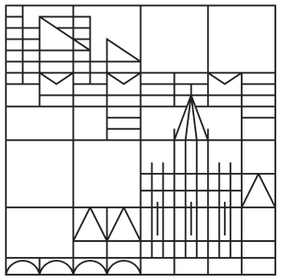
Universität
Konstanz



Database System Architecture and Implementation

Module 1

Storing Data: Disks and Files

October 22, 2018

Module Overview

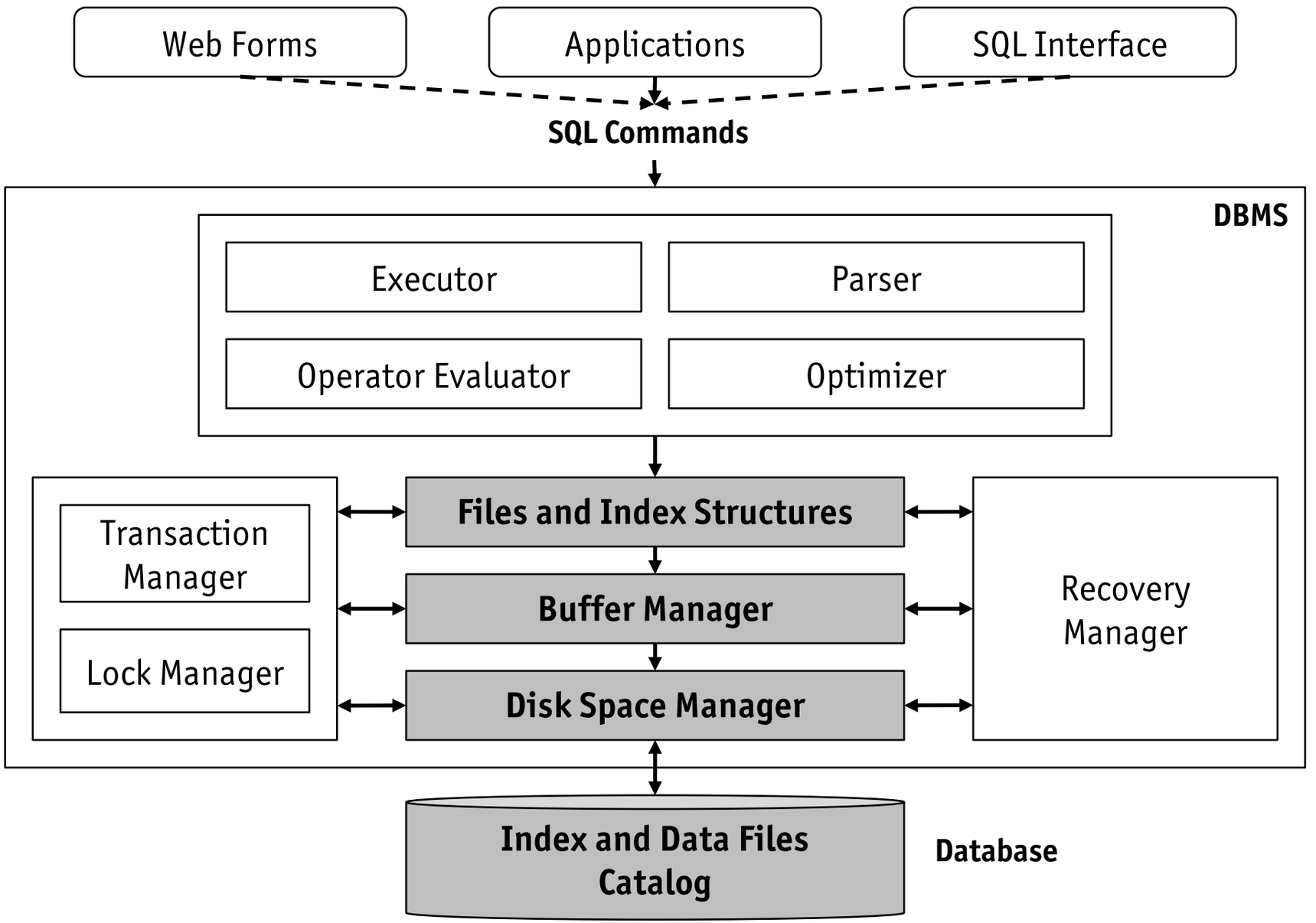


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Orientation

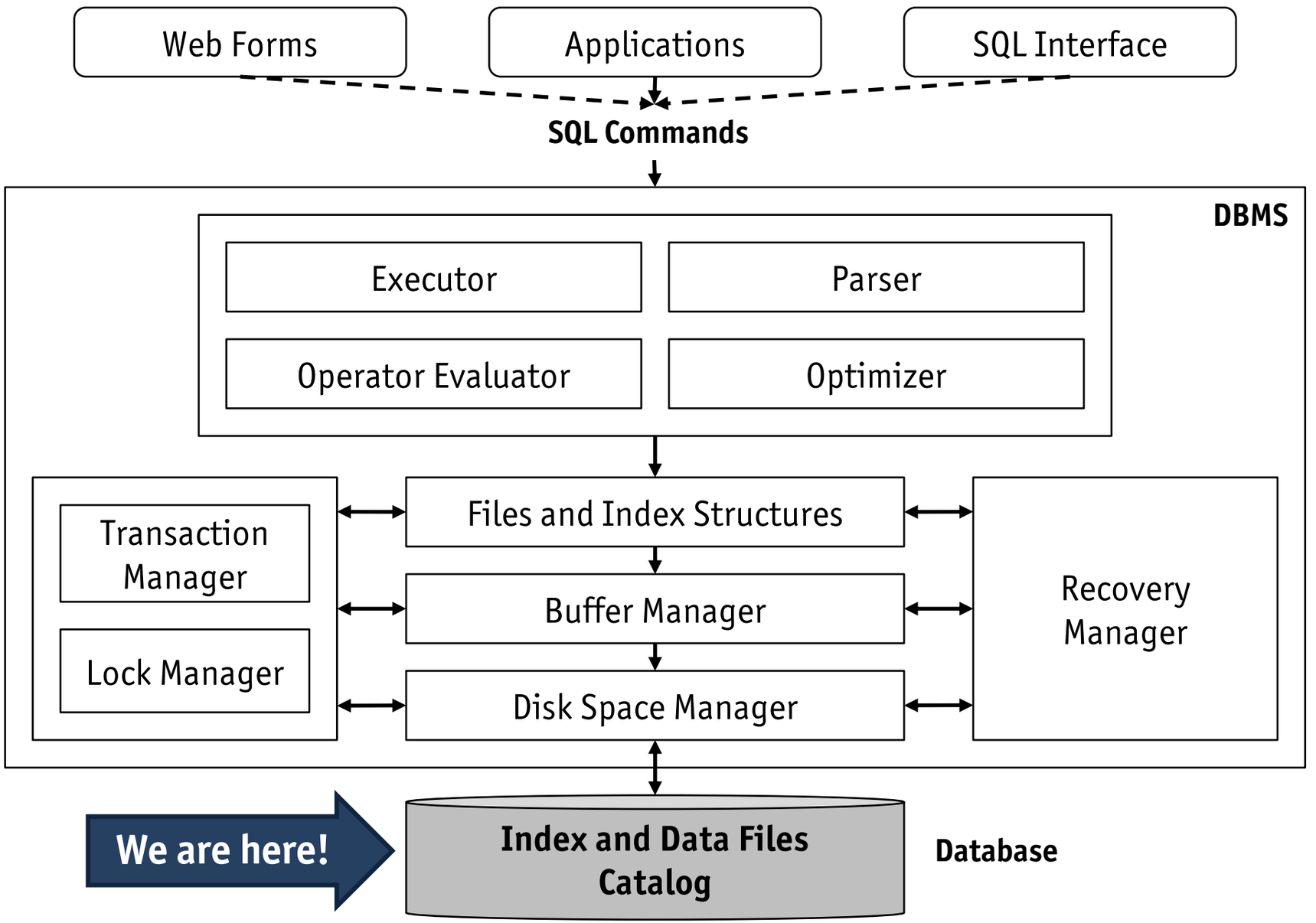
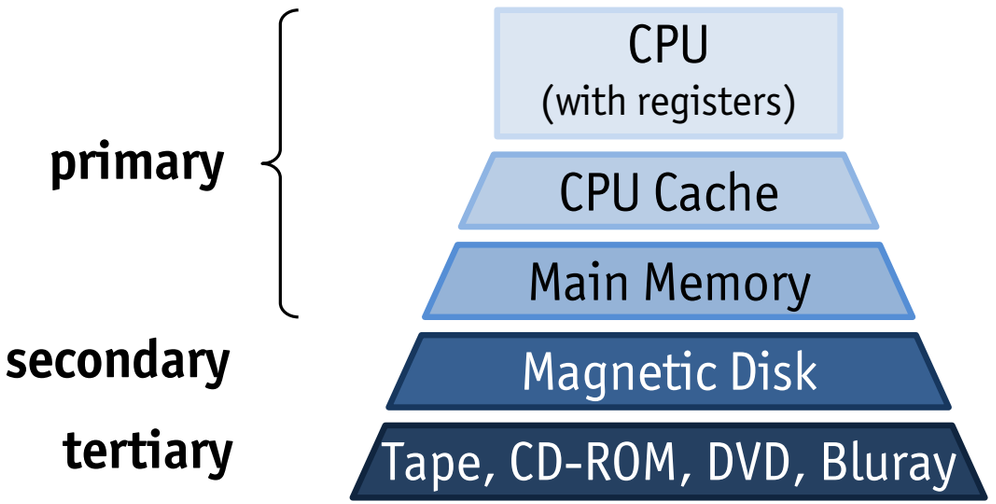


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

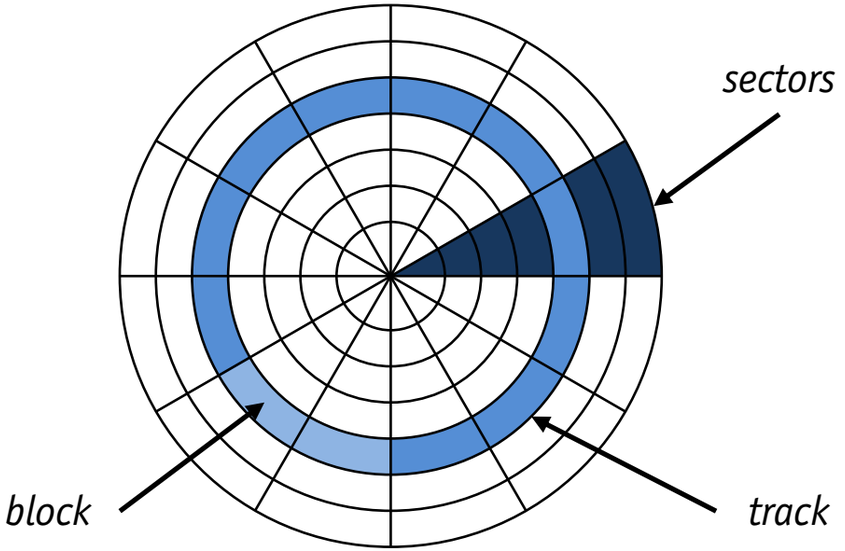
Memory Hierarchy

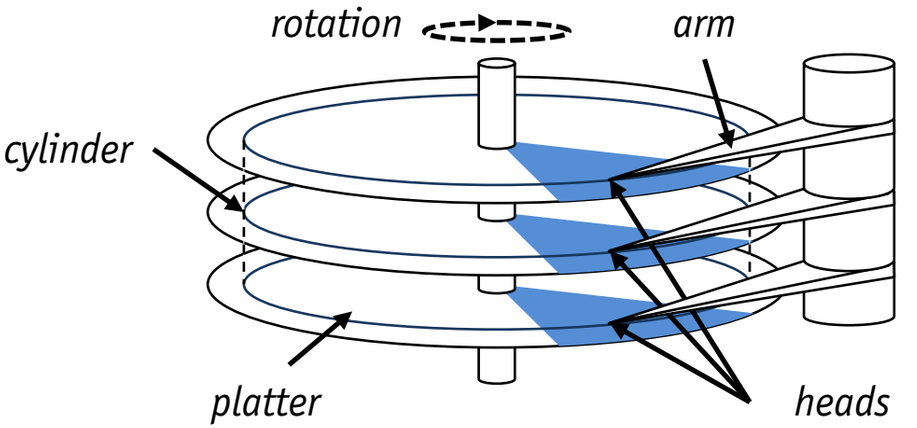


	capacity	latency
CPU (with registers)	bytes	< 1 ns
CPU Cache	kilo/megabytes	< 10 ns
Main Memory	gigabytes	70-100 ns
Magnetic Disk	terabytes	3-10 ms
Tape, CD-ROM, DVD, Bluray	petabytes	*varies*

- Memory in off-the-shelf computers is organized in a hierarchy
  - **cost** of primary memory ≈ 100 × cost of secondary storage of same size
  - **size** of address space in primary memory usually not large enough to map an entire database
- Only data stored in **non-volatile**, i.e., secondary, tertiary, storage persists across system shutdowns and crashes
- Goal is to **hide latency** by using the fast memory as **cache**

Magnetic Disks





- Data is arranged in concentric rings (**tracks**) on **platters** (one or two-sided)
- Tracks are divided into arc-shaped **sectors** (hardware characteristic)
- A **cylinder** is the set of all tracks with the same diameter
- A stepper motor moves an array of disk **arms** and **heads** from track to track as the disks steadily **rotate**
- Data is read from and written to disk one **block** at a time, which can be set to a multiple of sector size when formatting the disk, e.g., 4 kB or 8 kB

Access Time

- Data blocks can only be read and written if disk heads and platters are positioned accordingly
  - this design has implications on the **access time** required to read or write a given block
  - time to read or write data varies, depending on the location of the data
- **Definition**: access time is the sum $t = t_s + t_r + t_t$, where
  - **seek time** (t_s): disk heads have to be moved to desired track
  - **rotational delay** (t_r): disk controller has to wait for the desired block to rotate under disk head
  - **transfer time** (t_t): disk block data has to be written or read

Example

- Seagate Cheetah 15K.7 (2010)
  - 4 disks, 8 heads, 512 kB/track, 600 GB capacity
  - rotational speed 15,000 RPM (revolutions per minute)
  - average seek time 3.4 ms
  - transfer rate ≈ 163 MB/s

What is the access time to read an 8 kB block?

