

Chapter 10: Storage and File Structure

- Overview of Physical Storage Media
- Magnetic Disks
- RAID
- Tertiary Storage
- Storage Access
- File Organization
- Organization of Records in Files
- Data-Dictionary Storage
- Storage Structures for Object-Oriented Databases

Classification of Physical 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

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 hardware/operating system.
- Main memory:
 - general-purpose machine instructions operate on data resident in main memory
 - fast access, but generally too small to store the entire database
 - sometimes referred to as **core** memory
 - volatile — contents of main memory are usually lost if a power failure or system crash occurs

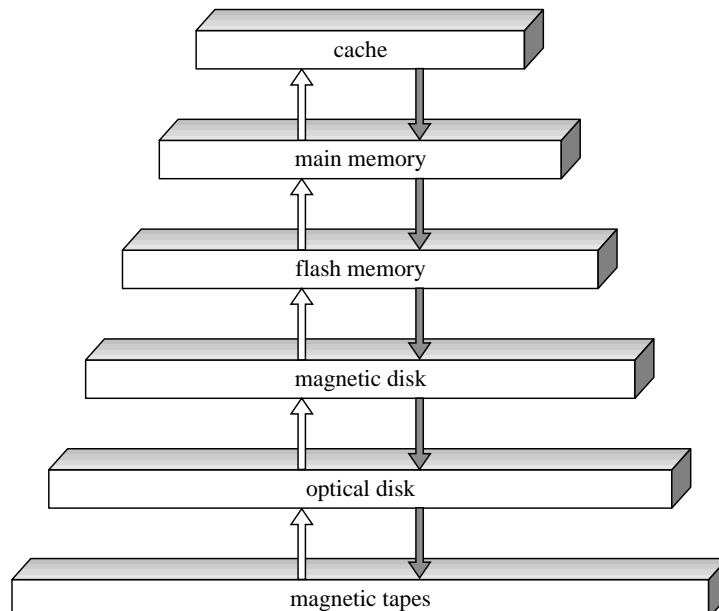
Physical Storage Media (Cont.)

- Flash memory – reads are roughly as fast as main memory; data survives power failure; but can support a only limited number of write/erase cycles
- Magnetic-disk storage – 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
 - usually survives power failures and system crashes; disk failure can destroy data, but is much less frequent than system crashes

Physical Storage Media (Cont.)

- Optical storage – non-volatile. CD-ROM most popular form. Write-once, read-many (WORM) optical disks used for archival storage.
- 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 (5GB tapes are common)
 - tape can be removed from drive \Rightarrow storage costs much cheaper than disk.

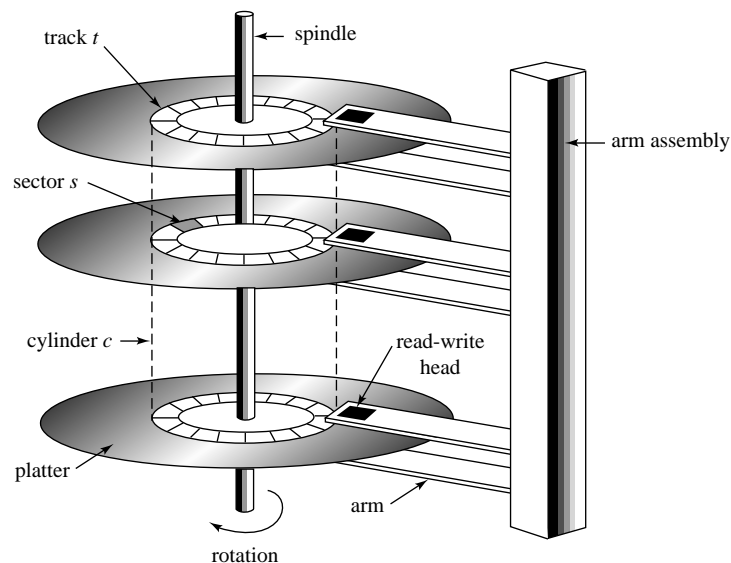
Storage Hierarchy



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** (flash memory, magnetic disks)
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time; also called **off-line storage** (magnetic tape, optical storage)

