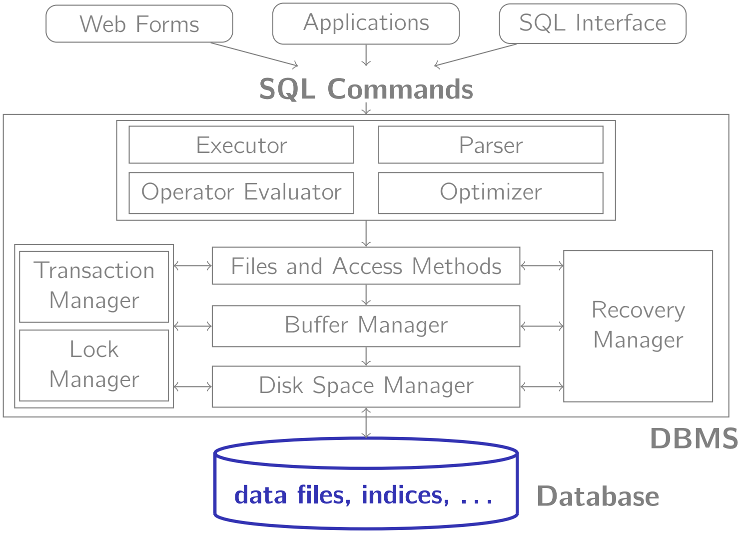
Architecture and Implementation of Database Systems (Winter 2015/16)

Jens Teubner, DBIS Group
`jens.teubner@cs.tu-dortmund.de`

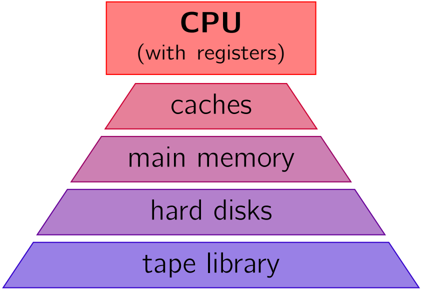
Winter 2015/16

Part II

Storage: Disks and Files


Web Forms
Applications
SQL Interface
SQL Commands
Executor
Parser
Operator Evaluator
Optimizer
Transaction Manager
Lock Manager
Files and Access Methods
Buffer Manager
Disk Space Manager
Recovery Manager
DBMS
data files, indices, . . .
Database

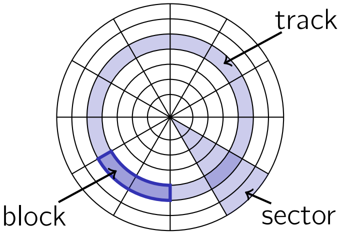
Memory Hierarchy

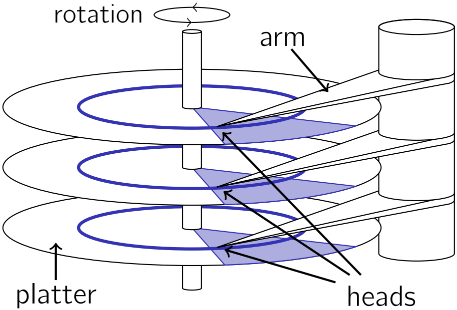


capacity	latency
bytes	< 1 ns
kilo-/megabytes	< 10 ns
gigabytes	70–100 ns
terabytes	3–10 ms
petabytes	varies

- fast, but expensive and small, memory close to CPU
- larger, slower memory at the periphery
- We'll try to hide latency by using the fast memory as a **cache**.

Magnetic Disks





- A stepper motor positions an array of disk heads on the requested track.
- Platters (disks) steadily rotate.
- Disks are managed in blocks: the system reads/writes data one block at a time.

Photo: http://www.metallurgy.utah.edu/

Access Time

This design has implications on the **access time** to read/write a given block:

1. Move disk arms to desired track (**seek time** t_s).
2. Wait for desired block to rotate under disk head (**rotational delay** t_r).
3. Read/write data (**transfer time** t_{tr})

$\rightarrow$ **access time:** $t = t_s + t_r + t_{tr}$

Example: Notebook drive Hitachi Travelstar 7K200

- 4 heads, 2 disks, 512 bytes/sector, 200 GB capacity
- rotational speed: 7200 rpm
- average seek time: 10 ms
- transfer rate: ≈ 50 MB/s

✎ **What is the access time to read an 8 KB data block?**

Sequential vs. Random Access

Example: Read 1000 blocks of size 8 KB

- **random access:**

  $t_{\text{rnd}} = 1000 \cdot 14.33\,\text{ms} = 14.33\,\text{s}$

- **sequential read:**

  $$\begin{aligned} t_{\text{seq}} &= t_s + t_r + 1000 \cdot t_{tr} + \tfrac{16 \cdot 1000}{63} \cdot t_{s,\text{track-to-track}} \\ &= 10\,\text{ms} + 4.14\,\text{ms} + 160\,\text{ms} + 254\,\text{ms} \approx 428\,\text{ms} \end{aligned}$$

  The Travelstar 7K200 has 63 sectors per track, with a 1 ms track-to-track seek time; one 8 KB block occupies 16 sectors.