Slide Credit: Torsten Grust, University of Tübingen, Germany

Sequential vs. Random Access

Example: Read 1,000 blocks of size 8 kB

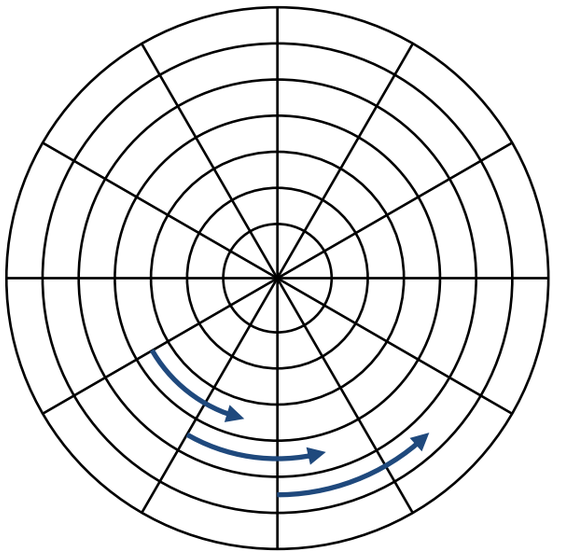
- The Seagate Cheetah 15K.7 stores an average of 512 kB per track, with a 0.2 ms track-to-tack seek time. Hence, our 8 kB blocks are spread across 16 tracks.
- **Random access**
  $t_{rnd} = 1{,}000 \times 5.45\ \text{ms} = \mathbf{5.45\ s}$
- **Sequential read of adjacent blocks**
  $t_{seq} = t_s + t_r + 1{,}000 \times t_t + 16 \times t_{s,\text{track-to-track}}$
  $= 3.40\ \text{ms} + 2.00\ \text{ms} + 50\ \text{ms} + 3.2\ \text{ms} \approx \mathbf{58.6\ ms}$

- Sequential access is **much** faster than random access
- **Avoid** random access whenever possible
- As soon as we need at least $\frac{58.6\ \text{ms}}{5{,}450\ \text{ms}} = 1.07\%$ of a file, we better read the **entire** file sequentially

Slide Credit: Torsten Grust, University of Tübingen, Germany

Performance Tricks

- Manufacturers use a number of tricks to improve performance
  - **track skewing:** align sector 0 of each track to avoid rotational delay using longer sequential scans
  - **request scheduling:** if multiple requests have to be served, chose the one that requires the smallest arm movement (SPTF: shortest positioning time first and elevator algorithms)
  - **zoning:** outer tracks are longer than inner ones and therefore can be divided into more sectors



Slide Credit: Torsten Grust, University of Tübingen, Germany

Evolution of Hard Disk Technology

- Disks are a potential bottleneck for system performance
  - performance of CPU has improved ≈ 50% per year
  - disk seek and rotational latencies have only marginally improved over the last years (≈ 10% per year)
- Sequential vs. random access ratio gets worse over time
  - throughput (i.e., transfer rates) improve by ≈ 50% per year
  - hard disk capacity grows by ≈ 50% every year

Example: Seagate Barracuda 7200.7 (10 years ago)

Read 1,000 blocks of 8 kB sequentially and randomly: 397 ms vs. 12,800 ms

Example Credit: Torsten Grust, University of Tübingen, Germany

Implications for DBMS

- Time for moving blocks to and from disk usually **dominates** time taken by a database operation
  - data must be **in memory** for DBMS to operate on it
  - a disk block is the **unit of data transfer** between disk and memory, which is called an I/O (input/output) operation
- DBMS takes geometry and mechanics of hard disk into account
  - transfer a whole track in one platter revolution
  - switch active disk head after each revolution
- This implies a **closeness measure** for data records r_1, r_2 on disk
  1. Place r_1 and r_2 in the same block (single I/O operation)
  2. Place r_2 inside a block adjacent to the block of r_1 on the **same track**
  3. Place r_2 in a block somewhere on the track of r_1
  4. Place r_2 in a track of the **same cylinder** as the track of r_1
  5. Place r_2 in a cylinder adjacent to the cylinder of r_1

Improving I/O Performance

1. Reduce **number** of I/O operations
   - DBMS buffer: cache data in fast memory to hide latency
   - physical database design: tables, indexes, ...
2. Reduce **duration** of I/O operations
   - access neighboring disk blocks (**clustering**) and bulk I/O
   - different I/O paths (**de-clustering**) with parallel access

Clustering vs. De-clustering

- Clustering
  - bulk I/O can be implemented on top of or inside disk controller
  - **advantages:** optimized seek time and rotational delay, minimized overhead (e.g., interrupt handling), no additional hardware needed
  - **disadvantage:** I/O path busy for a long time (💣concurrency)
  - **use case:** mid-sized data access (prefetching, sector buffering)
- De-clustering
  - advanced hardware or disk arrays (RAID systems)
  - **advantages:** parallel I/O operations, minimized transfer time with multiplied bandwidth
  - **disadvantages:** average seek time and rotational delay increased, blocking of parallel transactions, additional hardware needed
  - **use case:** large-size data access

RAID Systems

- **R**edundant **A**rray of **I**ndependent **D**isks
  - storage system built as an **arrangement** of several disks
  - designed to **improve reliability** and **increase performance**
- Reliability
  - **redundancy:** replicate data onto multiple disks
  - leverage redundant information to improve **mean-time-to-failure**
- Performance
  - **data striping:** distribute data over disks
  - exploit parallel I/O to emulate a single large and very fast disk
- Implementation of RAID logic
  - **hardware RAID:** inside the disk subsystem or disk controller
  - **software RAID:** inside the operating system

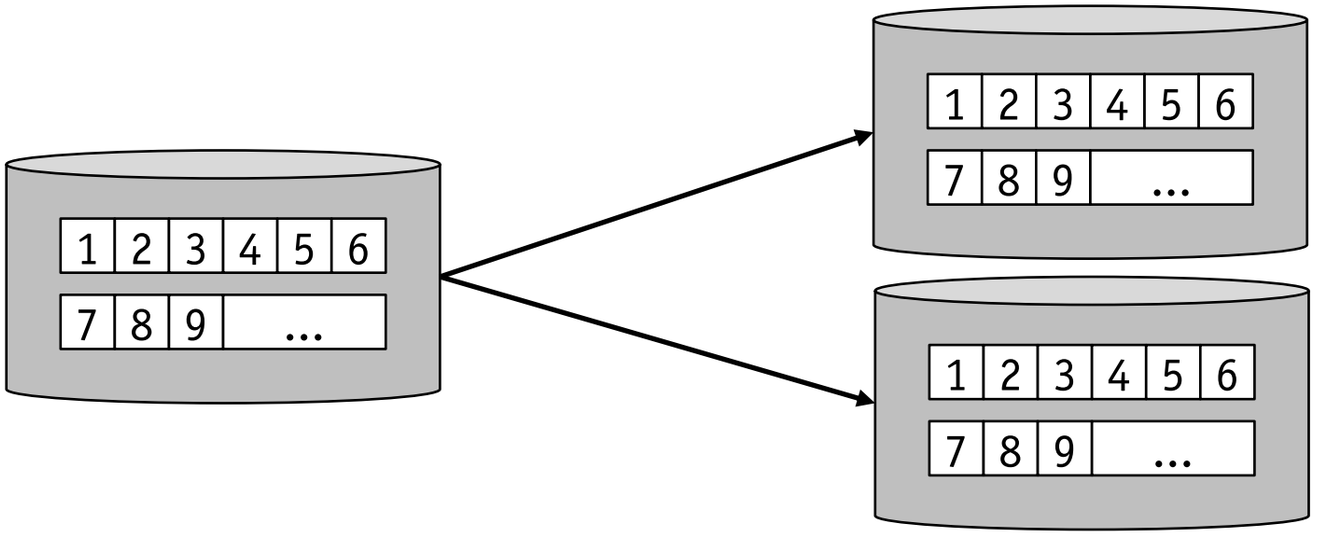
MTTF: Mean-Time-To-Failure

Example

If the MTTF of a single disk is 50,000 hours (≈ 5.7 years), then the MTTF of an array of 100 disks is only 50,000/100 = 500 hours (≈ 21 days)!

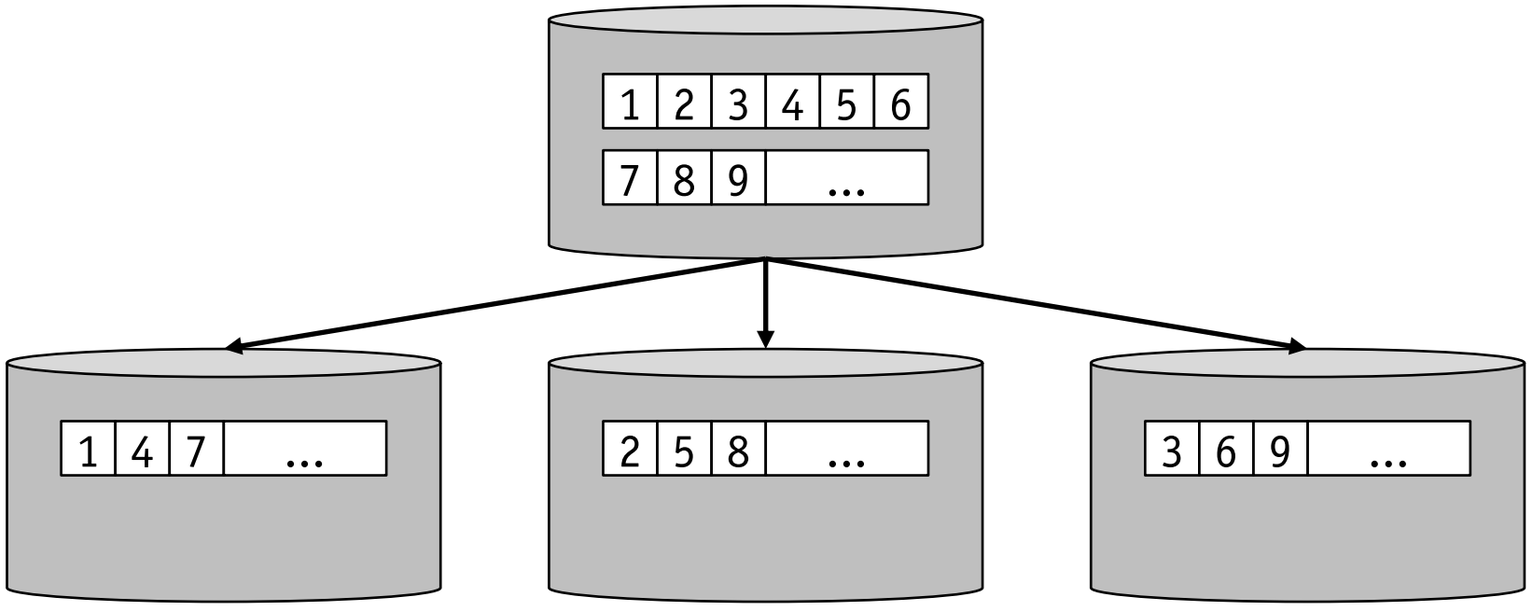
- Having more disks **increases** storage system performance, but also **decreases** overall storage system reliability
- 💣 Example is too simple!
  - disk failures do not occur independently, e.g., because of a "bad" batch
  - failure probability changes over time, i.e., disk failures are more likely to occur early and late in their lifetimes
- Use redundant information to reconstruct data of a failed disk
  - where is redundant information stored?
  - how is redundant information computed?

Disk Mirroring



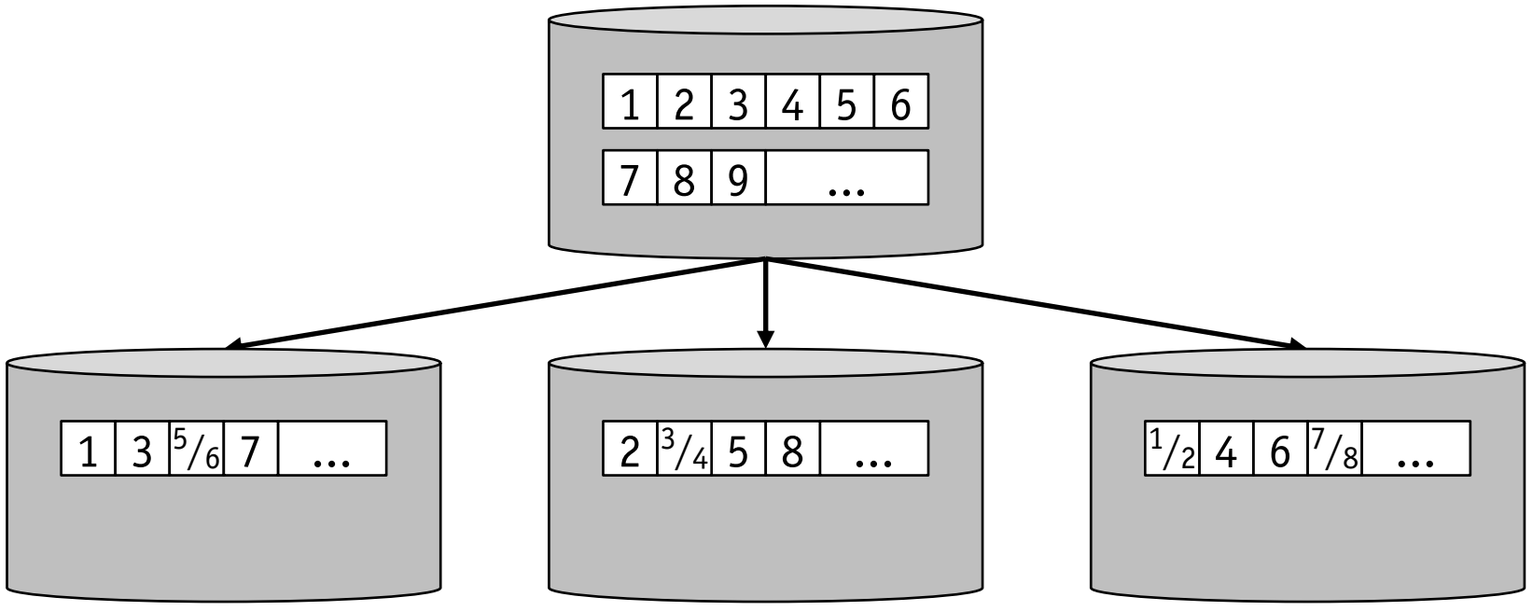
- Replicate copies of data onto multiple disks
  - write operations are executed on **all** disks
  - read operations are executed on **one** disk
- Benefits
  - **performance:** I/O parallelism only for reads
  - **reliability:** can survive failure of one disk

Disk Striping



- Partition data into equally-sized chunks of consecutive blocks
- Striping unit (i.e., chunk size) determines **degree of parallelism** for **single** I/O and **between different** I/O operations
  - **small chunks**: high intra-access parallelism, but many devices busy, i.e., not many I/O operations in parallel
  - **large chunks**: high inter-access parallelism, i.e., many I/O operations in parallel

Disk Striping with Parity



- Pure striping has a high failure risk as there is no redundancy
- Distribute data and parity information over ≥ 3 disks
- Benefits
  - **performance:** high I/O parallelism
  - **reliability:** depending on redundancy scheme, one or even two disks can fail at the same time

