Storage and File Structure

Thomas Heinis

t.heinis@imperial.ac.uk
wp.doc.ic.ac.uk/theinis

Imperial College
London

Database Servers

Also called **query server** systems or SQL *server* systems

- Clients send requests to the server
- Transactions are executed at the server
- Results are shipped back to the client.

Requests are specified in SQL, and communicated to the server through a *remote procedure call* (RPC) mechanism.

Transactional RPC allows many RPC calls to form a transaction.

Open Database Connectivity (ODBC) is a C language application program interface standard from Microsoft for connecting to a server, sending SQL requests, and receiving results. JDBC standard is similar to ODBC, for Java

Database Server Architecture

A typical transaction server consists of multiple processes accessing data in shared memory.

Server processes

- These receive user queries (transactions), execute them and send results back
- Processes may be **multithreaded**, allowing a single process to execute several user queries concurrently
- Typically multiple multithreaded server processes

Lock manager process

Database writer process

- Output modified buffer blocks to disks continually

Database Server Processes

Log writer process

- Server processes simply add log records to log record buffer
- Log writer process outputs log records to stable storage.

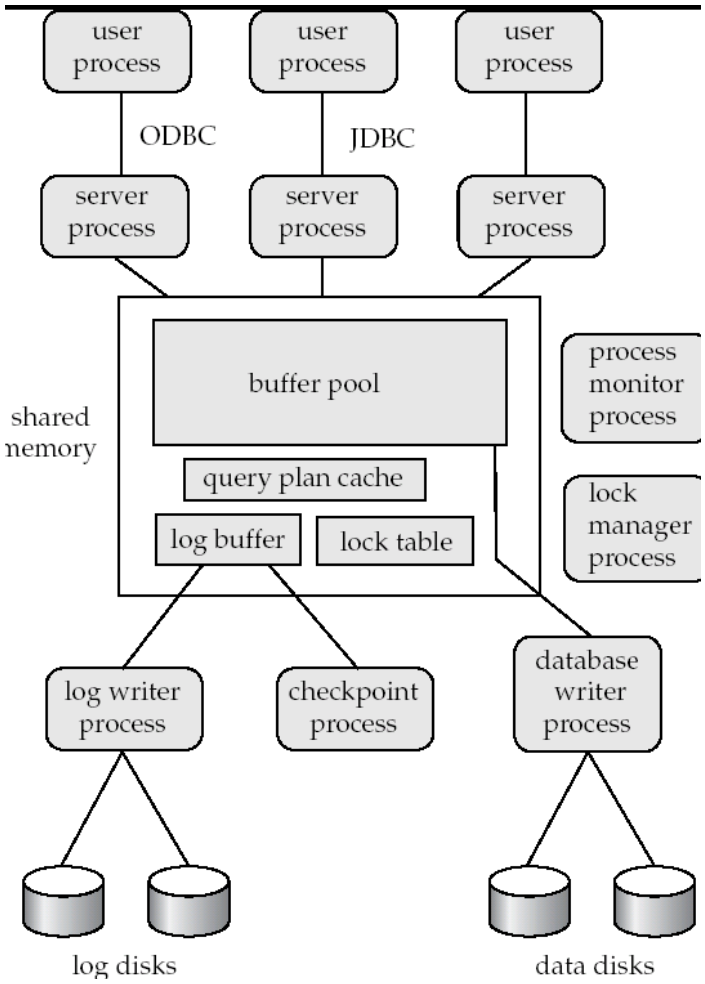
Checkpoint process

- Performs periodic checkpoints

Process monitor process

- Monitors other processes, and takes recovery actions if any of the other processes fail
  - aborting any transactions being executed by a server process and restarting it

Database System Processes



Database System Processes

Shared memory contains shared data

- Buffer pool
- Lock table
- Log buffer
- Cached query plans (reused if same query submitted again)

All database processes can access shared memory

To ensure that no two processes are accessing the same data structure at the same time, databases systems implement **mutual exclusion** using either

- Operating system semaphores
- Atomic instructions such as test-and-set

Classification of Physical Storage

- 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
- Can differentiate storage into:
  - **volatile storage:** loses contents when power is switched off
  - **non-volatile storage**:
    - Contents persist even when power is switched off.
    - Includes secondary and tertiary storage, as well as battery-backed up main-memory.

