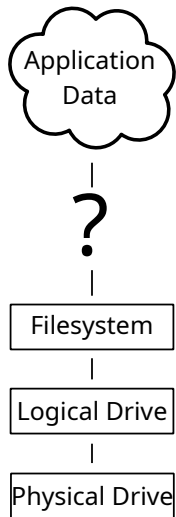


Storage

The Problem



Requirements

There are different classes of requirements:

- Data Independence
 - ▶ application is shielded from physical storage
 - ▶ physical storage can be reorganized
 - ▶ hardware can be changed
- Scalability
 - ▶ must scale to (nearly) arbitrary data sizes
 - ▶ fast retrieval
 - ▶ efficient access to individual data items
 - ▶ updating arbitrary data
- Reliability
 - ▶ data must never be lost
 - ▶ must cope with hardware and software failures
- ...

Layer Architecture

- implementing all these requirements on the “bare metal” is hard
- and not desirable
- a DBMS must be maintainable and extensible

Instead: use a **layer** architecture

- the DBMS logic is split into levels of functionality
- each level is implemented by a specific layer
- each layer interacts only with the next lower layer
- simplifies and modularizes the code

A Simple Layer Architecture

Purpose

query translation
and optimization

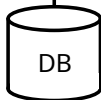
managing records
and access paths

DB buffer and
hardware interface

query layer

access layer

storage layer



Access Granularity

declarative queries
sets of records

records

page

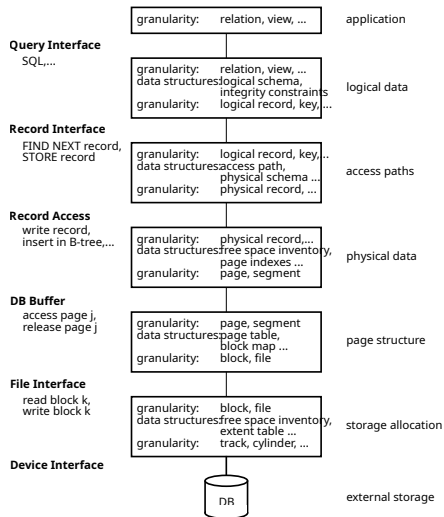
A Simple Layer Architecture (2)

- layers can be characterized by the data items they manipulate
- lower layer offers functionality for the next higher level
- keeps the complexity of individual layers reasonable
- rough structure: physical → low level → high level

This is a reasonable architecture, but simplified.

A more detailed architecture is needed for a complete DBMS.

A More Detailed Architecture



A More Detailed Architecture (2)

A few pieces are still missing:

- transaction isolation
- recovery

but otherwise it is a reasonable architecture.

Some system deviate slightly from this classical architecture

- most DBMSs nowadays delegate drive access to the OS
- some DBMSs delegate buffer management to the OS (tricky, though)
- a few DBMSs allow for direct logical record access
- ...

Influence of Hardware

Must take hardware into account when designing a storage system.

For a long time dominated by **Moore's Law**:

The number of transistors on a chip doubles every 18 month.

Indirectly drove a number of other parameters:

- main memory size
- CPU speed
 - ▶ no longer true!
- HD capacity
 - ▶ start getting problematic, too. density is very high
 - ▶ only capacity, not access time

Later we will look at these again.

Memory Hierarchy

capacity
latency

bytes
1ns

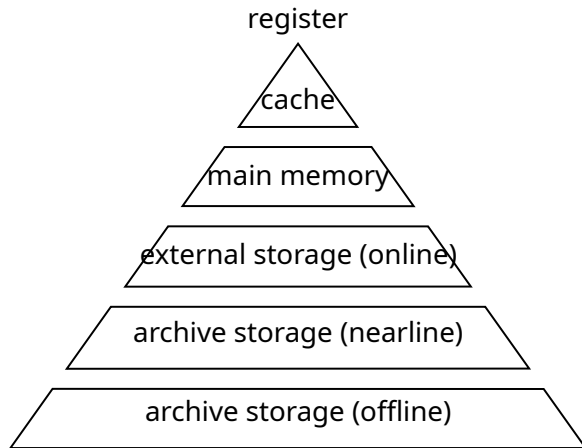
K-M bytes
<10ns

G bytes
<100ns

T bytes
ms

T bytes
sec

T-P bytes
sec-min



Memory Hierarchy (2)