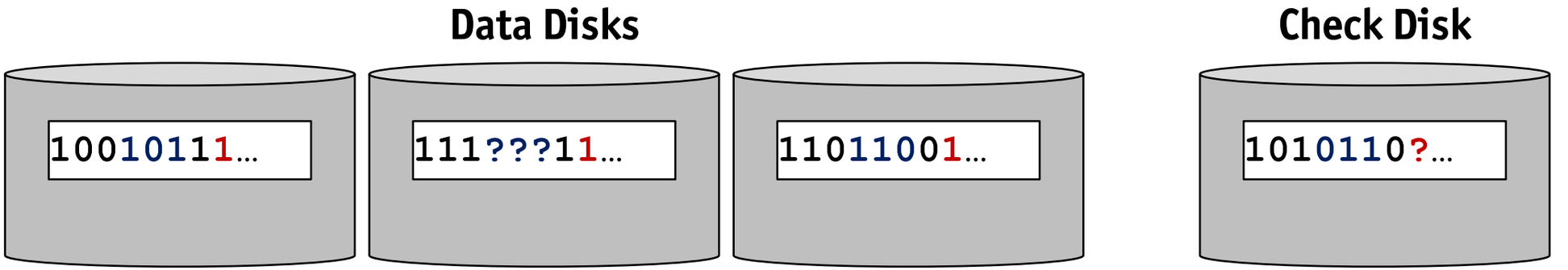
RAID Levels

- Based on the different options to maximize reliability and performance, so-called **RAID Levels** have been defined
- Striping unit (data interleaving)
  - **distribution**: how to scatter (primary) data across disks
  - **granularity**: fine (bits/bytes) or coarse (blocks)
- Computation and allocation of redundant information
  - **redundancy schemes**: parity, ECC, etc.
  - **distribution**: separate/few disks vs. all disks of the array
- Initially, five RAID Levels have been introduced, additional levels were later defined

Redundancy Schemes

- Parity scheme
  - requires one additional check disk
  - recovery from **one** disk is possible
- Hamming codes
  - requires $\log n$ check disks, assuming n data disks
  - recovery from **one** disk is possible, faulty disk can be identified
- Reed-Solomon codes
  - requires **two** check disks
  - recovery from **up to two** disks is possible

Parity Scheme



- Assume a disk array with D data disks ($D = 3$ above)
- **Initialization**
  - let $i(n)$ be the number of D data bits at the nth position that are **1**
  - nth **parity bit** on check disk is set to **1** if $i(n)$ is odd and to **0** otherwise
- **Recovery**
  - let $j(n)$ be the number of nth data bits that are **1** on $D - 1$ non-failed disks
  - if $j(n)$ is odd and the nth parity bit is **1**, or if $j(n)$ is even and the nth parity bit is **0**, then the value of the nth bit on the failed disk must be **0**
  - otherwise the value of the nth bit on the failed disk must be **1**

RAID Levels 0, 1, 10, and 2

- RAID Level 0: Non-redundant, just striping
  - least storage overhead (no redundancy)
  - **write**: no extra effort, **read**: not the best performance
- RAID Level 1: Mirroring
  - double necessary storage
  - **write**: doubles write access, **read**: optimized performance
- RAID Level 10 (a.k.a. 0+1): Striping and Mirroring
  - **write**: analogous to Level 1, **read**: improved performance over Level 0
- RAID Level 2: Error-Correcting Codes (ECC)
  - **initialization**: uses bit-level striping and Hamming code to compute ECC for data of n disks, which is store onto $n - 1$ additional disks
  - **recovery**: determine lost disk by using $n - 1$ extra disk and correct (reconstruct) its content from one of those
  - large requests benefit from aggregated bandwidth (💣 small requests)

RAID Levels 3, 4, 5, and 6

- RAID Level 3: Bit-Interleaved Parity
  - one parity disk suffices, since controller can easily identify faulty disk
  - distribute (primary) data bit-wise onto data disks
  - **read/write** go to all disks: no inter-I/O parallelism, but high bandwidth
- RAID Level 4: Block-Interleaved Parity
  - like RAID 3, but distribute data block-wise onto data disks
  - small **reads** go to one disk, but all **writes** go to one parity disk
- RAID Level 5: Block-Interleaved Distributed Parity
  - like RAID 4, but distribute parity blocks across all disks (load balancing)
  - best performance for small and large **reads** as well as large **writes**
- RAID Level 6: P+Q Redundancy
  - like RAID 5, but adds an additional parity block
  - performance equivalent to RAID 5, except that **small writes** involve more disks

Overview of RAID Levels

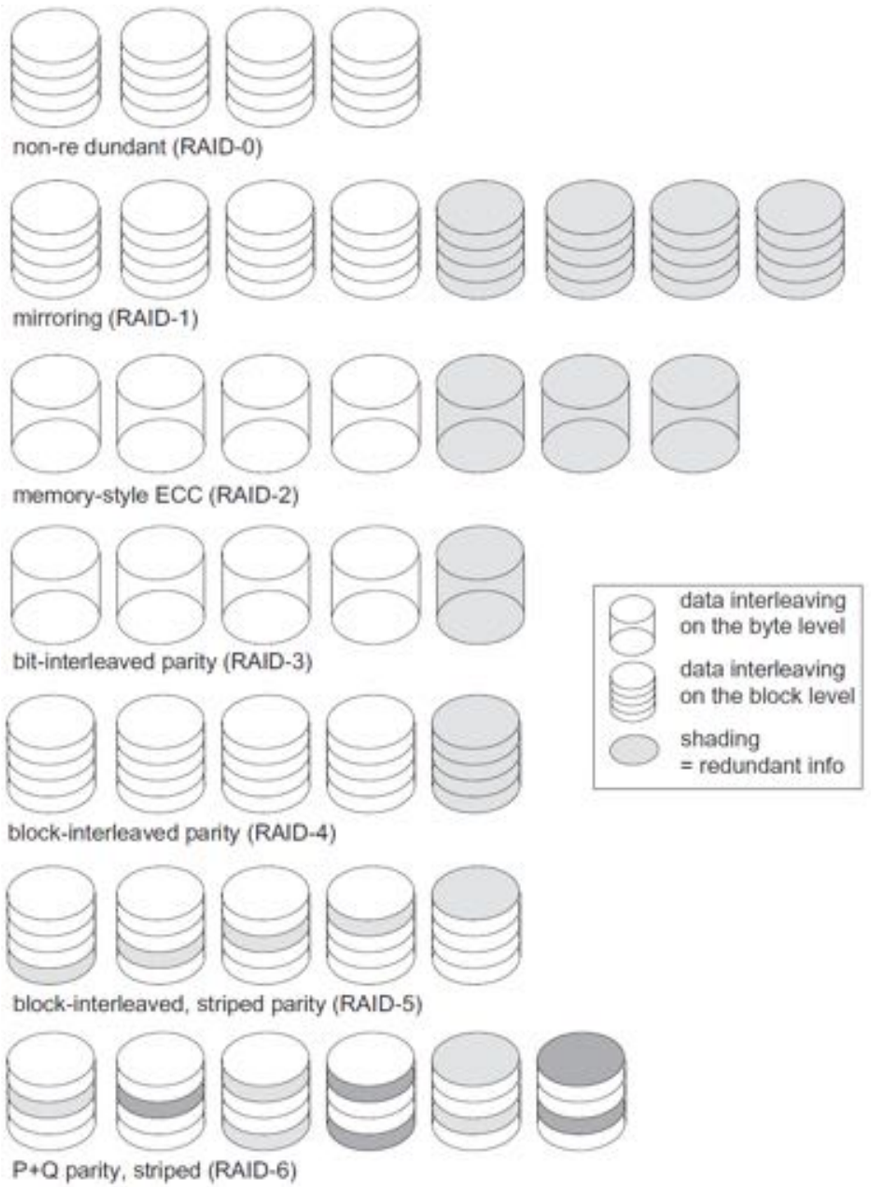
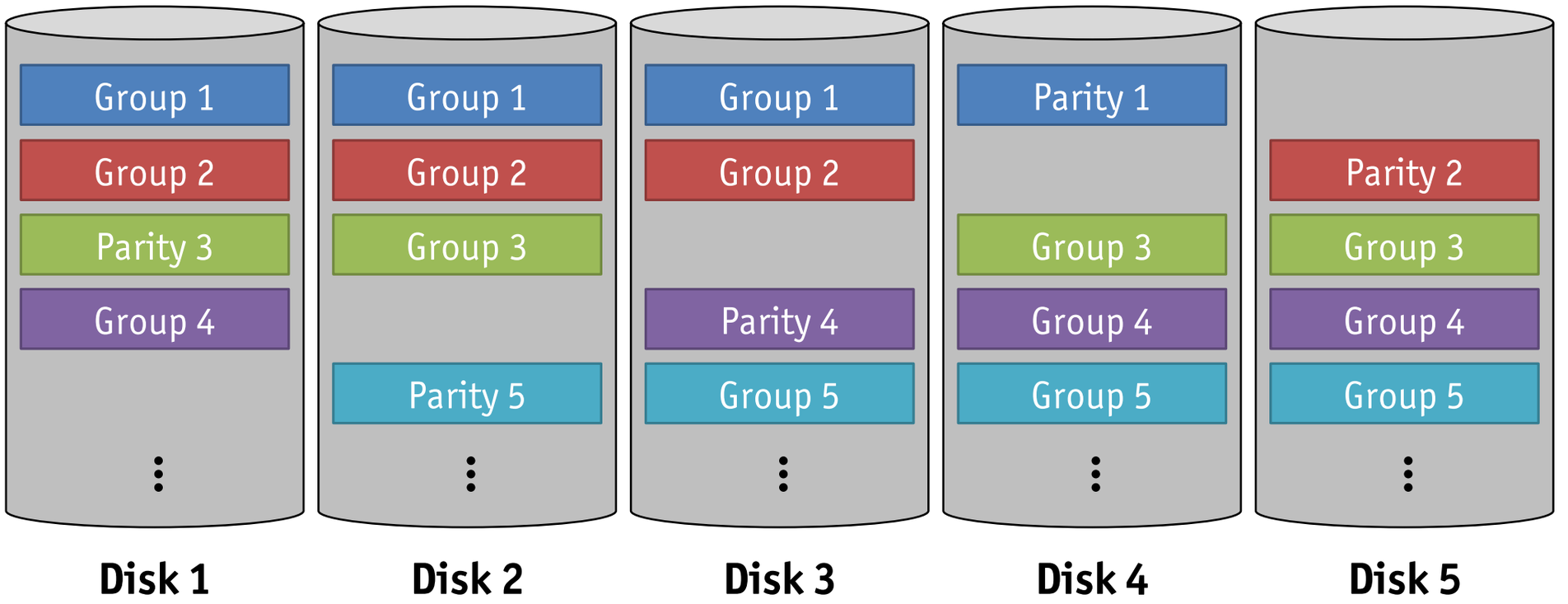


Figure Credit: Marc H. Scholl, University of Konstanz, Germany

Parity Groups

- Parity is not necessarily computed across all disks of an array
- Possible to define parity groups (of same or different sizes)



Selecting RAID Levels

- RAID Level 0
  - improves overall performance at lowest cost
  - no provision against data loss
  - best write performance, since no redundancy
- RAID Level 10
  - superior to level 1
  - main application area is small storage subsystems, sometimes write-intensive applications
- RAID Level 1
  - most expensive variant
  - typically serialize to necessary I/O operations for writes to avoid data loss in case of power failure, etc.

Selecting RAID Levels

- RAID Level 2 and 4
  - always inferior to levels 3 and 5, respectively
- RAID Level 3
  - appropriate for workloads with large request for contiguous blocks
  - bad for many small requests of a single block
- RAID Level 5
  - good general-purpose solution
  - best performance (with redundancy) for small and large read as well as large write operations
- RAID Level 6
  - choice for higher level or reliability

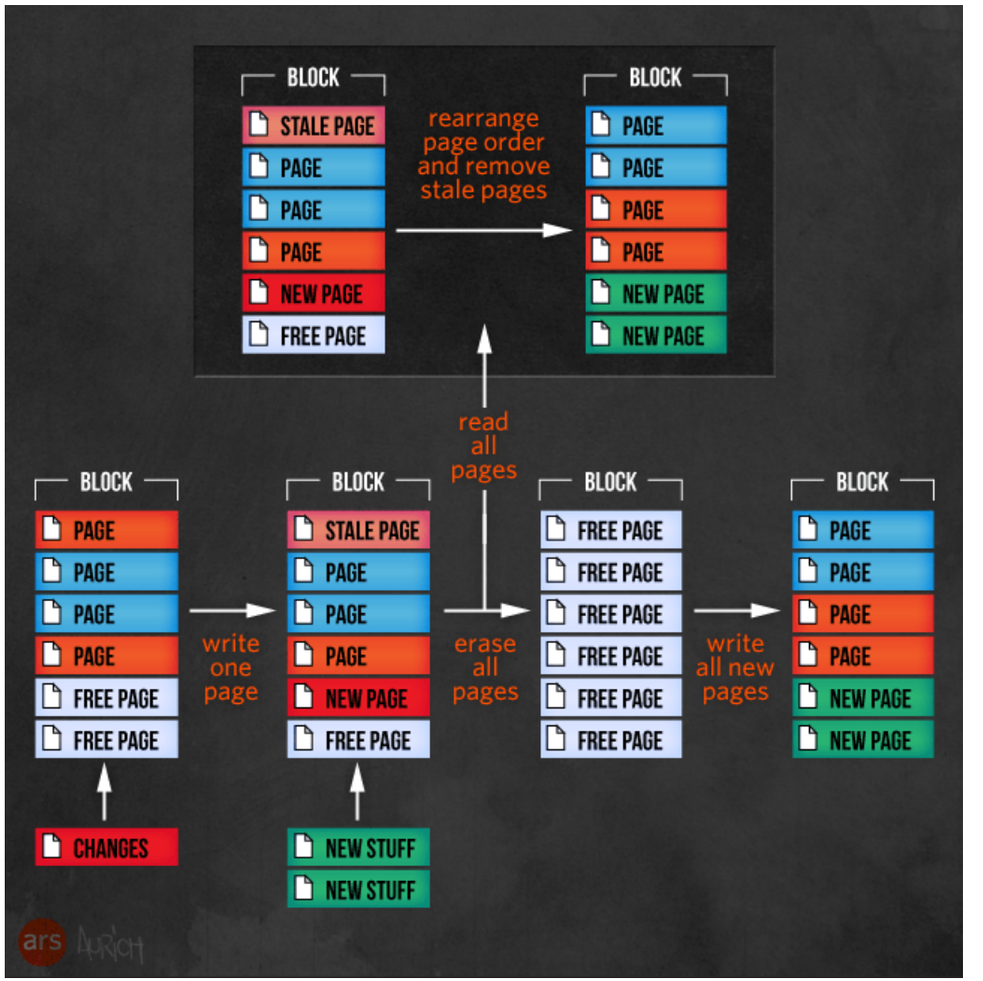
Solid-State Disks

- Solid-state disks (SSD) have emerged as an alternative to conventional hard disks (HDD)
- SSD provide **very low-latency read access** (< 0.01 ms)
- Random writes are **significantly slower** than on a HDD
  - (blocks of) pages have to be erased before they can be updated
  - once pages have been erased, sequentially writing them is almost as fast as reading

Slide Credit: Torsten Grust, University of Tübingen, Germany

Solid-State Disks

- Page-level writes
  - typical **page size** is 128 kB
- Block-level deletes
  - SSD erase **blocks of pages**
  - block ≈ 64 pages (8 MB)



Picture Credit: arstechnica.com (The SSD Revolution)

Slide Credit: Torsten Grust, University of Tübingen, Germany

Example

- Seagate Pulsar.2 (2012)
  - NAND flash memory, 800 GB capacity
  - standard 2.5” enclosure, no moving or rotating parts
  - data read and written in pages of size 128 kB
  - transfer rate ≈ 370 MB/s

What is the access time to read an 8 kB block?

