Database Systems
Spring 2013

Physical Database Design
SL07

- Disk Storage and Files
  - Physical Storage Media
  - Accessing the Storage
  - Organization of Files
- Index Structures
  - Types of Indexes
  - B+ Tree
  - Hashing
  - Index Definition in SQL

Literature and Acknowledgments

Reading List for SL07:

- Database Systems, Chapters 16 and 17, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

- Database Systems, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

Disk Storage and Files

- Physical Storage Media
- Storage Access
- File Organization

Physical Storage Media/1

- Several types of storage media exist in computer systems and are relevant for DBMS
- The storage media can be organized into a **storage hierarchy**.
- **Classification** of storage media
  - **Speed** with which data can be accessed
  - **Cost** per unit of data
  - **Reliability**
    - data loss on power failure or system crash
    - physical failure of the storage device
  - **Volatile** vs. **non-volatile** storage
    - Volatile storage: Loses contents when power is switched off
    - Non-Volatile storage: Contents persist when power is switched off

Physical Storage Media/2

- **Cache**
  - Volatile
  - Fastest and most costly form of storage
  - Managed by the computer system hardware
- **Main memory**
  - Volatile
  - Fast access (x0 to x00 of nanosecs; 1 nanosec $= 10^{-9}$ secs)
  - Generally only a part of a database is loaded into memory
    - Capacities of up to a few Gigabytes (or even Terabytes) widely used currently
    - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
    - If entire database is kept in memory we have a main memory database

Physical Storage Media/3

- **Flash memory (SSD)**
  - Non-volatile
  - Reads are roughly as fast as main memory
  - Writes are slow (few microseconds) and more complicated
    - Data cannot be overwritten, but a block must be erased and written over simultaneously
  - Cost per unit of storage roughly similar to main memory
  - Widely used in embedded devices such as digital cameras
  - Also known as EEPROM (Electrically Erasable Programmable Read-Only Memory)

Physical Storage Media/4

- **Magnetic disk**
  - Non-volatile
  - Data is stored on spinning disk, and read/written magnetically
  - Much slower access than main memory
  - Much larger capacities than main memory; typically up to roughly x00 GB - 2 TB currently
    - Growing rapidly with technology improvements (factor 2 to 3 every 2 years)
  - Primary medium for the long-term storage of data; typically stores entire DB.
  - Data must be moved from disk to main memory for access, and written back for storage
  - Direct data access, i.e., data on disk can be read in any order, unlike magnetic tape
  - Hard disks vs. floppy disks

Physical Storage Media/5

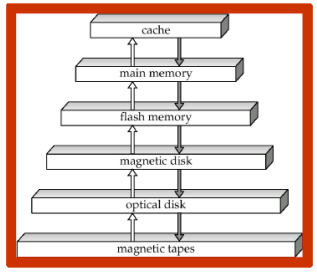
- **Optical disk**
  - Non-volatile
  - Data is read optically from a spinning disk using a laser
  - Reads and writes are slower than with magnetic disk
  - Different types
    - CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
    - Write-one, read-many (WORM) optical disks used for archival storage
    - Multiple write versions also available (CD-RW, DVD-RW, and DVD-RAM)
  - Juke-box systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data.

Physical Storage Media/6

- **Tape storage**
  - Non-volatile
  - Much slower than disk due to sequential access only
  - Very high capacity (up to tens of terabytes)
  - Used primarily for backup and for archival data
  - Tape can be removed from drive
  - Tape storage costs are much cheaper than disk storage costs.
  - Tape juke-boxes available for storing massive amounts of data
    - Hundreds of terabytes (1 terabyte = 10^{12} bytes) to even a petabyte (1 petabyte = 10^{15} bytes)

Physical Storage Media/7

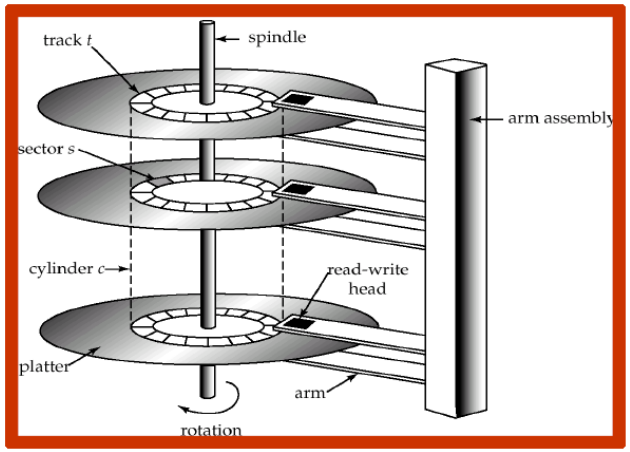
- The storage media can be organized in a hierarchy according to their speed and cost
- **Primary storage**: Fastest media, but volatile
  - e.g., cache, main memory
- **Secondary storage**: Non-volatile,moderately fast access
  - e.g., flash memory, magnetic disks
  - also called on-line storage
- **Tertiary storage**: Non-volatile, slow access time
  - e.g., magnetic tape, optical storage
  - also called off-line storage



- DBMS must explicitly deal with storage media at all levels of the hierarchy

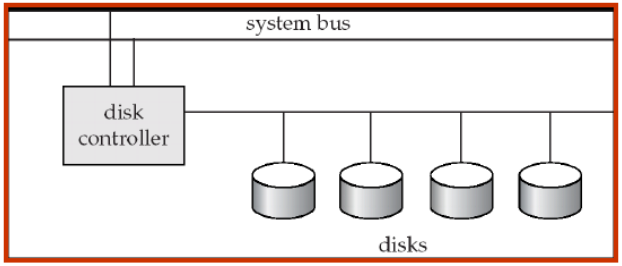
Magnetic Hard Disks/1

- Most DBs are stored on magnetic disks for the following reasons:
  - Generally, DBs are too large to fit entirely in main memory
  - Data on disks is non-volatile
  - Disk storage is cheaper than main memory
- Simplified and schematic structure of a magnetic disk



Magnetic Hard Disks/2

- **Disk controller:** Interface between the computer system and the HW of the disk drive. Performs the following tasks:
  - Translates high-level commands, such as read or write a sector, into actions of the disk HW, such as moving the disk arm or reading/writing the sector.
  - Adds a checksum to each sector
  - Ensures successful writing by reading back a sector after writing it



Magnetic Hard Disks/3

- **Performance measures** of hard disks
  - **Access time:** the time it takes from when a read or write request is issued to when the data transfer begins. Is composed of:
    - **Seek time:** time it takes to reposition the arm over the correct track
    - Avg. seek time is 1/2 the worst case seek time (2-10 ms on typical disks)
    - **Rotational latency:** time it takes for the sector to be accessed to appear under the head
    - Avg. latency is 1/2 the worst case latency (e.g., 4-11 ms for 5400-15000 rpm)
  - **Data-transfer rate:** rate at which data can be retrieved from or stored to disk (e.g., 25-100 MB/s)
    - Multiple disks may share a single controller
  - **Mean time to failure (MTTF)**: average time the disk is expected to run continuously without any failure
    - Typically several years

Storage Access Through Blocks/1

- A **block** is a contiguous sequence of sectors from a single track.
- Blocks are separated by **interblock gaps**, which hold control information created during disk initialization.
- Logically, a block is a unit of storage allocation and data transfer.
  - Data between disk and main memory is transferred in blocks.
  - A database file is partitioned into fixed-length blocks.
  - Typical block sizes range from 4 to 16 kilobytes
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks

Review 7.1

Consider relations $r(A)$ and $s(A)$. r is ordered, s is unordered. Block size $B = 2KB$. Tuple size $t = 100$Bytes. $|r| = |s| = 800'000$ tuples. The values of A are uniformly distributed between 5M and 10M. The time for 1 IO is 0.025 sec. Determine the execution times for the following queries where $x = r$ or $x = s$: $\sigma_{A=6M}(x)$, $\sigma_{A<5'000'500}(x)$, $\sigma_{A\neq 6M}(x)$.