Physical Storage Media

Cache – fastest and most costly form of storage; volatile; managed by the computer system hardware

Main memory:

- fast access (10's to 100's of nanoseconds)
- generally too small (or too expensive) to store the entire database
  - capacities of up to a few Gigabytes widely used currently
  - Capacities have gone up and per-byte costs have decreased steadily and rapidly
- **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.

Physical Storage Media (Cont.)

Flash memory

- Data survives power failure
- Data can be written at a location only once, but location can be erased and written to again
  - Can support only a limited number (10K – 1M) of write/erase cycles.
  - Erasing of memory has to be done to an entire bank of memory
- Reads are roughly as fast as main memory
- But writes are slow (few microseconds), erase is slower

Physical Storage Media (Cont.)

Flash memory

- NOR Flash
  - Fast reads, very slow erase, lower capacity
  - Used to store program code in many embedded devices
- NAND Flash
  - Page-at-a-time read/write, multi-page erase
  - High capacity (several GB)
  - Widely used as data storage mechanism in portable devices

Physical Storage Media (Cont.)

Magnetic-disk

- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- Data must be moved from disk to main memory for access, and written back for storage
- **direct-access** – possible to read data on disk in any order, unlike magnetic tape
- Survives power failures and system crashes
  - disk failure can destroy data: is rare but does happen

Physical Storage Media (Cont.)

Optical storage

- non-volatile, data is read optically from a spinning disk using a laser
- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- Reads and writes are slower than with magnetic disk
- **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

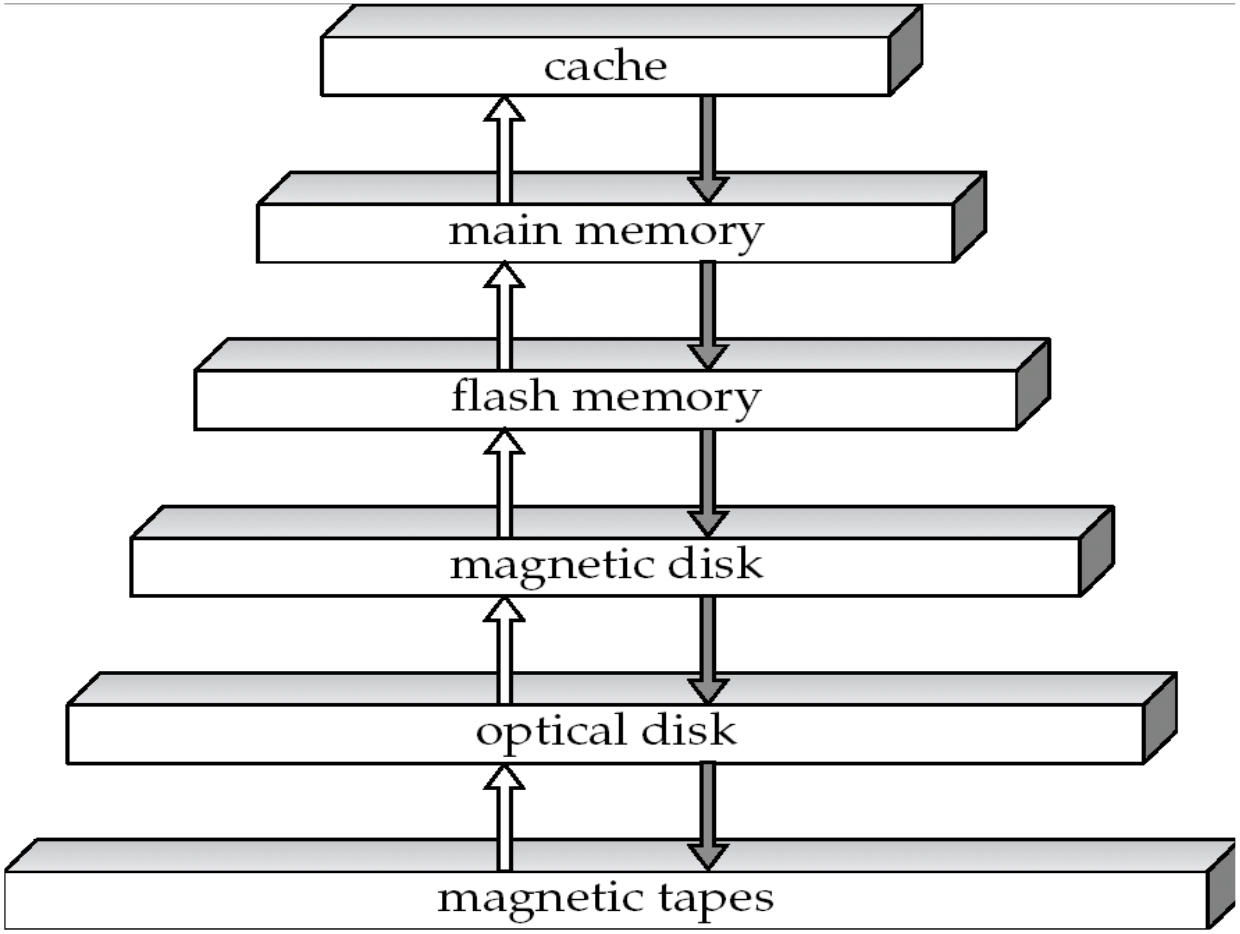
Physical Storage Media (Cont.)