There are huge gaps between hierarchy levels

- traditionally, main memory vs. disk is most important
- but memory vs. cache etc. also relevant

The DBMS must aim to maximize locality.

Hard Disk Access

Hard Disks are still the dominant external storage:

- rotating platters, mechanical effects
- transfer rate: ca. 150MB/s
- seek time ca. 3ms
- huge imbalance in random vs. sequential I/O!

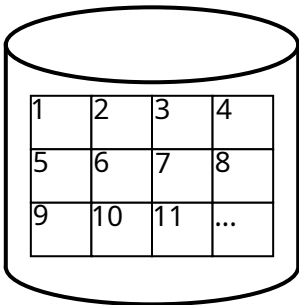
The DBMS must take these effects into account

- sequential access is much more efficient
- gap is growing instead of shrinking
- even SSDs are slightly asymmetric (and have other problems)

Hard Disk Access (2)

Techniques to speed up disk access:

- do not move the head for every single tuple
- instead, load larger chunks
- typical granularity: one **page**
- page size varies. traditionally 4KB, nowadays often 16K and more
- page size is a trade-off



Hard Disk Access (3)

The page structure is very prominent within the DBMS

- granularity of I/O
- granularity of buffering/memory management
- granularity of recovery

Page is still too small to hide random I/O though

- sequential page access is important
- DBMSs use read-ahead techniques
- asynchronous write-back

Buffer Management

Some pages are accessed very frequently

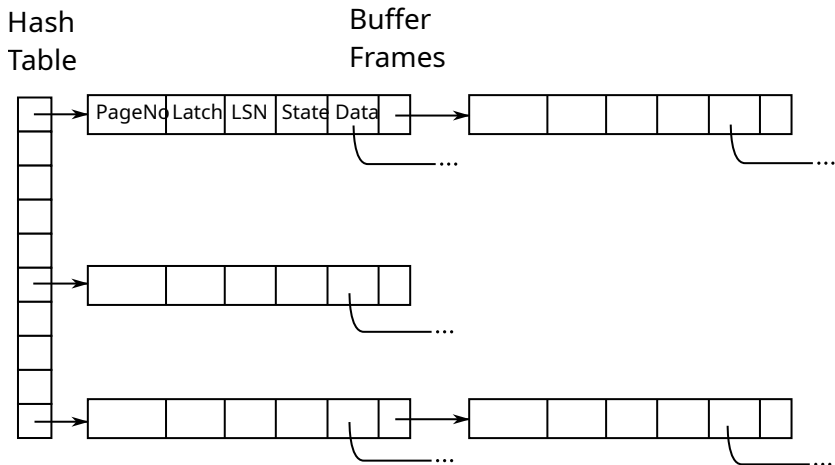
- reduce I/O by buffering/caching
- buffer manager keeps active pages in memory
- limited size, discards/write back when needed
- coupled with recovery, in particular logging

Basic interface:

1. FIX(pageNo,shared)
2. UNFIX(pageNo,dirty)

Pages can only be accessed (or modified) when they are **fixed**.

Buffer Management



The buffer manager itself is protected by one or more latches.