Storage Access Through Blocks/2

- A major **goal** of DBMSs: Make the transfer of data between disk and main memory as efficient as possible by
  - Optimizing/Minimizing the disk-block access time
  - Minimizing the number of block transfers
  - Keeping as many blocks as possible in memory ($\rightarrow$ buffer manager)
- Techniques to optimize disk-block access:
  1. Disk arm scheduling
  2. Appropriate file organization
  3. Write buffers and log disks

Storage Access Through Blocks/3

- **Disk arm scheduling algorithms:** Order pending accesses to tracks so that disk arm movement is minimized
- **Elevator algorithm**
  - Disk controller orders the requests by track (from outer to inner or vice versa)
  - Move disk arm in that direction, processing the next request in that direction, till no more requests in that direction
  - Then reverse the direction (i.e., inner to outer) and repeat the previous two steps

Storage Access Through Blocks/4

- **File organization:** Optimize block access time by organizing the blocks to correspond to how data will be accessed
  - e.g., store related information on the same or nearby cylinders.
  - Files may get fragmented over time
    - e.g., if data is inserted to or deleted from the file
    - e.g., if free blocks on disk are scattered, which means that a newly created file has its blocks scattered over the disk
    - Sequential access to a fragmented file results in increased disk arm movement
  - Some systems have utilities to defragment the file system, in order to speed up file access

Storage Access Through Blocks/5

Updated blocks can be written asychnronously to increase the write speed.

- **Non-volatile write buffers:** Speed up disk writes by writing blocks to a non-volatile, battery backed up RAM or flash memory immediately; the controller then writes to disk whenever the disk has no other requests or request has been pending for some time.
  - Even if power fails, the data is safe.
  - Writes can be reordered to minimize disk arm movement.
  - Database operations that require data to be safely stored before continuing can continue immediately.
- **Log disk:** A disk devoted to write a sequential log of block updates
  - Used exactly like non-volatile RAM
  - Write to log disk is very fast since no seeks are required
  - No need for special hardware.

Review 7.2

Consider a disk as follows: block size B = 512 Bytes, interblock gap size G = 128 Bytes, blocks per track B/T = 20, tracks per surface T/S = 400, double-sided disks D = 15, seek time st = 30 msec, 2400 rotations per minute. Determine the following values:

1. total capacity per track
2. useful capacity per track
3. number of cylinders
4. useful capacity per cylinder

Review 7.2

5. transfer rate tr
6. block transfer time btt
7. rotational delay rd
8. bulk transfer rate btr
9. block read time
10. time for 20 random reads
11. time for 20 sequential reads

Buffer Manager/1

- **Buffer:** Portion of main memory available to store copies of disk blocks.
- **Buffer Manager:** Subsystem that is responsible for buffering disk blocks in main memory.
  - The overall goal is to minimize the number of disk accesses.
  - Buffer manager is similar to a virtual-memory manager of an operating system.

Buffer Manager/2

- Programs call the buffer manager when they need a block from disk.
- Buffer manager **algorithm**
  1. Programs call the buffer manager when they need a block from disk.
     - The requesting program is given the address of the block in main memory.
  2. If the block is not in the buffer:
     - The buffer manager allocates space in the buffer for the new block (replacing/throwing out some other block, if required).
     - 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 memory to the requesting program.
- There exist different strategies/policies to replace buffers

Buffer Replacement Policies/1

- **LRU strategy:** Replace the block least recently used
  - Idea: Use past pattern of block references to predict future references
  - Applied successfully by most operating systems
- **MRU strategy:** Replace the block most recently used
- LRU can be a bad strategy in DBMS for certain access patterns involving repeated scans of data
- Queries in DBs have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references

Buffer Replacement Policies/2

- **Example:** compute a join with nested loops

```
for each tuple tr of r do
  for each tuple ts of s do
    if the tuples tr and ts match then ...
```

- Different access pattern for r and s
  - An r-block is no longer needed, after the last tuple is processed (even if it has been used recently), thus should be removed immediately
  - An s-block is needed again after all other s-blocks are processed, thus MRU is the best strategy
  - A mixed strategy with hints on replacement strategy provided by the query optimizer is preferable

Review 7.3

Assume a relation r with 3 tuples and a relation s with three tuples. Assume a block can fit 2 tuples. Illustrate how a nested loop join processes tuples and how blocks can be used effectively if 2 blocks are available for the join.

Buffer Replacement Policies/3

- **Pinned block:** Memory block that is not allowed to be written back to disk.
  - e.g., the r-block before processing the last tuple tr
- **Toss immediate strategy:** Frees the space occupied by a block as soon as the final tuple of that block has been processed
  - e.g., the r-block after processing the last tuple tr
- MRU + pinned block is the best choice for the join

Buffer Replacement Policies/4

- **Buffer replacement policies in DBMS can use various information**
  - Queries have **well-defined access patterns** (e.g., sequential scans)
  - **Information in a query** to predict future references
  - **Statistical information** regarding the probability that a request will reference a particular relation.
    - e.g., the data dictionary is frequently accessed.
    - Heuristic: keep data dictionary blocks in main memory buffer

File Organization

- **File:** A file is logically a **sequence of records**, where
  - a record is a sequence of fields;
  - the file header contains information about the file.
- Usually, a relational table is mapped to a file and a tuple to a record.
- A DBMS has the choice to
  - Use the file system of the operating system (reuse code).
  - Manage disk space on its own (OS independent, better optimization, e.g., Oracle)
- Two approaches to represent files (or records) on disk blocks:
  - **Fixed length** records (fixed-length records are simple, inflexible, and inefficient in terms of memory)
  - **Variable length** records (variable-length records are complex, flexible, and efficient in terms of memory)

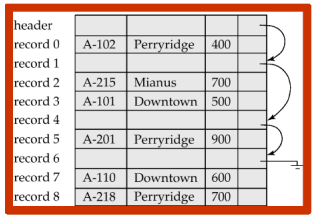
Fixed-Length Records/1

- Store record i starting from byte $m * (i - 1)$, where m is the size of each record.
- Record access is simple but records may cross blocks
  - **spanned** organization: records can be split and span accross block boundaries using pointers
  - **unspanned** organization: records may not cross block boundaries; leave free space in blocks if records do not fit
- Deletion of record i is more complicated. Several alternatives exist:
  - Move records $i + 1, ..., n$ to $i, ..., n - 1$
  - Move record n to i
  - Do not move records, but link all free records in a free list

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/2

- **Free list**
  - 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
  - Note the additional field to store pointers.
  - More space efficient representations are possible: No pointers need to be stored in records that contain data.