→ Sequential I/O is **much** faster than random I/O.

→ **Avoid random I/O** whenever possible.

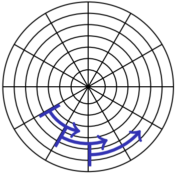
→ As soon as we need at least $\frac{428\,\text{ms}}{14330\,\text{ms}} = 3\,\%$ of a file, we better read the **entire** file!

Performance Tricks

System builders play a number of tricks to improve performance.

track skewing

Align sector 0 of each track to avoid rotational delay during sequential scans.



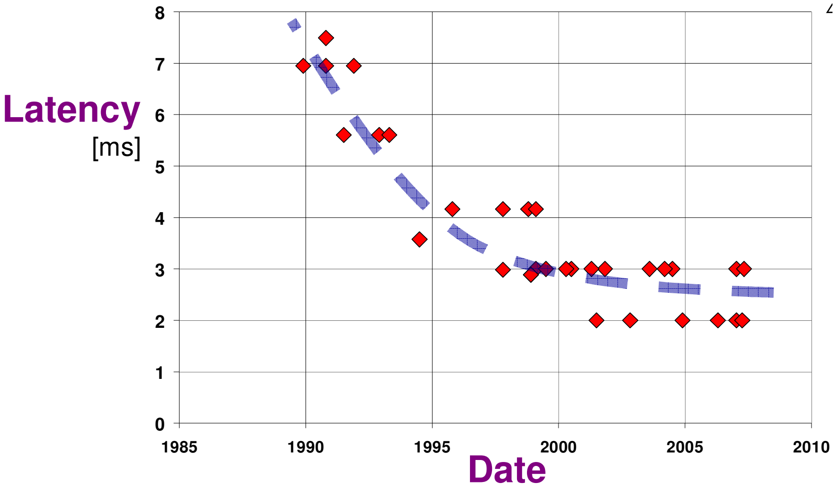
request scheduling

If multiple requests have to be served, choose the one that requires the smallest arm movement (SPTF: shortest positioning time first).

zoning

Outer tracks are longer than the inner ones. Therefore, divide outer tracks into more sectors than inners.

Hard Disk Latency



source: Freitas, Chiu. Solid-State Storage: Technology, Design, and Applications. FAST 2010.

Evolution of Hard Disk Technology

Disk latencies have only marginally improved over the last years ($\approx 10\,\%$ per year).

But:

- Throughput (i.e., transfer rates) improve by $\approx 50\,\%$ per year.
- Hard disk capacity grows by $\approx 50\,\%$ every year.

Therefore:

- Random access cost hurts even more as time progresses.

Ways to Improve I/O Performance

The latency penalty is hard to avoid.

But:

- Throughput can be increased rather easily by exploiting **parallelism**.
- **Idea:** Use multiple disks and access them in parallel.

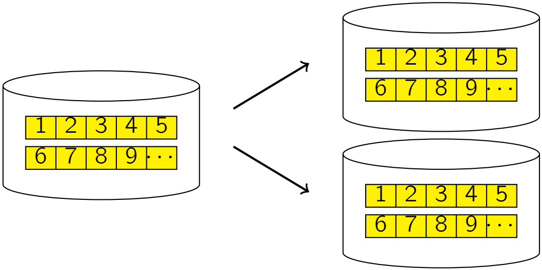
TPC-C: An industry benchmark for OLTP

Some while ago, the number one system (DB2 9.5 on AIX) used

- 10,992 disk drives (73.4 GB each, 15,000 rpm) **(!)**
  (plus 8 internal SCSI drives with 146.8 GB each),
- connected with 68 $\times$ 4 Gbit Fibre Channel adapters,
- yielding 6 mio transactions per minute.

Disk Mirroring

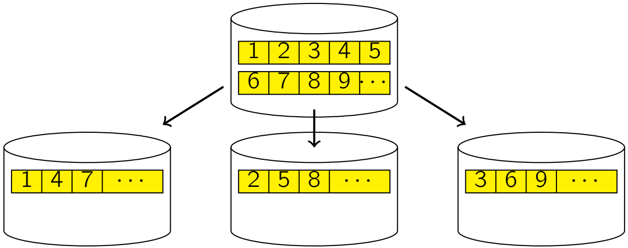
- Replicate data onto multiple disks



- I/O parallelism only for **reads**.
- Improved failure tolerance (can survive one disk failure).
- This is also known as **RAID 1** (mirroring without parity).
  (RAID: Redundant Array of Inexpensive Disks)

Disk Striping

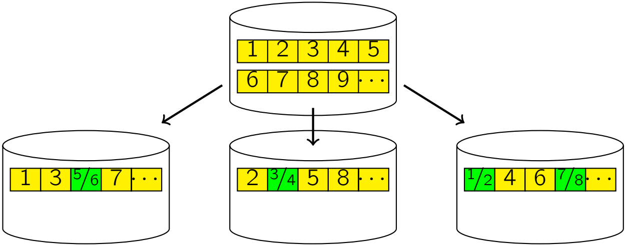
- Distribute data over disks



- Full I/O parallelism.
- High failure risk (here: 3 times risk of single disk failure)!
- Also known as **RAID 0** (striping without parity).

