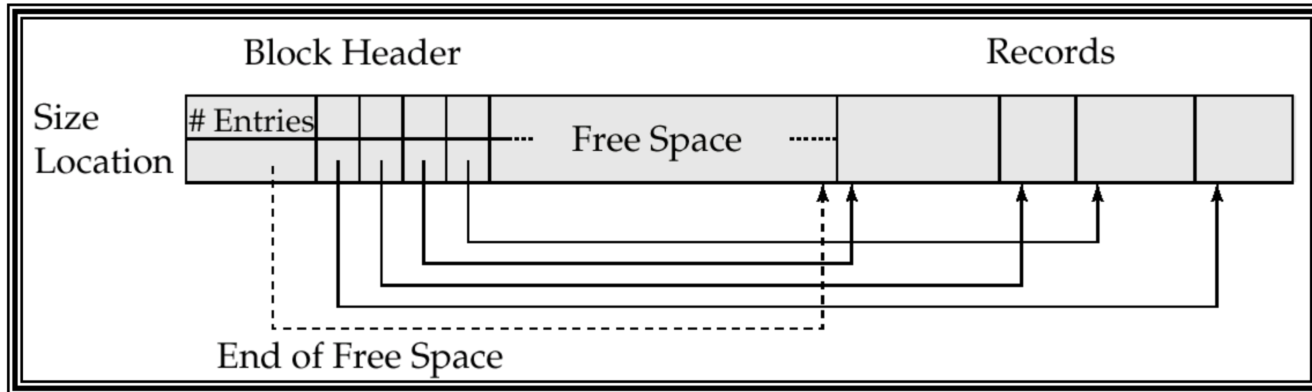
- Do not move records in page. Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)



Variable-Length Records
Byte String Representation

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields (used in some older data models).
- Simple (but bad) solution: Byte string representation
  - Attach an *end-of-record* (⊥) control character to the end of each record
  - Difficulty with deletion (fragmentation of free space)
  - Difficulty with growth (movement of records is difficult).

Variable-Length Records
Slotted Page Structure



- Slotted page header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers do not point directly to record — instead they point to the entry for the record in header.

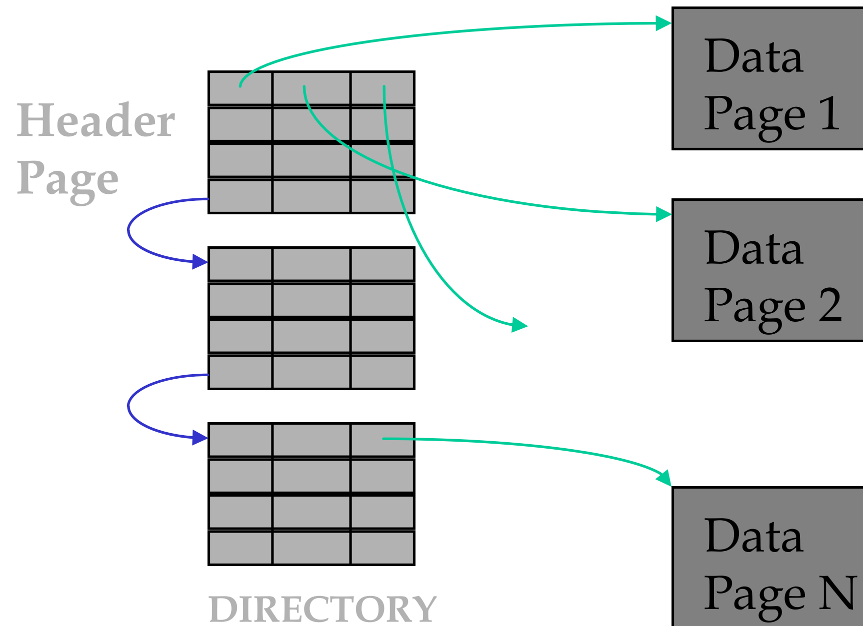
Organization of Records in Files

- **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 computed on some attribute of each record; the result specifies in which block of the file the record should be placed

Unordered (Heap) Files

- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* in pages
  - keep track of the *records* in a page
- There are many alternatives for keeping track of this.