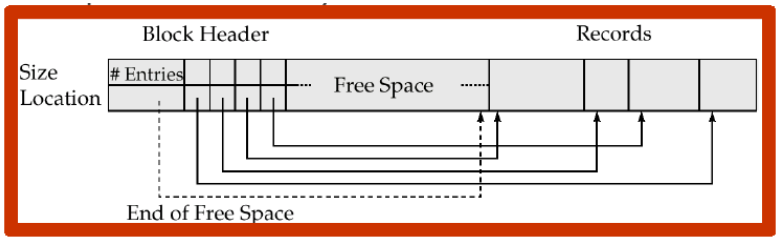


Variable-Length Records/1

- Variable-length records arise in DBMS in several ways:
  - Records types that allow variable lengths for one or more fields.
  - Storage of multiple record types in a file.
  - Record types that allow repeating fields (used in some older models).
- There exist different methods to represent variable-length records:
  - Slotted page structure is the most flexible organization of variable-length records.
  - A slotted page structure maintains a directory of slots for each page.

Variable-Length Records/2

- **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 in a page to keep them contiguous with no empty space between them; entry in the header must be updated.
  - Pointers should not point directly to record - instead they should point to the entry for the record in header.



Organization of Records in Files/1

There are different ways to logically organize records in a file (this is called the primary file organization):

- **Heap file organization:** A record can be placed anywhere in the file where there is space; there is no ordering in the file.
- **Sequential file organization:** Store records in sequential order based on the value of the search key of each record.
- **Hash file organization:** A hash function is computed on some attribute of each record; the result specifies in which block of the file the record is placed.

- Generally, each relation is stored in a separate file.

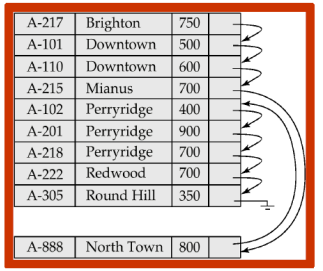
Organization of Records in Files/2

- **Sequential file:** The records in the file are ordered by a search key (one or more attributes)
  - Records are chained together by pointers
  - Suitable for applications that require sequential processing of the entire file
  - To be efficient, records should also be stored physically in search key order (or close to it).
  - **Example:** account(account-number,branch-name,balance)

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	

Organization of Records in Files/3

- It is difficult to maintain the physical order as records are inserted and deleted.
  - Deletion: Store a deletion marker with each record; use pointer chains to build a free list
- 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
- Need to reorganize the file from time to time to restore (physical) sequential order



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	
A-888	North Town	800	

Review 7.4

Assume a disk with the following characteristics: block size B = 512 Bytes, blocks per track = 20, tracks per surface = 400, number of double-sided disks = 15, rotations per minute = 2400 rpm, seek time = 30 msec.

Assume a relation Emp(N 30 Bytes, SSN 9 Bytes, A 40 Bytes, P 9 Bytes) with 20'000 tuples.

Determine the following values:

1. HD capacity
2. size of 1 Emp tuple

Review 7.4

1. blocking factory (bfr) of Emp (= number of tuples per block)
2. number of blocks used by Emp (unspanned organization)
3. number of blocks used by Emp (spanned organization)
4. average time for a linear search in Emp (contiguous file)

Index Structures for Files

- Basic Concepts, Types of Indexes
- B+ Tree
- Hashing
- Ordered Indexing versus Hashing
- Index Definition in SQL

Basic Concepts/1

- Indexing mechanism are used to speed up access to data
  - e.g., author catalog in library, book index
- **Index file:** Consists of records (called **index entries**) of the form (*search key, pointer*) where
  - **search key** is an attribute or set of attributes used to look up records in a data file
  - **pointer** is a pointer to a record (database tuple) in a data file
- Duplicated search keys in an index file are allowed
- Index files are typically much smaller than the original file

Basic Concepts/2

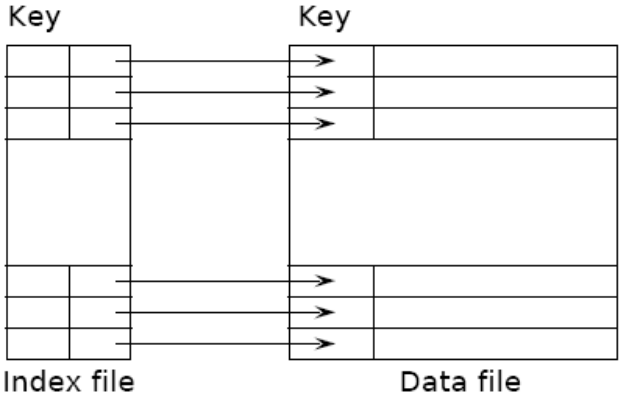
- **Evaluation of an index** must include
  - Access Time
  - Insertion Time
  - Deletion Time
  - Space overhead
  - Access Type supported efficiently, e.g.,
    - Records with a **specific value** in the attribute, e.g., persons who were born 1970
    - Reocrds with an attribute value falling in a **specific range of values**, e.g., persons who were born between 1970 and 1976

Basic Concepts/3

- Depending on the ordering of the data and the index file we can have a
  - **primary index** (same order of data and index; unique keys)
  - **clustering index** (same order of data and index)
  - **secondary index** (different order of data and index)
- Depending on what we put into the index we have a
  - **sparse index** (index entry for some tuples only)
  - **dense index** (index entry for each tuple)
- A primary index is usually sparse
- A clustering index is usually sparse
- A secondary index must be dense
- Note: terminology is not consistent across textbooks

Primary Index/1

- **Primary index**
  - In a primary index the search key order corresponds to the sequential order of the records in the data file.
  - The search key of a primary index is unique. Thus, the search key is a candidate key (and is often the primary key).

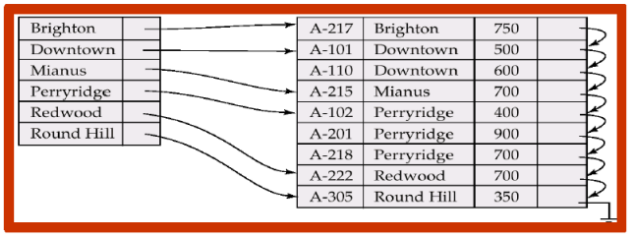


Primary Index/2

- **Index file:**
  - The index file is a sequential (ordered) file. The search key appears only once in the index file.
  - For a primary index both index and data are stored on sequential files (index file and data file)
  - Designed for both efficient sequential access and random access
  - One of the oldest indexing techniques in DB

Clustering Index

- **Index file:**
  - The index file is a sequential file. The same search key may occur multiple times (ordered on nonkey field).
  - Both index and data are stored on sequential files (index file and data file)
  - Designed for both efficient sequential access and random access
  - One of the oldest indexing techniques in DB

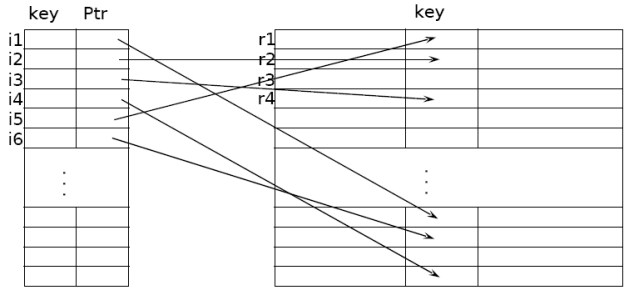


Secondary Indexes/1

- Secondary indexes are used to quickly find all records whose values in a certain field (which is not the search key of the primary index) satisfy some condition.
- **Example:** Consider an account relation that is stored sequentially by account number
  - Find all accounts in a particular branch
  - Find all accounts with a specified balance or range of balances
- The above query can only be answered by retrieving and checking all records (very inefficient).
- An additional (secondary) index is needed to answer such queries efficiently.