Disk Striping with Parity

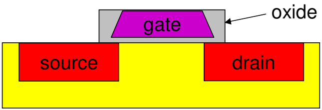
- Distribute data and parity information over disks.

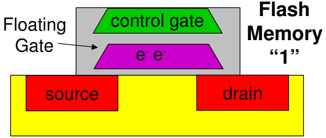


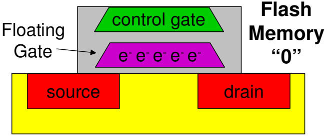
- High I/O parallelism.
- Fault tolerance: one disk can fail without data loss (two disks with dual parity/RAID 6).
- Also known as **RAID 5** (striping with distributed parity).

Solid-State Drives: Technology

Basis: MOS transistor







Flash cell:

- Add (fully isolated) **floating gate** in-between.
- Charge on floating gate shifts characteristics of the source/control gate/drain transistor.
  - → Use to "read" charge state
- (Dis-)charging of floating gate only through high voltage (tunnel effect)
  - → Charge "trapped" → persistence

source: Freitas, Chiu. Solid-State Storage: Technology, Design, and Applications. FAST 2010.

Solid-State Drives: Technology

Miniaturization:

- Combine many cells to achieve tight packing
  - $\rightarrow$ NAND Flash
  - $\rightarrow$ Must read blocks of data at once ($\leadsto$ hard disks)
- Single-level cells (SLC) vs. Multi-level cells (MLC)
  - $\rightarrow$ Cost/density $\leftrightarrow$ reliability trade-off

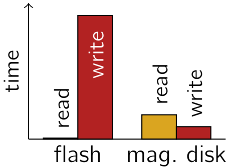
Challenges:

- Feature size $\searrow$ $\Rightarrow$ reliability $\searrow$
  - Fewer electrons of charge, thinner isolation layers
  - $\rightarrow$ Limited **retention**
- Over time, writes damage isolation layer
  - $\rightarrow$ Limited **endurance** ($10^4 \sim 10^5$ writes per cell)
- **Block based erasure** ($\rightarrow$ no update in place)
  - $\rightarrow$ **Write amplification**, slow writes

Solid-State Drives

Solid state drives (SSDs) as an alternative to conventional hard disks?

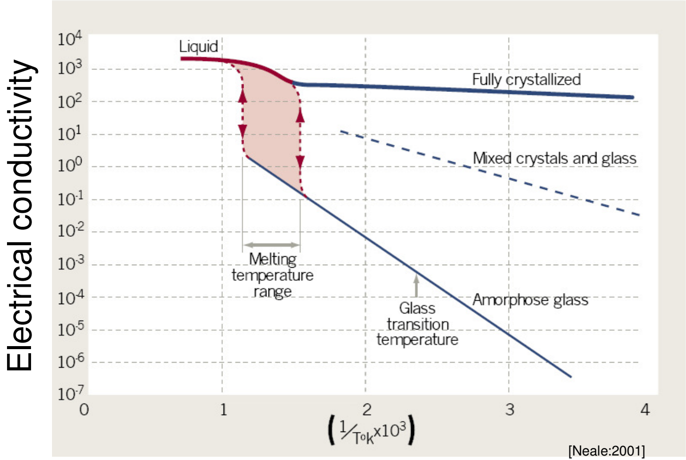
- SSDs provide **very low-latency random read access**.
- **Random writes**, however, are significantly **slower** than on traditional magnetic drives.
  - Pages have to be **erased** before they can be updated.
  - Once pages have been erased, sequentially writing them is almost as fast as reading.



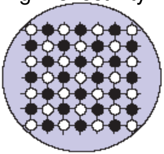
- Adapting databases to these characteristics is a current research topic.

Samsung 32 GB flash disk; 4096 bytes read/written randomly. Source: Koltsidas and Viglas. Flashing up the Storage Layer. *VLDB 2008*.

