Einführung in Datenbanksysteme

Prof. Dr. Ralf Möller

TUHH

Speicherstrukturen und Datenbankarchitektur

Danksagung

- Diese Vorlesung basiert auf dem Kurs

  Architecture and Implementation of Database Systems
  von Jens Teubner, ETH Zürich

- Ich bedanke mich für die Bereitstellung des Materials

Architecture of a DBMS / Course Outline

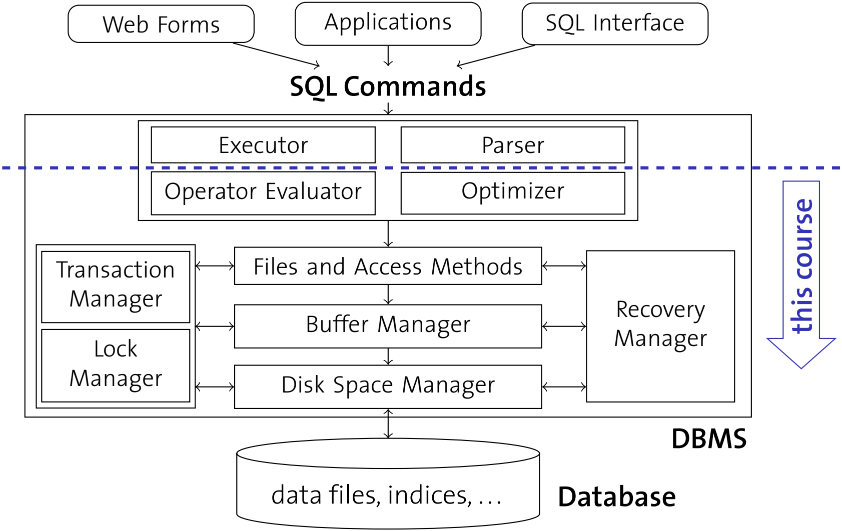
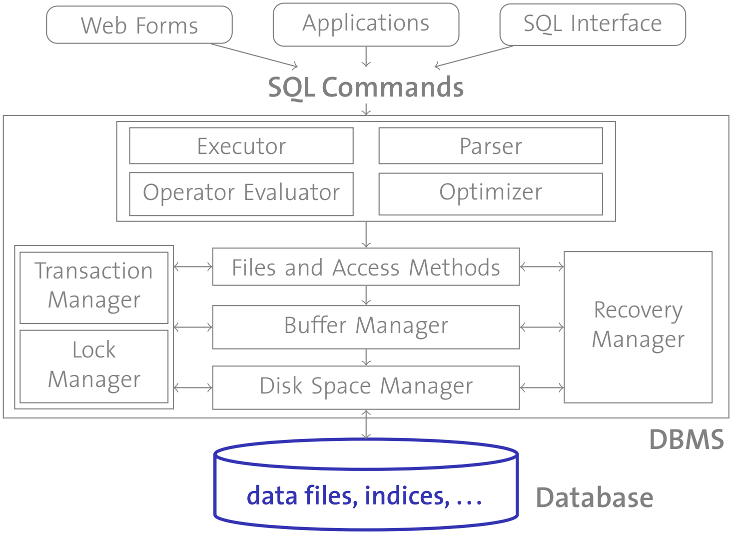


Figure inspired by Ramakrishnan/Gehrke: "Database Management Systems", McGraw-Hill 2003.

Part I

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

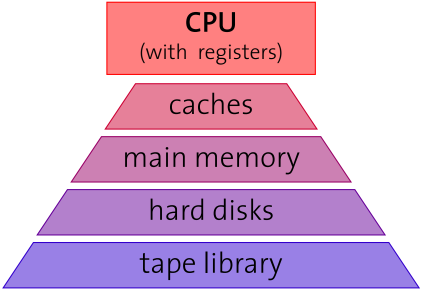
Recovery Manager

DBMS

data files, indices, . . .

Database

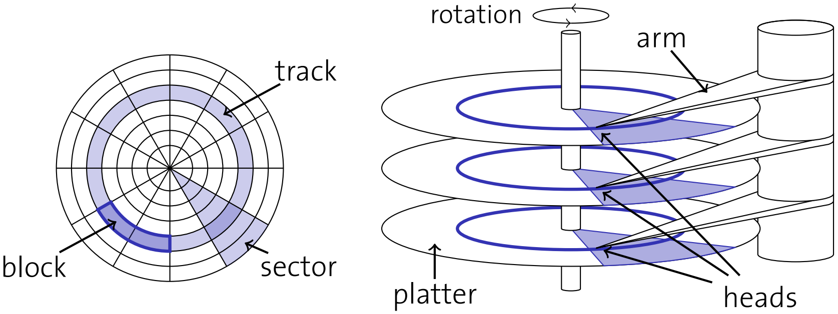
Memory Hierarchy



	capacity	**latency**
CPU (with registers)	bytes	< 1 ns
caches	kilo-/megabytes	< 10 ns
main memory	gigabytes	70–100 ns
hard disks	terabytes	3–10 ms
tape library	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})

→ **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:**

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

  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.