Buffer Frame

Maintains the state of a certain page within the buffer

pageNo	the page number
latch	a read/writer lock to protect the page (note: must not block unrelated pages!)
LSN	LSN of the last change, for recovery (buffer manager must force the log before writing)
state	clean/dirty/newly created etc.
data	the actual data contained on the page

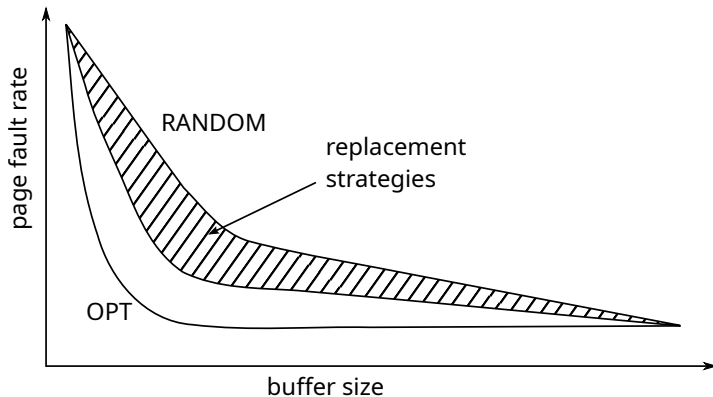
(will usually contain extra information for buffer replacement)

Usually kept in a hash table.

Buffer Replacement

When memory is full, some buffer pages have to be replaced

- clean pages can be simply discarded
- dirty pages have to be written back first
- discarded pages are replaced with new pages



Buffer Replacement - FIFO

First In - First Out

- simple replacement strategy
- buffer frames are kept in a linked list
- inserted at the end, remove from the head
- “old” pages are removed first

Does not take locality into account.

Buffer Replacement - LRU

Least Recently Used

- similar to FIFO, buffer frames are kept in a double-linked list
- remove from the head
- when a frame is unfixed, move it to the end of the list
- “hot” pages are kept in the buffer

A very popular strategy. Latching requires some care.

Buffer Replacement - LFU

Least Frequently Used

- remembers the number of access per page
- in-frequently used pages are remove first
- maintains a priority queue of pages

Sounds plausible, but too expensive in practice.

Buffer Replacement - Second Chance

LRU is nice, but the LRU list is a hot spot.

Idea: use a simpler mechanism to simulate LRU

- one bit per page
- bit is set when page is unfixed
- when replacing, replace pages with unset bit
- set bits are cleared during the process
- strategy is called “second chance” or “clock”

Easy to implement, but a bit crude.

Buffer Replacement - 2Q

Maintain not one queue but two

- many pages are referenced only once
 - some pages are hot and reference frequently
 - maintain a separate list for those
1. maintain all pages in FIFO queue
 2. when a page is referenced again that is currently in FIFO, move it into an LRU queue
 3. prefer evicting from FIFO

Hot pages are in LRU, read-once pages in FIFO. Good strategy for common DBMS operations.

Buffer Replacement - Hints

Application knowledge can help buffer replacement

- 2Q tries to recognize read-once pages
- these occur when scanning over data
- but the DBMS knows this anyway!
- it could therefore give **hints** when unfixing
- e.g., *will-need*, or *will-not-need* (changes placement in queue)

Segments

While page granularity is fine for I/O, it is somewhat unwieldy

- most structures within a DBMS span multiple pages
- relations, indexes, free space management, etc.
- convenient to treat these as one entity
- all DBMS pages are partitioned into sets of pages

Such a set of pages is called a **segment**.

Conceptually similar to a file or virtual memory.

Segments (2)

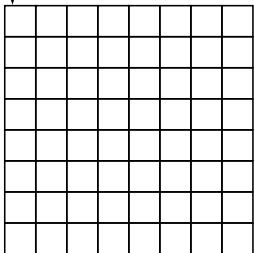
A segment offers a virtual address space within the DBMS

- can allocate and release new pages
- can iterate over all pages
- can drop the whole segment
- optionally offers a linear address space

Greatly simplifies the logic of higher layers.