Secondary Indexes/2

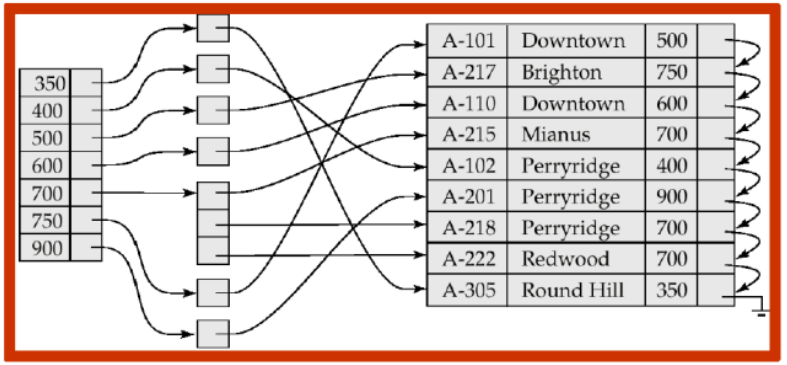
- **Secondary index:** Index whose search key specifies an order **different** from the sequential order of the file



- Secondary indexes are non-clustering
- Secondary indexes **must be dense**, i.e., they must include an index entry for every search key value and a pointer to every record in the data file.

Secondary Indexes/3

- Two options for data pointers
  - Duplicate index entries: an index record for every data record
  - Buckets: An index record for each search key value; index record points to a **bucket** that contains pointers to all the actual records with that particular search key value
- **Example:** Secondary index on the balance field of the account relation using buckets

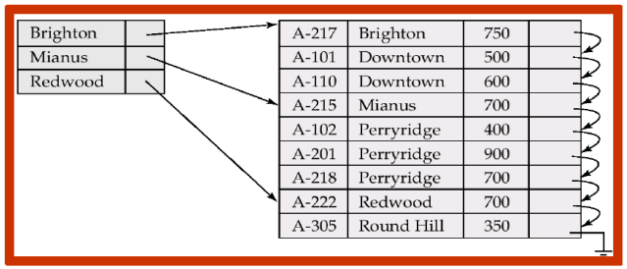


Review 7.5

- Relation $R(A, B, ...)$ with 6M tuples.
- Primary index on A. Secondary index on B.
- 200 index entries per block. 50 tuples per block.
- Values of A and B are uniformly distributed in $[0, 100M]$.
- Use indexes to answer $Q1 = \sigma[A > 75M](r)$ and $Q2 = \sigma[B > 75M](r)$. Determine the number of IOs. Interpret the result.

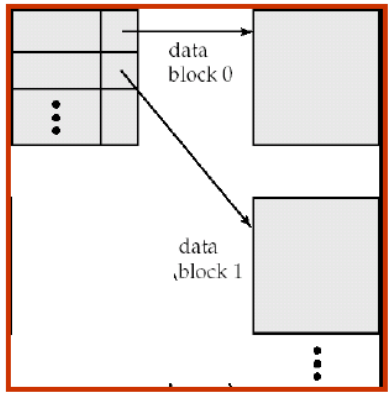
Sparse Index/1

- **Sparse index**
  - Contains index records for only some search key values.
  - Sparse index has (much) fewer entries than records in a table.
  - Applicable when records are sequentially ordered on the search key



Sparse Index/2

- Often a sparse index contains an index entry for every block in file.
- The index entry stores the least search key value of the block it points to.



Dense Index/1

- **Dense index:**
  - An index record appears for **each record** in the data file.
  - Dense index can get large (but still is much smaller than the data file).
  - Handling gets easier if there is exactly one entry for each record.
  - Alternative definition (used in some textbooks/systems: the index contains a record for each search key; the index record points to the first data record with that search key value; the remaining data records with that search key are stored sequentially

Primary versus Secondary Indexes

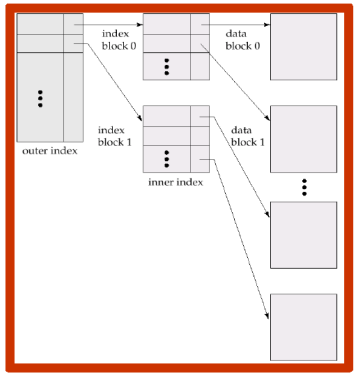
- Indexes offer substantial benefits for lookups
- Updating indexes imposes overhead on DB modifications: whenever data are modified, all index on this data must be updated too
- Primary indexes can be dense or sparse
- Secondary indexes must be dense
- Sequential scan using primary index is efficient
- A sequential scan using a secondary index is expensive since each record access may fetch a new block from disk
- A sparse index uses less space than a dense index
- The maintenance overhead for insertion and deletion is less for a sparse index than for a dense index
- In general a sparse index is slower than a dense index for locating records.

Multilevel Index/1

- If an index grows the handling becomes more expensive
  - A search for a data record requires several disk block reads from the index file
  - Binary search might be used on index file: $log_2 b$ disk block reads, where b is the total number of index blocks
  - If overflow blocks are used in the index file, binary search is not applicable, and sequential scan is required: b disk block reads are required
- To reduce the number of index block I/Os, treat primary index kept on disk as a sequential file (like any other data file) and construct a sparse index on it
  - $\Rightarrow$ multilevel index

Multilevel Index/2

- **Multilevel index**
  - Inner index: The primary index file on the data
  - Outer index: A sparse index on the index
- If even outer index is too large to fit in main memory, yet another level of index can be created, etc.