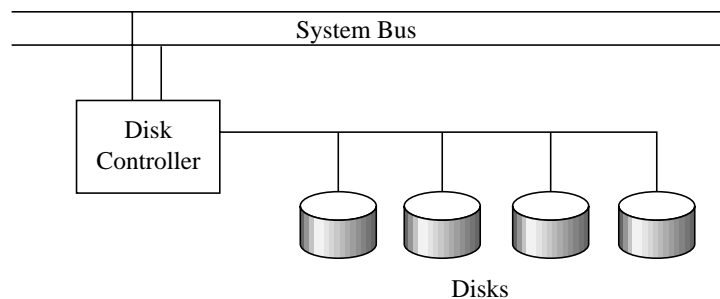
Magnetic Disks Mechanism



Magnetic Disks

- Read–write head – device positioned close to the platter surface; reads or writes magnetically encoded information.
- Surface of platter divided into circular **tracks**, and each track is divided into **sectors**. A sector is the smallest unit of data that can be read or written.
- To read/write a sector
 - disk arm swings to position head on right track
 - platter spins continually; data is read/written when sector comes under head
- Head–disk assemblies – multiple disk platters on a single spindle, with multiple heads (one per platter) mounted on a common arm.
- **Cylinder i** consists of i th track of all the platters

Disk Subsystem



- Disk controller – interfaces between the computer system and the disk drive hardware.
 - accepts high-level 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.

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 1/3rd the worst case seek time.
 - **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.
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.

Optimization of Disk-Block Access

- **Block** – a contiguous sequence of sectors from a single track
 - data is transferred between disk and main memory in blocks
 - sizes range from 512 bytes to several kilobytes
- Disk-arm-scheduling algorithms order accesses to tracks so that disk arm movement is minimized (**elevator algorithm** is often used)
- File organization – optimize block access time by organizing the blocks to correspond to how data will be accessed. Store related information on the same or nearby cylinders.
- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM buffer immediately; controller then writes to disk whenever the disk has no other requests.
- **Log disk** – a disk devoted to writing a sequential log of block updates; this eliminates seek time. Used like nonvolatile RAM.

RAID

- **Redundant Arrays of Inexpensive Disks** – disk organization techniques that take advantage of utilizing large numbers of inexpensive, mass-market disks.
- Originally a cost-effective alternative to large, expensive disks
- Today RAIDs are used for their higher reliability and bandwidth, rather than for economic reasons. Hence the “I” is interpreted as **independent**, instead of **inexpensive**.

Improvement of Reliability via Redundancy

- The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail. E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days).
- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g. **Mirroring** (or **shadowing**)
 - duplicate every disk. Logical disk consists of two physical disks.
 - every write is carried out on both disks
 - if one disk in a pair fails, data still available in the other

Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
 1. Load balance multiple small accesses to increase throughput
 2. Parallelize large accesses to reduce response time
- Improve transfer rate by striping data across multiple disks.
- **Bit-level striping** – split the bits of each byte across multiple disks
 - In an array of eight disks, write bit i of each byte to disk i .
 - Each access can read data at eight times the rate of a single disk.
 - But seek/access time worse than for a single disk.
- **Block-level striping** – with n disks, block i of a file goes to disk $(i \bmod n) + 1$.

RAID Levels

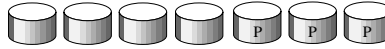
- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
- Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- **Level 0**: Striping at the level of blocks; non-redundant. Used in high-performance applications where data loss is not critical.
- **Level 1**: Mirrored disks; offers best write performance. Popular for applications such as storing log files in a database system.



(a) RAID 0: Non-Redundant Striping



(b) RAID 1: Mirrored Disks



(c) RAID 2: Memory Style Error Correcting Codes



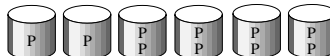
(d) RAID 3: Bit Interleaved Parity



(e) RAID 4: Block Interleaved Parity



(f) RAID 5: Block-Interleaved Distributed Parity



(g) RAID 6: P + Q Redundancy

RAID Levels (Cont.)

- **Level 2:** Memory-Style Error-Correcting-Codes (ECC) with bit striping.
- **Level 3:** Bit-Interleaved Parity; a single parity bit can be used for error correction, not just detection.
 - When writing data, parity bit must also be computed and written
 - Faster data transfer than with a single disk, but fewer I/Os per second since every disk has to participate in every I/O.
 - Subsumes Level 2 (provides all its benefits, at lower cost).

RAID Levels (Cont.)