Tape storage

- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** – much slower than disk
- very high capacity (40 to 300 GB)
- tape can be removed from drive $\Rightarrow$ storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data:

  hundreds of terabytes (1 terabyte = 10^9 bytes) to even a petabyte (1 petabyte = 10^{12} bytes)

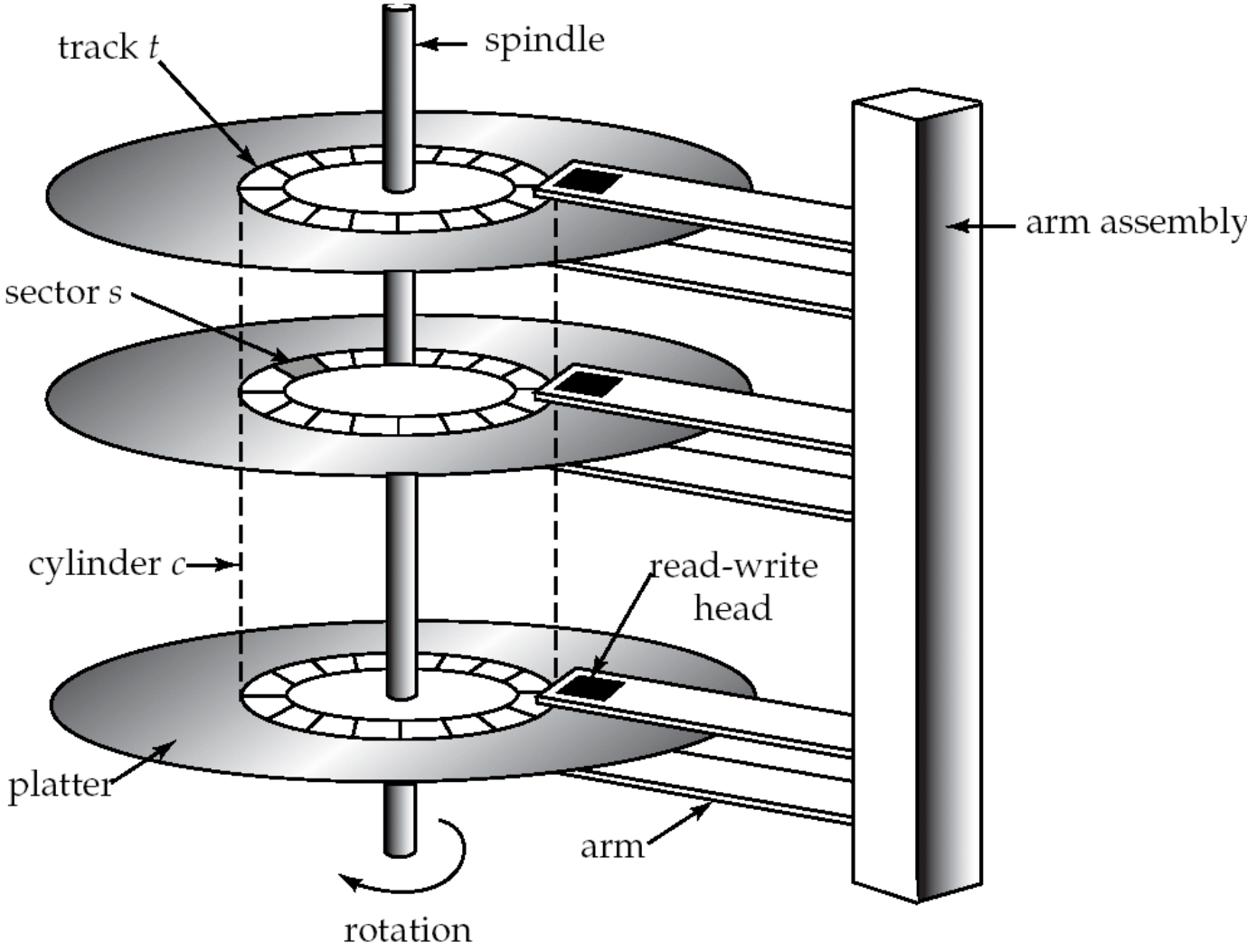
Storage Hierarchy



Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage

Magnetic Hard Disk Mechanism



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

Magnetic Disks

- **Read-write head**
  - Positioned very close to the platter surface
  - Reads or writes magnetically encoded information.
- Surface of platter divided into circular **tracks**
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors.**
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 (on inner tracks) to 1000 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head

Magnetic Disks (Cont.)

- Head-disk assemblies
  - multiple disk platters on a single spindle (1 to 5 usually)
  - one head per platter, mounted on a common arm
- **Cylinder** *i* consists of i^{th} track of all the platters
- Earlier generation disks were susceptible to "head-crashes" leading to loss of all data on disk
- Current generation disks are less susceptible to such disastrous failures, but individual sectors may get corrupted

Disk Controller

Disk controller – interfaces between the computer system and the disk drive hardware.

- Accepts commands to read or write a sector
- initiates actions such as moving the disk arm to the right track and actually reading or writing the data
