

Systems Infrastructure for Data Science

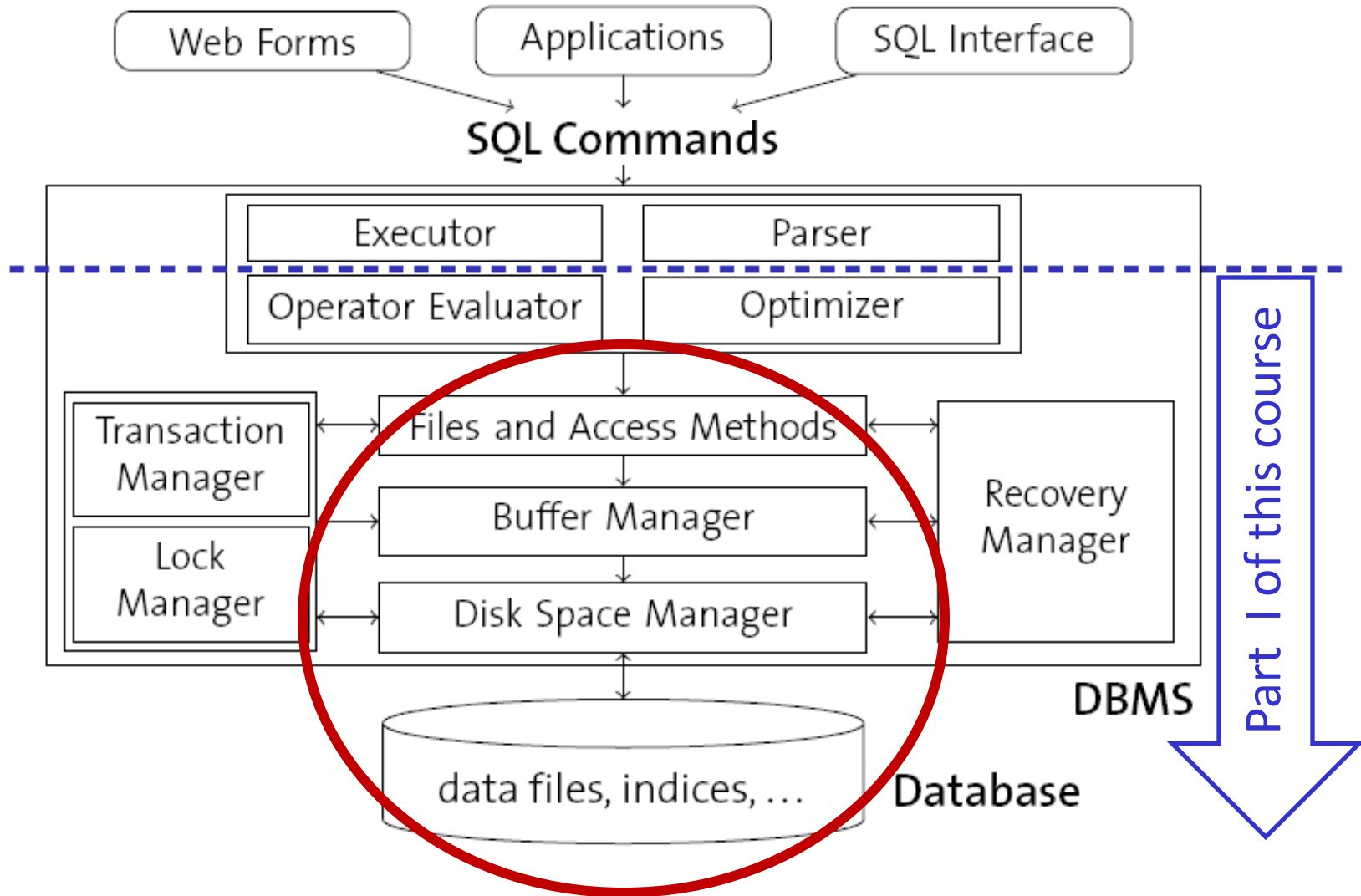
Web Science Group

Uni Freiburg

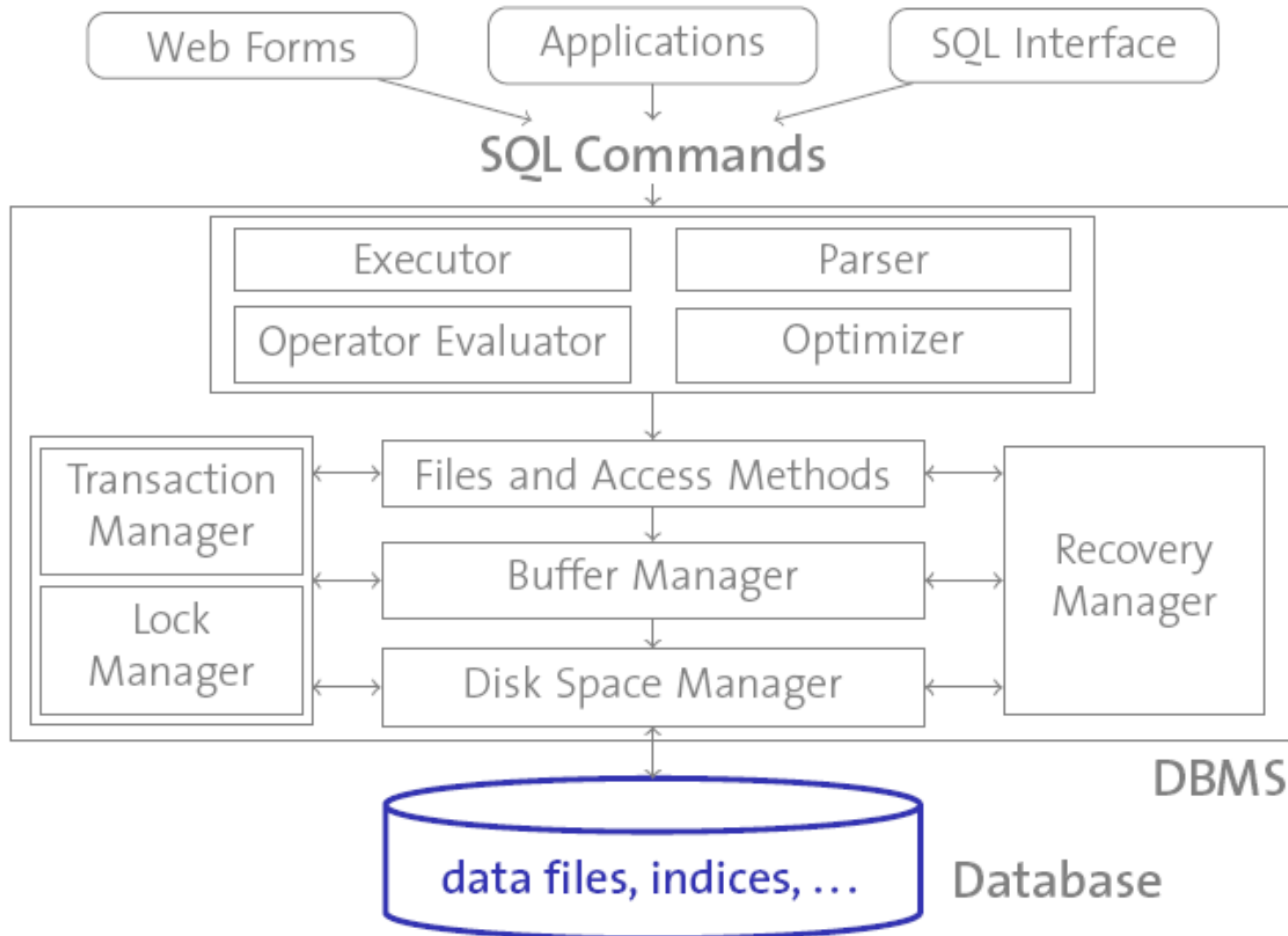
WS 2013/14

Lecture I: Storage

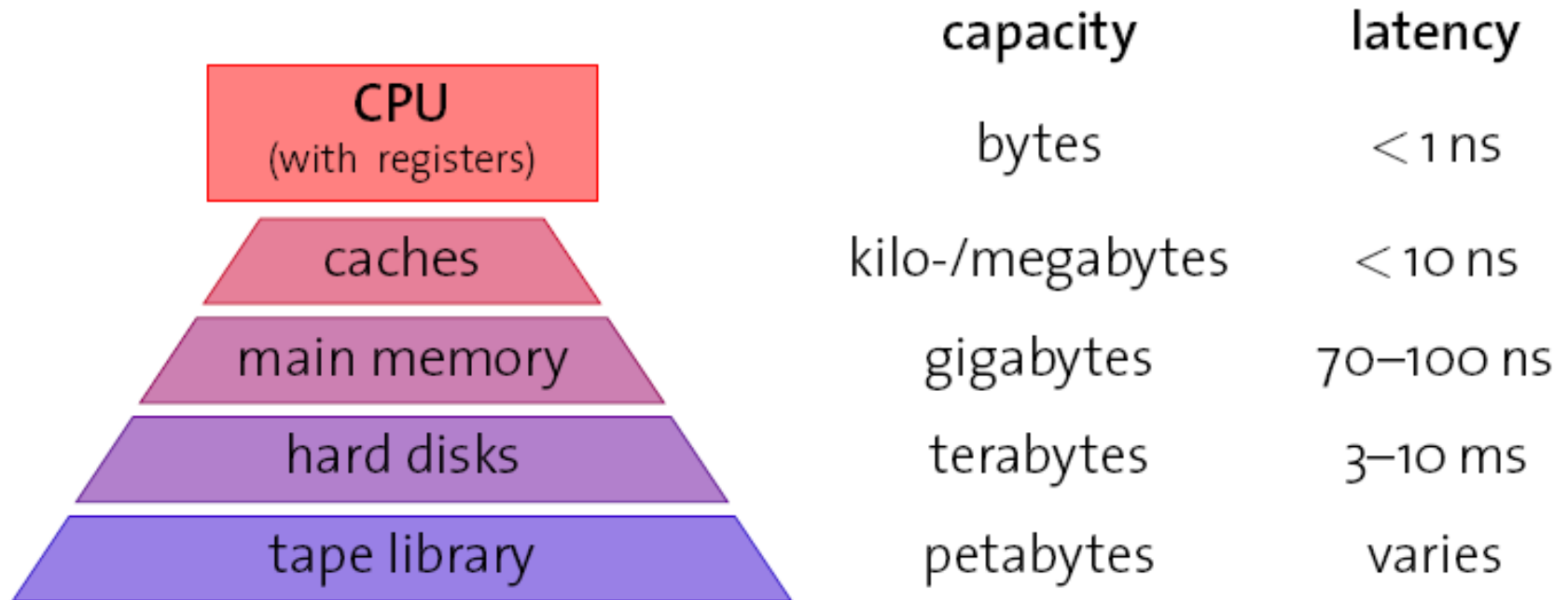
Storage