Slide Credit: Torsten Grust, University of Tübingen, Germany

Sequential vs. Random Access

Example: Read 1,000 blocks of size 8 kB

- The Seagate Pulsar.2 (sequentially) reads data in 128 kB chunks.
- **Random access**
  $t_{rnd} = 1{,}000 \times 0.30\ \text{ms} = \mathbf{0.30\ s}$
- **Sequential read of adjacent blocks**
  $t_{seq} = \left\lceil \frac{1{,}000 \times 8\ \text{kB}}{128\ \text{kB}} \right\rceil \times t_t \approx \mathbf{18.9\ ms}$

- Sequential access still beats random access
  - but random access is once again more feasible
- Adapting database technology to these characteristics is a current research topic

Slide Credit: Torsten Grust, University of Tübingen, Germany

Network and Cloud-based Storage

- Today the network is **not** a bottleneck anymore
- Storage area network (SAN)
  - emulate interface of block-structured disks
  - hardware acceleration and simplified maintainability by abstracting from RAID or physical disk
  - fault tolerance and increased flexibility through multiple servers and storage resources
- Data management in the Cloud
  - University of Konstanz: Course INF-12820
  - M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska: **Building a Database on S3**. In *Proc. Intl. Conf. on Management of Data (SIGMOD), pp. 251-264, 2008.*

Orientation

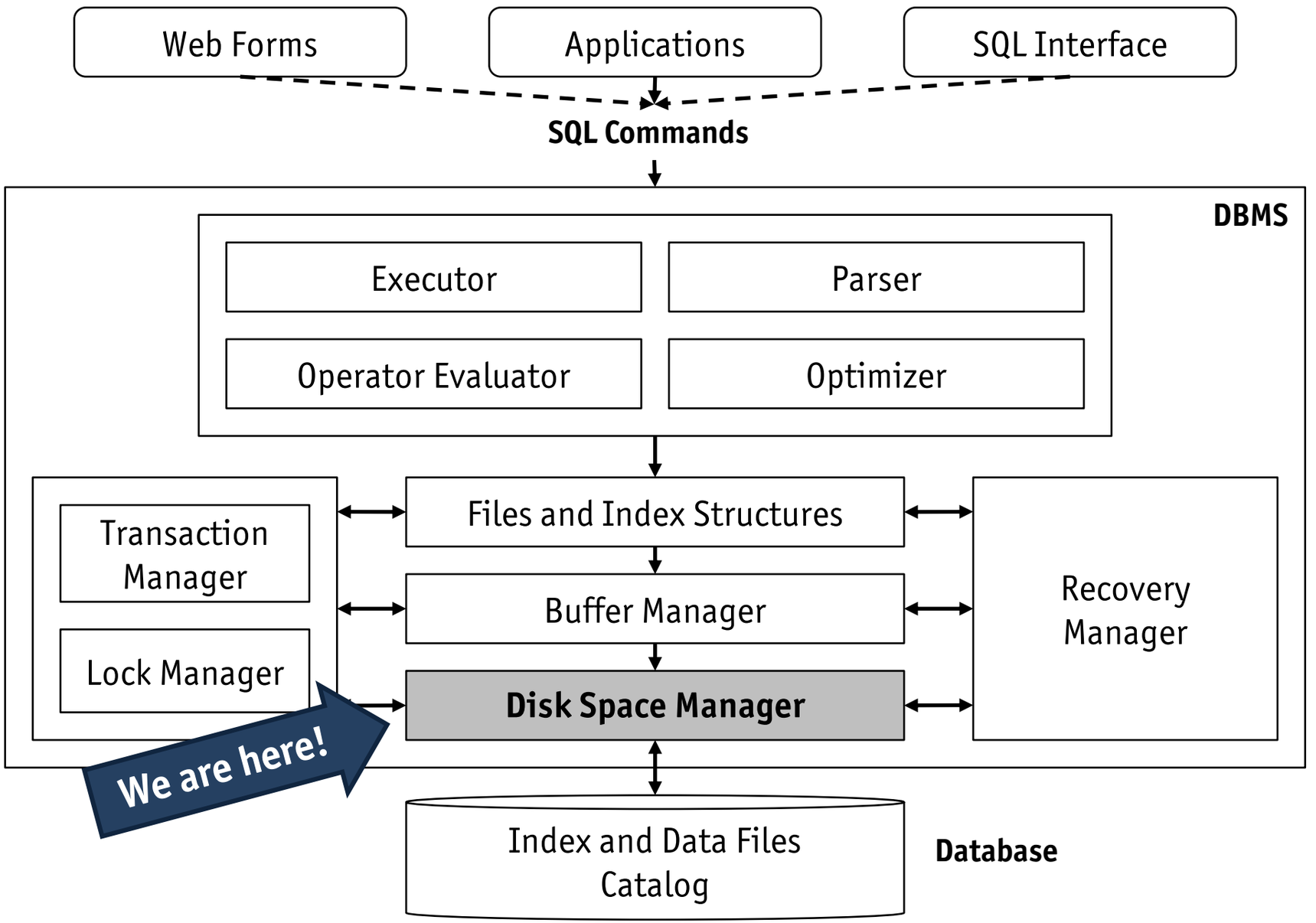


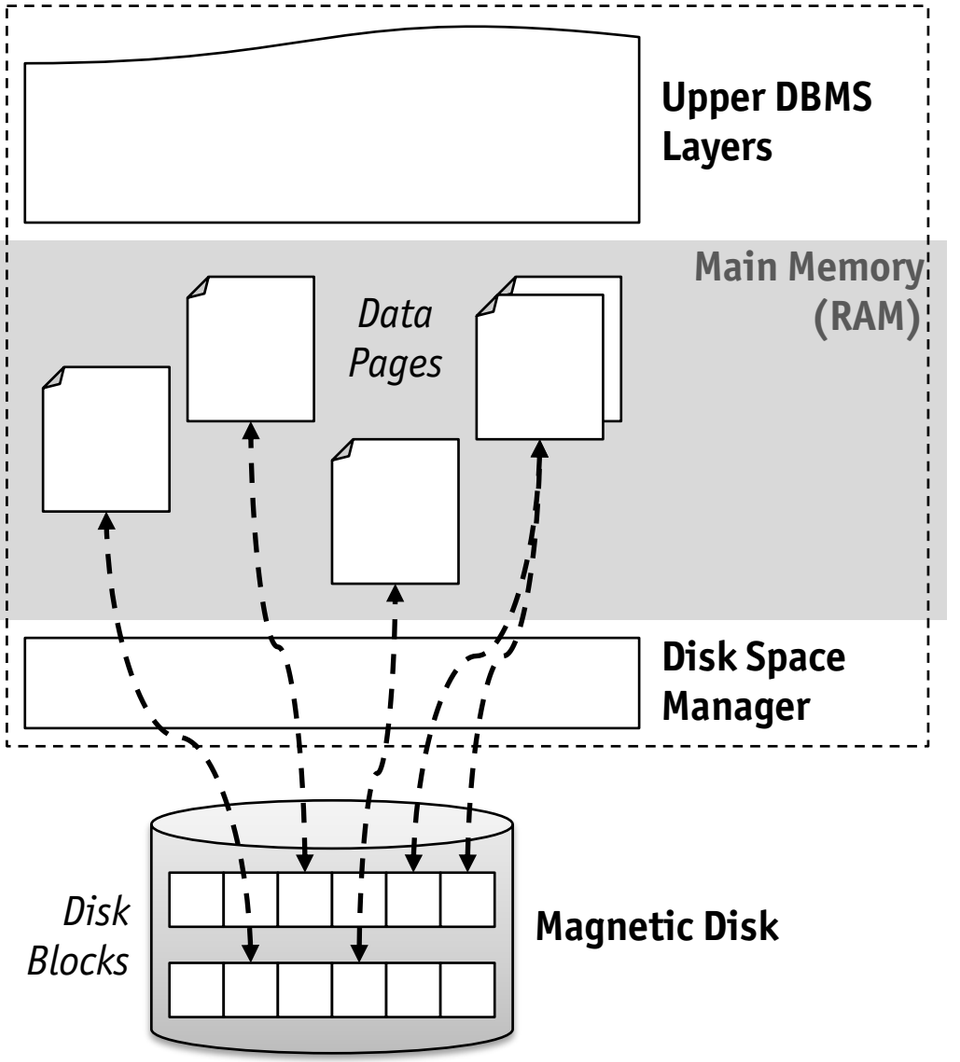
Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Disk Space Manager

- Abstracts from the gory details of the underlying storage
  - disk space manager talks to disk controller and initiates I/O operations
  - DBMS issues **`allocate`**/**`deallocate`** and **`read`**/**`write`** commands to disk space manager
- Provides the concept of a **page**
  - a page is a disk block that has been brought **into memory**
  - disk blocks and pages are of the **same size**
  - **sequences** of pages are mapped onto **contiguous sequences** of blocks
  - **unit of storage** for all system components in higher layers

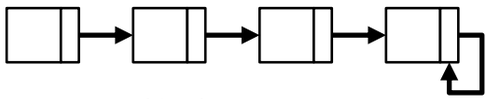
Disk Space Manager

- Locality-preserving mapping
  - track page locations and block usage internally
  - page number ↔ physical location
- Abstract physical location
  - OS file name and offset within that file
  - head, sector, and track of a hard disk drive
  - tape number and offset for data stored in tape library
  - …

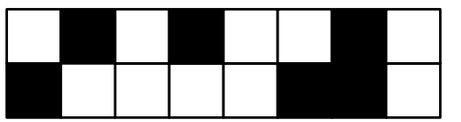


Managing Free Space

- Disk space manager also keeps track of **used** and **free blocks** to reclaim space that has been freed
  - blocks can usually be allocated contiguously on database/table creation
  - subsequent de-allocation and new allocation may create **holes**
- **Linked list** of free blocks



  1. keep a pointer to the **first free block** in a known location on disk
  2. when a block is no longer needed, append/prepend it to the list
  3. the **`next`** pointer may be stored in blocks themselves
- Free block **bitmap**



  1. reserve a block whose bytes are interpreted bitwise (bit $n = 0$: block n is free)
  2. toggle bit n whenever block n is (de-)allocated

Contiguous Sequences of Pages

Exercise

To exploit **sequential access**, it is useful to allocate **contiguous sequences** of pages

Which technique would you use, a linked list or a bitmap of free blocks?

Slide Credit: Torsten Grust, University of Tübingen, Germany

Segments, Table Spaces, and Partitions

- Most DBMS do not manage the entire data store as **one** contiguous storage space
- Data store can be partitioned into smaller units
  - segments
  - table spaces
  - partitions
  - storage pools
  - *and many other names*
- These units are arranged in a layered stack on top of the physical disks
  - **simple use of OS file** *table*←[1:1]→*segment*←[1:1]→*(OS) file*
  - **more flexible** *table*←[n:1]→*segment*←[1:1]→*(OS) file*
  - **advanced** *table*←[n:1]→*segment*←[n:1]→ *storage group*…
    …←[n:1]→*logical (OS) volume*