Multilevel Index/3

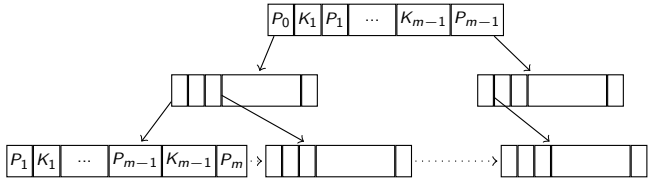
- **Index search: querying**
  - Start at the root
  - Check all entries (the entries are sorted) and follow the appropriate pointer
  - Repeat until you arrive at a leaf where the pointer points to the tuple
- **Index update: deletion and insertion**
  - Indexes at all levels must be updated on insertion and deletion in the data file
  - Update starts with the inner index
  - Algorithms are extensions of the single-level algorithms

B+ Tree/1

- The **B+ tree** is a multi-level index and is an alternative to sequential index files
  - Advantage of B+ tree index files
    - Automatically maintains as many levels of index as appropriate
    - Automatically reorganizes itself with small, local changes in the face of insertions and deletions
    - reorganization of entire file is not required to maintain performance
  - Disadvantage of B+ tree
    - Extra insertion and deletion overhead as well as space overhead
  - Advantages of B+ trees by far outweigh disadvantages, and they are used extensively

B+ Tree/2

- **B+ tree:** a rooted tree with the following properties



  - **Balanced tree**, i.e., all paths from root to leaf are of the same length (at most $\lceil log_{\lceil m/2 \rceil}(K) \rceil$ for K search key values)
  - A **node** contains up to $m-1$ search key values and m pointers, and the search key values within a node are sorted
  - Nodes are between half and completely full
  - **Internal nodes** have between $\lceil m/2 \rceil$ and m children
  - **Leaf nodes** have between $\lceil (m-1)/2 \rceil$ and m - 1 search key values
  - **Root node**: If it is a leaf, it can have between 0 and $m-1$ search key values; otherwise, it has at least 2 children

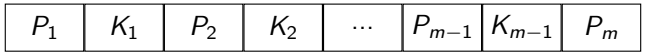
Terminology and Notation

- A pair (P_i, K_i) in a leaf node is an **entry**
- A pair (K_i, P_i) in an internal (i.e., non-leaf) node is an **entry**
- $L[i]$ denotes the value of the ith entry in node L
- Data pointers are stored at leaf nodes only
- Leaf nodes are linked together: the last pointer in a node points to the next leaf node.

- Note that there are many small variations of B+ tree; textbooks differ; stick to approach described on these slides.

B+ Tree Node Structure/1

- **Leaf nodes**



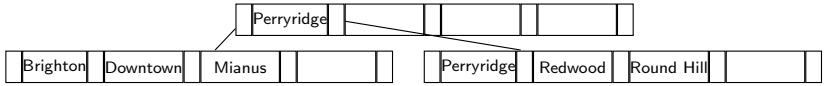
  - $K_1, \ldots, K_{m-1}$ are the search key values
  - $P_1, ..., P_{m-1}$ are pointers to records or buckets of records (for leaf nodes)
  - The search keys in a node are ordered: $K_1 < K_2 < K_3 < ... < K_{m-1}$
  - P_i points to the database tuples with search keys X equal to K_i
  - $P_1, \ldots, P_{m-1}$ either point to a file record with search key value K_i (unique) or to a bucket of pointers to file records with search key value K_i (non-unique)
  - Bucket structure is only needed if search key does not form a primary key
  - Pointer P_m points to next leaf node in search key order

B+ Tree Node Structure/2

- **Internal nodes** | P_0 | K_1 | P_1 | K_2 | P_2 | ... | K_{m-1} | P_{m-1} |
  - Form a multi-level sparse index on the leaf nodes
  - $P_0, ..., P_{m-1}$ are pointers to children (for non-leaf nodes)
  - P_i points to a subtree with search keys X such that $K_i \leq X < K_{i+1}$
    - P_0 points to the subtree where all search key values are less than K_1
    - For $1 \leq i < m-1$: Pointer P_i points to the subtree where all search key values are greater than or equal to K_i and less than K_{i+1}
    - Pointer P_{m-1} points to the subtree where all search key values are greater than or equal to K_{m-1}

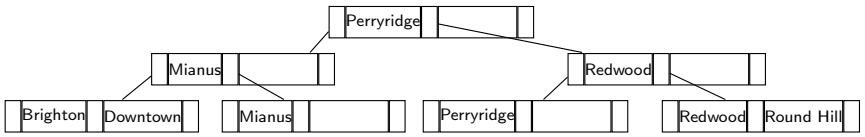
Example of B+ Tree/1

- **B+ tree for account file (m=5 pointers per node)**
  - Leaf nodes: between 2 and 4 search key values ($\lceil (m-1)/2 \rceil$ and $m-1$)
  - Non-leaf nodes other than root: between 3 and 5 children ($\lceil (m/2 \rceil$ and m))
  - Root node: at least 2 children



Example of B+ Tree/2

- **B+ tree for account file (m=3)**
  - Leaf nodes: between 1 and 2 search key values ($\lceil (m-1)/2 \rceil$ and $m-1$)
  - Non-leaf nodes other than root: between 2 and 3 children ($\lceil (m/2 \rceil$ and m))
  - Root node: at least 2 children



Observations about B+ Trees/1

- **B+ tree for account file**
  - Since the inter-node connections are done by pointers, logically close blocks need not be physically close
    - gives flexibility
    - increases times for seeks and latency
  - The non-leaf levels of the B+ tree form a hierarchy of sparse indexes (= multilevel index on leaf nodes)
  - The B+ tree contains a relatively small number of levels
    - $\lceil log_{\lceil m/2 \rceil}(K) \rceil$ for K search key values in the file

Observations about B+ Trees/2

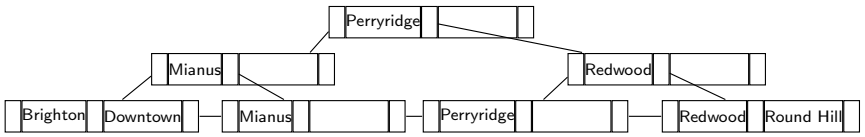
- **Search is efficient, since only a small number of index blocks need to be read**
  - Compare to the $log_2(b)$ disk block reads for binary search in sequential index files
  - Typically the root node and perhaps the first level nodes are kept in main memory, which further reduces the disk block reads.
- **Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time**

Queries on B+ Trees/1

- **Steps to find all records with a search key value of k (we assume a dense index):**
  1. Set C = root node
  2. **while** C is not a leaf node **do**
     Search for the largest search key value $\leq k$
     if such a value exists, assume it is K_i
     then set C = the node pointed to by P_i
     else set C = the node pointed to by P_0
  3. If there is a key value K_i in C such that $K_i = k$
     then follow pointer P_i to the desired record or bucket
     else no record with search key value k exists

Queries on B+ Trees/2

- **Example:** Find all records with a search key value equal to Mianus
  - Start from the root node
  - No search key $\leq$ Mianus exists, thus follow P_0
  - Mianus is the largest search key $\leq$ Mianus, thus follow P_1
  - search key = Mianus exists, thus follow the first data pointer to fetch record



Queries on B+ Trees/3

- In processing a query, a path is traversed in the tree from the root to some leaf node
- For K search key values in the data file, the path length is at most $\lceil log_{\lceil m/2 \rceil}(K) \rceil$
- A node generally corresponds to a disk block, typically 4KB, and m is typically $\approx$ 400 (10 bytes per index entry)
- With 1 million search key values and m = 400, at most $log_{200}(1,000,000) = 3$ nodes are accessed in a lookup
  - Contrast this with a balanced binary tree (or binary search) with 1 million search key values: around 20 nodes are accessed in a lookup
  - This difference is significant since every node access may need a disk I/O, costing around 20 milliseconds.