The Physical Layer



The Memory Hierarchy

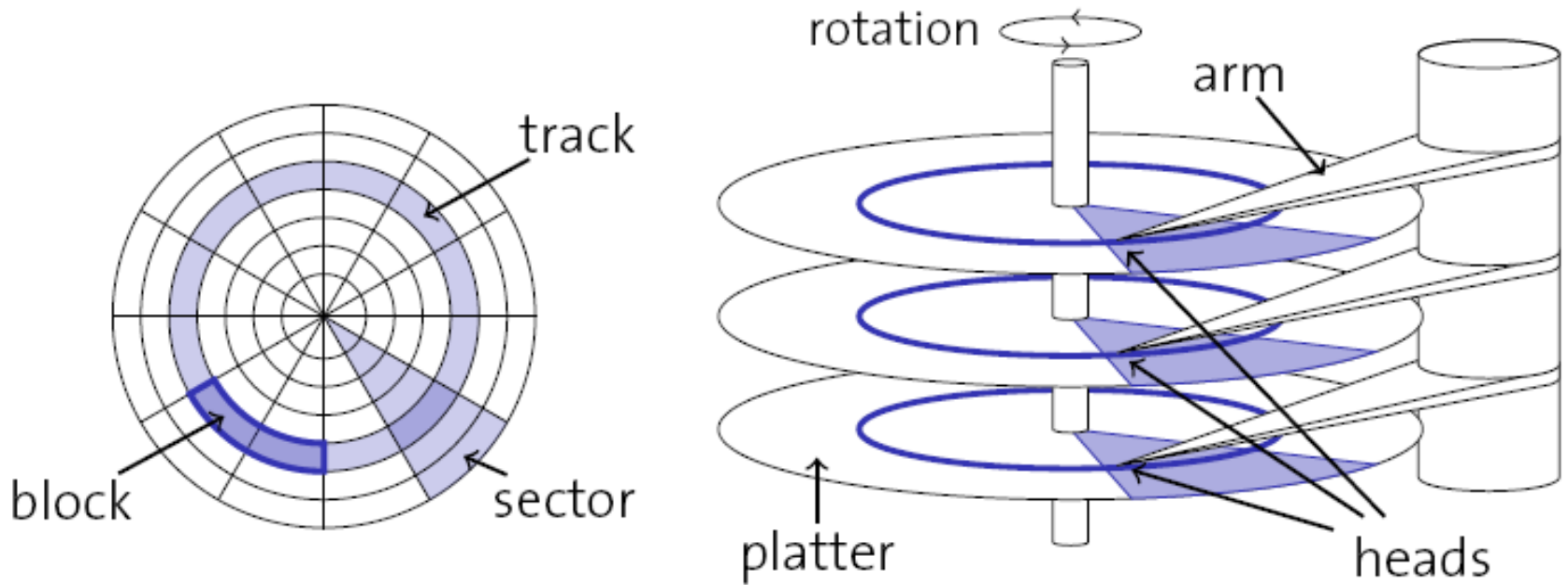


- Fast, but expensive and small memory close to CPU
- Larger, slower memory at the periphery
- We'll try to hide latency by using the fast memory as a **cache**.