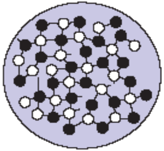
Phase-Change Memory: Physics



Crystalline phase
Low resistance
High reflectivity

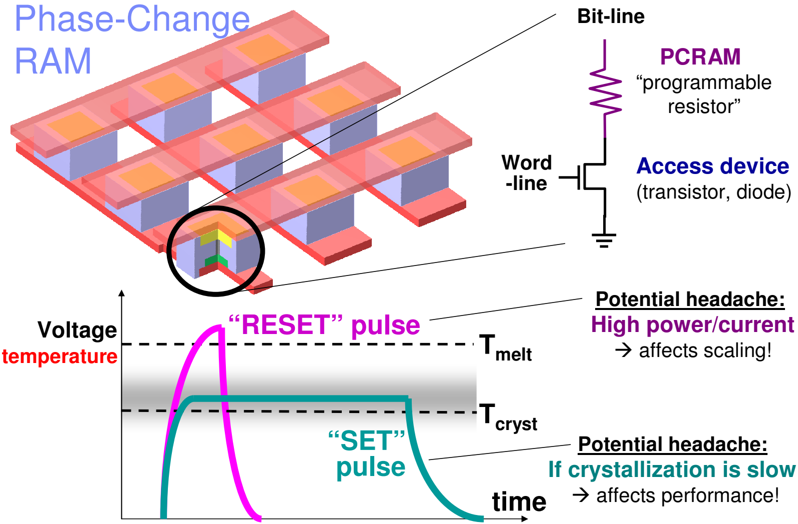


Amorphous phase
High resistance
Low reflectivity



source: Freitas, Chiu. Solid-State Storage: Technology, Design, and Applications. FAST 2010.

Phase-Change Memory: Technology



source: Freitas, Chiu. Solid-State Storage: Technology, Design, and Applications. FAST 2010.

Storage-Class Memory (SCM)

Phase-Change Memory is one promising technology to realize **storage-class memory**:

- **Persistent** (like disks or SSDs)
- **RAM-like access characteristics**
  → Speed-wise, but also with byte-level addressing

First prototypes exist!

Challenges/Questions:

- How **scalable** are SCM technologies? (so far looks good)
- How can SCM be **integrated** into a system?
  → Access SCM like a block device or like RAM?
- What does fast, byte-addressable storage mean for software?
  → *E.g.*, database recovery mechanisms

Network-Based Storage

The network is **not** a bottleneck any more:

- Hard disk: 50–100 MB/s
- Serial ATA: 375 MB/s (600 MB/s soon)
  Ultra-640 SCSI: 640 MB/s
- 10 gigabit Ethernet: 1,250 MB/s (latency: $\sim \mu$s)
  Infiniband QDR: 12,000 MB/s (latency: $\sim \mu$s)
- for comparison:
  PC2-5300 DDR2-SDRAM (dual channel): 10.6 GB/s
  PC3-12800 DDR3-SDRAM (dual channel): 25.6 GB/s

$\rightarrow$ Why not use the network for database storage?

Storage Area Network

- **Block-based** network access to storage
  - Seen as logical disks ("give me block 4711 from disk 42")
  - Unlike network file systems (*e.g.*, NFS, CIFS)
- SAN storage devices typically abstract from RAID or physical disks and present logical drives to the DBMS
  - Hardware acceleration and simplified maintainability
- Typically local networks with multiple servers and storage resources participating
  - Failure tolerance and increased flexibility

Grid or Cloud Storage

Some big enterprises employ clusters with **thousands** of commodity PCs (*e.g.*, Google, Amazon):

- **system cost** $\leftrightarrow$ **reliability** and **performance**,
- use **massive replication** for data storage.

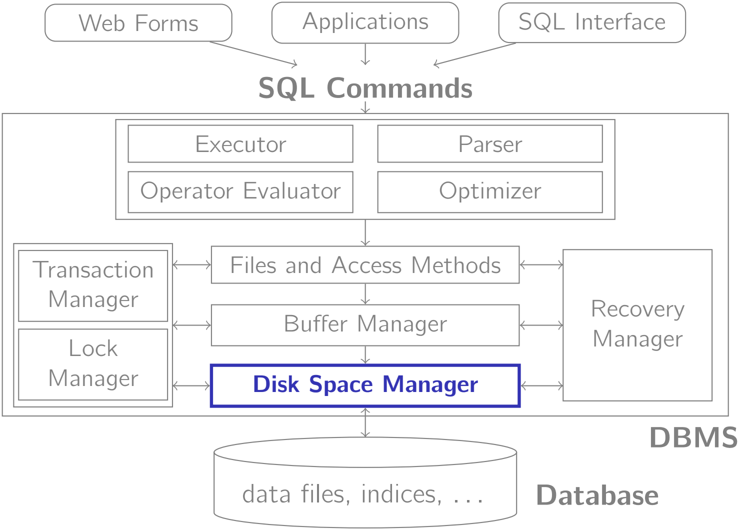
Spare CPU cycles and disk space can be sold as a **service**.

Amazon's "Elastic Computing Cloud (EC2)"
: Use Amazon's compute cluster by the hour ($\sim$ 10 ¢/hour).