Orientation

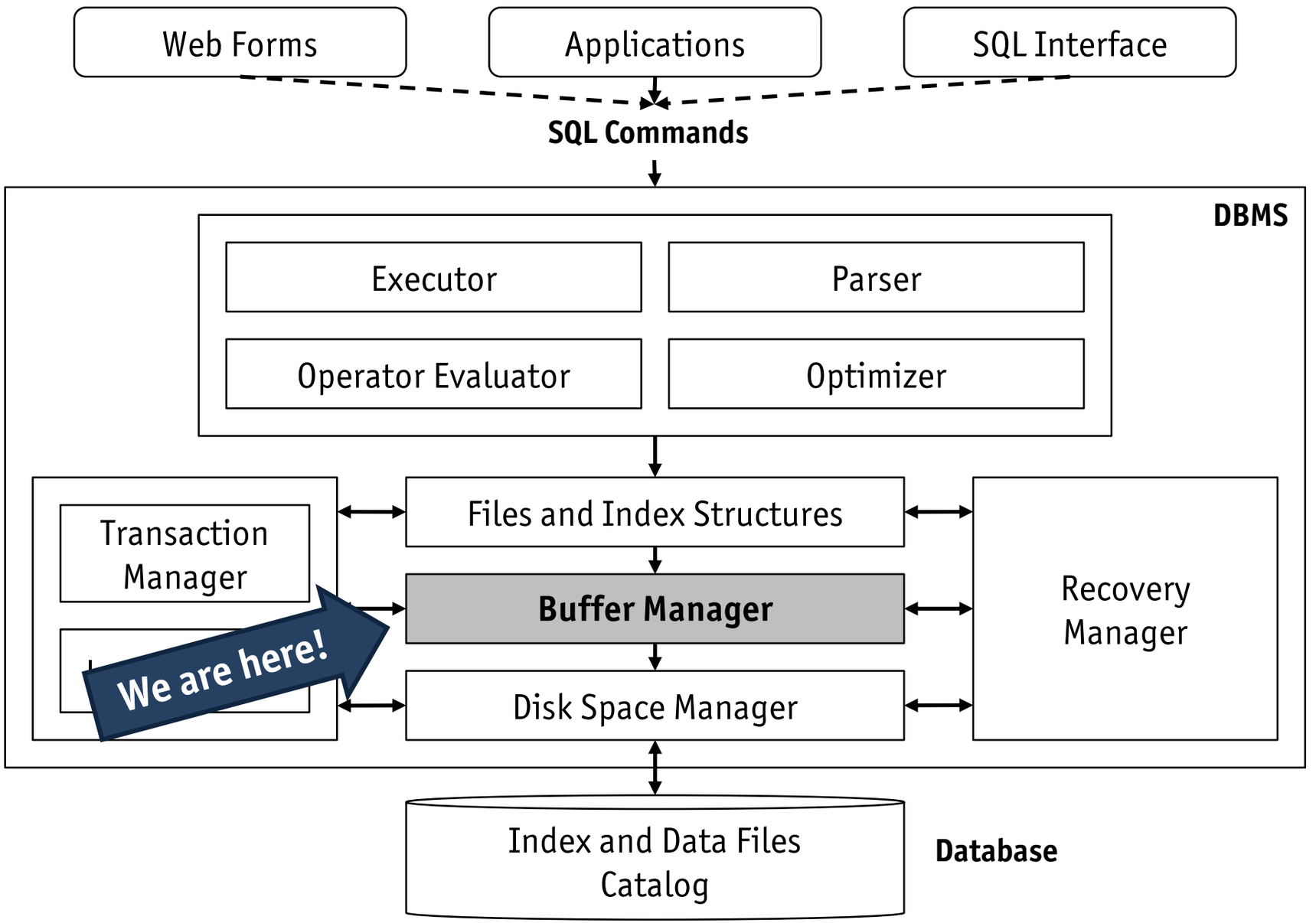


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Buffer Manager

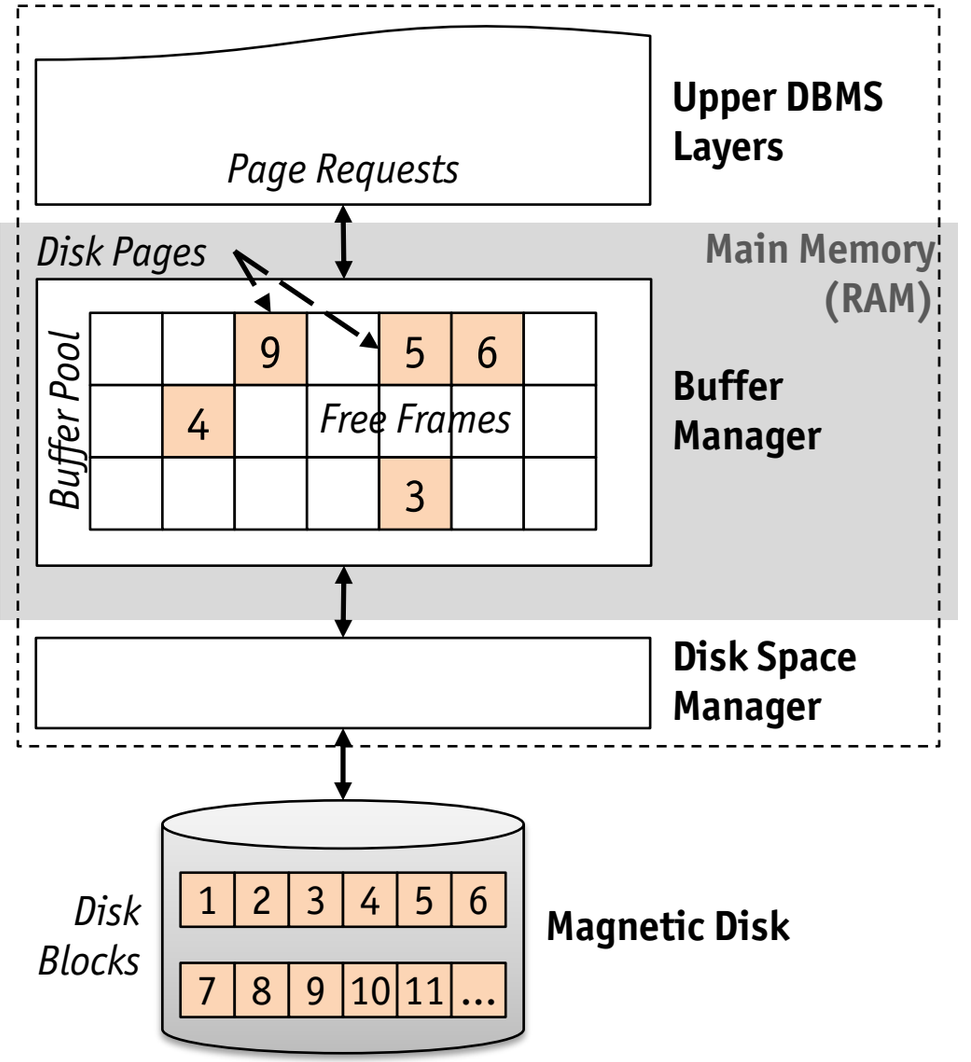
Recall

size of database on secondary storage
≫
size of available primary memory to hold user data

- To scan the entire pages of a 50 GB table (`SELECT * FROM ...`), the DBMS needs to
  - **bring pages into memory** as they are needed for processing
  - **overwrite** (replace) **pages** when they become obsolete and new pages are required
- The **buffer manager** mediates between external storage and main memory

Buffer Pool

- **Buffer pool** is a designated main memory area managed by the buffer manager
  - organized as a collection of **frames** (empty slots)
  - pages brought into memory and are loaded into frames as needed
  - a **replacement policy** decides, which page to evict when the buffer is full



Buffer Manager Interface

- 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
  - if necessary, load page into memory and mark page as clean (¬*dirty*)
  - return a reference to the frame containing page number *pageNo*
- **`unpin`**`(`*pageNo*, *dirty*`)`
  - release page number *pageNo*, making it a candidate for eviction
  - if page was modified, *dirty* must be set to **`true`**

Why do we need the *dirty* bit?

Slide Credit: Torsten Grust, University of Tübingen, Germany

Proper Nesting of `pin()` and `unpin()`

A read-only page operation

```
a ← pin(p);
{ ...
  read data on page at
  memory address a
  ... }
unpin(p, false);
```

A read/write page operation

```
a ← pin(p);
{ ...
  read and modify data (records)
  on page at memory address a
  ... }
unpin(p, true);
```

- All database transactions are required to properly "bracket" page operations using **`pin()`** and **`unpin()`** calls
- Proper bracketing enables the system keeps a count of active users (e.g., transactions) accessing a page (**`pinCount`**)

Writing Pages Back to Disk

- Pages are written back to disk when evicted from buffer pool
  - "clean" victim pages are not written back to disk
  - call to `unpin()` does not trigger any I/O operations, even if the `pinCount` of the page becomes `0`
  - pages with `pinCount = 0` are simply a suitable **victim** for eviction

Digression: Transaction Management

Wait! Stop! This makes no sense... If pages are written only when they are evicted from the buffer pool, then how can we ensure proper transaction management, i.e., guarantee durability?

A buffer manager typically offers at least one or more interface calls (e.g., `flushPage(`*p*`)`) to **force** a page *p* (synchronously) back to disk