Observations and Trends

- For which gaps were systems designed traditionally?
- Within the same technology:
 - Storages capacities grow fastest
 - Transfer speeds grow moderately
 - Latencies only see minimal changes
- Between the levels
 - Widening latency gap

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.



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}).

$$\text{access time } t = t_s + t_r + t_{tr}$$

Example

Notebook drive Hitachi TravelStar 7K200

- 4 heads, 2 disks, 512 bytes/sector, 200 GB capacity
- average seek time = **10 ms**
- rotational speed = **7200 rpm** (revolutions per minute)
- transfer rate = \approx **50 MB/s**
- What is the access time to read an **8 KB** data block?

$$t = t_s + t_r + t_{tr}$$

$$t_s = 10 \text{ ms}$$

$$t_r = (60,000/7200)/2 = 4.17 \text{ ms} \leftarrow$$

$$t_{tr} = (8/50,000) * 1,000 = 0.16 \text{ ms}$$

$$t = 10 + 4.17 + 0.16 = 14.33 \text{ ms}$$

$$\begin{aligned} \text{max} &= 60,000/7200 \text{ ms} \\ \text{avg} &= \text{max}/2 \end{aligned}$$

Sequential vs. Random Access

➤ What is the access time to read **1000 blocks** of size **8 KB**?

- Random access:

$$\begin{aligned}t_{\text{rnd}} &= 1000 * t \\ &= 1000 * (t_s + t_r + t_{\text{tr}}) \\ &= 1000 * (10 + 4.17 + 0.16) = 1000 * 14.33 = \mathbf{14330 \text{ ms}}\end{aligned}$$

- Sequential access:

$$\begin{aligned}t_{\text{seq}} &= t_s + t_r + 1000 * t_{\text{tr}} + N * t_{\text{track-to-track seek time}} \\ &= t_s + t_r + 1000 * 0.16 \text{ ms} + (16 * 1000)/63 * 1 \text{ ms} \\ &= 10 \text{ ms} + 4.17 \text{ ms} + 160 \text{ ms} + 254 \text{ ms} \approx \mathbf{428 \text{ ms}}\end{aligned}$$

// N: number of tracks

// TravelStar 7K200: There are 63 sectors per track.
Each 8 KB block occupies 16 sectors.

$$t_{\text{track-to-track seek time}} = 1 \text{ ms}$$

Sequential vs. Random Access

➤ What is the access time to read **1000 blocks** of size **8 KB**?

- Random access:

$$\begin{aligned}t_{\text{rnd}} &= 1000 * t \\ &= 1000 * (t_s + t_r + t_{\text{tr}}) \\ &= 1000 * (10 + 4.17 + 0.16) = 1000 * 14.33 = \mathbf{14330 \text{ ms}}\end{aligned}$$

- Sequential access:

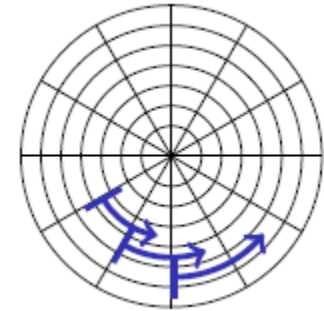
$$\begin{aligned}t_{\text{seq}} &= t_s + t_r + 1000 * t_{\text{tr}} + N * t_{\text{track-to-track seek time}} \\ &= t_s + t_r + 1000 * 0.16 \text{ ms} + (16 * 1000)/63 * 1 \text{ ms} \\ &= 10 \text{ ms} + 4.17 \text{ ms} + 160 \text{ ms} + 254 \text{ ms} \approx \mathbf{428 \text{ ms}}\end{aligned}$$

- Sequential I/O is **much** faster than random I/O.
- Avoid **random I/O** whenever possible.

Performance Tricks

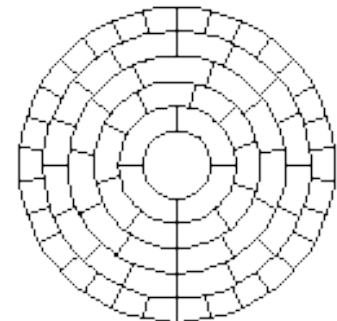
- System builders play a number of tricks to improve performance:

- **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: **S**hortest **P**ositioning **T**ime **F**irst).

- **Zoning:** Outer tracks are longer than the inner ones. Therefore, divide outer tracks into more sectors than inner ones.



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

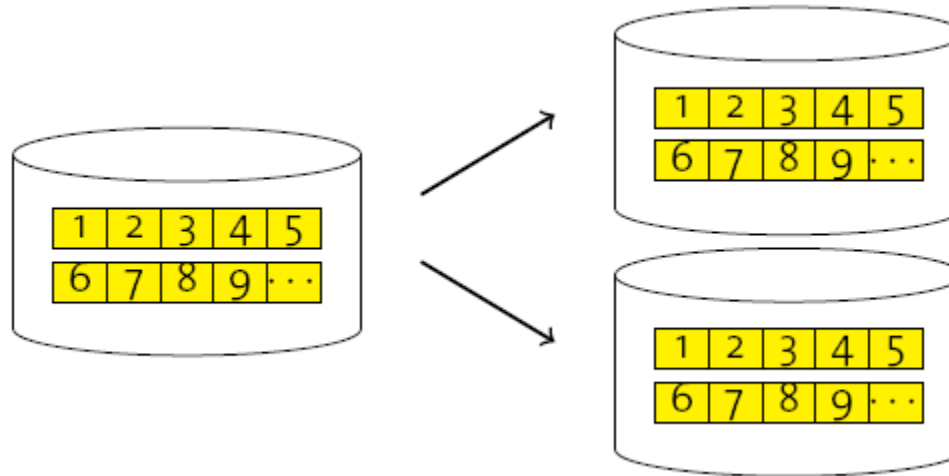
- The latency penalty is hard to avoid.
- **But:**
 - Throughput can be increased rather easily by exploiting **parallelism**.
 - **Idea:** Use multiple disks and access them in parallel.

➤ **TPC-C: An industry benchmark for OLTP**

- The #1 system in 2008 (an IBM DB2 9.5 database on AIX) uses:
 - 10,992 disk drives (73.4 GB each, 15,000 rpm) (!)
 - connected with 68 x 4 Gbit Fibre Channel adapters,
 - yielding 6M transactions per minute.