- Computes and attaches **checksums** to each sector to verify that data is read back correctly
  - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
- Ensures successful writing by reading back sector after writing it
- Performs remapping of bad sectors

Performance Measures of Disks

Access time – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:

- **Seek time** – time it takes to reposition the arm over the correct track.
  - Average seek time is 1/2 the worst case seek time.
    - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
  - 4 to 10 milliseconds on typical disks
- **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
  - Average latency is 1/2 of the worst case latency.
  - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)

Performance Measures (Cont.)

Data-transfer rate – the rate at which data can be retrieved from or stored to the disk.

- 25 to 100 MB per second max rate
- Multiple disks may share a controller, so rate that controller can handle is also important
  - E.g. ATA-5: 66 MB/sec, SATA: 150 MB/sec, Ultra 320 SCSI: 320 MB/s
  - Fiber Channel (FC2Gb): 256 MB/s

Optimization for Block Access (SSD & HDD)

Block: a contiguous sequence of sectors from a single track

- data is transferred between disk and main memory in blocks
- Typical block sizes today range from 4 to 16 kilobytes

Disk-arm-scheduling algorithms order pending accesses to tracks so that disk arm movement is minimized

- **elevator algorithm**: move disk arm in one direction (from outer to inner tracks or vice versa), processing next request in that direction, until no more requests in that direction, then reverse direction and repeat

Optimization of Disk Block Access (Cont.)