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(page in v) then
        write page in v to disk
    read page pageNo from disk into frame v
    pinCount(pageNo) ← 1
    dirty(pageNo) ← false
    return address of frame v
```


Slide Credit: Torsten Grust, University of Tübingen, Germany

Implementation of `unpin()`

Function `unpin` (*pageNo, dirty*)

`pinCount(`*pageNo*`)` ← `pinCount(`*pageNo*`) - 1`
`dirty(`*pageNo*`)` ← `dirty(`*pageNo*`)` ∨ *dirty*

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

Slide Credit: Torsten Grust, University of Tübingen, Germany

Transaction Management (oh no, not again...)

Digression: Conflicting concurrent writes to a block

Assumptions

1. the same page *p* is requested by **more than one** transaction, i.e., the `pinCount(`*p*`)` > 1
2. those transactions perform **conflicting writes** on *p*

Conflicts of this kind are resolved by the system's **concurrency control**, a layer on top of the buffer manager. For more information, see courses "Database Systems" (INF-12040) and "Transactional Information Systems" (INF-11610).

↳ The buffer manager can assume that everything is in order whenever it receives an `unpin(`*p*`, true)` call.

Slide Credit: Torsten Grust, University of Tübingen, Germany

Two Strategic Questions

1. **Buffer Allocation Problem**: How much precious buffer space should be allocated to each active transaction?
   - **static** assignment
   - **dynamic** assignment
2. **Page Replacement Problem**: Which page will be replaced when a new request arrives and the buffer pool is full?
   - decide without knowledge of reference pattern
   - presume knowledge of (expected) reference pattern

- Additional complexity is introduced when we take into account that DBMS may manage "segments" of different page sizes
  - **one buffer pool**: good space utilization, but fragmentation problem
  - **multiple buffer pools**: no fragmentation, but worse utilization, global allocation/replacement strategy may get complicated
- A possible solution is to support set-oriented `pin(`{*p*}`)` calls

Buffer Allocation Policies

- **Problem:** allocate parts of buffer pool to each transaction (TX) or let replacement policy decide who gets how much space?
- **Local** policies…
  - allocate buffer frames to a specific TX **without** taking the reference behavior of concurrent TX into account
  - have to be supplemented with a mechanism for handling the allocation of buffer frames for shared pages
- **Global** policies…
  - consider **not only** the reference pattern of the transaction currently executing, **but also** the reference behavior of all other transactions
  - base their allocation decision on data obtained from all transactions

Local vs. Global Policies

- **Properties** of a local policy
  - ⊕ one TX **cannot hurt** other transactions
  - ⊕ all TX are treated **equally**
  - ⊖ possibly, bad **overall** utilization of buffer space
  - ⊖ some TX may occupy vast amounts of buffer space with "old" pages, while others suffer from too little space (**"internal page thrashing"**)
- **Problem** with a global policy *(assume TX reading a huge relation sequentially)*
  - all page accesses are references to **newly loaded pages**
  - hence, almost **all other pages** are likely to be replaced (following a standard replacement policy)
  - other TX cannot proceed without loading their pages again (**"external page thrashing"**)

Buffer Allocation Policies

- **Global** one buffer pool for all transactions
- **Local** use various information (e.g., catalog, index, data, ...)
- **Local** each TX gets a certain fraction of the buffer pool
  - **static partitioning** *assign buffer budget once for each TX*
  - **dynamic partitioning** *adjust buffer budget of TX according to its past reference pattern or some kind of semantic information*
- It is also possible to apply **mixed** policies
  - have **different** buffer pools with **different** approaches
  - mixed policies **complicate** matters significantly!

Dynamic Buffer Allocation Policies

- **Local LRU** *(cf. LRU replacement policy, later)*
  - keep a separate **LRU-stack** for each active TX
  - keep a global **free list** for pages not pinned by any TX

  Strategy

  1. replace a page from the free list
  2. replace a page from the LRU-stack of the requesting TX
  3. replace a page from the TX with the largest LRU-stack
- **Working Set Model** (cf. virtual memory management of OS)

  goal: avoid thrashing by allocating "just enough" buffers to each TX

  approach: observe number of different page requests by each TX in a certain interval of time (window size τ)
  - deduce "optimal" buffer budget from this observation
  - allocate buffer budgets according to ratio between those optimal sizes

Buffer Replacement Policies

- Choice of **buffer replacement policy** (victim frame selection) can considerably affect DBMS performance
- Large number of policies in OS and DBMS

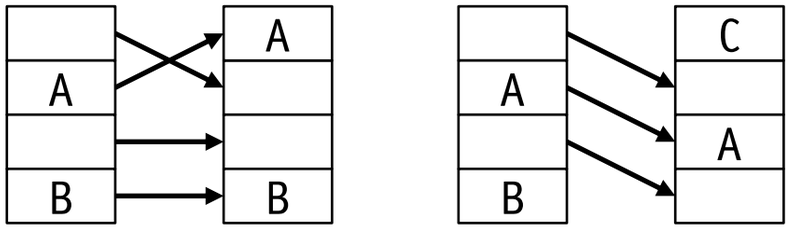
Criteria		Age of page in buffer		
		no	since last reference	total age
References	none	Random		FIFO
	last		LRU CLOCK GCLOCK(V1)	
	all	LFU	GCLOCK(V2) DGCLOCK	LRD(V1)
			LRD(V2)	

Criteria matrix for victim selection

Typical Buffer Replacement Policies

- DBMS typically implement more than one replacement policy
  - **FIFO** ("first in, first out")
  - **LRU** ("least recently used"): evicts the page whose latest **`unpin()`** is longest ago
  - **LRU-*k*:** like LRU, but evicts the *k* latest **`unpin()`** call, not just the latest
  - **MRU** ("most recently used"): evicts the page that has been unpinned most recently
  - **LFU** ("least frequently used")
  - **LRD** ("least reference density")
  - **CLOCK** ("second chance"): simulates LRU with less overhead (no LRU queue reorganization on every frame reference)
  - **GCLOCK** ("generalized clock")
  - **WS, HS** ("working set", "hot set")
  - **Random:** evicts a random page

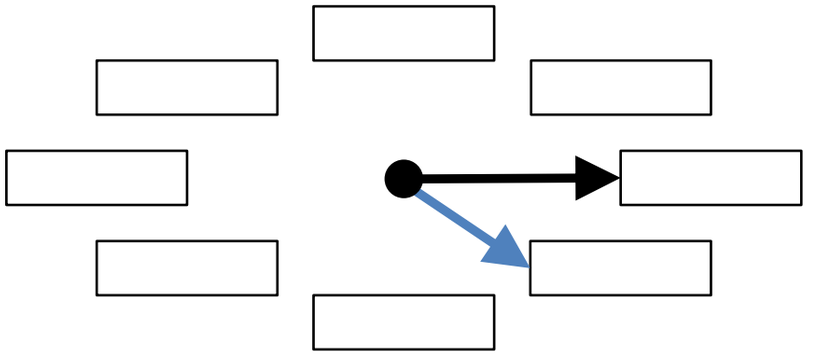
LRU



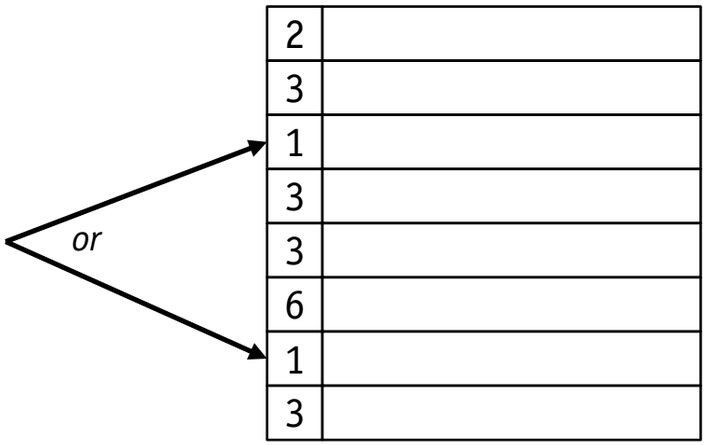
Reference to A in buffer

Reference to C not in buffer

FIFO



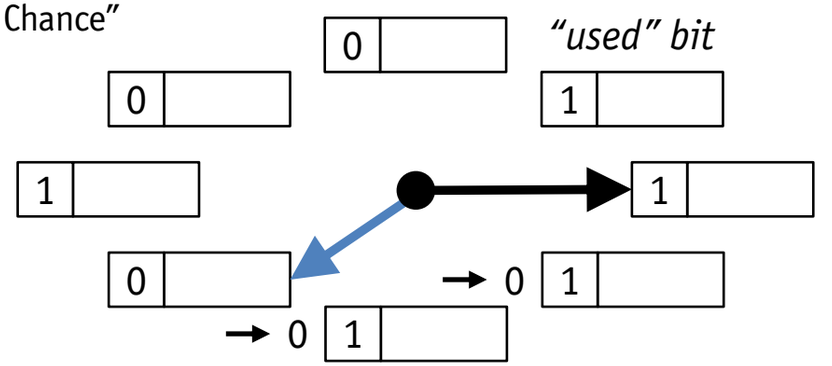
LFU



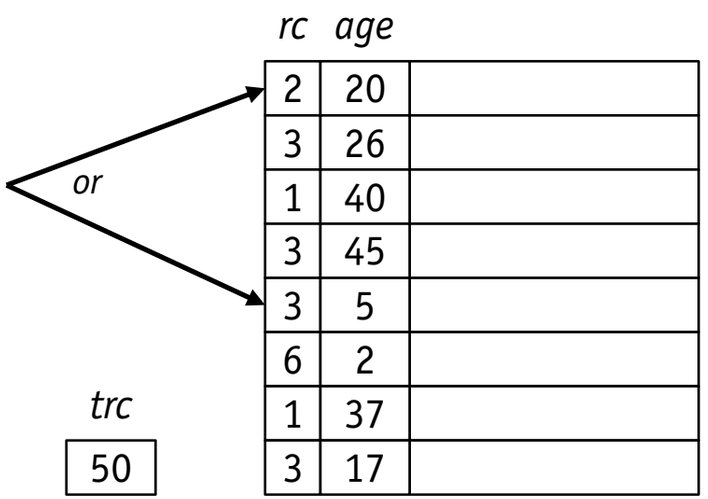
Victim page *or*

CLOCK

"Second Chance"

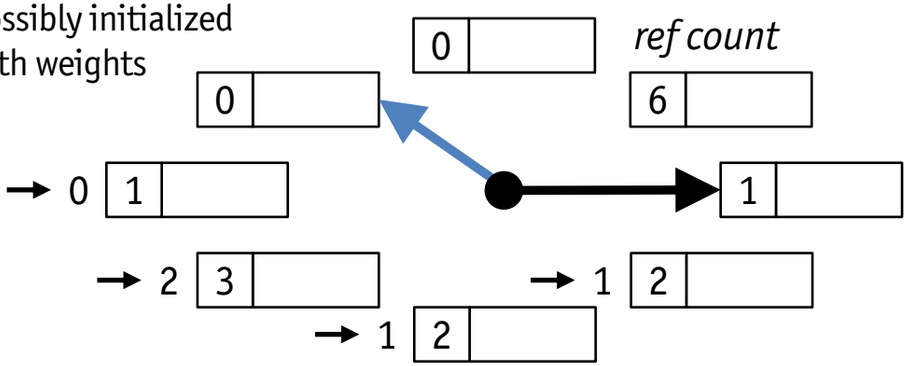


LRD(V1)



GCLOCK

possibly initialized with weights



Details of LRU and CLOCK Policy

Example: LRU Buffer Replacement Policy

1. keep a **queue** (often described as a **stack**) of pointers to frames
2. in `unpin(`*pageNo*, *dirty*`)`, append *p* to the **tail** of queue, if `pinCount(`*pageNo*`)` is decremented to `0`
3. to find next victim, search through the queue from its **head** and find the first page *p* with `pinCount(`pageNo`) = 0`

Example: CLOCK Buffer Replacement Policy

1. number the *N* buffer frames `0 ... N-1`, initialize `current` ← `0`, maintain a bit array `referenced[0 ... N-1]`, which is initialized to all `0`
2. in `pin(`*pageNo*`)`, do `referenced[`*pageNo*`]` ← `1`
3. to find the next victim, consider page `current`: if `pinCount(current) = 0` and `referenced[current] = 0`, `current` is the victim; otherwise, `referenced[current]` ← `0`, `current` ← (`current + 1`) *mod* `N`; repeat 3.

Heuristic Policies Can Fail

- These buffer replacement policies are **heuristics** only and can fail miserably in certain scenarios

Example: A Challenge for LRU

A number of transactions want to scan the same sequence of pages (as for example a repeated `SELECT * FROM R`). Assume a buffer pool capacity of 10 pages.

1. Let the size of relation *R* be 10 or less page.
   How many I/O operations do you expect?
2. Let the size of the relation *R* be 11 pages.
   What about the number of I/O operations in this case?

Details of LRD

- Record the following three parameters
  - $trc(t)$ total reference count of transaction t
  - $age(p)$ value of $trc(t)$ at the time of loading p into buffer
  - $rc(p)$ reference count of page p
- Update these parameters during a transaction's page references (`pin(`*pageNo*`)` calls)
- Compute **mean reference density** of a page p at time t as

$$rd(p,t) := \frac{rc(p)}{trc(t)-age(p)}, \text{ where } trc(t) - rc(p) \geq 1$$

- **Strategy** for victim selection
  - chose page with least reference density $rd(p,t)$
  - many variants exist, e.g., for gradually disregarding old references

Exploiting Semantic Knowledge

- **Background**: query compiler/optimizer already…
  - selects access plan, e.g., sequential scan vs. index
  - estimates number of page I/O operations for cost-based optimization
- **Idea**: use this information to determine query-specific, optimal buffer budget, i.e., a **Hot Set**
- Goals of **Query Hot Set** model
  - optimize overall system throughput
  - avoiding thrashing is the most important goal

Hot Set with Disjoint Page Sets

Mode of Operation

1. only those queries are activated, whose Hot Set buffer budget can be satisfied immediately
2. queries with higher demand have to wait until their budget becomes available
3. within its own buffer budget, each transaction applies a local LRU policy

- Properties
  - ⊖ **no sharing** of buffered pages between transactions
  - ⊖ risk of **internal thrashing** when Hot Set estimates are wrong
  - ⊖ queries with large Hot Sets **block** following small queries (or, if bypassing is permitted, many small queries can lead to **starvation** of large queries)

Hot Set with Non-Disjoint Page Set

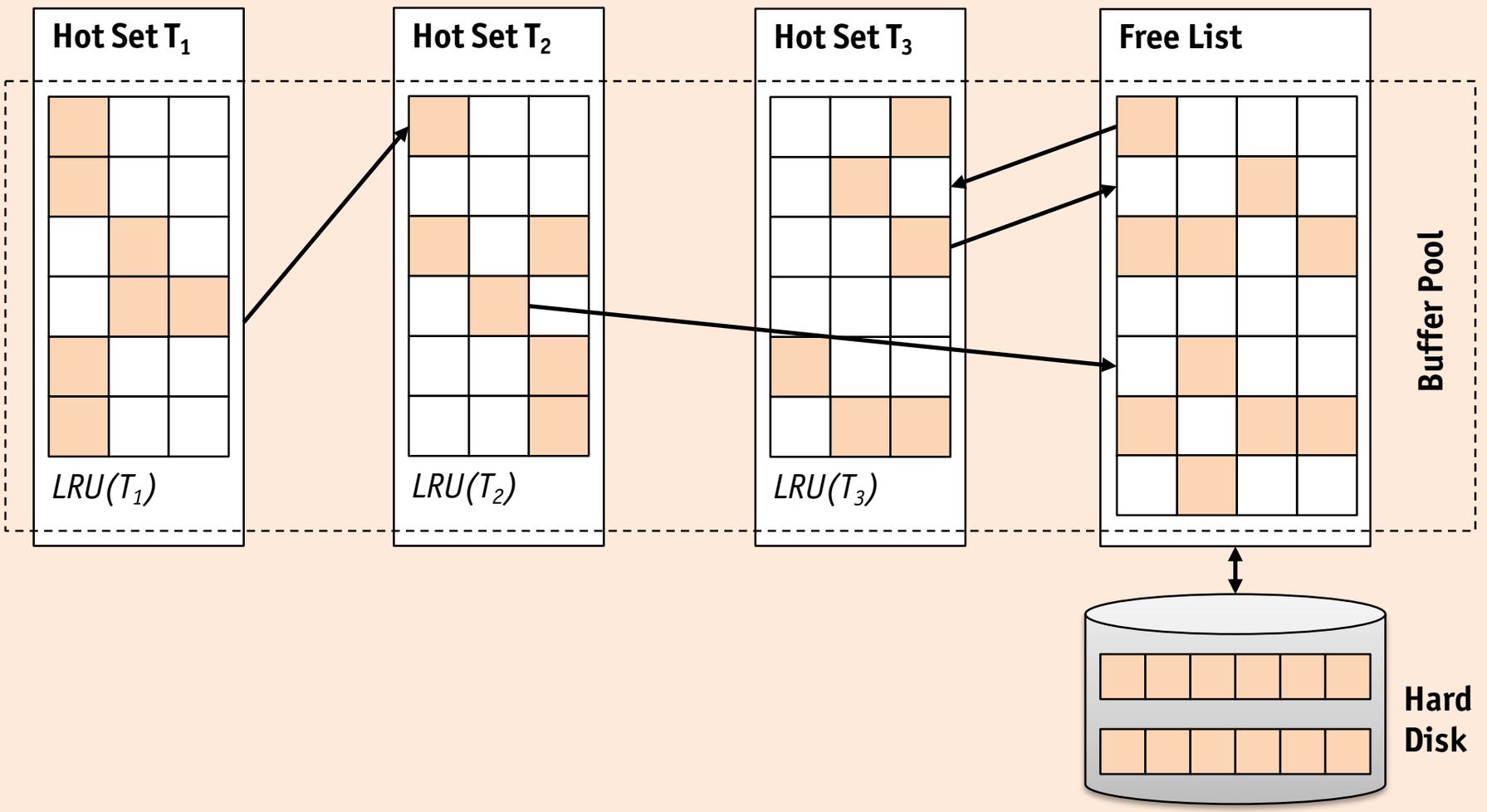
Mode of Operation

1. queries allocate their budget stepwise, up to the size of their Hot Set
2. local LRU stacks are used for replacement
3. request for a page *p*
   i. if found in **own LRU stack**: update LRU stack
   ii. if found in **another transaction's LRU stack**: access page, but do not update the other LRU stack
   iii. if found in **free list**: push page on own LRU stack
4. call to **`unpin`**`(`*pageNo*`)` : push page onto free list stack
5. filling empty buffer frames: taken from the bottom of the free list stack

- As long as a page is in a local LRU stack, it cannot be replaced
- If a page drops out of a local LRU stack, it is pushed onto free list stack
- A page is replaced only if it reaches the bottom of the free list stack before some transaction pins it again

Hot Set with Non-Disjoint Page Set

Mode of Operation



Priority Hints

- **Idea**: with **`unpin(`***pageNo***`)`**, a transaction gives one of two possible indications to the buffer manager
  - **preferred page** managed in a transaction-local partition
  - **ordinary page** managed in a global partition
- **Strategy**: when a page needs to be replaced…
  1. try to replace an ordinary page from the global partition using LRU
  2. replace a preferred page of the requesting transaction using MRU
- **Advantages**
  - much simpler than Hot Set, but similar performance
  - easy to deal with “too small” partitions

Variant: Fixing and Hating Pages

Operator can **fix** a page if it may be useful in the near future (e.g., *nested-loop join*) or **hate** a page it will not access any time soon (e.g., *pages in a sequential scan*)

Prefetching

- Buffer manager can try to **anticipate** page requests
  - asynchronously read ahead even if only a single page is requested
  - improve performance by overlapping CPU and I/O operations
- Prefetching techniques
  - **prefetch lists: on-demand, asynchronous read-ahead**