Disk Mirroring

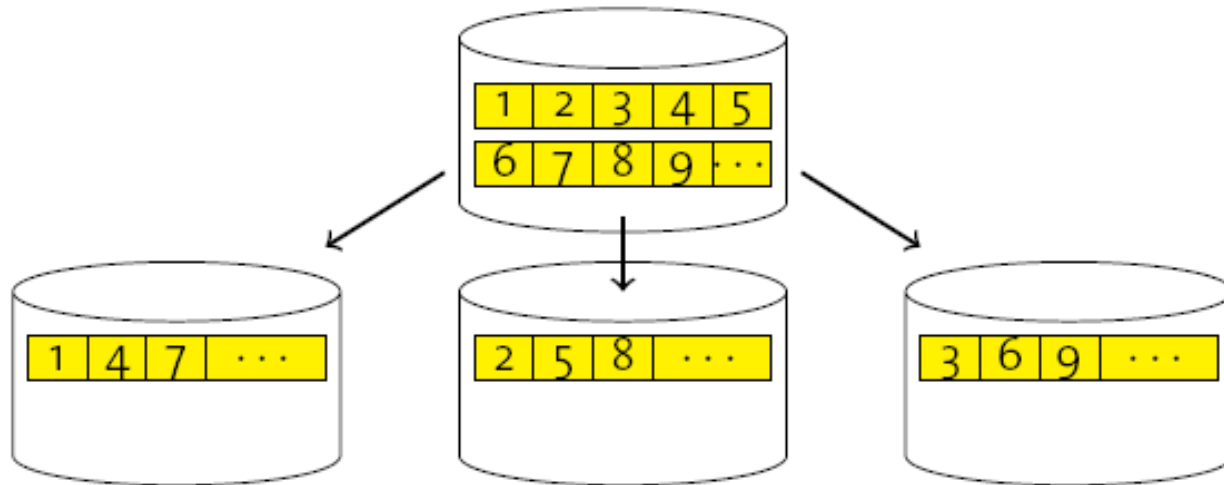
- Replicate data onto multiple disks:



- I/O parallelism only for **reads** (writes must be sequential to keep consistency).
- Improved **failure tolerance** (can survive one disk failure).
- **No parity** (no extra information kept to recover from disk failures).
- This is also known as **RAID 1** ("mirroring without parity").
(RAID = **R**edundant **A**rray of **I**nexpensive **D**isks)

Disk Striping

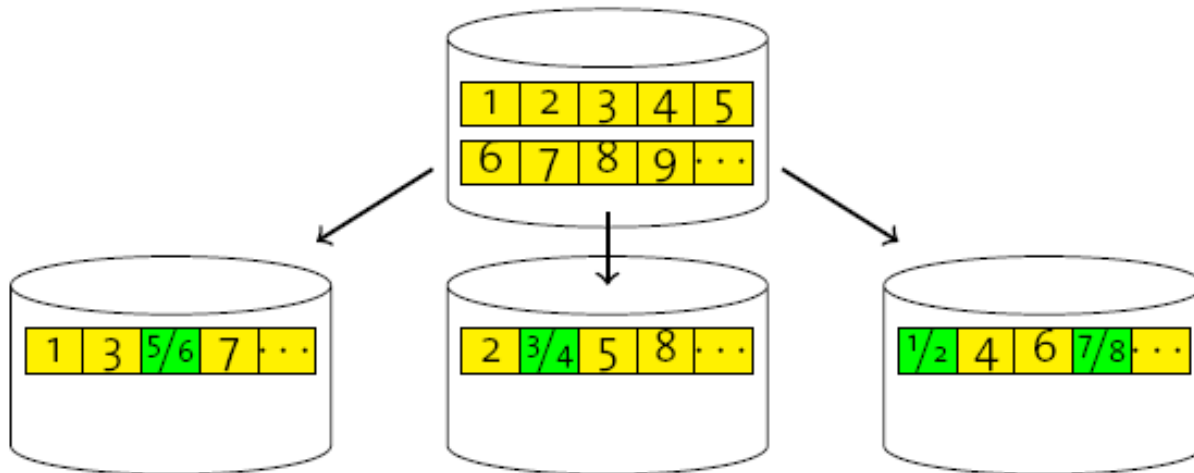
- Distribute data into equal-size partitions over multiple disks:



- Full I/O parallelism (both reads and writes).
- No parity.
- High failure risk (here: 3 times risk of single disk failure)!
- This is 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.
- This is also known as **RAID 5** (“striping with distributed parity”).

Other RAID Levels

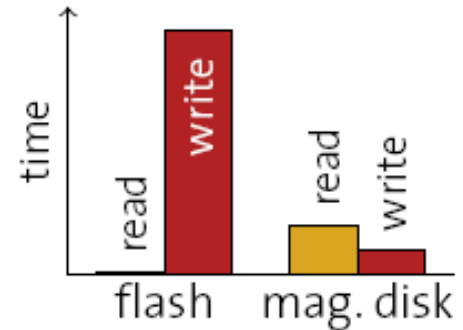
- **RAID 0**: block-level **striping** without parity or mirroring
- **RAID 1**: **mirroring** without parity or striping
- RAID 2: bit-level striping with dedicated parity
- RAID 3: byte-level striping with dedicated parity
- RAID 4: block-level striping with dedicated parity
- **RAID 5**: block-level striping with **distributed parity**
- RAID 6: block-level striping with double distributed parity

Modern Storage Alternatives

- (Flash-based) Solid-State Disk (SSD)
- Phase-Change Memory (PCM)
- Storage-Area Network (SAN)
- Cloud-based Storage (e.g., Amazon S3)

Solid-State Disks

- Solid-State Disks (SSDs), mostly based on flash memory chips, 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.
- Koltsidas and Viglas, “Flashing up the Storage Layer”, VLDB Conference, 2008.



Phase-Change Memory

- More recently, Phase-Change Memory (PCM) has been emerging as an alternative to flash.
 - It incurs lower read and write latency compared to both flash memory and magnetic disks.
 - Currently mostly used in mobile devices; is expected to become more common in the near future.
- Chen, Gibbons, Nath, “Rethinking Database Algorithms for Phase Change Memory”, CIDR Conference, 2011.

Network-based Storage

- The network is **not** a bottleneck any more:
 - Hard disk: 150 MB/s
 - Serial ATA: 600 MB/s
 - Ultra-640 SCSI: 640 MB/s
 - 10 gigabit Ethernet: 1,250 MB/s (latency $\sim \mu\text{s}$)
 - Infiniband QDR: 12,000 MB/s (latency $\sim \mu\text{s}$)
 - For comparison:
 - PC2-5300 DDR2-SDRAM (dual channel) = 10.6 GB/s
 - PC3-12800 DDR3-SDRAM (dual channel) = 25.6 GB/s
- Why not use the network for database storage?