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 blocks/cylinders.
  - File systems attempt to allocate contiguous chunks of blocks (e.g. 8 or 16 blocks) to a file
- Files may get **fragmented** over time
  - E.g. if data is inserted to/deleted from the file
  - Or free blocks on disk are scattered, and 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

Optimization of Disk Block Access (Cont.)

Nonvolatile write buffers speed up disk writes by writing blocks to a non-volatile RAM buffer immediately

- Non-volatile RAM: battery backed up RAM or flash memory
  - Even if power fails, the data is safe and will be written to disk when power returns
- Controller then writes to disk whenever the disk has no other requests or request has been pending for some time
- Database operations that require data to be safely stored before continuing can continue without waiting for data to be written to disk
- *Writes can be reordered to minimize disk arm movement*

Optimization of Disk Block Access (Cont.)

Log disk – a disk devoted to writing a sequential log of block updates

- Used exactly like nonvolatile RAM
  - Write to log disk is very fast since no seeks are required
  - No need for special hardware (NV-RAM)

File systems typically reorder writes to disk to improve performance

- **Journaling file systems** write data in safe order to NV-RAM or log disk
- Reordering without journaling: risk of corruption of file system data

Optical Disks

- Compact disk-read only memory (CD-ROM)
  - Seek time about 100 msec (optical read head is heavier and slower)
  - Higher latency (3000 RPM) and lower data-transfer rates (3-6 MB/s) compared to magnetic disks
- Digital Video Disk (DVD)
  - DVD-5 holds 4.7 GB , variants up to 17 GB
  - Slow seek time, for same reasons as CD-ROM
- Record once versions (CD-R and DVD-R)

Magnetic Tapes

- Hold large volumes of data and provide high transfer rates
  - Transfer rates from few to 10s of MB/s
- Currently the cheapest storage medium
  - Tapes are cheap, but cost of drives is very high
- Very slow access time in comparison to magnetic disks and optical disks
  - limited to sequential access.
- Used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another.
- Tape jukeboxes used for very large capacity storage
  - (terabyte (10^{12} bytes) to petabyte (10^{15} bytes)

Storage Access

- A database file is partitioned into fixed-length storage units called **blocks**. 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.

Buffer manager does the following:

- If the block is already in the buffer, return the address of the block in main memory
- If the block is not in the buffer
  1. Allocate space in the buffer for the block
     1. Replacing (throwing out) some other block, if required, to make space for the new block.
     2. Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
  2. Read the block from the disk to the buffer, and return the address of the block in main memory to requester.

Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
- Idea behind LRU – use past pattern of block references as a predictor of future references
- Queries 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
  - LRU can be a bad strategy for certain access patterns involving repeated scans of data
  - Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable

Buffer-Replacement Policies

- **Pinned block** – memory block that is not allowed to be written back to disk.
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- Most recently used (MRU) strategy – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use 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

- The database is stored as a collection of *files*. Each file is a sequence of *records.* A record is a sequence of fields.
- One 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.