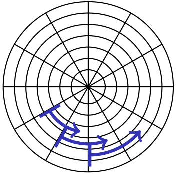
→ As soon as we need at least $\frac{428\,\text{ms}}{14330\,\text{ms}} = 3\,\%$ of a file, we better read the **entire** file!

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.

Evolution of Hard Disk Technology

Disk latencies have only marginally improved over the last years (≈ 10 % per year).

But:

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

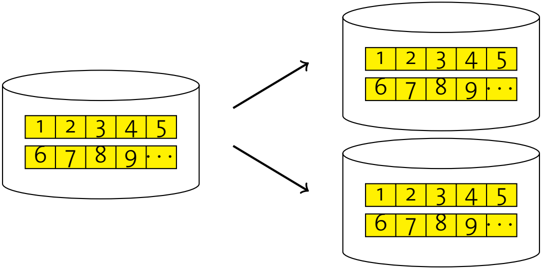
📈 TPC-C: An industry benchmark for OLTP

The current number one system (a DB2 9.5 database on AIX) uses

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

Disk Mirroring

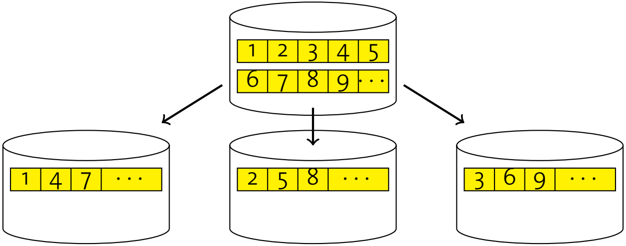
- Replicate data onto multiple disks



- 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

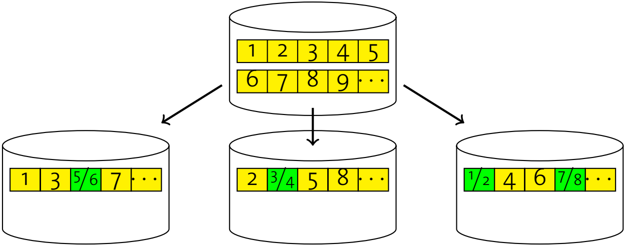
- Distribute data over disks



- 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.

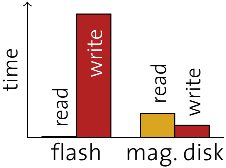


- High I/O parallelism.
- Fault tolerance: one disk can fail without data loss (two disks with dual parity/RAID 6).
- Also known as **RAID 5** (striping with distributed parity).

Solid-State Disks

Solid state disks (SSDs) have emerged 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*.

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

Switch

→ 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** ↔ **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).

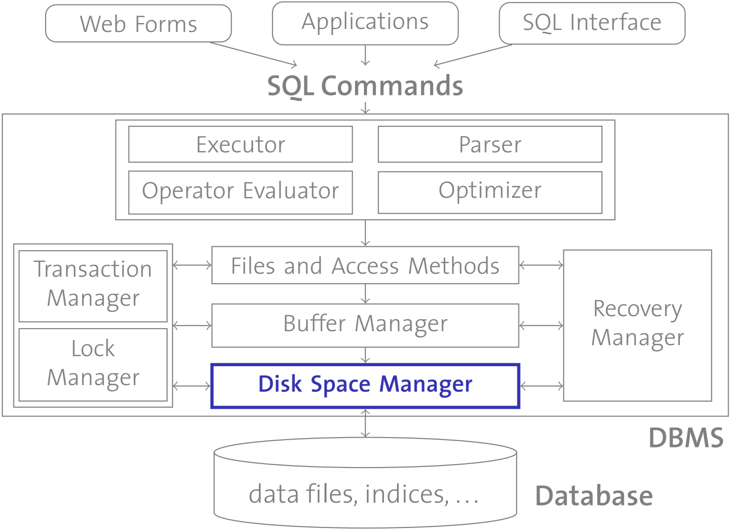
Amazon's "Simple Storage Systems (S3)"

"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)

→ **Build a database on S3?** (↗ 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