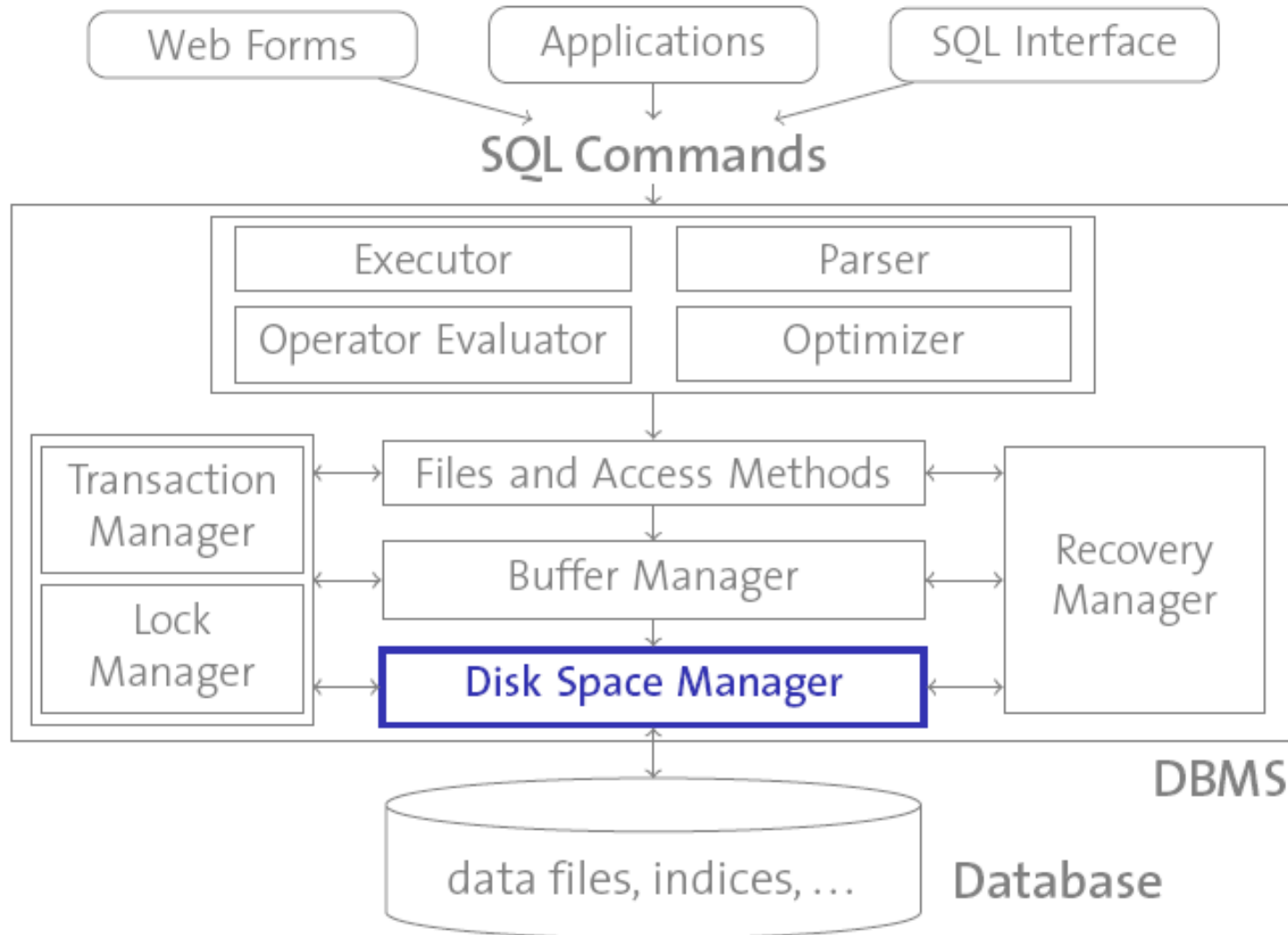
Storage Area Network (SAN)

- **Block-based** network access to storage
 - Seen as logical disks (“Give me block 4711 from disk 42.”)
 - Unlike network file systems (e.g., NFS)
- 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 cents/hour).
 - Amazon’s “Simple Storage Systems (S3)”
 - “Infinite” store for objects between 1 Byte and 5 GB in size, with a simple key \rightarrow value interface.
 - Latency: 100 ms to 1 s (not impacted by load)
 - Pricing \approx disk drives (but additional cost for access)
- Build a database on S3? (Brantner et al., SIGMOD’08 Conference)

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

page number → physical location

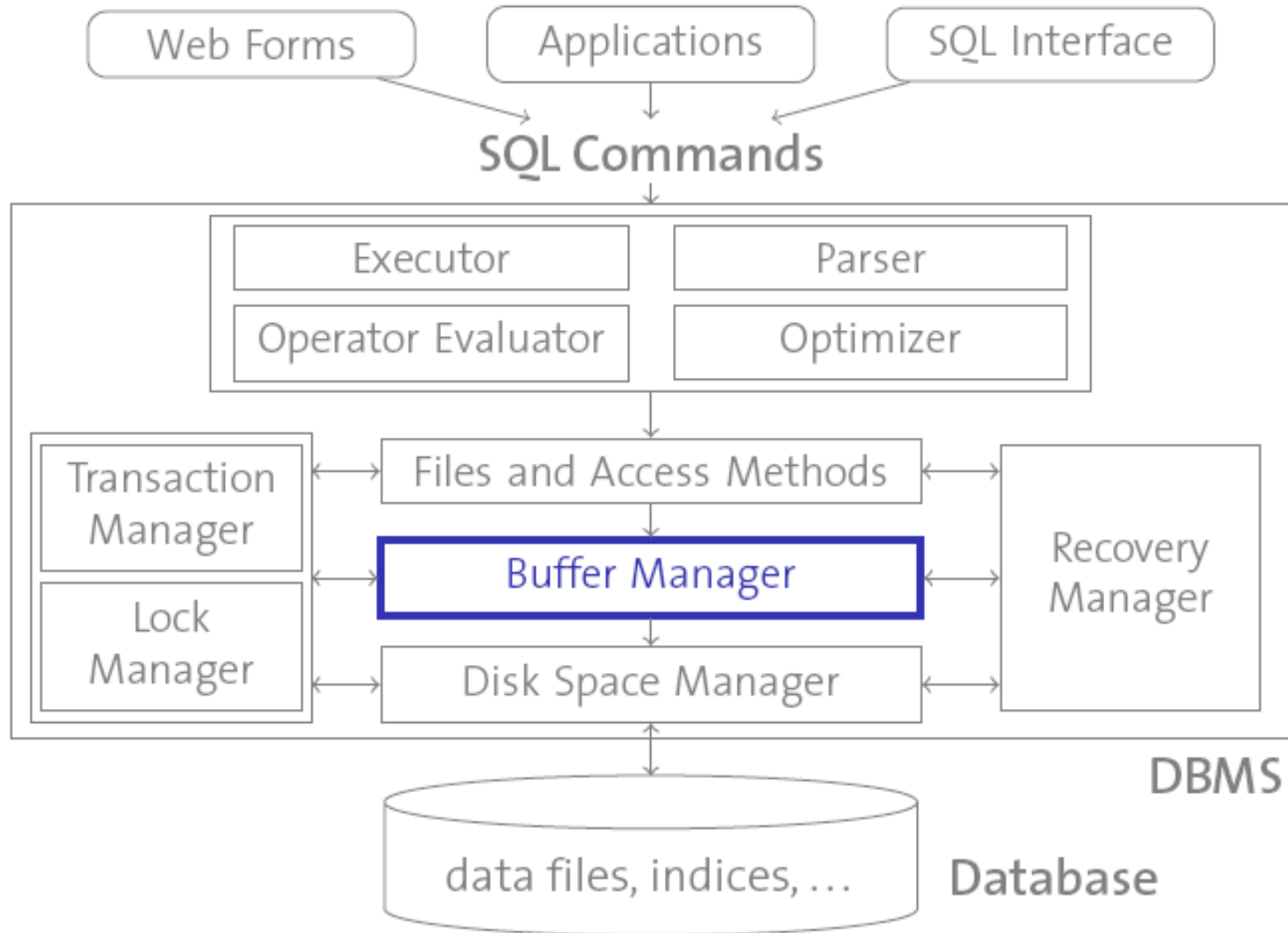
where a physical location could be, e.g.,