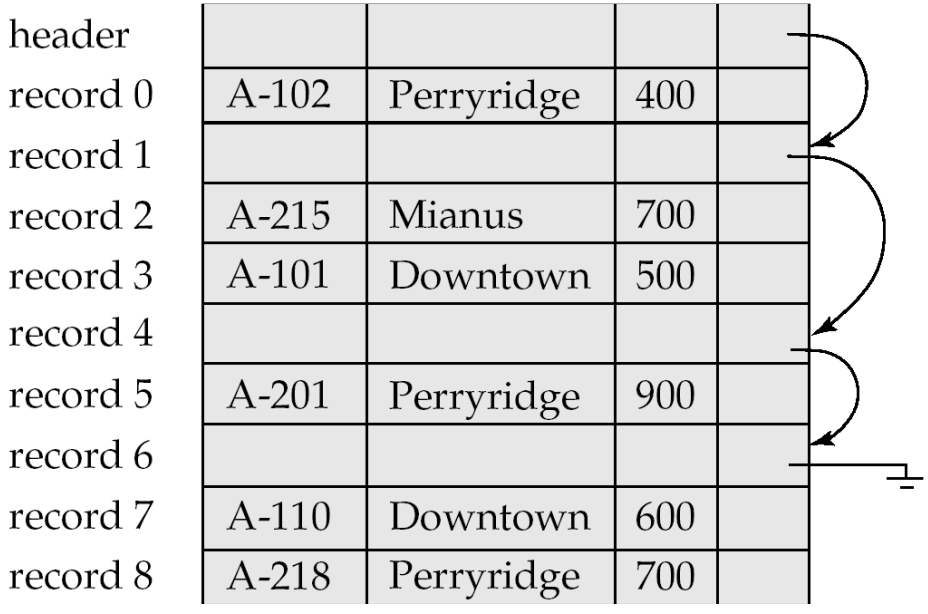
Fixed-Length Records

- Simple approach:
  - 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
    - Modification: do not allow records to cross block boundaries
- Deletion of record *i:* alternatives*:*
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record *n* to *i*
  - do not move records, but link all free records on 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

Free Lists

- 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.)

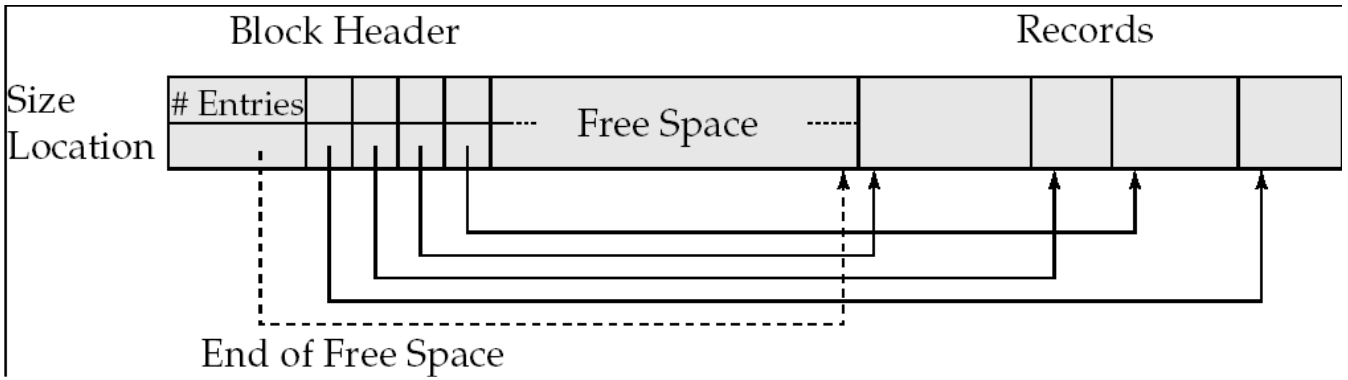


Variable-Length Records

- 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).

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 should not point directly to record — instead they should 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
- Records of each relation may be stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