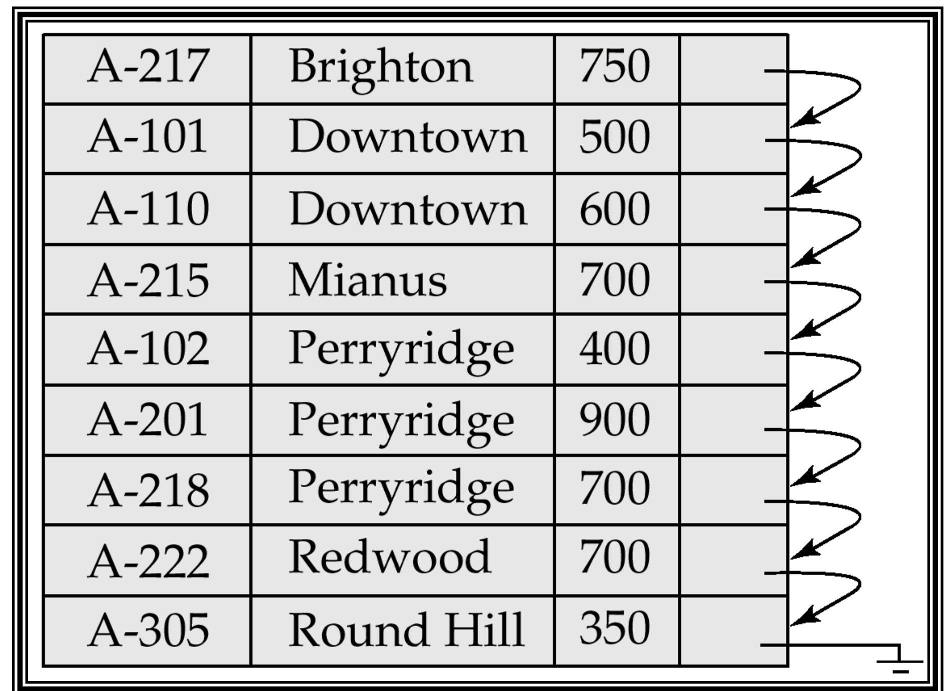
Heap File Using a Page Directory



- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.

Sequential File Organization

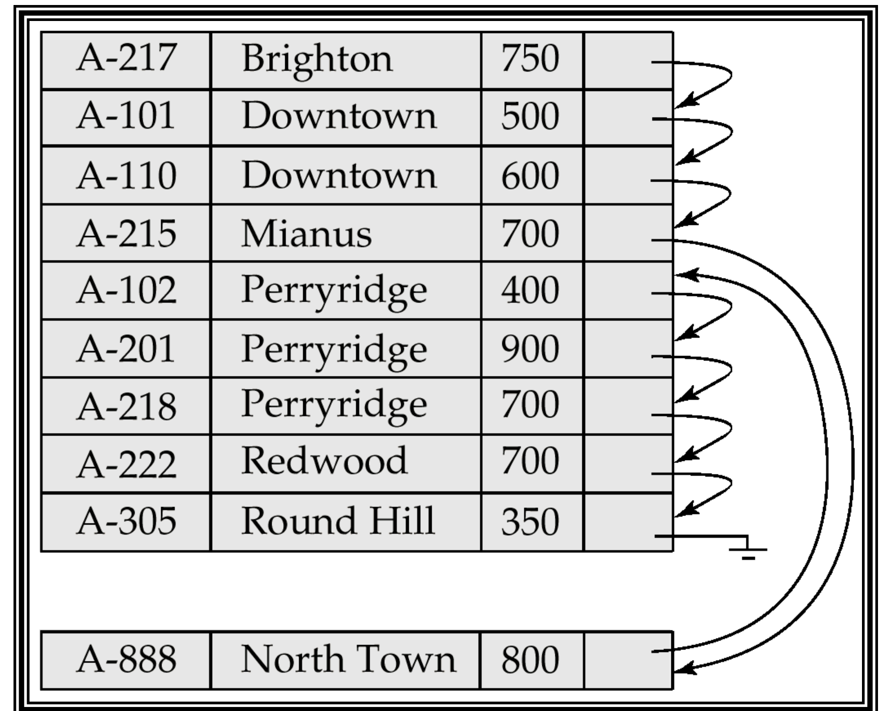
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key (not the same concept a key in the relational model)