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

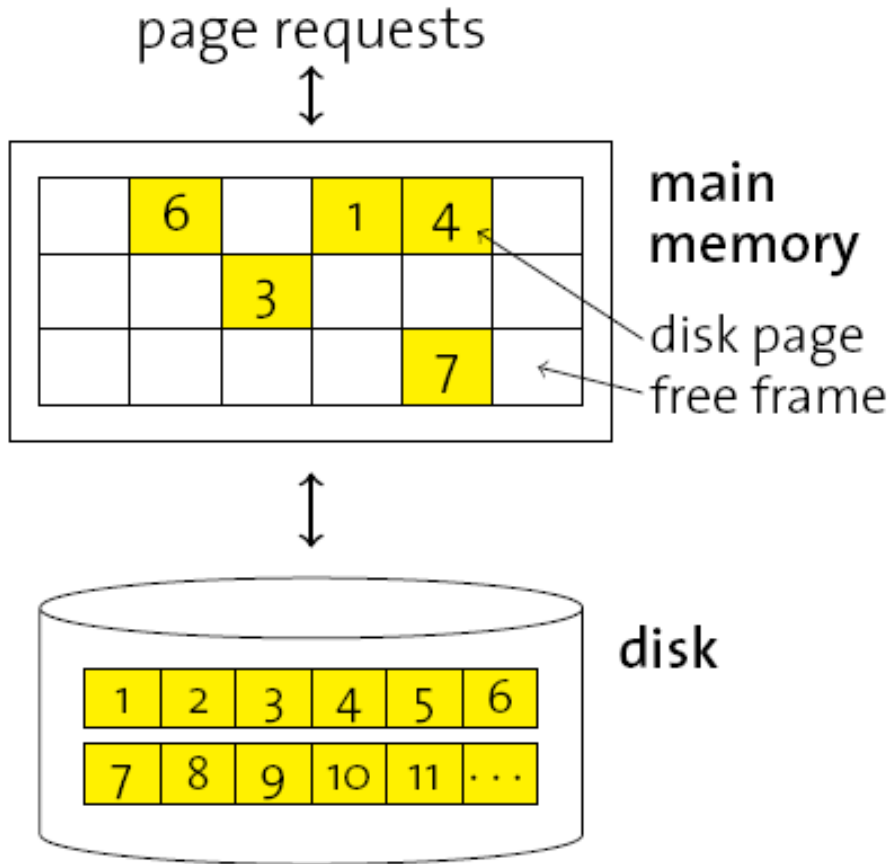
Empty Pages

- 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.

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 the page was modified.

Implementation of `pin()`

```
1 Function: pin(pageno)  
2 if buffer pool already contains pageno then  
3   | pinCount (pageno) ← pinCount (pageno) + 1;  
4   | return address of frame holding pageno ;  
5 else  
6   | select a victim frame v using the replacement policy ;  
7   | if dirty (v) then  
8   |   | write v to disk ;  
9   |   | read page pageno from disk into frame v ;  
10  |   | pinCount (pageno) ← 1;  
11  |   | dirty (pageno) ← false ;  
12  |   | return address of frame v ;
```

Implementation of `unpin()`

```
1 Function: unpin(pageno, dirty)  
2 pinCount(pageno) ← pinCount(pageno) - 1;  
3 if dirty then  
4   dirty(pageno) ← dirty;
```


Page Replacement

- Only frames with `pinCount=0` can be chosen for replacement.
- If no such frames, the buffer manager has to wait until there is one.
- If many such frames, one is chosen based on the buffer manager's replacement policy.

Page 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 the longest ago.
- **Most Recently Used (MRU)**
 - Evict the page that has been unpinned the most recently.
- **Random**
 - Pick a victim randomly.

Buffer Management in Reality

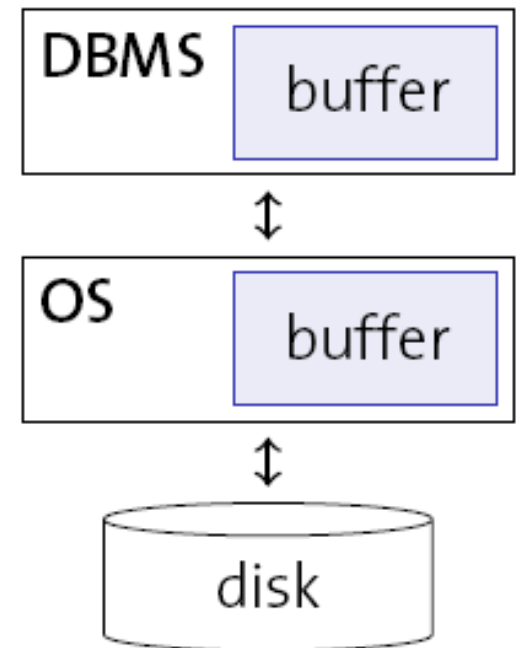
- Prefetching
 - Buffer managers try to anticipate page requests to overlap CPU and I/O operations.
 - **Speculative prefetching:** Assume sequential scan and automatically read ahead.
 - **Prefetch lists:** Some database algorithms can instruct the buffer manager with a list of pages to prefetch.
- 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 indices and tables.