Amazon's "Simple Storage Systems (S3)"
: "Infinite" store for objects between 1 Byte and 5 GB in size, with a simple key $\mapsto$ value interface.
  - Latency: 100 ms to 1 s (not impacted by load)
  - pricing $\approx$ disk drives (but addl. cost for access)

$\rightarrow$ **Build a database on S3?** ($\nearrow$ Brantner *et al.*, SIGMOD 2008)

Managing Space



Managing Space

The **disk space manager**

- abstracts from the gory details of the underlying storage
- provides the concept of a **page** (typically 4–64 KB) as a unit of storage to the remaining system components
- maintains the mapping

$$\text{page number} \mapsto \text{physical location} \quad ,$$

  where a physical location could be, *e.g.*,

  - an OS file name and an offset within that file,
  - head, sector, and track of a hard drive, or
  - tape number and offset for data stored in a tape library

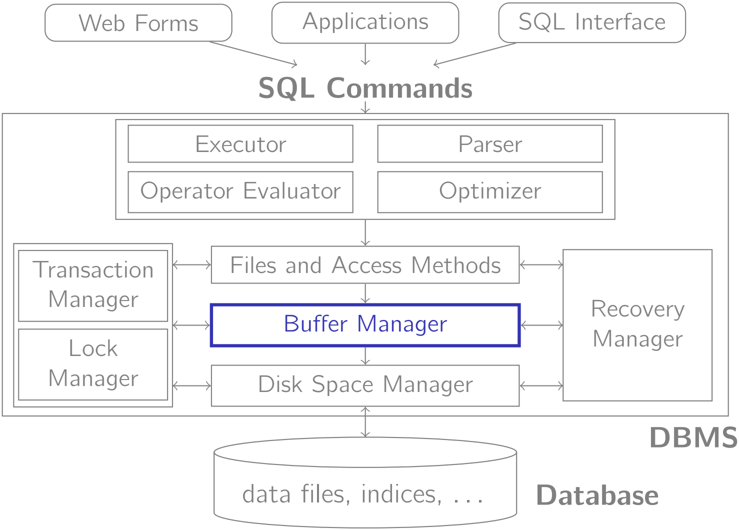
Empty Pages

The disk space manager also keeps track of used/free blocks.

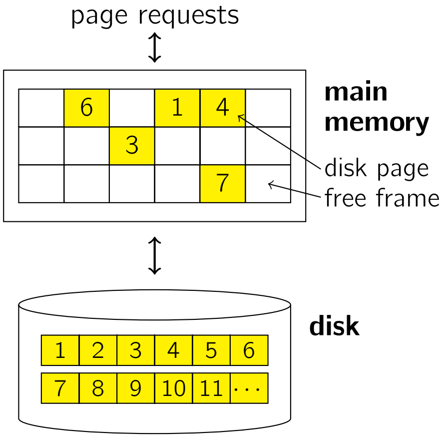
1. Maintain a linked list of free pages
   - When a page is no longer needed, add it to the list.
2. Maintain a bitmap with one bit for each page
   - Toggle bit *n* when page *n* is (de-)allocated.

✎ To exploit **sequential access**, it may be useful to allocate **contiguous** sequences of pages. **Which of the techniques 1 or 2 would you choose to support this?**

Buffer Manager



Buffer Manager



The **buffer manager**

- mediates between external storage and main memory,
- manages a designated main memory area, the **buffer pool** for this task.

Disk pages are brought into memory as needed and loaded into memory **frames**.

A **replacement policy** decides which page to evict when the buffer is full.

Interface to the Buffer Manager

Higher-level code requests (pins) pages from the buffer manager and releases (unpins) pages after use.

`pin` (*pageno*)
: Request page number *pageno* from the buffer manager, load it into memory if necessary. Returns a reference to the frame containing *pageno*.

`unpin` (*pageno*, *dirty*)
: Release page number *pageno*, making it a candidate for eviction. Must set *dirty* = true if page was modified.

✎ **Why do we need the *dirty* bit?**

Implementation of pin ()

```
Function: pin(pageno)
if buffer pool already contains pageno then
    pinCount (pageno) ← pinCount (pageno) + 1 ;
    return address of frame holding pageno ;
else
    select a victim frame v using the replacement policy ;
    if dirty (v) then
        write v to disk ;
    read page pageno from disk into frame v ;
    pinCount (pageno) ← 1 ;
    dirty (pageno) ← false ;
    return address of frame v ;
```

Implementation of `unpin ()`

Function: `unpin`(*pageno*, *dirty*)
`pinCount` (*pageno*) ← `pinCount` (*pageno*) − 1 ;
if *dirty* **then**
    `dirty` (*pageno*) ← *dirty* ;

✎ **Why don't we write pages back to disk during `unpin ()`?**

Replacement Policies

The effectiveness of the buffer manager's **caching** functionality can depend on the **replacement policy** it uses, *e.g.*,

Least Recently Used (LRU)
: Evict the page whose latest `unpin()` is longest ago.