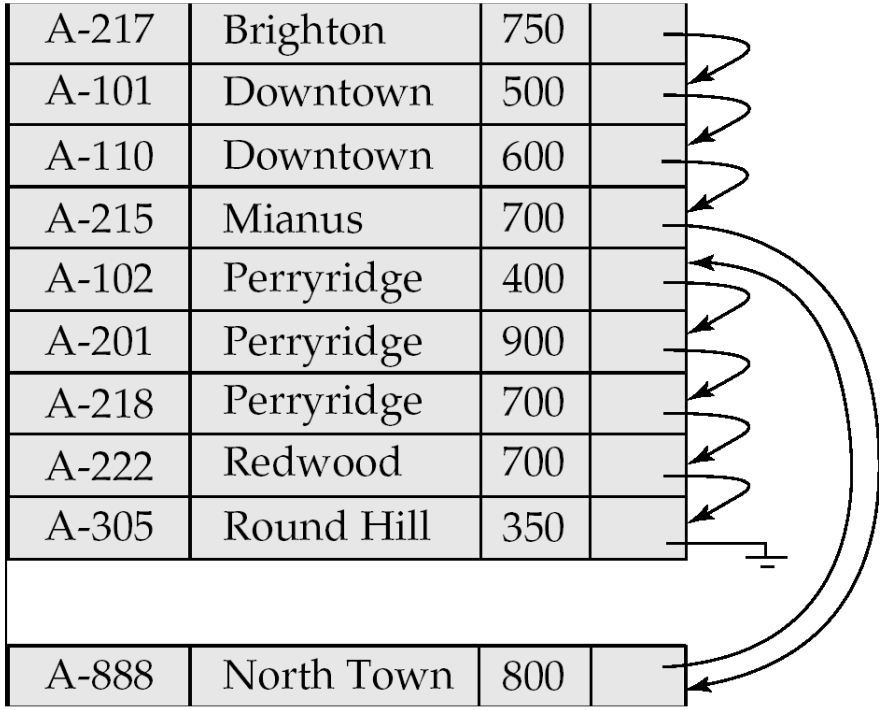
Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key

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	

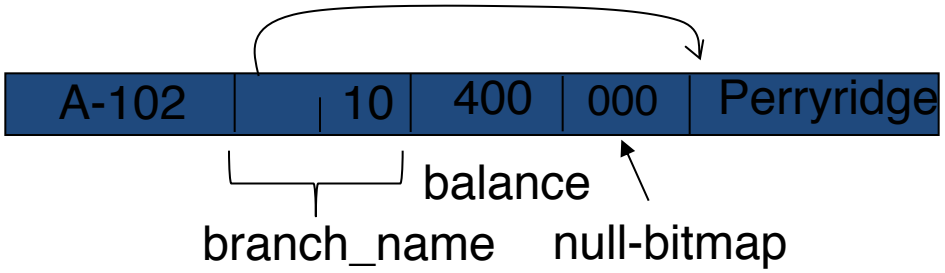
Sequential File Organization

- 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



Record Representation

- Records with fixed length fields are easy to represent
  - Similar to records (structs) in programming languages
  - Extensions to represent null values
    - E.g. a bitmap indicating which attributes are null
- Variable length fields can be represented by a pair (offset, length)
  offset: the location within the record, length: field length.
  - All fields start at predefined location, but extra indirection required for variable length fields



Example record structure of *account* record

Blocks & Records

Row-oriented

Works well for queries like:
select * from table

Page 1:

Record 1 Att 1	Record 1 Att 2	Record 1 Att 3
Record 2 Att 1	Record 2 Att 2	Record 2 Att 3
Record 3 Att 1	Record 3 Att 2	Record 3 Att 3
Record 4 Att 1	Record 4 Att 2	Record 4 Att 3

Page 2:

Record 5 Att 1	Record 5 Att 2	Record 5 Att 3
Record 6 Att 1	Record 6 Att 2	Record 6 Att 3
Record 7 Att 1	Record 7 Att 2	Record 7 Att 3
Record 8 Att 1	Record 8 Att 2	Record 8 Att 3

Column-oriented

Works well for queries like:
select att1 from table

Page 1:

Record 1 Att 1	Record 2 Att 1	Record 3 Att 1
Record 4 Att 1	Record 5 Att 1	Record 6 Att 1
Record 7 Att 1	Record 8 Att 1	Record 1 Att 2
Record 2 Att 2	Record 3 Att 2	Record 4 Att 2

Optimisation

Independent of the layout, how can we optimise:

select * from table where att1 > 120?

Sort records!

Record 2 Att 1: **100**	Record 1 Att 2	Record 1 Att 3
Record 4 Att 1: **120**	Record 2 Att 2	Record 2 Att 3
Record 1 Att 1: **250**	Record 3 Att 2	Record 3 Att 3
Record 3 Att 1: **310**	Record 4 Att 2	Record 4 Att 3

What about:

select * from table where att1 > 120 & att2 < 40?