A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Sequential File Organization (Cont.)

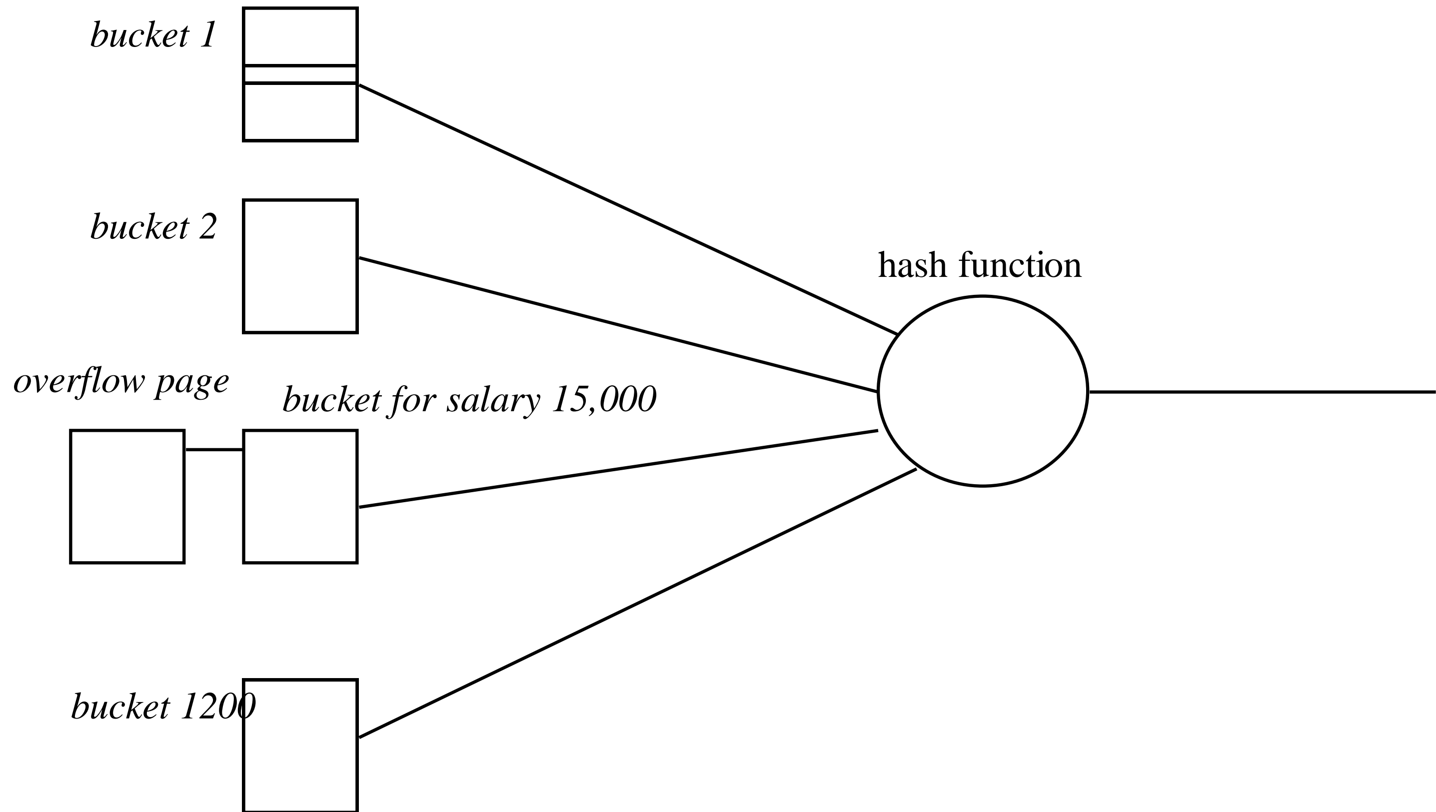
- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