Intuition for B+ Tree Insertions/1

- **Insert a record with search key value of k**
  1. Find the leaf node in which the search key value would appear
  2. **If** the search key value is already there **then**
     - Add record to the data file
  3. **If** the search key value is not there and leaf is not full **then**
     - Add record to the data file
     - Insert (pointer, key-value) pair in the leaf node such that the search keys are still in order
  4. **If** search value is not there and leaf is full **then**
     - 4.1 Take all entries (including the new one being inserted) in sorted order; place the first half in the original node and the rest in a new node
     - 4.2 Insert the smallest entry of the new node into the parent of the node being split
     - 4.3 **If** the parent is full **then** split it and propagate the split further up
- **Splitting proceeds upwards until a node that is not full is found**
  - In the worst case the root node may be split, increasing the height of the tree by 1

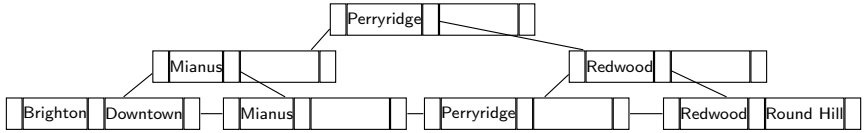
B+ Tree Insertion Algorithm

Algorithm 1: B+TreeInsert(L,k,p)

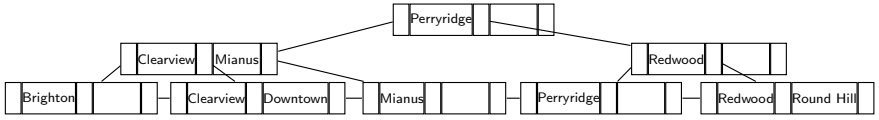
if *L is not yet full* **then**
  insert (k,p) into L
else
  create new node L';
  if *L is a leaf* **then**
    $L := L + (k,p); k' := L\lceil (m+1)/2 \rceil$;
    move entries greater or equal to k' from L to L';
  else
    $L := L + (k,p); k' := L\lceil m/2 \rceil$;
    move entries greater or equal to k' from L to L';
    delete entry with value k' from L'
  if *L is not the root* **then** B+TreeInsert(parent(L),k',L');
  else create new root with children L and L' and value k'

B+ Tree Insertions/3

- **Example:**
- B+ tree before insertion of Clearview



- B+ tree after insertion of Clearview



Review 7.6

Assume an empty B+ tree of order 4. Show the B+ tree after the following insertions: +2 +3 +5 ; +7 ; +11 ; +17 ; +19 +23 ; +29 +31 ; +8 +9 ; Show the B+ tree at the points indicated by a semicolon.

Intuition for B+ Tree Deletions/1

- **Deletion of a record with search key k**
  1. Find leaf node with (pointer, key-value) entry; remove entry
  2. **If** the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node **then**
     - **Coalesce** siblings, i.e., insert all search key values in the two nodes into a single node (the one on the left if it exists; the right otherwise) and delete the other node
     - Delete the entry in parent node that is between the two nodes by applying the deletion procedure recursively
  3. **If** the node has too few entries due to the removal, and the entries in the node and a sibling do not fit into a single node **then**
     - **Redistribute** the pointers between the node and a sibling such that both have more than the minimum number of entries
     - Update the corresponding search key value in the parent of the node
- Node deletions may cascade upwards till a node with $\lceil m/2 \rceil$ or more pointers is found. If the root node has only one pointer after deletion, it is deleted and the child becomes the root.

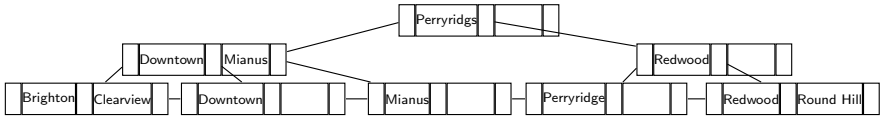
B+ Tree Deletion Algorithm

Algorithm 2: B+TreeDelete(L,k,p)

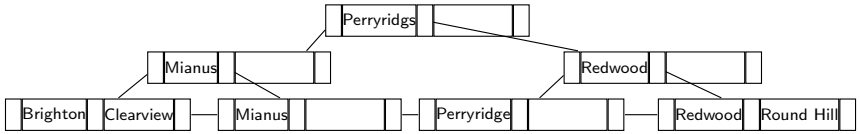
```
delete (p,k) from L;
if L is root with one child then root := child;
else if L has too few entries then
    L' is previous sibling of L [next if there is no previous] ;
    k' is value in parent that is between L and L';
    if entries L and L' fit on one page then
        if L is leaf then move entries from L to L';
        else move k' and all entries from L to L';
        B+TreeDelete(parent(L),k',L)
    else
        if L is leaf then
            move last [first] entry of L' to L;
            replace k' in parent(L) by value of first entry in L [L'];
        else
            move [first] last entry of L' to L;
            replace k' in parent(L) by value of first entry of L [L'];
            replace value of first entry in L [L'] by k';
```


B+ Tree Deletions/3

- **Example:** Before deleting Downtown



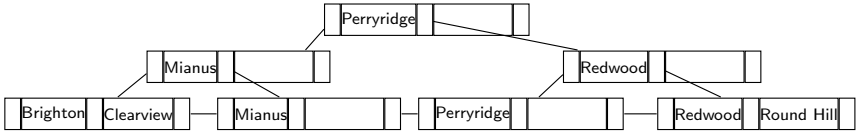
- After deleting Downtown



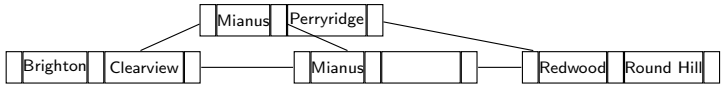
- The removal of the leaf node containing Downtown do not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node's parent.

B+ Tree Deletions/4

- **Example:** Before deleting Perryridge



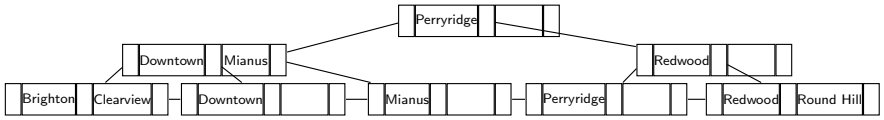
- After deleting Perryridge



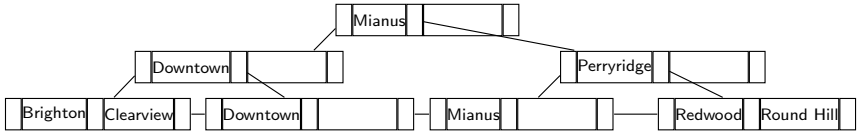
- Node with Perryridge becomes underfull and is merged with its sibling.
- As a result Perryridge node's parent becomes underfull, and is **coalesced** with its sibling (and an entry is deleted from their parent).
- Root node then has only one child and is deleted.

B+ Tree Deletions/5

- **Example:** Before deleting Perryridge



- After deleting Perryridge



- Parent of leaf containing Perryridge became underfull and borrowed a pointer from its left sibling (**redistribute** entries).
- Search key value in the parent's parent changes as a result.

Review 7.7

Consider the final B+ tree from review 7.6. Show the B+ trees after the following operations: -19 ; -17 -11 ; -9 ; -8 ; Show the B+ tree at the points indicated by a semicolon.