- **Level 4:** Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from N other disks.
 - Provides higher I/O rates for independent block reads than Level 3 (block read goes to a single disk, so blocks stored on different disks can be read in parallel)
 - Provides high transfer rates for reads of multiple blocks
 - However, parity block becomes a bottleneck for independent block writes since every block write also writes to parity disk

RAID Levels (Cont.)

- **Level 5:** Block-Interleaved Distributed Parity; partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.
 - E.g., with 5 disks, parity block for n th set of blocks is stored on disk $(n \bmod 5) + 1$, with the data blocks stored on the other 4 disks.
 - Higher I/O rates than Level 4. (Block writes occur in parallel if the blocks and their parity blocks are on different disks.)
 - Subsumes Level 4
- **Level 6:** P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures. Better reliability than Level 5 at a higher cost; not used as widely.

Optical Disks

- Compact disk–read only memory (CD-ROM)
 - Disks can be loaded into or removed from a drive
 - High storage capacity (about 500 MG on a disk).
 - High seek times and latency; lower data-transfer rates than magnetic disks
- Digital video disk – new optical format; holds 4.7 to 17 GB
- WORM disks (Write-Once Read-Many) – can be written using the same drive from which they are read.
 - data can only be written once, and cannot be erased.
 - high capacity and long lifetime; used for archival storage
 - WORM jukeboxes

Magnetic Tapes

- Hold large volumes of data (5GB tapes are common)
- Currently the cheapest storage medium
- Very slow access time in comparison to magnetic 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 (terabyte (10^{12}) to petabyte (10^{15})) storage

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
 - The requesting program is given the address of the block in main memory, if it is already present in the buffer.
 - 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 in the block from the disk to the buffer, and passes the address of the block in main memory to the requester.

Buffer-Replacement Policies

- Most operating systems replace the block least recently used (LRU)
- 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 (Cont.)

- **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.
- Deletion of record i — alternatives:
 - move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 - move record n to i
 - Link all free records on a *free list*

Free Lists

- Store the address of the first record whose contents are deleted in the file header.
- Use this first record to store the address of the second available record, and so on
- Can think of these stored addresses as *pointers* since they “point” to the location of a record.

| | | | | |
|----------|------------|-------|-----|--|
| header | | | | |
| record 0 | Perryridge | A-102 | 400 | |
| record 1 | | | | |
| record 2 | Mianus | A-215 | 700 | |
| record 3 | Downtown | A-101 | 500 | |
| record 4 | | | | |
| record 5 | Perryridge | A-201 | 900 | |
| record 6 | | | | |
| record 7 | Downtown | A-110 | 600 | |
| record 8 | Perryridge | A-218 | 700 | |

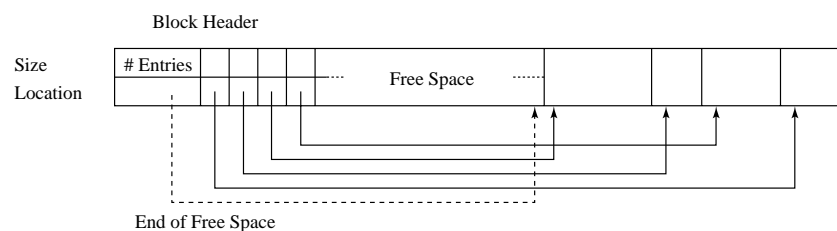
Free Lists (Cont.)

- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)
- **Dangling pointers** occur if we move or delete a record to which another record contains a pointer; that pointer no longer points to the desired record.
- Avoid moving or deleting records that are pointed to by other records; such records are **pinned**.

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).
- Byte string representation
 - Attach an *end-of-record* (\perp) control character to the end of each record
 - Difficulty with deletion
 - Difficulty with growth

Variable-Length Records: Slotted Page Structure



- 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 then be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.

Variable-Length Records (Cont.)

- Fixed-length representation:
 - reserved space
 - pointers
- Reserved space – can use fixed-length records of a known maximum length; unused space in shorter records filled with a null or end-of-record symbol.

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

Pointer Method

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

- Pointers – the maximum record length is not known; a variable-length record is represented by a list of fixed-length records, chained together via pointers.

Pointer Method (Cont.)

- Disadvantage to pointer structure; space is wasted in all records except the first in a chain.
- Solution is to allow two kinds of block in file:
 - Anchor block – contains the first records of chain
 - Overflow block – contains records other than those that are the first records of chains.