Database Systems vs. Operating Systems

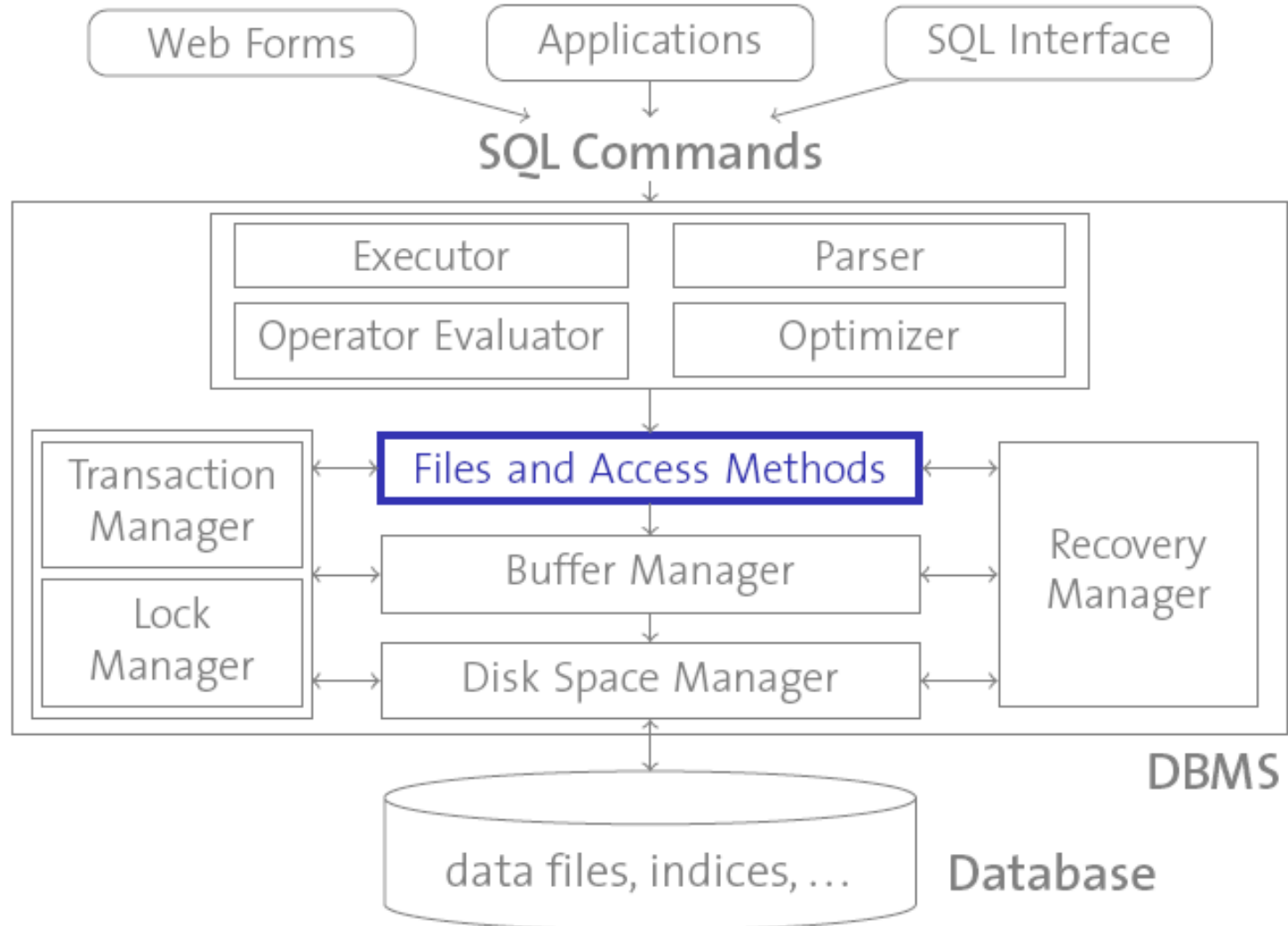
- Didn't we just re-invent the operating system?
- Yes,
 - disk space management and buffer management very much look like **file management** and **virtual memory** in OSs.
- But,
 - a DBMS may be much more aware of the **access patterns** of certain operators (prefetching, page fixing/hating),
 - concurrency 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).

Database Systems vs. Operating Systems

- In fact, DBMS and OS 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.
 - Similar problems: scheduling (Linux 3.6 vs. Postgres)
 - Classical approach: databases try to **turn off** OS functionality as much as possible.
 - Ongoing Research: New interfaces, cooperation, e.g. COD at CIDR 2013

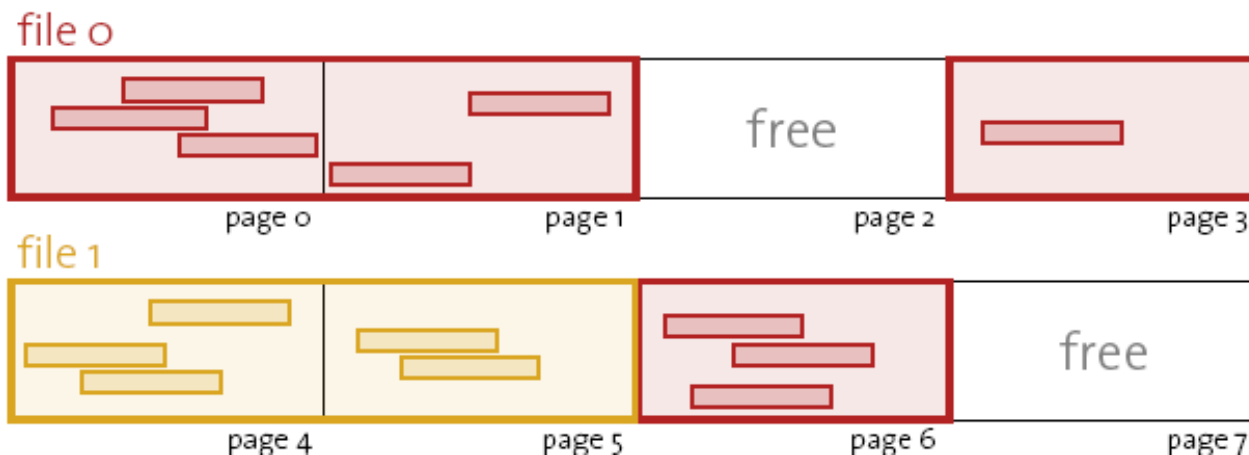


Files and Records



Database Files

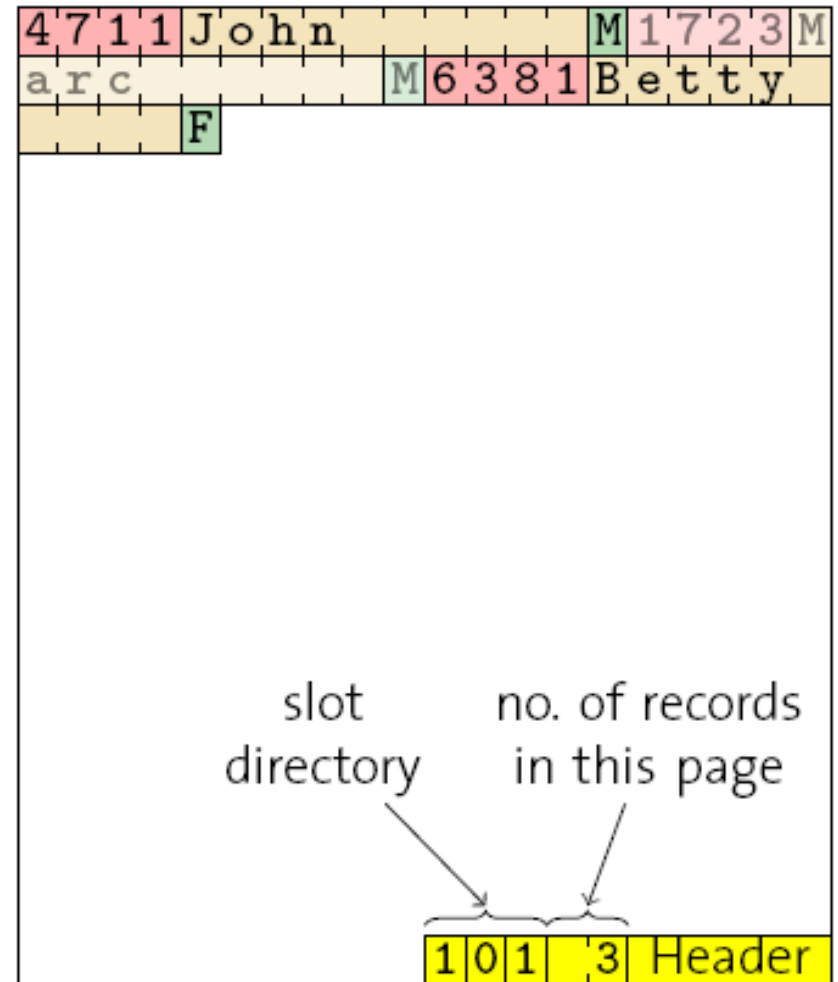
- So far we have talked about **pages**. Their management is oblivious with respect to their actual content.
- On the conceptual level, a DBMS manages **tables of tuples** and **indices** (among others).
- Such tables are implemented as **files of records**:
 - A **file** consists of **one or more pages**.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