Hashing as a file organization

- Assume 100,000 employee records – we can put 100 records per page. Therefore, we need 1,000 pages.
- Lets say that we want to organize the file so that we can efficiently answer equality selections on salary e.g., "find all employee records whose salary is 15,000".
- We allocate 1,200 buckets (pages) so there is some space for future insertions.
- The hash function will have the form h(salary)=(a*salary+b) modulo 1,200.

Hashing as a file organization (cont)



- When we *insert* a new record we compute the hash function of the salary and insert the record in the appropriate bucket. If the bucket is full we create an *overflow* page/bucket and insert the new record there.

Hashing as a file organization (cont)

- This organization can efficiently answer queries of the form: "*find all employees whose salary is 15,000*".
- In order to answer this query we compute the hash value of 15,000 and then search only in the corresponding bucket.
- A bucket may contain records of employees with different salaries that produce the same hash value (e.g., employees with salary 30,000). This is not a problem because since we read the page, we can check all its records and select only the ones that satisfy the query condition.
- If there are no overflow buckets, answering the query requires just a single read.
- If there are overflow buckets, we have to read all of them.
- Hashing is not good for range search ("find all employees with salaries between 15,000 and 16,000") because the records of these employees maybe distributed in totally different buckets.

Simplistic Analysis

- We ignore CPU costs, for simplicity:
  - **B:** Is the number of data pages in the file
  - Measuring number of page I/O's ignores gains of pre-fetching blocks of pages; thus, even I/O cost is only approximated.
  - Average-case analysis; based on several simplistic assumptions:
- Single record insert and delete.
- Heap Files:
  - Equality selection on key; exactly one match.
  - Insert always at end of file.
- Sorted Files:
  - Files compacted after deletions.
  - Selections on sort field(s).
- Hashed Files:
  - No overflow buckets, 80% page occupancy.

Cost of Operations

	Heap File	Sorted File	Hashed File
Scan all recs	**B**	**B**	**1.25 B**
Equality Search	**0.5 B**	**$\log_2 B$**	**1**
Range Search	**B**	**$\log_2 B$ + # of pages with matches)**	**1.25 B**
Insert	**2**	**Search + B**	**2**
Delete	**Search + 1**	**Search + B**	**2**

Several assumptions underlie these (rough) estimates!

Introduction to Indexing

1. Assume that you work in a government office, and you maintain the records of 10 million HK residents
2. The record of each resident contains the HK ID, name, address, telephone etc.
3. People come to your office, and ask you to retrieve the records of persons given their HK ID (i.e., "show me the record of person with HK ID: 5634569")
4. Lets forget computers for now. You just want to print the records in a catalog, so you answer these queries by manually looking up the catalog
   - Assuming that you can put 10 records per printed page, the catalog will be 1 million pages

Introduction to Indexing (2)