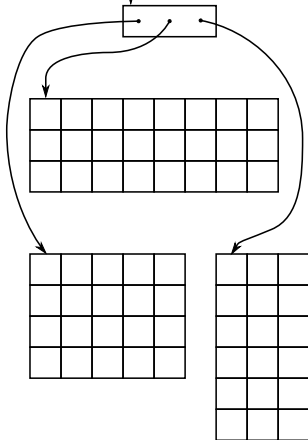
Block Allocation

Catalog



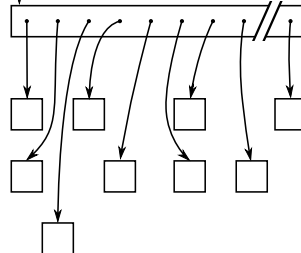
static file-mapping

Catalog



dynamic extent-mapping

Catalog



dynamic block-mapping

Block Allocation

All approaches have pros and cons:

- static file-mapping
 - ▶ very simple, low overhead
 - ▶ resizing is difficult
- dynamic block-mapping
 - ▶ maximum flexibility
 - ▶ administrative overhead, additional indirection
- dynamic extent-mapping
 - ▶ can handle growth
 - ▶ slight overhead

In most cases extent-based mapping is preferable.

Block Allocation (5)

Dynamic extent-mapping:

- grows by adding a new extent
- should grow exponentially (e.g., factor 1.25)
- bounds the number of extents
- reduces both complexity and space consumption
- and helps with sequential I/O!

Segment Types

Segments can be classified into types

- private vs. public
- permanent vs. temporary
- automatic vs. manual
- with recovery vs. without recovery

Differ in complexity and required effort.

Standard Segments

Most DBMS will need at least two low-level segments:

- segment inventory
 - ▶ keeps track of all pages allocated to segments
 - ▶ keeps extent lists or page tables or ...
- free space inventory
 - ▶ keeps track of free pages
 - ▶ maintains bitmaps or free extents or ...

High-level segments built upon these.

Common high-level segments:

- schema
- relations
- temp segments (created and discard on demand)
- ...

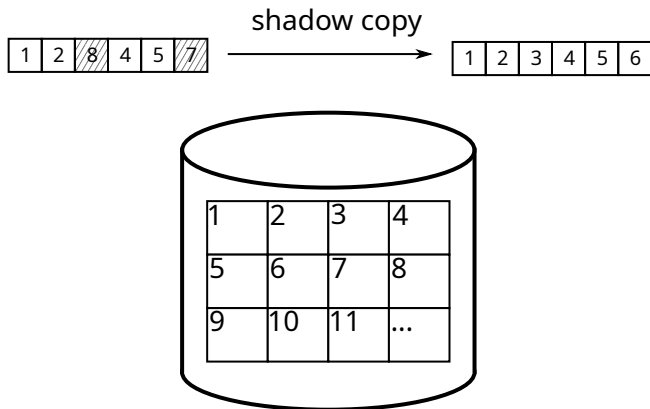
Update Strategies

DBMSs have different update behavior

	steal	\neg steal
force		
\neg force		

- usually one prefers steal, \neg force
- but then pages contain dirty data
- when using *update-in-place*, dirty data is on on disk
- complicates recovery
- transactions could see dirty data

Shadow Paging



uses a page table, dirty pages are stored in a shadow copy.

Shadow Paging (2)

Advantages:

- the clean data is always available on disk
- greatly simplified recovery
- can be used for transaction isolation, too

Disadvantages:

- complicates the page access logic
- destroys data locality

Nowadays rarely used in disk-based systems.

Delta Files

Similar idea to shadow paging:

- on change pages are copied to a separate file
- a copied page can be changed in-place
- on commit discard the file, on abort copy back

Can be implemented in two flavors:

- store a clean copy in the delta
- store the dirty data in the delta

Both have pros and cons.

Delta Files (2)

Delta files have some advantages over shadow paging:

- preserve data locality
- no mixture of clean and dirty pages

Disadvantages:

- cause more I/O
- abort (or commit) becomes expensive
- keeping track of delta pages is non-trivial

Still, often preferable over shadow paging.