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 is computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- **Clustering** – records of several different relations can be stored in the same file; related records are stored on the same block

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*

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



Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion – must locate the position in the file 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

Clustering File Organization

- Simple file structure stores each relation in a separate file
- Can instead store several relations in one file using a **clustering** file organization
- E.g., clustering organization of *customer* and *depositor*.

| | | |
|--------|--------|----------|
| Hayes | Main | Brooklyn |
| Hayes | A-102 | |
| Hayes | A-220 | |
| Hayes | A-503 | |
| Turner | Putnam | Stamford |
| Turner | A-305 | |

- good for queries involving *depositor* ⋈ *customer*, and for queries involving one single customer and his accounts
- bad for queries involving only *customer*
- results in variable size records

Data Dictionary Storage

Data dictionary (also called system catalog) stores metadata: that is, data about data, such as

- Information about relations
 - names of relations
 - names and types of attributes
 - physical file organization information
 - statistical data such as number of tuples in each relation
- integrity constraints
- view definitions
- user and accounting information
- information about indices (Chapter 11)

Data Dictionary Storage (Cont.)

- Catalog structure: can use either
 - specialized data structures designed for efficient access
 - a set of relations, with existing system features used to ensure efficient access

The latter alternative is usually preferred

- A possible catalog representation:

System-catalog-schema = (*relation-name*, *number-of-attributes*)

Attribute-schema = (*attribute-name*, *relation-name*, *domain-type*,
position, *length*)

User-schema = (*user-name*, *encrypted-password*, *group*)

Index-schema = (*index-name*, *relation-name*, *index-type*,
index-attributes)

View-schema = (*view-name*, *definition*)

Mapping of Objects to Files

- Mapping objects to files is similar to mapping tuples to files in a relational system; object data can be stored using file structures.
- Objects in O-O databases may lack uniformity and may be very large; such objects have to be managed differently from records in a relational system.
 - Set fields with a small number of elements may be implemented using data structures such as linked lists.
 - Set fields with a larger number of elements may be implemented as B-trees, or as separate relations in the database.
 - Set fields can also be eliminated at the storage level by normalization.

Mapping of Objects to Files (Cont.)

- Objects are identified by an object identifier (OID); the storage system needs a mechanism to locate an object given its OID.
 - **logical identifiers** do not directly specify an object's physical location; must maintain an index that maps an OID to the object's actual location.
 - **physical identifiers** encode the location of the object so the object can be found directly. Physical OIDs typically have the following parts:
 1. a volume or file identifier
 2. a page identifier within the volume or file
 3. an offset within the page

Management of Persistent Pointers

- Physical OIDs may have a *unique identifier*. This identifier is stored in the object also and is used to detect references via dangling pointers.

| Physical Object Identifier | | | | Object | |
|----------------------------|------|--------|-----------|-----------|------------|
| Vol. | Page | Offset | Unique-Id | Unique-Id | Data |

(a) General Structure

| Location | Unique-Id | Data |
|------------|------------|---------------|
| 6.32.45608 | 51 |data |
| Good OID | 6.32.45608 | 51 |
| Bad OID | 6.32.45608 | 50 |

(b) Example of use

Management of Persistent Pointers (Cont.)

- Implement persistent pointers using OIDs; persistent pointers are substantially longer than are in-memory pointers
- Pointer swizzling cuts down on cost of locating persistent objects already in memory.
- Software swizzling (swizzling on pointer dereference)
 - When a persistent pointer is first dereferenced, it is **swizzled** (replaced by an in-memory pointer) after the object is located in memory.
 - Subsequent dereferences of the same pointer become cheap
 - The physical location of an object in memory must not change if swizzled pointers point to it; the solution is to pin pages in memory
 - When an object is written back to disk, any swizzled pointers it contains need to be **unswizzled**.