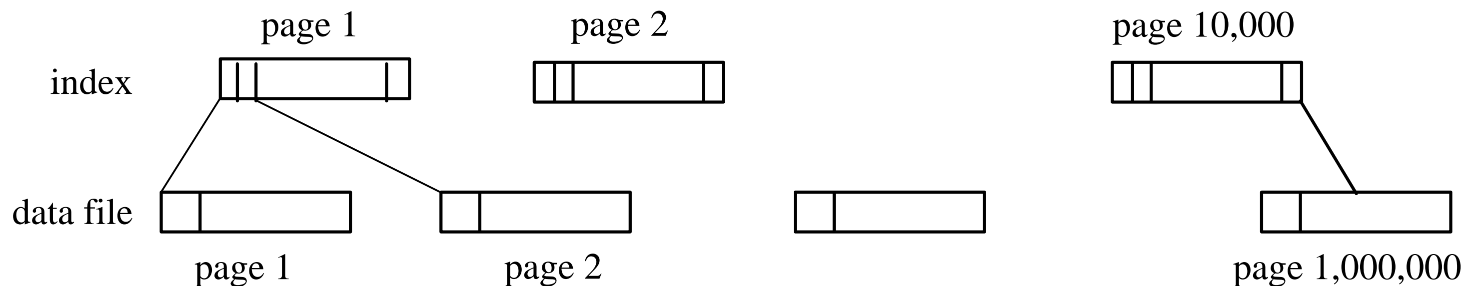
- How would you arrange the records in the catalog?
  - Your goal is to minimize the effort of finding records
  - We measure this cost as the number of pages you have to "open" before finding the record
- Solution 1 - random order (e.g., heap file organization)
  - If the catalog contains records in random order of HK ID, then in the worst case you have to search the entire catalog (cost = 10^6) before finding a record, or to determine that the HK ID does not exist in the catalog
- Solution 2 - records sorted on HK ID (e.g., sequential file)
  - You can apply binary search with cost $\lceil \log_2 10^6 \rceil = 20$
- Same discussion applies when we use computers; instead of the printed pages, we have disk pages (e.g., size 8 KB)
  - Every time we read something from the disk, we need to bring an entire page in main memory. The major cost is how many pages we read because disk operations are much more expensive than CPU operations
- Can you make it even faster?

Introduction to Indexing (3)

- Lets keep the sorted file and build an additional index (e.g., at the beginning of the catalog).
  - Each index entry is a small record, that contains a HK ID and the page where you can find this ID (e.g., <5634569, 259> means that HK ID 5634569 is on page 259 of the catalog.
  - HK ID is called the search key of the index
  - Since each index entry is much smaller than the record, lets assume that we can fit 100 entries per page.
  - The index entries are also sorted on HK ID
- Do we need an entry for each of the 10,000,000 records?
  - No: we only need an entry for the first record of each page
    - Example: If I have two consecutive entries <5634569, 259>, <5700000, 260> in the index, then I know that every HK ID between 5634569 and 5700000 must be on page 259.
  - Therefore, I only need only 1,000,000 index entries (one for each page of the main catalog).

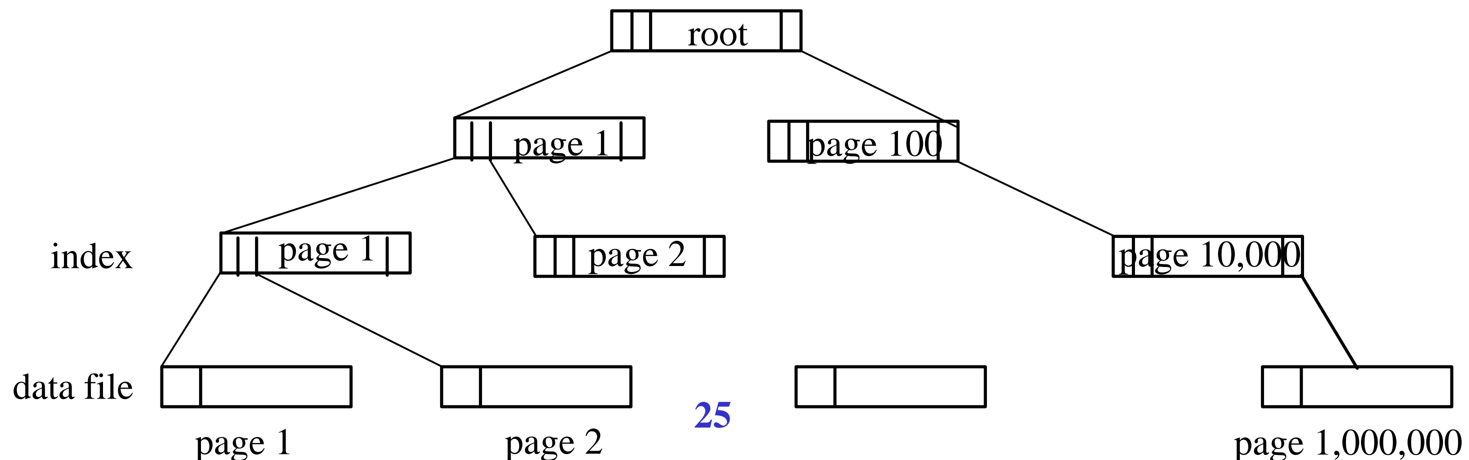
Introduction to Indexing (4)