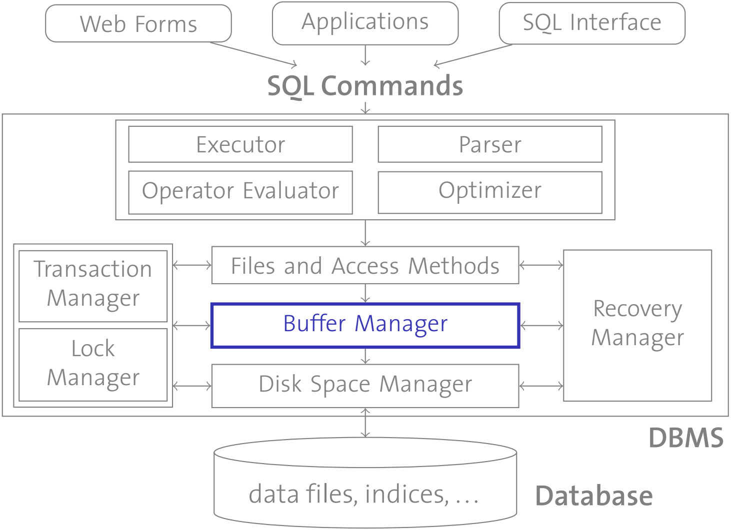
Empty Pages

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



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

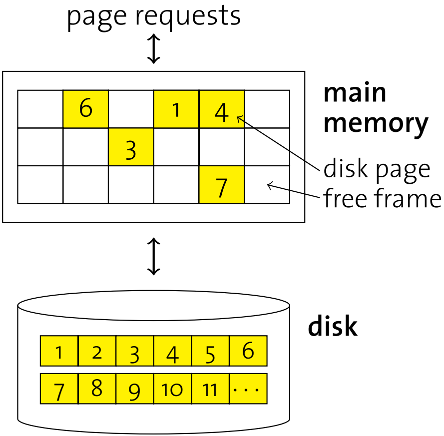
Recovery Manager

DBMS

data files, indices, . . .

Database

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.

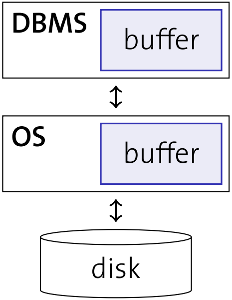
But,

- a DBMS may be much more aware of the **access patterns** of certain operators (→ prefetching, page fixing/hating),
- concurrency control 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.
  - → **Raw disk** access instead of OS files.


DBMS

buffer

↕

OS

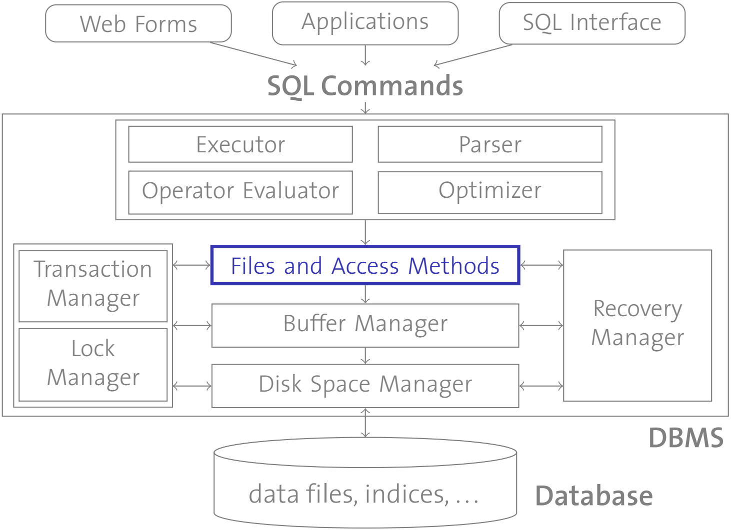
buffer

↕

disk

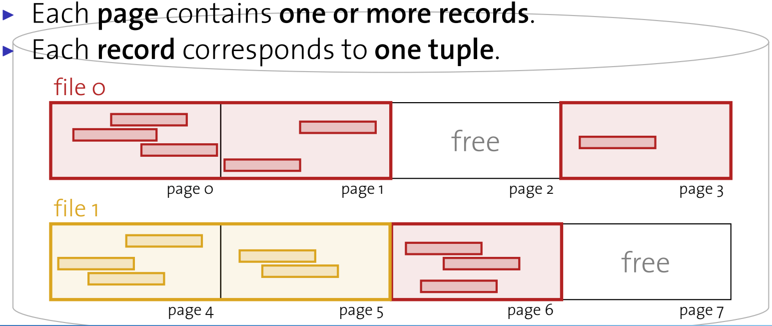
(Similar story: DBMS TX management vs. journaling file systems.)