Inside a Page

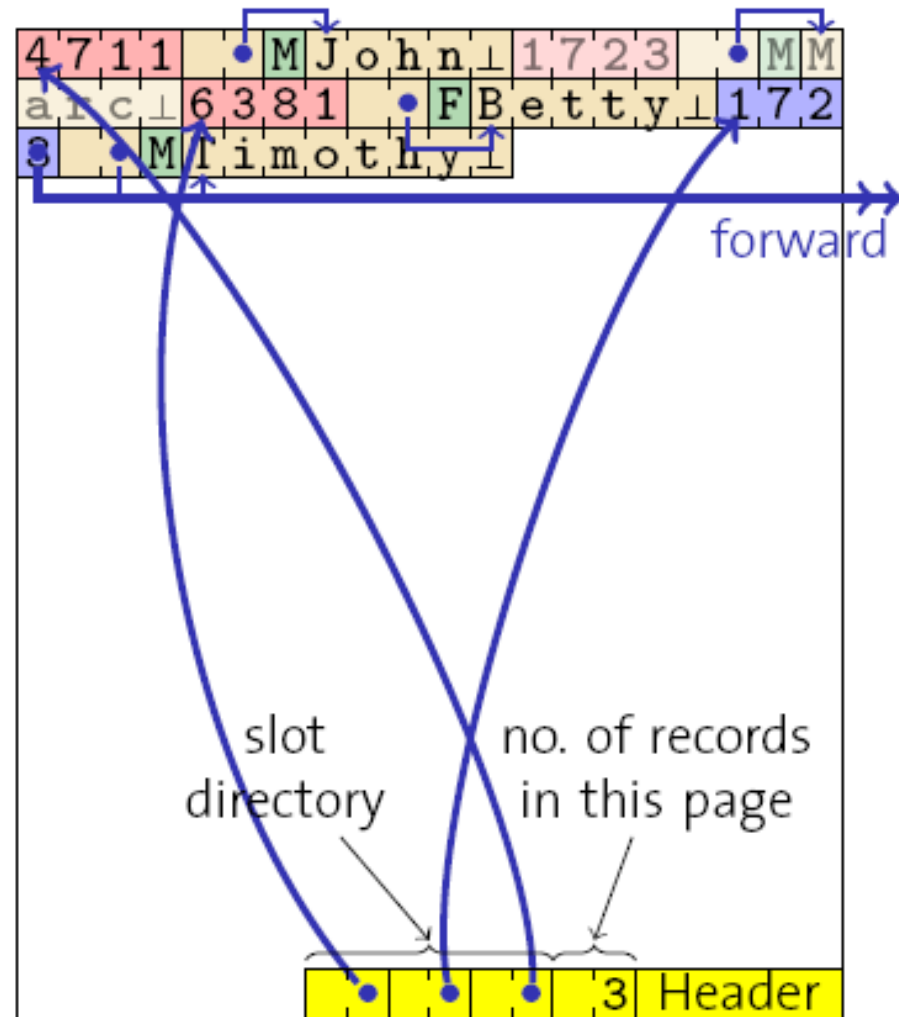
ID	NAME	SEX
4711	John	M
1723	Marc	M
6381	Betty	F

- **record identifier (rid):**
 <pageno, slotno>
- record position (within page):
 slotno x bytes per slot
- **Tuple deletion?**
 - record id shouldn't change
 - slot directory (bitmap)



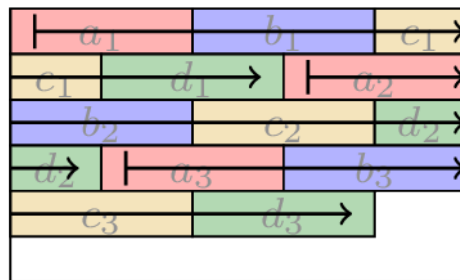
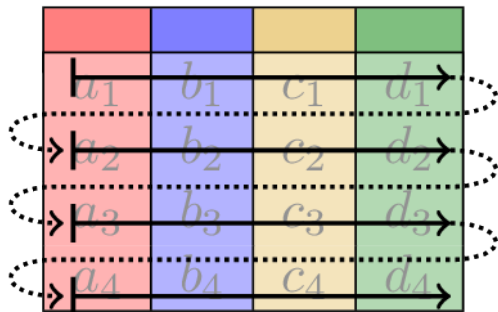
Inside a Page: Variable-sized Fields

- Variable-sized fields moved to **end** of each record.
 - Placeholder points to location.
- 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.

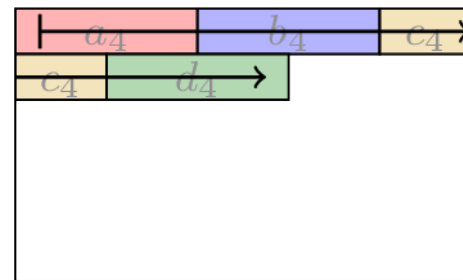


Variants of page layout

- Row-wise

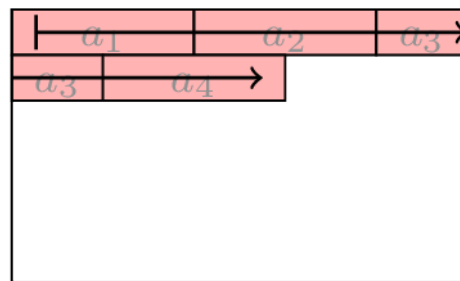
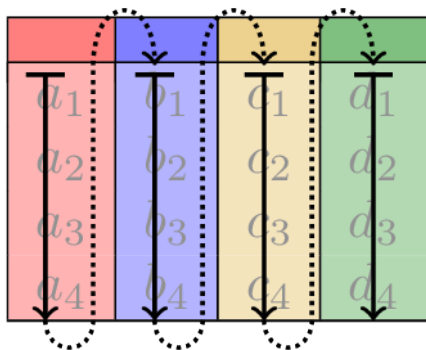


page 0



page 1

- Column-wise



page 0



page 1

(Figures by Jens Teubner, ETH)

Row Stores vs. Column Stores

- Which one is better?
- For what?
- Some ideas for evaluation:
 - How many attributes to tuples have? How many of them are read in a typical query
 - How would they deal with different levels of the storage hierarchy?
 - What happens on an update?
 - Which one would compress better?

Summary