- Given that I can fit 100 entries per page, and I have 1,000,000 entries, my index is 10,000 pages.
- How I can use the index to speed up search for a record?
  - Use binary search on the index to find the largest HK ID that is smaller or equal to input HK ID. The cost is $\lceil \log_2 10^4 \rceil = 14$.
  - Then, follow the pointer from that entry to the actual catalog (cost 1)
  - Total cost: 14+1 = 15



Introduction to Indexing (5)

- Can I drop the cost even further?
- Yes: Build an index on the index
  - The second level index contains 10,000 entries (one for each page of the first index) in 100 pages.
  - Use binary search on the second level index to find the largest HK ID that is smaller or equal to input HK ID. The cost is $\lceil \log_2 10^2 \rceil = 7$.
  - Then, follow the pointer from that entry to first level index and finally to the actual catalog (cost 2)
  - Total cost: 7+2 = 9
- Finally build a third level index containing 100 entries, one for each page of the second level index.
  - These entries fit in one page. Read this page - find the largest HK ID that is smaller or equal to input HK ID and follow the pointers to second level index, first level index and file.
  - Total cost: 4



Basic Concepts of Indexing

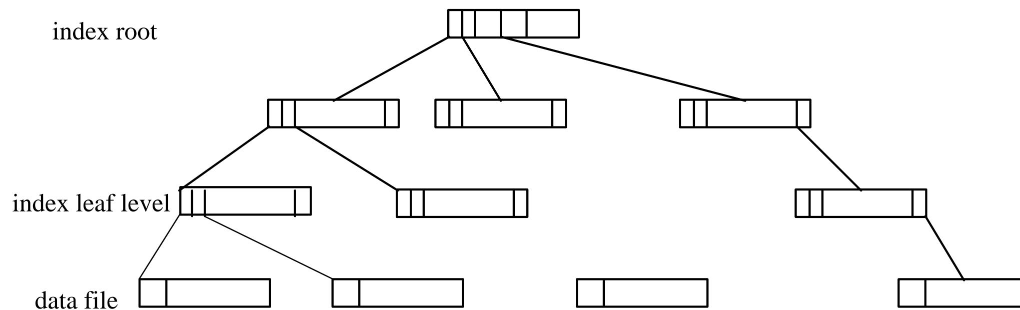
- Indexes speed up access to desired data.
- **Search Key** - attribute to set of attributes used to look up records in a file.
  - Not to be confused with the concept primary or candidate key
  - In the previous slides the search key was HKID - we can find records given the HKID.
  - If we want to find records given the name (or another attribute), we need to build additional indexes
- An **index file** consists of records (called **index entries**) of the form <search key, pointer>
  - Index files are typically much smaller than the original file (they skip most attributes)