Files and Records


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

Database Files

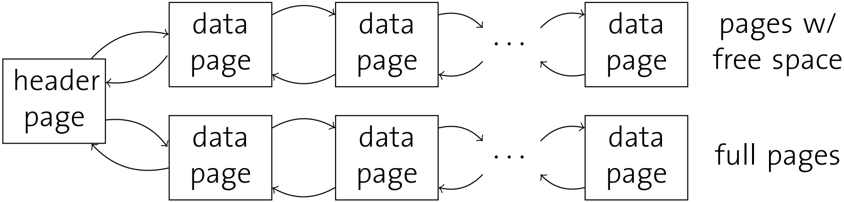
- So far we have talked about **pages**. Their management is oblivious with respect to their actual content.
- On the conceptual level, a DBMS manages **tables of tuples** and **indexes** (among others).
- Such tables are implemented as **files of records**:
  - A **file** consists of **one or more pages**.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

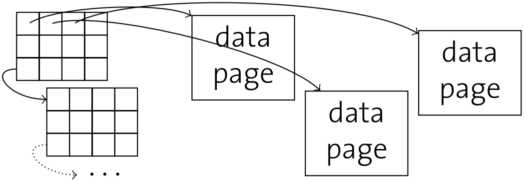
Linked list of pages



- \+ 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.

Inside a Page

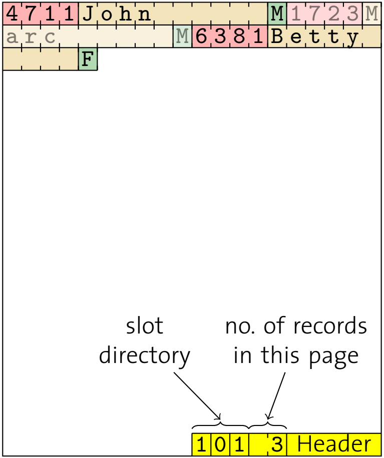
ID	NAME	SEX
4711	John	M
~~1723~~	~~Marc~~	~~M~~
6381	Betty	F

- **record identifier (rid):**

  $\langle pageno, slotno \rangle$
- record position (within page):

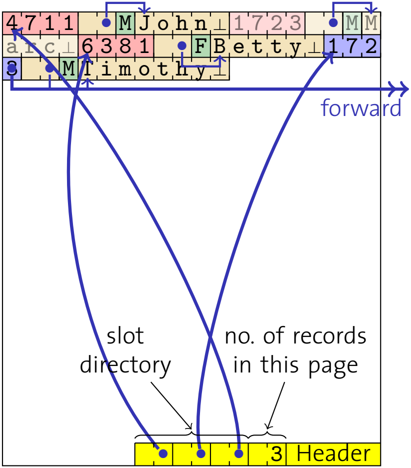
  $slotno \times bytes\ per\ slot$
- Tuple **deletion**?
  - record id shouldn't change

  → **slot directory** (bitmap)



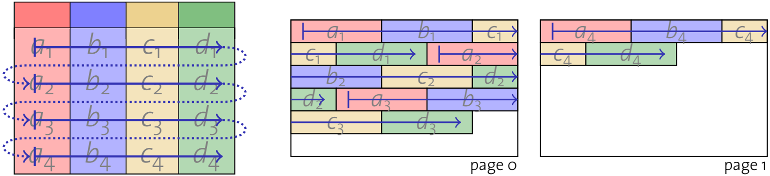
Inside a Page—Variable-Sized Fields

- 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?**

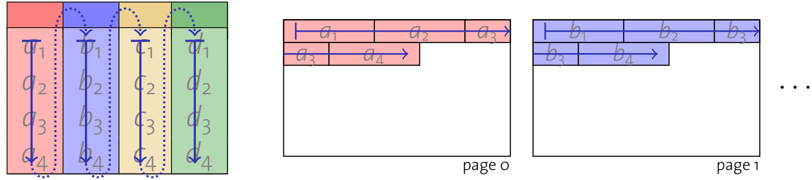


Alternative Page Layouts

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



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



Alternative Page Layouts

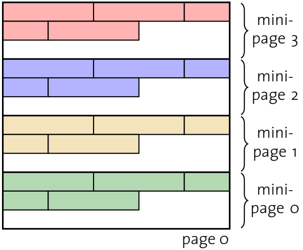
These two approaches are also known as **NSM (n-ary storage model)** and **DSM (decomposition storage model)**.[1]

- 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 ↦ 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.