LRU-k
: Like LRU, but considers k-latest `unpin()`, not just latest.

Most Recently Used (MRU)
: Evict the page that has been unpinned most recently.

Random
: Pick a victim randomly.

✎ What could be the rationales behind each of these strategies?

Buffer Management in Reality

Prefetching

Buffer managers try to anticipate page requests to overlap CPU and I/O operations.

- **Speculative prefetching:** Assume sequential scan and automatically read ahead.
- **Prefetch lists:** Some database algorithms can instruct the buffer manager with a list of pages to prefetch.

Page fixing/hating

Higher-level code may request to **fix** a page if it may be useful in the near future (*e.g.*, index pages).
Likewise, an operator that **hates** a page won't access it any time soon (*e.g.*, table pages in a sequential scan).

Partitioned buffer pools

E.g., separate pools for indexes and tables.

Databases vs. Operating Systems

Hmm... Didn't we just re-invent the operating system?

Yes,

- disk space management and buffer management very much look like **file management** and **virtual memory** in OSs.

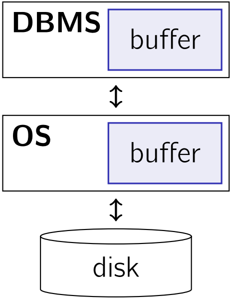
But,

- a DBMS may be much more aware of the **access patterns** of certain operators ($\rightarrow$ prefetching, page fixing/hating),
- transaction management often calls for a **defined order** of write operations,
- technical reasons may make OS tools unsuitable for a database (*e.g.*, file size limitation, platform independence).

Databases vs. Operating Systems

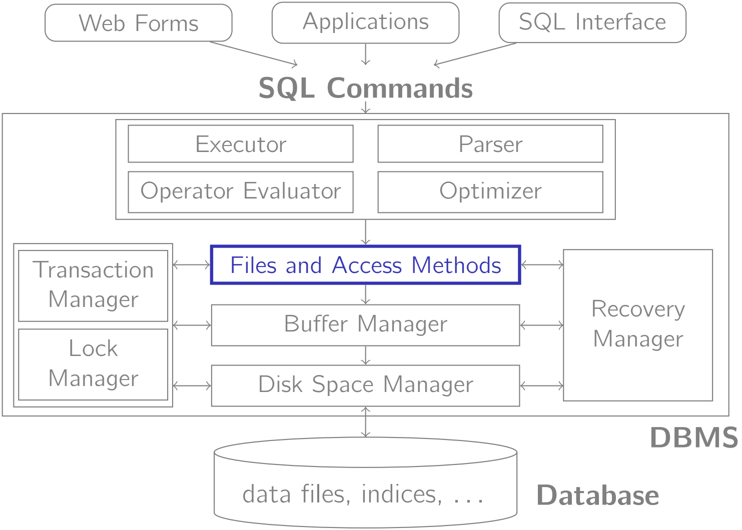
In fact, databases and operating systems sometimes interfere.

- Operating system and buffer manager effectively buffer the same data twice.
- Things get really bad if parts of the DBMS buffer get swapped out to disk by OS VM manager.
- Therefore, databases try to **turn off** OS functionality as much as possible.
  $\rightarrow$ **Raw disk** access instead of OS files.



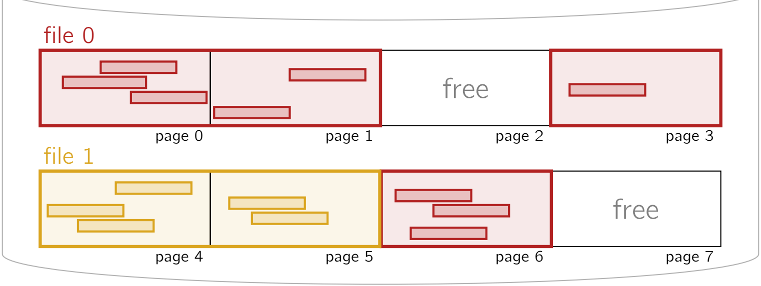
(Similar story: DBMS TX management vs. journaling file systems.)

Files and Records



Database Files

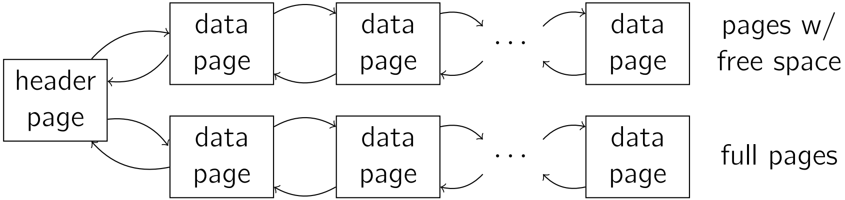
- So far we have talked about **pages**. Their management is oblivious with respect to their actual content.
- On the conceptual level, a DBMS manages **tables of tuples** and **indexes** (among others).
- Such tables are implemented as **files of records**:
  - A **file** consists of **one or more pages**.
  - Each **page** contains **one or more records**.
  - Each **record** corresponds to **one tuple**.