    e.g., when traversing the sequence set of an index, during a sequential scan of a relation
  - **heuristic (speculative) prefetching**
    e.g., sequential *n*-block look-ahead (cf. drive or controller buffers in hard disks), semantically determined supersets, index prefetch, ...

Prefetching

The Real World

- **IBM DB2**
  - supports both **sequential** and **list** prefetch (prefetching a list of pages)
  - **default prefetch size** is 32 4 kB pages (user-definable), but for some utilities (e.g., COPY, RUNSTAT) pages up to 64 4 kB are prefetched
  - for small buffer pools (i.e., < 1000 buffers) prefetch adjusted to 8 or 16 pages
  - prefetch size can be defined by user (sometimes it makes sense to prefetch 1000 pages)
- **Oracle 8**
  - uses prefetching for **sequential scan**, retrieving **large objects**, and certain **index scans**
- **Microsoft SQL Server**
  - supports prefetching for **sequential scan** and for scans along the leaf-level of a **B+ tree index**
  - prefetch size can be adjusted during a scan
  - extensive use of asynchronous (speculative) prefetching

Database vs. Operating System

- **Stop!** What you are describing is an **operating system** (OS)!

- Well, yes…
  - disk space management and buffer management very much look like **file management** and **virtual memory** (VM) in an operating system
- But, no…
  - DBMS can predict the **access patterns** of certain operators a lot better than the operating system (prefetching, priority hints, etc.)
  - **concurrency control** is based on protocols that prescribe the order in which pages have to be written back to disk
  - **technical reasons** can make operating systems tools unsuitable for a database (e.g., file size limitation, platform independence)

Double Buffering

- DBMS buffer manager within VM of DBMS server process can **interfere** with OS VM manager
  - **virtual page fault:** page resides in DBMS buffer, but the frame has been swapped out by operating system VM manager
    ↳ **one I/O operation** is necessary that is not visible to the DBMS
  - **buffer fault:** page does not reside in DBMS buffer, but frame is in physical memory
    ↳ regular DBMS page replacement requiring **one I/O operation**
  - **double page fault:** pages does not reside in DBMS buffer and frame has been swapped out of physical memory by operating system VM manager
    ↳ **two I/O operations** necessary: one to bring in the frame (OS) and another one to replace the page in that frame (DBMS)
- DBMS buffer needs to be **memory resident** in OS

Buffer Management in Practice

The Real World

- **IBM DB2**
  - buffers can be partitioned into named pools
  - each database, table, or index can be bound to a pool
  - each pool uses FIFO, LRU or (variant of) Clock buffer replacement policy
  - supports “hating” of pages
- **Oracle 7**
  - maintains a single global buffer pool using LRU
- **Microsoft SQL Server**
  - uses a single buffer pool with Clock replacement
  - supports a “reservation” of pages by queries that require large amount of memory (e.g., queries involving sorting of hashing)

Orientation

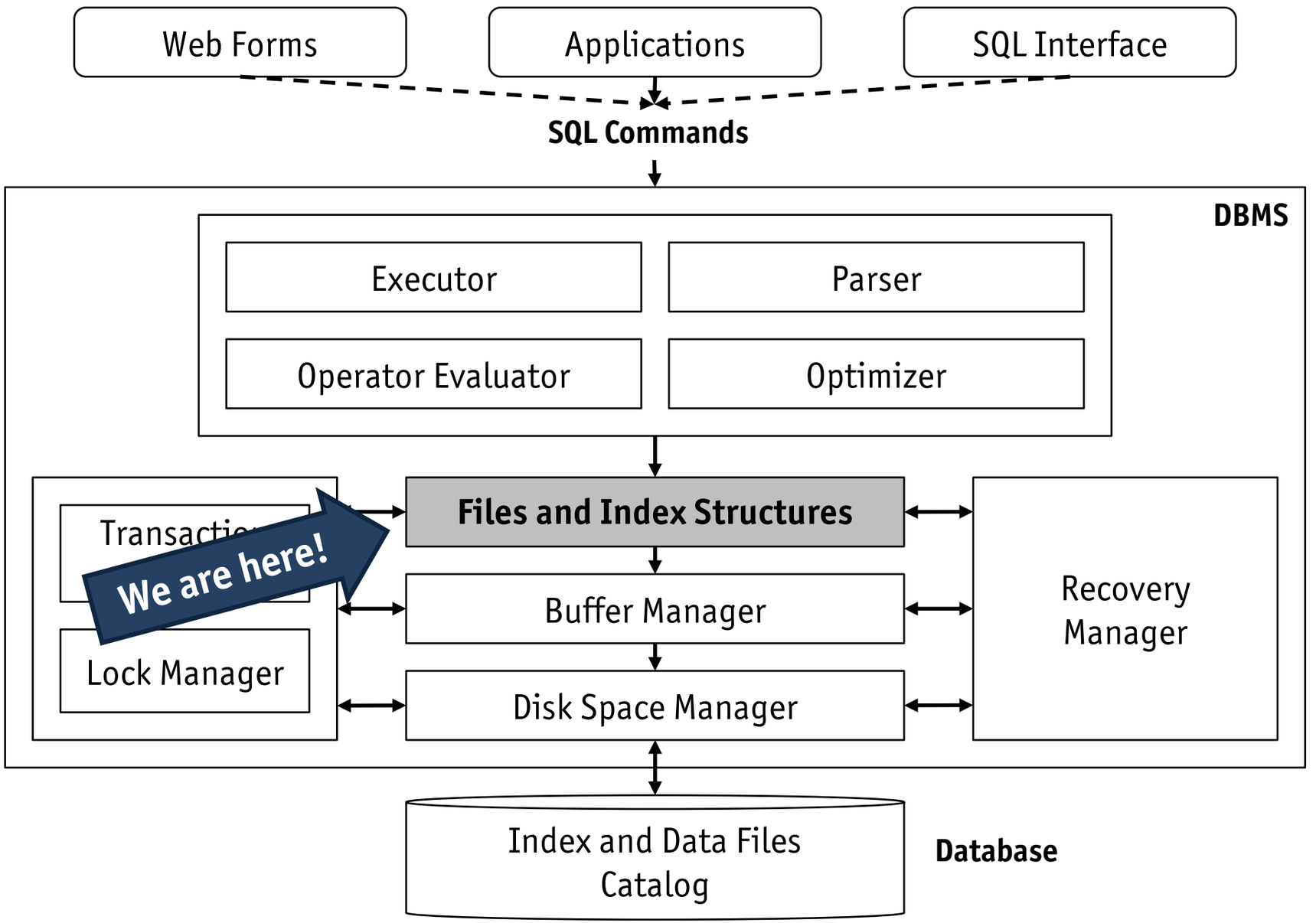
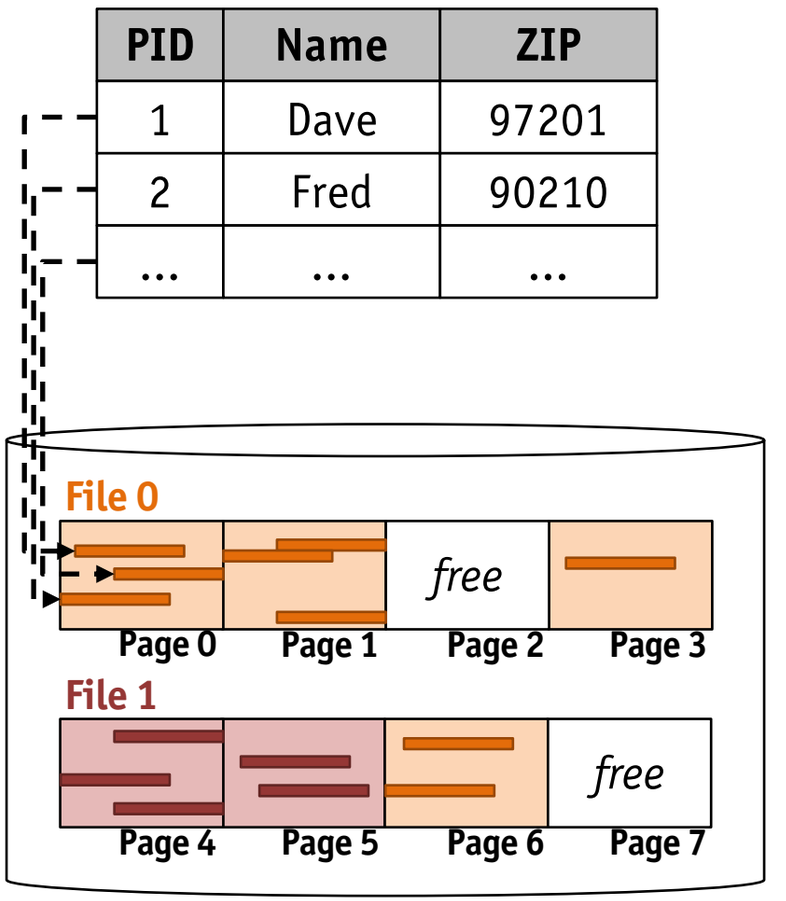


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Database Files

- Focus change
  - **the road so far:** how does a DBMS manage pages?
  - **now:** how does a DBMS use pages to store data?
- On the conceptual level, a **relational** DBMS manages **tables of rows** and **indexes**
- On the physical level, these data structures are implemented as **files of records**
  - each file consists of **one or more pages**
  - each page contains **one or more records**
  - each record corresponds to **one row**



Database Heap Files

- The most important and most simple file structure in a database is the heap file
  - represents an **unordered** collection of records (cf. SQL semantics)
  - each record in a heap file has a **unique record identifier** (*rid*)
- Record ids (*rids*) are used like **record addresses** (or pointers)
  - internally, the heap file structure must be able to **map** a given rid to the page containing the record

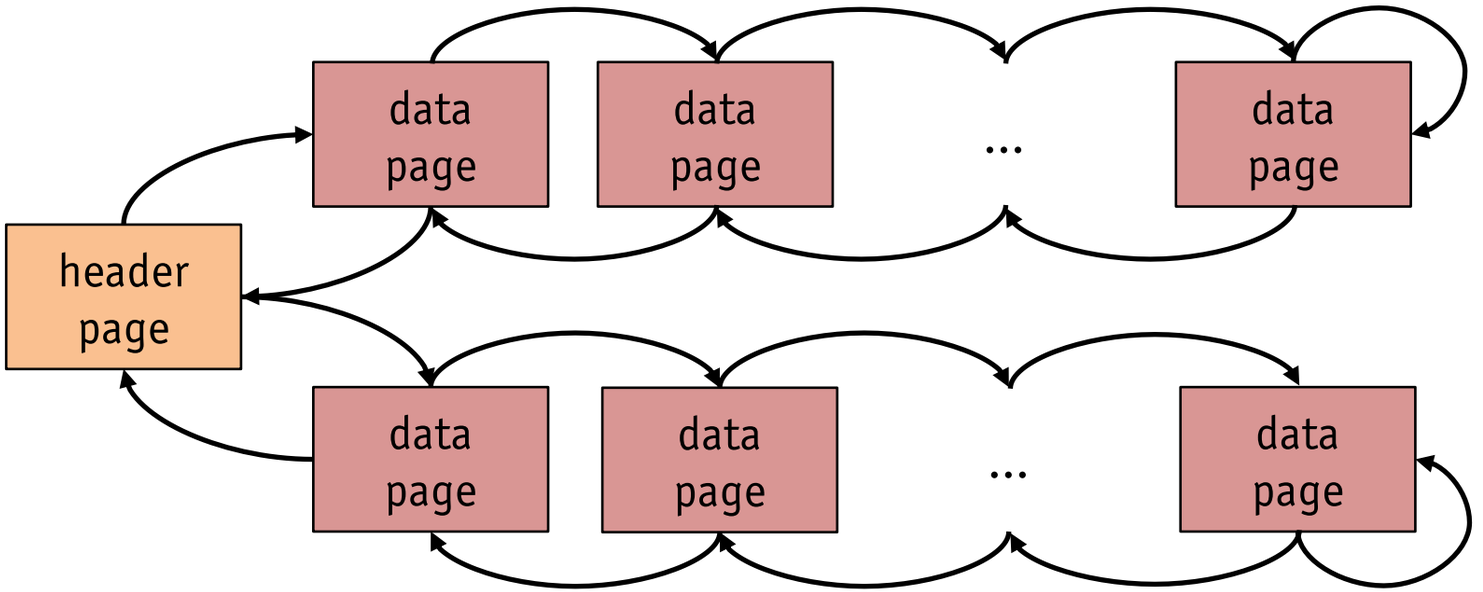
Typical heap file interface

- **create**/**destroy** heap file *f* named *n*: *f* ← `createFile(`*n*`)` and `deleteFile(`*n*`)`
- **insert** record *r* and return its *rid*: *rid* ← `insertRecord(`*f*, *r*`)`
- **delete** a record with a given *rid*: `deleteRecord(`*f*, *rid*`)`
- **get** a record with a given *rid*: *r* ← `getRecord(`*f*, *rid*`)`
- initiate a **sequential scan** over the whole heap file: `openScan(`*f*`)`

Free Space Management

- Implications of the heap file interface
  - to support **`openScan`**`(`*f*`)` , the heap file structure has to **keep track of all pages in the file** *f*
  - to support **`insertRecord`**`(`*f*`,` *r*`)` efficiently, the heap file structure also needs to **keep track of all pages with free space in the file** *f*
- In this course, we will look at two simple structures that can offer this support
  - (doubly) **linked list** of pages
  - **directory** of pages

Linked List of Pages



Operation $f \leftarrow$ `createFile(`n`)`

1. DBMS allocates a free page (called file **header page**) and stores an entry <*n*, *header page*> to a known location on disk
2. header page is initialized to point to two doubly linked list of pages: one containing **full pages** and one containing **pages with free space**
3. initially, both lists are **empty**

Linked List of Pages

Operation *rid* ← `insertRecord` (*f*, *r*)

1. try to **find a page** *p* in the free list with space > |*r*|
2. should this fail ask the disk space manager **to allocate a new page** *p*
3. record *r* is **written** to page *p*
4. since generally |*r*| ≪ |*p*|, *p* will belong to the **list of pages with free space**
5. a **unique** *rid* for *r* is computed and returned to the caller

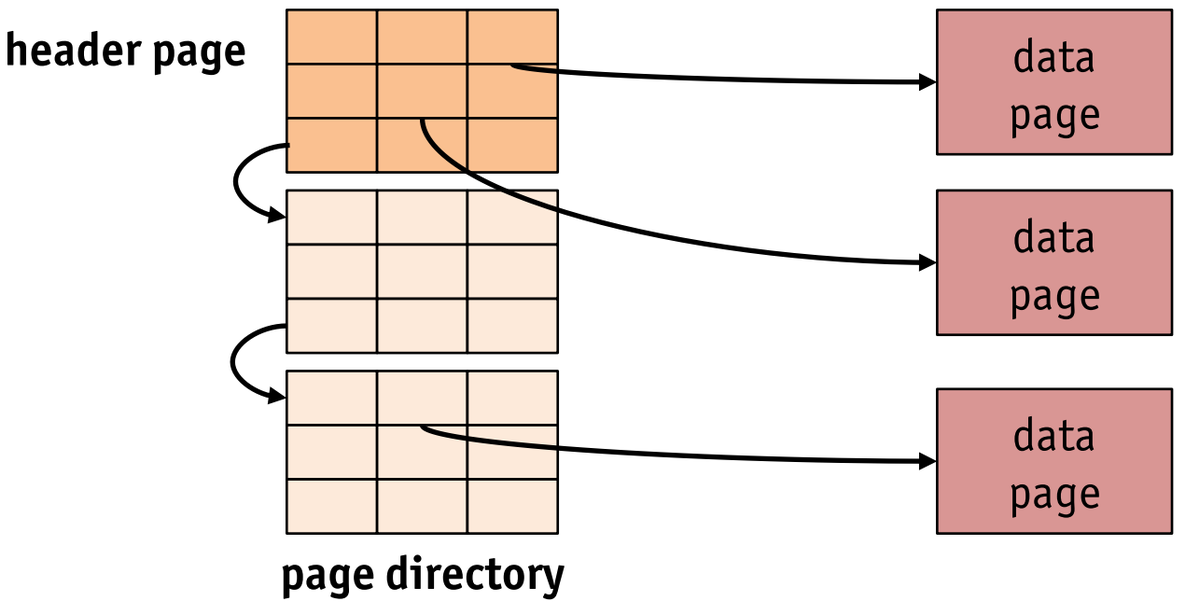
Operation `deleteRecord` (*f*, *r*)

1. may result in **moving the containing page** from the list of full pages to the list of pages with free space
2. may even lead to **page deallocation** if the page is completely free after deletion

Operation `openScan` (*f*)

1. **both** page lists have to be traversed

Directory of Pages



- **Header page** contains first page of a chain of **directory pages**
  - each entry in a directory page identifies a page of the file
  - |page directory| ≪ |data pages|
- Free space management is also done through the directory
  - space vs. accuracy **trade-off**: from *open/closed* flag to exact information
  - for example, entries could be of the form <*page addr p*, *nfree*>, where *nfree* indicates **actual amount of free space** (e.g., in bytes) on page *p*

Free Space Management

- Keeping **exact** counter value (e.g., *nfree*) during updates may produce a performance bottleneck in multi-user operations
  - each transaction, whose update changes the amount of available free space, needs to update this meta-information in the directory
  - locking (or some other form of synchronization) needs to be applied to avoid lost updates
  - frequent updates of the same record lead to “hot data item”, introducing lock-wait queues and thus reducing degree of parallelism
- Keep **fuzzy information** on available free space in directory, e.g., $\lfloor nfree/8 \rfloor$ (units of 8 bytes or some other granularity)
  - directory only needs to be updates for “large” changes in the available free space on a page
  - page itself contains the exact information (of course)

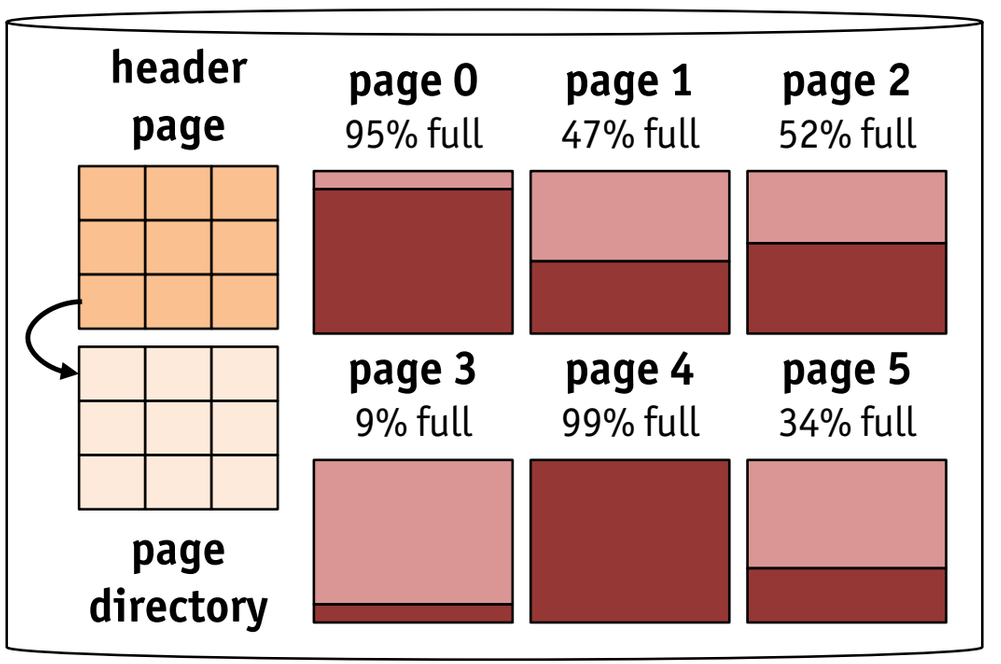
Record Insertion Strategies

- **Append Only**
  - always insert into the last page
  - otherwise, create a new page
- **Best Fit**
  - search from beginning and pick the page with the amount of free space that most closely matches the required space
  - reduces fragmentation, but requires search the entire list or directory
- **First Fit**
  - search from beginning and pick the first page with sufficient space
  - since first pages fill up quickly, system may waste a lot of search effort in these pages later on
- **Next Fit**
  - maintain cursor and continue searching where last search ended

Free Space Witnesses

- Accelerate search by remembering **witness pages**
  - classify pages into **buckets**
  - for each bucket, remember a **witness page**, i.e., any page that falls into that bucket
  - only perform standard best/first/next fit search, if no witness page is recorded for the specific bucket
  - populate witness information, e.g., as a side effect when searching

Bucket#	%Full	Witness
0	75%□100%	0
1	50%□75%	2
2	25%□50%	5
3	0%□25%	3



Linked List vs. Directory

I/O operations and free space management

For a file of 10,000 pages, give lower and upper bound for the number of I/O operations during an `insertRecord(`*f*, *r*`)` call for a heap file organized using

1. a **linked list** of pages

2. a **directory** of pages (1,000 directory entries/page)

Linked List vs. Directory

- Linked list of free pages
  - ⊕ easy to implement
  - ⊖ most pages will end up in the list of pages with free space
  - ⊖ might have to search many pages to place a (large) record
- Directory of free pages
  - ⊕ free space management more efficient
  - ⊖ memory overhead to host the page directory

Page Formats

- Locating the page containing a given *rid* is not the whole story
  - **internal structure of pages** plays a crucial role
  - we think of a page as sequence of **slots**, each of which contains a record

Generating sensible record ids (*rids*)

Given that rids are used like record addresses, what would be a feasible *rid* generation method?

Record Ids in Commercial Systems

The Real World

- **IBM DB2**, **Oracle 8**, and **Microsoft SQL Server**
  - record id is implemented as a **page id** and a **slot number**

Fixed-Length Records

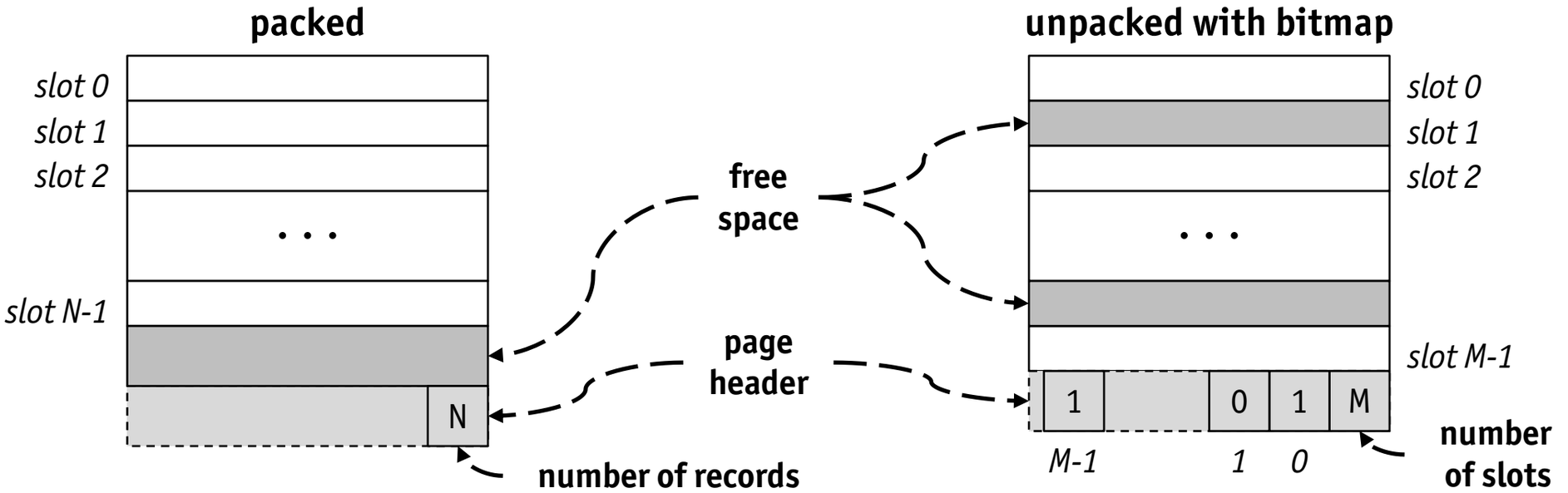
- All records on the page (in the file) are the **same size** *s*
  - **`getRecord`** **(***f*, <*p*, *n*>**)** : given the *rid* <*p*, *n*> we know that the record is to be found at (byte) offset *n* × *s* on page *p*
  - **`deleteRecord`** **(***f*, <*p*, *n*>**)** : copy the bytes of the last occupied slot on page *p* to offset *n* × *s*, mark slot as free (page is **packed**, i.e., all occupied slots appear together at the start if the page)
  - **`insertRecord`** **(***f*, *r***)** : find a page *p* with free space >= *s* (see previous discussion) and copy *r* to the first free slot on *p*, then mark the slot as occupied

Packed pages and deletions

One problem with packed pages remains as calling **`deleteRecord`** **(***f*, <*p*, *n*>**)** modifies the *rid* of a **different record** <*p*, *n′*> on the same page

↳ if any external references to this record exist, we need to chase through the whole database and **update** *rid* references <*p*, *n′*> → <*p*, *n*>… **Bad!**

Free Slot Bitmap



- Avoid record copying and therefore rid modifications
  - **`deleteRecord`**`(`*f*, <*p*, *n*>`)` simply needs to set bit *n* in bitmap to 0
  - **no other** *rids* affected

Page header or trailer?

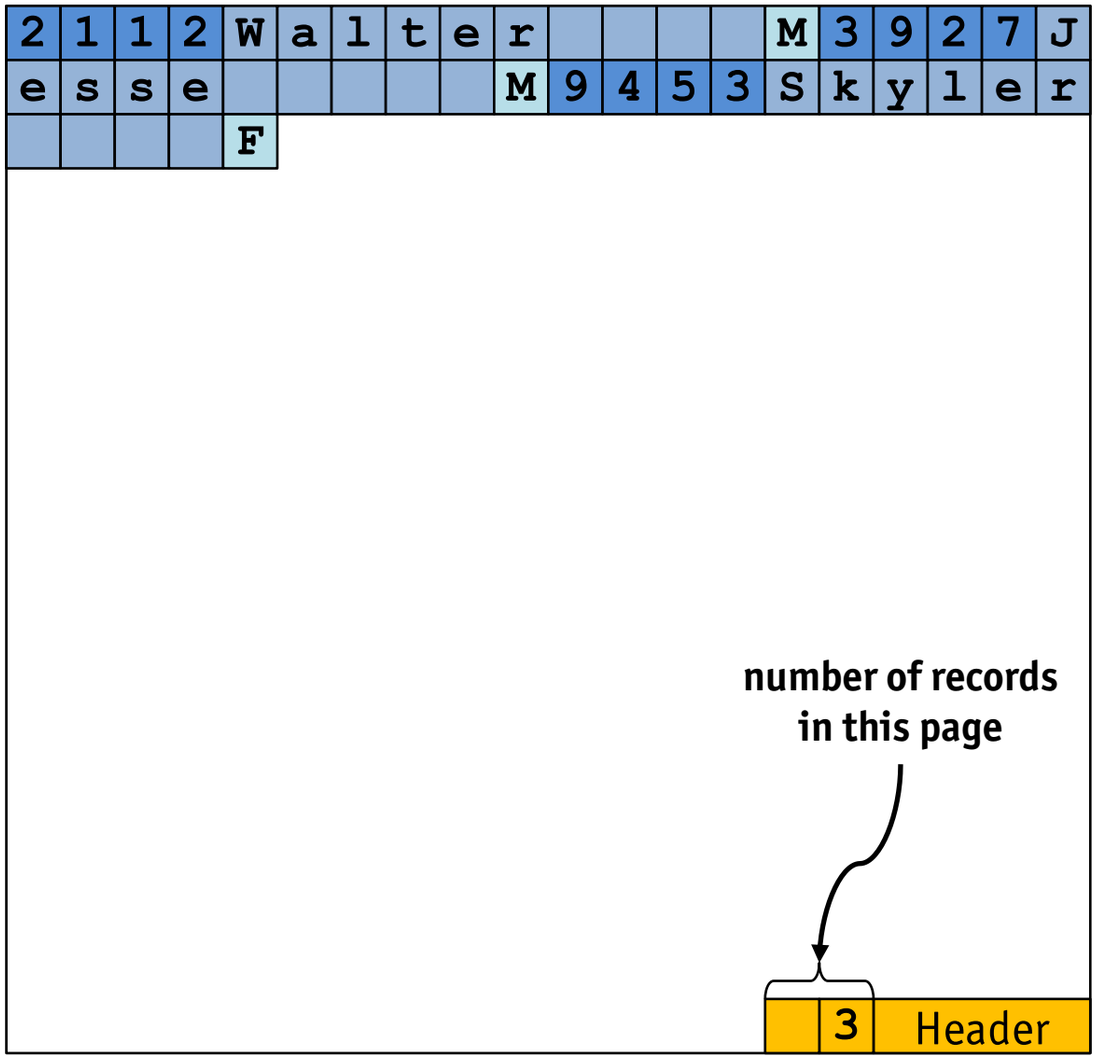
In both page organization schemes, we have positioned the page header **at the end of the page**. How would you justify this design decision?

Inserting Fixed-Length Records

Creating a simple table in SQL