Ordered Indexes

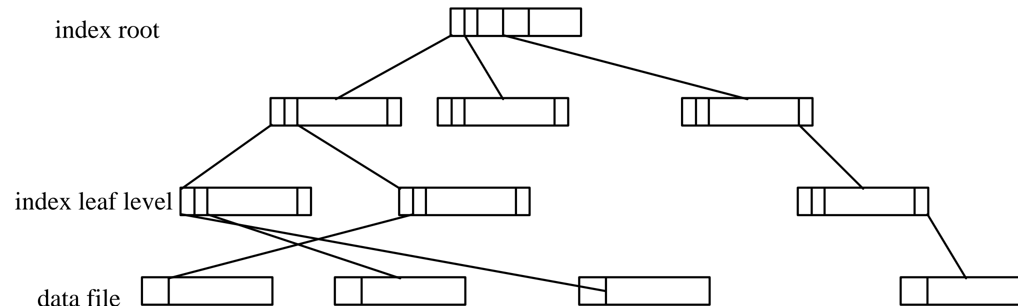
- The index that we built is an ordered - also called tree – index
  - Because the index entries are sorted on the search key (e.g., HKID)
  - There are other types of indexes (e.g., hash indexes, bitmaps)
  - Good for equality and range search
- When looking for a record, we always start from the root, and follow a single path to the leaf that contains the search key of the record. Then, we perform an additional access to get the record from the data file
  - The cost, in terms of page accesses, is the height of the tree (number of levels) plus 1
- An index page is also called index node
- The number of children (pointers) of a node is called the fanout.
  - In our example, the fanout is 100
- The height of the tree is $\lceil \log_{fanout}(\#\text{index entries}) \rceil$
  - In our example, the height is $\lceil \log_{100}(10^6) \rceil = 3$

Primary vs. Secondary Indexes

- **Primary index** (also called **clustering index**)**:** when the file is sorted on search key of the index (e.g., our index on HKID)
  - Index-sequential file**:** ordered sequential file with a primary index (also called ISAM - indexed sequential access method).



- **Secondary index** (also called **non-clustering index**)**:** when the file is not sorted on search key of the index



Sparse vs. Dense Indexes

- Sparse Index: contains index records for only some search-key values.
  - Only applicable to primary indexes
  - In our HKID index, we only have index entries for the first record in each page of the file
  - In general, we have an index entry for every file page, corresponding to the minimum search-key value in the page.
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value <= *K*
  - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- An index that has an entry for every search key value is called dense.

Example of Secondary Dense Index

- Continuing the previous example, assume again the file with the 10 million records and the primary index on HKID
- In addition to HKID, you want to be able to find records given the name.
  - Somebody, gives you the name of a person and wants his/her record. How to find the record fast?
- Answer: build another index on the name
  - Since the file is sorted on the HKID, the new index must be secondary and dense.
  - Assuming that all names are distinct, your index will contain 10 million entries.
  - Assuming that the fanout is again 100, cost of finding a record given the name is $\lceil \log_{100}(10^7) \rceil$
- Secondary index almost as good as primary index (in terms of cost) when retrieving a single record
  - However, maybe very expensive when retrieving many records (e.g., for range queries). More on this in subsequent classes.

Ordered Indexes on Non-Candidate Keys

- Assume that I want to build an index on the name, but there may be several people with the same name
- If the index is primary and sparse, this is not a problem. Otherwise, there are three options.
- Option 1: use variable length index entries
  - Each entry contains a name, and pointers to all records with this name
  - Example: <Qiong Luo, pnt1, pnt2, …., pntn>
  - Problem: complicated implementation as it needs a file organization that supports records of variable length.
- Option 2: use multiple index entries per name
  - There is an entry for every person, if he/she shares the same name with other people
  - Example: <Qiong Luo, pnt1>, <Qiong Luo, pnt2>,…., <Qiong Luo, pntn>
  - Problem: Redundancy - you repeat the name many times
- Option 3: use an extra level of indirection (most common option)
  - Index entry points to a bucket that contains pointers to all the actual records with that particular name

Secondary Index on ***balance*** field of ***account***