Heap Files

The most important type of files in a database is the **heap file**. It stores records in **no particular order** (in line with, *e.g.*, SQL).

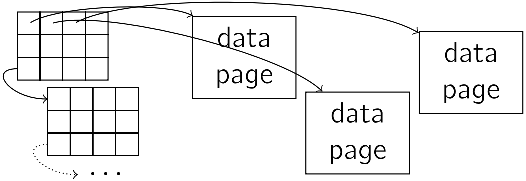
Linked list of pages



- + easy to implement
- − most pages will end up in free page list
- − might have to search many pages to place a (large) record

Heap Files

Directory of pages



- use as **space map** with information about free page
  - granularity as trade-off space ↔ accuracy
    (range from *open*/*closed* bit to exact information)

\+ free space search more efficient

− small memory overhead to host directory

Free Space Management

Which page to pick for the insertion of a new record?

Append Only
: Always insert into last page. Otherwise, create a new page.

Best Fit
: Reduces fragmentation, but requires searching the entire space map for each insert.

First Fit
: Search from beginning, take first page with enough space.
(→ These pages quickly fill up, and we waste a lot of search effort in first pages afterwards.)

Next Fit
: Maintain **cursor** and continue searching where search stopped last time.

Free Space Witnesses

We can accelerate the search by remembering **witnesses**:

- Classify pages into **buckets**, *e.g.*, "75 %–100 % full", "50 %–75 % full", "25 %–50 % full", and "0 %–25 % full".
- For each bucket, remember some **witness pages**.
- Do a regular best/first/next fit search only if no witness is recorded for the specific bucket.
- Populate witness information, *e.g.*, as a side effect when searching for a best/first/next fit page.

ID	NAME	SEX
4711	John	M
~~1723~~	~~Marc~~	~~M~~
6381	Betty	F

- **record identifier (rid):**

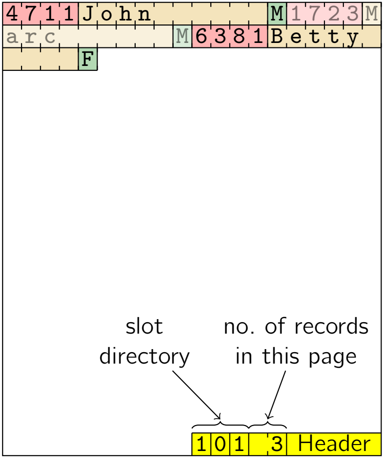
  $\langle \mathit{pageno}, \mathit{slotno} \rangle$

- record position (within page):

  $\mathit{slotno} \times \mathit{bytes\ per\ slot}$

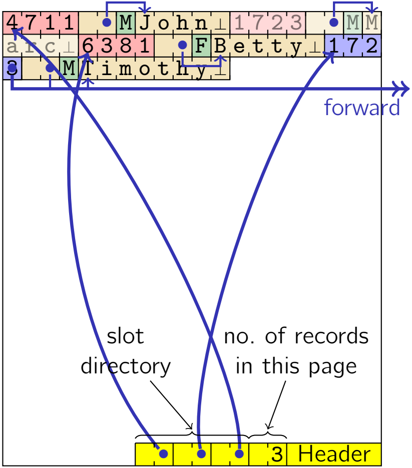
- Tuple **deletion**?
  - record id shouldn't change

  $\rightarrow$ **slot directory** (bitmap)



Inside a Page—Variable-Sized Fields

- Variable-sized fields moved to **end** of each record.
  - Placeholder points to location.
  - ✎ **Why?**
- Slot directory points to start of each record.
- Records **can move** on page.
  - *E.g.*, if field size changes.
- Create **"forward address"** if record won't fit on page.
  - ✎ **Future updates?**



Slotted Pages — IBM DB2

In DB2, the slot directory grows from the front, data grows from the end:

```
BPS Page Header:
      Page Data Offset = 48
      Page Data Length = 4048
              Page LSN = 0000 0438 8F85
                  ...
           Object Type = Data Object
   Data Page Header:
            Slot Count = 103
      Total Free Space = 48
                  ...
     Free Space Offset = 216
   Maximum Record Size = 37
   Data Records:
 Slot 0:
   Offset Location = 3991  (xF97)
   Record Length = 37  (x25)
   Record Type = Table Data Record
                 (FIXEDVAR) (PUNC)
   ...
 Slot 1:
   Offset Location = 3954  (xF72)
   Record Length = 37  (x25)
   Record Type = Table Data Record
                 (FIXEDVAR) (PUNC)
   ...
```

```
0000  3000D00F 50020000 00040000 04388F85  0...P........8..
0010  01010000 0500000A 542FE4C5 01000000  ........T.......
0020  02000000 09000000 00000000 00000000  ................
0030  67003000 25002500 2500D800 00000000  g.0.............
0040  CDDF0000 970F720F 4D0F280F FFFFDE0E  ......r.M.......
0050  B90E940E 6F0E4A0E 250E000E DB0DB60D  ....o.J.........
0060  910D6C0D 470D220D FD0CD80C B30C8E0C  ..l.G...........
      ...
0F90  00100025 0001001D 00C71800 00620D00  .............b..
0FA0  00737472 2D383337 37372020 20202020  .str.83777......
0FB0  20202020 20001000 25000100 1D003B18  ................
0FC0  00003AA9 00007374 722D3837 31383720  ......str.87187.
0FD0  20202020 20202020 20200010 00250001  ................
0FE0  001D00CF 1C000056 AB000073 74722D39  .......V...str.9
0FF0  30303533 20202020 20202020 20202000  0053............
```