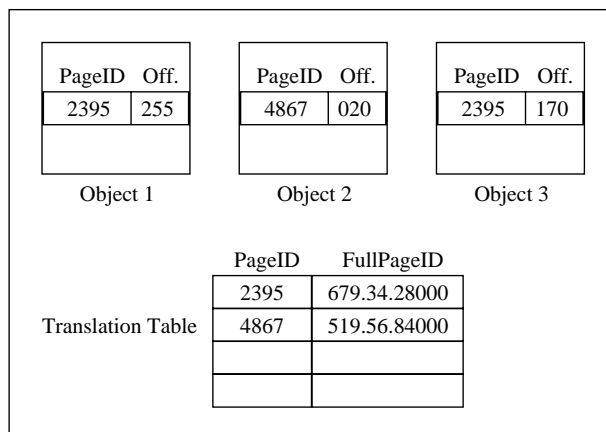
Hardware Swizzling

- Persistent pointers in objects need the same amount of space as in-memory pointers — extra storage external to the object is used to store rest of pointer information
- Uses virtual memory translation mechanism to efficiently and transparently convert between persistent pointers and in-memory pointers.
- All persistent pointers in a page are swizzled when the page is first read in.
 - thus programmers have to work with just one type of pointer, i.e. in-memory pointer.
 - some of the swizzled pointers may point to virtual memory addresses that are currently not allocated any real memory

Hardware Swizzling

- Persistent pointer is conceptually split into two parts: a page identifier, and an offset within the page.
 - The page identifier in a pointer is a short indirect pointer: Each page has a translation table that provides a mapping from the short page identifiers to full database page identifiers.
 - Translation table for a page is small (at most 1024 pointers in a 4096 byte page with 4 byte pointers)
 - Multiple pointers in a page to the same page share same entry in the translation table.

Hardware Swizzling (Cont.)



- Page image when on disk (before swizzling)

Hardware Swizzling (Cont.)

- When an in-memory pointer is dereferenced, if the operating system detects the page it points to has not yet been allocated storage, a **segmentation violation** occurs.
- `mmap` call associates function to be called on segmentation violation
- The function allocates storage for the page and reads in the page from disk.
- Swizzling is then done for all persistent pointers in the page (located using object type information).
 - If pointer points to a page not already allocated a virtual memory address, a virtual memory address is allocated (preferably the address in the short page identifier if it is unused). Storage is not yet allocated for the page.
 - The page identifier in pointer (and translation table entry) are changed to the virtual memory address of the page

Hardware Swizzling (Cont.)

| PageID | Off. | PageID | Off. | PageID | Off. |
|----------|------|----------|------|----------|------|
| 5001 | 255 | 4867 | 020 | 5001 | 170 |
| | | | | | |
| Object 1 | | Object 2 | | Object 3 | |

| PageID | FullPageID |
|--------|--------------|
| 5001 | 679.34.28000 |
| 4867 | 519.56.84000 |
| | |
| | |

Translation Table

Page image after swizzling

- Page with short page identifier 2395 was allocated address 5001. Observe change in pointers and translation table.
- Page with short page identifier 4867 has been allocated address 4867. No change in pointers and translation table

Hardware Swizzling (Cont.)

- After swizzling, all short page identifiers point to virtual memory address allocated for the page
 - functions accessing the objects need not know it has persistent pointers!
 - can reuse existing code and libraries that use in-memory pointers
- If all pages are allocated the same address as in the short page identifier, no changes required in the page!
- No need for deswizzling — page after swizzling can be saved back directly to disk
- A process should not access more pages than size of virtual memory — reuse of virtual memory addresses for other pages is expensive

Disk versus Memory Structure of Objects

- The format in which objects are stored in memory may be different from the format in which they are stored on disk in the database. Reasons are :-
 - software swizzling – structure of persistent and in-memory pointers are different
 - database accessible from different machines, with different data representations
- Make the physical representation of objects in the database independent of the machine and the compiler.
- Can transparently convert from disk representation to form required on the specific machine, language, and compiler, when the object (or page) is brought into memory.

Large Objects

- Very large objects are called **binary large objects (blobs)** because they typically contain binary data. Examples include:
 - text documents
 - graphical data such as images and computer aided designs
 - audio and video data
- Large objects may need to be stored in a contiguous sequence of bytes when brought into memory.
 - If an object is bigger than a page, contiguous pages of the buffer pool must be allocated to store it.
 - May be preferable to disallow direct access to data, and only allow access through a file-system-like API, to remove need for contiguous storage.

Modifying Large Objects

- Use B-tree structures to represent object: permits reading the entire object as well as updating, inserting and deleting bytes from specified regions of the object.
- Special-purpose application programs outside the database are used to manipulate large objects:
 - Text data treated as a byte string manipulated by editors and formatters.
 - Graphical data is represented as a bit map or as a set of geometric objects; can be managed within the database system or by special software (i.e., VLSI design).
 - Audio/video data is typically created and displayed by separate application software and modified using special purpose editing software.
 - *checkout/checkin* method for concurrency control and creation of versions