Secondary Storage Management [13]
Outline

In implementing DBMS we need to answer

- How should the system store and manage very large amounts of data?
- What data structures should be used in order to support efficient manipulation of these data?

Focus on 1st question

- Consider memory hierarchy
- Algorithms to move data between main and secondary/tertiary memory
- Techniques to reduce read/write times
- Methods to improve reliability of disks
- Introduce a block model of secondary storage

Then, consider how to represent data elements
Storage Devices

We can classify storage devices on

- Speed to access the data
- Cost per byte
- Reliability
- Persistence
  - volatile storage
    - loses contents after power is removed
  - non-volatile storage
    - retains contents even after power is removed
Memory Hierarchy

- Cache
- Main memory
- Virtual memory
- Secondary storage
  - Floppies
  - Hard-disks
- Tertiary storage
  - Ad-hoc tape storage (heap of tapes)
  - Optical-disk juke boxes
  - Tape silos
Disks

Physical characteristics

- Rotation speed (e.g., 7200 rpm)
- Number of platters per unit
- Number of tracks per surface
- Number of sectors per track
- Number of bytes per sector
- Number of bits per inch (radial, peripheral)

Inner tracks will have higher density if the capacity of all tracks is fixed.
Disks

Figure from DSC

- track $t$
- spindle
- sector $s$
- cylinder $c$
- platter
- rotation
- arm assembly
- read-write head
- arm
Disk Access Characteristics

- Latency to move data between memory and disk
  - Read data from disk
    - Seek time
      - Max is linear in #cylinders crossed
      - Avg #cylinders = 1/3 of total #cylinders
    - Rotational latency
    - Transfer time
    - Other delays
      - CPU time to issue I/O
      - Contention for controller
      - Contention for bus, memory
  - Write data to disk
    - Same as read
    - Verifying data written doubles latency
**Disk Access**

- Random access to disk blocks is expensive, while sequential access is much cheaper
  - ~20ms vs ~1ms for 1KB blocks

- Block pointers
  - Dealing with bad blocks is messy (requires some kind of map)
Using disks efficiently

Often, the primary cost for operations on large amounts of data is disk I/O

Merge-Sort for Secondary storage

- Two-phase Multi-way Merge-sort
  - partition data into segments that fit in memory – sort each segment
  - (multi-way) merge segments to get progressively larger segments

For example, with memory of size M, blocks of size B, and records of size R

- we can sort a relation with up to \( N = \frac{M}{R} \left( \frac{M}{B} - 1 \right) \) records by doing \( 4NR/B \) disk I/Os
Accelerating access to disks

- Organize data by cylinders since the seek time for adjacent cylinders is much smaller than for random cylinders
  - Good when access is predictable and one process is using the disk

- Use multiple smaller disks to get parallelism as long the other delays are kept small
  - Watch for contention for disk controller, bus, and main memory
  - Placement of data onto disks critical to get parallelism from using many disks
  - Cost of multiple disks is larger than the cost of single larger disk
Accelerating access to disks

Use mirroring to disks
- Decreases read latency for batches of blocks due to parallel reads, but does not affect write latency due to sequential writes
- Improves fault tolerance
- Pay for more capacity than is effectively used

Use an appropriate disk-scheduling algorithm that decides in what order outstanding requests for disk blocks will be serviced
- Elevator algorithm
- It is most effective when there are many outstanding requests
Prefetching
- when reading a block, prefetch a few more neighboring blocks, while
- Good when spatial locality in block accesses exists, but timing of requests if unpredictable

Double buffering
- overlap the processing of one block with the reading of another block

Both requires extra main memory
Disk failures

- Intermittent failures
  - Attempt to access a block fails, but repeated tries succeed

- Checksums
  - Detect failure to read/write correctly using checksum bits in each sector

- Media decay
  - Some bits are permanently corrupted

- Write failure
  - Can not rewrite or read a sector

- Stable storage
  - use two actual sectors to store the contents of one and following a write-all-read-one strategy
  - Handles media decay and write failures
Disk failures

Disk crashes

- Suddenly entire disk becomes permanently inaccessible
- Mirroring can help with crashes but may use too many extra disks
- Redundancy together with error-correcting coding techniques can handle crashes without substantial increase in the number of extra disks (e.g., RAID)
Representing data elements
Representing SQL datatypes

- **NUMBER**
  - Use a fixed number of bits depending on type of number (short, long, etc, int vs real, etc)

- **CHAR(n)**
  - Array of n bytes

- **VARCHAR(n)**
  - Length plus content
    - Array of n+1 bytes, with 1\textsuperscript{st} byte being the length of the content stored
    - Null-terminated string
      - Array of n+1 bytes to store contents after padding them with nulls to length n+1

- **DATE and TIME**
  - Store as strings with known length (except for time with unlimited precision)

- **BITS(n)**
  - Use array of ceiling(n/8) bytes

- **Enumerated types**
  - Use integers
Fields, Records, and Relations

- A field is the smallest piece of data that can be manipulated
  - Each field has a specific datatype

- A record is a collection of related fields

- Records can have
  - Fixed or variable format
  - Fixed of variable length

- A relation is a bag of records
Building records

CPU architecture may require fields to start at certain addresses
- Assume all fields and records always start at addresses that are multiple of 4.