- Such data can be obtained with `db2dart`.
- Observe how slot 4 is marked 'deleted' (`FFFF`).

Sparse Columns

An alternative is **interpreted storage**.

Interpreted Storage
$\langle \texttt{ID}, \texttt{4711} \rangle$, $\langle \texttt{NAME}, \texttt{John} \rangle$, $\langle \texttt{SEX}, \texttt{M} \rangle$
$\langle \texttt{ID}, \texttt{1723} \rangle$, $\langle \texttt{NAME}, \texttt{Marc} \rangle$, $\langle \texttt{SEX}, \texttt{M} \rangle$
$\langle \texttt{ID}, \texttt{6381} \rangle$, $\langle \texttt{NAME}, \texttt{Betty} \rangle$, $\langle \texttt{SEX}, \texttt{F} \rangle$

✎ **Why would one want to do this?**

Sparse Columns in MS SQL Server

Microsoft SQL Server 2008 provides support for sparse columns.

Columns marked as `SPARSE` are put into an interpreted storage.

```
CREATE TABLE Products
  (···, Card VARCHAR(10) SPARSE NULL, ···)
```

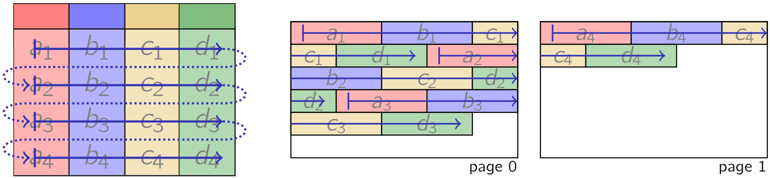
The internal storage is designed for fast access.

Interpreted Storage					
Header	Range	Mask	Column IDs	Value Offsets	Values
	4 ··· 100	1	4, 5, 6	0, 10, 18	`SD␣␣␣␣␣␣␣␣`
	101 ··· 200	0			`␣␣␣␣␣␣␣5`
	201 ··· 300	1			`␣␣␣␣␣␣10`

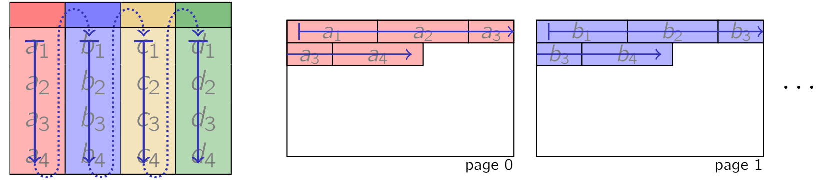
↗Acharya *et al.*. Relational Support for Flexible Schema Scenarios. *VLDB 2008*.

Alternative Page Layouts

We have just populated data pages in a **row-wise** fashion:



We could as well do that **column-wise**:



Alternative Page Layouts

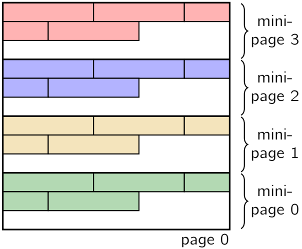
These two approaches are also known as **NSM (n-ary storage model)** and **DSM (decomposition storage model)**.[1]

- Tuning knob for certain workload types (*e.g.*, OLAP)
- Different behavior with respect to **compression**.

A hybrid approach is the **PAX (Partition Attributes Accross)** layout:

- Divide each page into **minipages**.
- Group attributes into them.

↗ Ailamaki *et al.* Weaving Relations for Cache Performance. *VLDB 2001*.



[1] Recently, the terms **row-store** and **column-store** have become popular, too.

Recap

Magnetic Disks

Random access **orders of magnitude** slower than sequential.

Disk Space Manager

Abstracts from hardware details and maps
page number $\mapsto$ physical location.

Buffer Manager

Page **caching** in main memory; `pin()`/`unpin()` interface;
replacement policy crucial for effectiveness.

File Organization

Stable **record identifiers (rids)**; maintenance with fixed-sized records and variable-sized fields; NSM vs. DSM.