Static Hashing/1

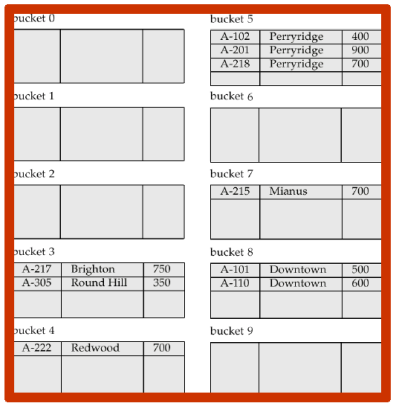
- **Disadvantage of sequential and B+ tree index file organization**
  - B+ tree: index structure must be accessed to locate data
  - Sequential file: binary search on large file might be required
  - This leads to additional block IO
- **Hashing**
  - provides a way to avoid index structures and to access data directly
  - provides also a way of constructing indexes
- A **bucket** is a unit of storage containing one or more records (typically a disk block; possibly multiple contiguous disk blocks).

Static Hashing/2

- **Hash file organization**
  - We obtain the bucket where a record is stored directly from its search key value using a hash function.
    - Constant access time
    - Avoids the use of an index
  - **Hash function h:** A function from the set of all search key values K to the set of all bucket addresses B.
  - Function h is used to locate records for access, insertion, and deletion.
  - Records with different search key values may map to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Static Hashing/3

- **Example:** Hash file organization of account file, using branch-name as key
- 10 buckets
- Binary representation of the ith character is assumed to be *i*, e.g. binary(B) = 2
- Hash function *h*
  - Sum of the binary representations of the characters modulo 10, e.g.,
  - h(Perryridge) = 5
  - h(Round Hill) = 3
  - h(Brighton) = 3



Hash Functions/1

- Worst hash function maps all search key values to the same bucket
  - This makes access time proportional to the number of search key values in the file.
- An **ideal hash function** has the following properties:
  - The distribution is **uniform**, i.e., each bucket is assigned the same number of search key values from the set of all possible values.
  - The distribution is **random**, so in the average case each bucket will have the same number of records assigned to it irrespective of the actual distribution of search key values in the file.

Hash Functions/2

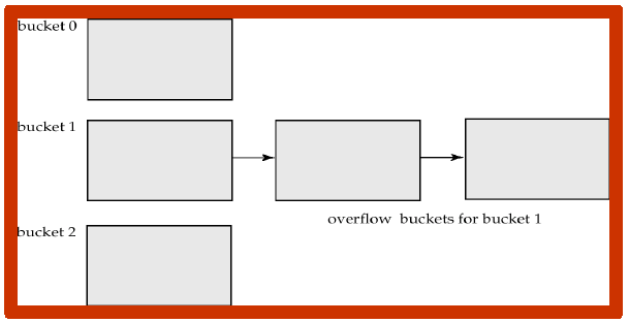
- **Example:** 26 buckets and a hash function that maps branch names beginning with the *i*-th letter of the alphabet to the *i*-th bucket
  - Simple, but not a uniform distribution, since we expect more branch names to begin, e.g., with B and R than Q and X.
- **Example:** Hash function on the search key balance by splitting the balance into equal ranges: 1 - 10000, 10001 - 20000, etc.
  - Uniform but not random distribution
- **Typical hash function:** Perform computation on the internal binary representation of the search key.
  - e.g., for a string search key, add the binary representations of all characters in the string and return the sum modulo the number of buckets

Bucket Overflow/1

- **Bucket overflow:** If a bucket has not enough space, a bucket overflow occurs; two reasons for bucket overflow
  - **Insufficient buckets**: the number of buckets n_B must be chosen to be $n_B > n/f$, where n = total number of records and f = number of records in bucket
  - **Skew in distribution of records:** A bucket may overflow even when other buckets still have space. This can occur due to two reasons:
    - multiple records have same search key value
    - hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it **cannot** be eliminated!
  - Handled by using overflow buckets

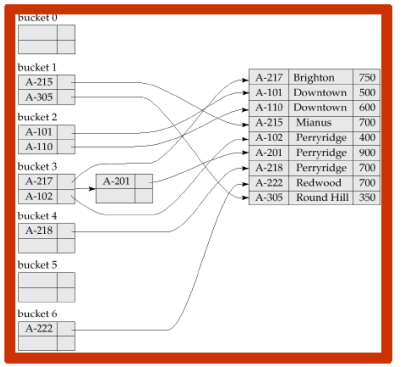
Bucket Overflow/2

- **Overflow chaining (closed hashing)**
  - If a record is inserted into bucket b, and b is already full, an **overflow bucket** is provided, where the record is inserted
    - The overflow buckets of a given bucket are chained together in a list.



Hash Indexes

- **Hash index:** organizes the search key values with their associated record pointers into a hash file structure.
  - Buckets contain search keys and pointers to the data records
  - Multiple (search key, pointer)-pairs might be required (different from index-sequential file)
- **Example:** Index on account
  - h: Sum of digits in account-number modulo 7



Deficiencies of Static Hashing/1

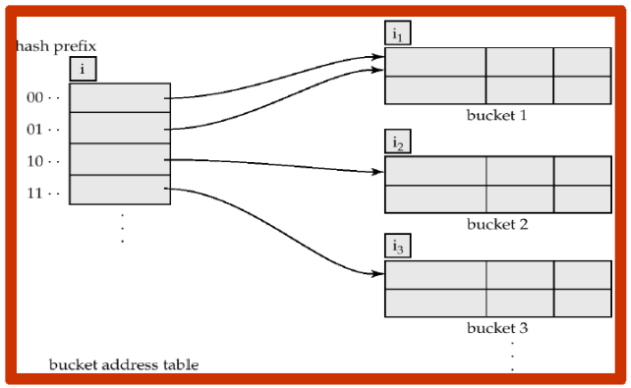
- In static hashing, the **fixed** set B of bucket addresses presents a serious problem
  - Databases grow and shrink with time
  - If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space is wasted initially.
  - If database shrinks, again space will be wasted
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically
  - $\Rightarrow$ dynamic hashing

Deficiencies of Static Hashing/2

- **Dynamic hashing:** Allows the hash function to be modified dynamically.
- **Extendable hashing:** one form of dynamic hashing
  - Hash function h generates values over a large range - typically b-bit integers, with $b = 32$.
  - At any time use only a prefix of h to index into the bucket address table
  - Let the size of the prefix be i bits, $0 \leq i \leq 32$
    - Bucket address table has size $= 2^i$
    - Value of i grows and shrinks as the size of DB grows and shrinks; initially $i = 0$
  - The actual number of buckets is $\leq 2^i$
    - Multiple entries in the bucket address table may point to the same bucket.
    - All such entries have a common hash prefix, $i_j \leq i$, which is stored with each bucket j
    - The number of buckets changes dynamically due to coalescing and splitting of buckets.