Record schema
- Each record should indicate its schema since
  - Schema may change
  - Blocks may contain records from many relations with different schemas
  - Can not determine record schema from the record’s location in disk

Record header
- Schema
- Record length
- Read/write timestamps
Packing fixed-length records into blocks

- Place records sequentially, while they fit, following an optional block header

- Block header
  - Directory of records within block
  - Relation to which records belong to
  - Timestamps of last access or modification
  - Block ID
  - Links to other related blocks in a network of blocks
  - Role of this block in the network

- What if a record is larger than a block?
Block and record addresses

- Blocks and records are referenceable items
- Referenceable items exist in address spaces
  - Database address space
  - Memory address space

Address types are

- Physical address
  - A string indicating the physical location of the item (e.g., machine, disk, cylinder, track, block, offset)
- Logical address
  - Fixed length string

Map of physical-logical addresses

Logical addresses offer considerable flexibility in managing blocks/records and referring to them
**Pointer swizzling**

- Pointers are often parts of records and blocks
- There is need to manage such pointers when moving referenceable between memory and secondary storage
  - Logical and physical addresses are both database addresses
- Techniques to avoid translating repeatedly between database and memory addresses are known as pointer swizzling
- Pointer swizzling uses
  - Extra bit in each pointer to indicate whether it has been swizzled (i.e., converted to memory address)
  - A translation table between database and memory addresses
**Pointer swizzling**

- **Automatic swizzling**
  - Upon moving an item in memory
    - Enter the database-memory address pair for the item in the translation table
    - locate all the pointers within the item and swizzle them, if the item referred to is in memory
  - When following an unswizzled pointer, swizzled upon reading the referenced item
- **Swizzling on demand**
  - Swizzle a point when following it and find the referenced item in memory
- **No swizzling**
- **Programmer control**
  - An referenceable item is pinned if it cannot currently be moved back to disk safely
  - An item that has swizzled pointers to it, is pinned
Variable-length records

Records with variable-length fields
- Place fixed-length fields followed by the variable length fields
- Use record header to indicate length of record, and offsets of its fields

Records with repeating fields
- Use similar approach as for variable-length fields

Alternative – split records
- Use separate block to store variable-length or repeated fields
- Use record header to contain pointer(s) to these fields

Compromise
- Store some of the repeated fields with the record, storing additional occurrences in another block
Variable-format records

The fields of a record or their order is not determined by the relation the record represents

- Use tagged fields
  - Information about the role of the field (e.g., name, type, length, etc)
  - Value

- Tagged fields are useful for records from
  - Information integration applications
  - Records with very flexible schemas
Spanned records

Sometimes is desirable to have records split occupying more than one block

- Records that don’t fit in a block
- Large records that lead to high internal fragmentation

A fragment is the portion of a record that occupies a single block

- A record is thus a sequence of fragments

Record/fragment headers contain information indicating

- Whether it is a fragment
- Whether it is the first/last fragment
- Pointers to the next/previous fragments
Storing BLOBs

- BLOBs are really large fields
  - Can be stored in
    - a linked list of blocks
    - in a contiguous sequence of blocks
  - Can be stripped across multiple disks

When a client retrieves a BLOB

- We ship few blocks of the BLOB at a time as needed
- While we ship all the other small fields at once
Inserting records

Upon locating block that record should be inserted to

- If block has space for new record insert the record
  - If records are stored in a particular order within the block, and other records can be moved within the block, then do so
  - adjusting the block’s offset

- Otherwise
  - Create a overflow block and insert the record there
    - A special block holding records that would had been in another block if it had space
  - Find space in a close-by block
    - Possible moving some other records to that block, due to an order requirement for the records of a block
Deleting and updating records

When deleting a record from a block

- If records can be moved within the block, then do so, else need to keep track of free space within the block
- Update the block’s offset table
- Reclaim any free space
- Care is required to avoid dangling pointers
  - ensure that there are no database pointers to the deleted item from elsewhere
  - If there are or there could be, then place a tombstone that will persist until the database is reconstructed

Updating a record can be handled using the methods for inserting/deleting a record