Extendable Hashing

- **General structure of extendable hashing**
  - i indicates the number of bits that are used from the hash value.
  - Consecutive entries may point to the same bucket (leads to a smaller prefix associated with this bucket).
  - In this structure, $i_2 = i_3 = i = 2$, whereas $i_1 = i - 1 = 1$ (thus, two entries point to bucket 1)



Lookup in Extendable Hashing

- **Lookup:** Locate the bucket containing search key value K_j
  1. Compute $h(K_j) = X$
  2. Use the first i (hash prefix) high order bits of X as a displacement into the bucket address table, and follow the pointer to the appropriate bucket

Updates in Extendable Hashing/1

- **Insertion** of a record with search key value K_j
  1. Use lookup to locate the bucket, say bucket j
  2. **If** there is room in bucket j **then**
     - Insert the record in the bucket.
  3. **Else**
     - The bucket must be split and insertion re-attempted

Updates in Extendable Hashing/2

- **Split a bucket j when inserting search key value** K_j
  - **If** $i > i_j$ (more than one pointer to bucket j) **then**
    - Allocate a new bucket z, and set i_j and i_z to the old $i_j + 1$.
    - Update bucket address table entries that point to j according to prefix (some will now point to z)
    - Remove and reinsert each record in bucket j.
    - Recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full).
  - **If** $i = i_j$ (only one pointer to bucket j) **then**
    - Increment i and double the size of the bucket address table.
    - Replace each entry in the table by two entries that point to the same bucket.
    - Recompute new bucket address table entry for K_j
- Overflow buckets needed instead of splitting (or in addition) in some cases, e.g., too many records with same hash value.

Updates in Extendable Hashing/3

- **Deletion** of a key value K
  1. Locate K in its bucket and remove it (search key from bucket and record from the file).
  2. The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  3. Coalescing of buckets can be done (can coalesce only with a buddy bucket having same value of i_j and same i_j -1 prefix, if it is present).
  4. Decreasing bucket address table size is also possible.
- **Note:** Decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Review 7.8

Consider the following hash function: h(Brighton) = 0010, h(Downtown) = 1010, h(Mianus) = 1100, h(Perryridge) = 1111, h(Redwood) = 0011. Assume a bucket size of two and extendable hashing with an address table size of 1. Show the hash table after the following modifications:

- insert 1 Brighton and 2 Downtown records
- insert 1 Mianus record
- insert 1 Redwood record
- insert 3 Perryridge records

Extendable Hashing: Discussion

- **Benefits** of extendable hashing
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
  - No buckets are reserved for future growth, but are allocated dynamically.
- **Disadvantages** of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Need a tree structure to locate desired record in the structure
  - Changing size of bucket address table is expensive

Ordered Indexing versus Hashing

- Cost of periodic re-organization
  - Hashmaps (e.g., Google's sparse and dense hash maps) do not provide constant insert/lookup time because of reorganization
- Relative frequency of insertions and deletions
  - B+ trees are better than hashing if there are many database updates
- Is it desirable to optimize average access time at the expense of worst-case access time?
  - Hashing has a better average time but no worst case guarantees
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indexes are to be preferred
    - There is no ordering in hash organization, and hence there is no notion of "next record in sort order".

Berkeley DB

o Database access methods:
- B+tree
- Hash (Extended Linear Hashing)
- Fixed/Variable Length Records
- Duplicate records per key in the B+tree and Hash access methods.
- Retrieval by record number in the B+tree access method.
- Keyed and sequential (forward and reverse) retrieval, insertion, modification and deletion.
- Memory-mapped read-only databases.
- Retrieval into user-specified or allocated memory.
- Partial-record data storage and retrieval.
- Architecture independent databases.
- Maximum B+tree depth of 255.
- Individual database files up to 2^{48} bytes, individual key or data elements up to 2^{32} bytes (or available memory).

Berkeley DB

The Berkeley Database Package (DB) is being used by many different organizations in many different applications! Here are a few of which you've probably heard:

Netscape SuiteSpot, an integrated suite of intranet and Internet server software, lets you communicate, access, and share information throughout your organization. The Enterprise, Catalog, Directory and Mail servers all use Berkeley DB.

The Isode Ltd. LDAP/X.500 Enterprise Directory Server uses Berkeley DB as its primary database. Berkeley DB is also used in its X.400/Internet Message Switch and X.400 Message Store products. Here's the press release from Isode about Berkeley DB.

Sendmail is the program that routes electronic mail throughout the Internet. Sendmail is used on almost every UNIX-like system, and it uses Berkeley DB.

The OSF Distributed Computing Environment (DCE) is an industry-standard, vendor-neutral set of distributed computing technologies. It provides security services to protect and control access to data, name services that make it easy to find distributed resources, and a highly scalable model for organizing widely scattered users, services, and data. The DCE backing store library (Open Group RFC #45) is a subset of Berkeley DB.

Index Definition in SQL

- SQL-92 does not define syntax for indexes because these are not considered part of the logical data model
- All DBMSs (must) provide support for indexes
- Create an index:

  create index <IdxName> **on** <RelName> (<AttrList>)
  E.g,: **create index** BrNaIdx **on** branch (branch-name)
- **Create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index: drop index <index-name>

  E.g,: drop index BrNaIdx

Indexes in PostgreSQL

- **CREATE [UNIQUE] INDEX** name **ON** table_name "(" col [**DESC**] { "," col [**DESC**] } ")" [...]
- **CREATE INDEX** MajIdx **ON** Enroll (Major);
- **CREATE INDEX** MajIdx **ON** Enroll **USING HASH** (Major);
- **CREATE INDEX** MajMinIdx **ON** Enroll (Major, Minor);

- Properties of indexes:
  - Indexes are automatically maintained as data are inserted, deleted, and updated.
  - Indexes slow down database modification statements.
  - Creating an index can take a long time.

Indexes in Oracle

- B+ tree indexes in Oracle

  CREATE [UNIQUE] INDEX name **ON** table_name "(" col [**DESC**] { "," col [**DESC**] } ")" [**pctfree** n] [...]
  - pct_free specifies how many percent of a index page are left unfilled initially (dafult to 10%)
  - In index definitions UNIQUE should not be used because it is a logical concept.
  - Oracle creates a B+ Tree index for each unique (and primary key) declaration.

  CREATE TABLE BOOK (
  ISBN **INTEGER**, Author **VARCHAR2 (30)** , ...);
  CREATE INDEX book_auth **ON** book(Author);

- Creating a hash-partitioned global index:

  CREATE INDEX CustLNameIX **ON** customers (LName)
  GLOBAL PARTITION BY HASH (LName) **PARTITIONS** 4;

Summary/1

- Physical storage media
  - storage hierarchy: cache, RAM, flash, disk, optical disk, tape, ...
- Accessing the storage
  - block-based access:
    - know characteristics of disks
    - compute number of IOs
    - compute execution time
  - buffer manager
- Organization of files
  - fixed-length record, variable-length record
  - heap file (unordered), sequential file (ordered), hash file

Summary/2

- Definition of and differences between index types
  - primary, clustering and secondary index
  - dense and sparse index
- B+ tree
  - universal database access structure; also for range predicates
  - definition (node, leaf, non-leaf, entry)
  - insertion and deletion
- Hashing
  - static and extendable hashing
  - no index structure needed for primary index (hash function gives record location directly)
  - good for equality predicates (used heavily in applications)
- Index definition in SQL