```
CREATE TABLE Persons (
  ID    INTEGER,
  NAME  CHAR(10),
  SEX   CHAR(1)
);
```

ID	NAME	SEX
2112	Walter____	M
3927	Jesse_____	M
9453	Skyler____	F

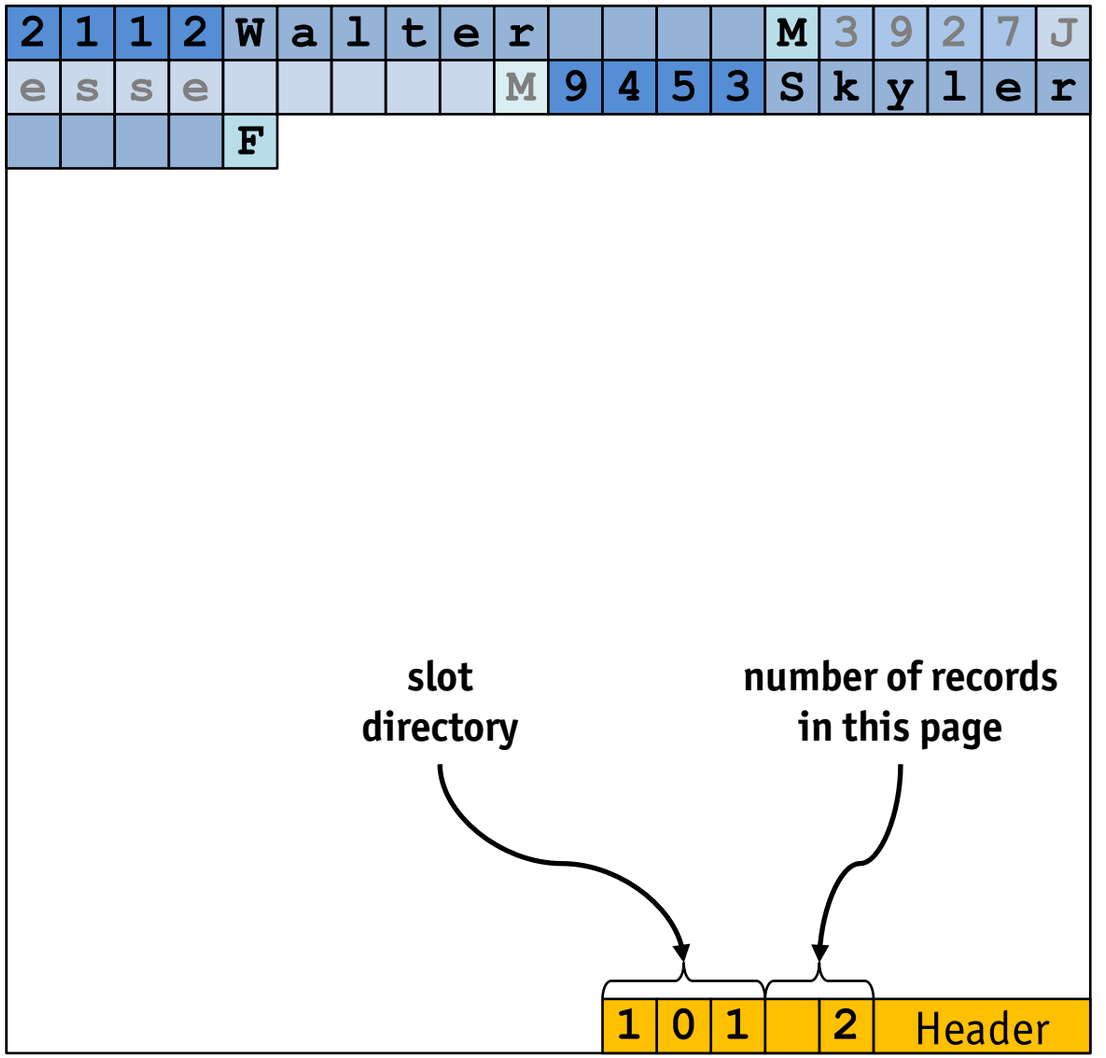


Deleting Fixed-Sized Records

Creating a simple table in SQL

```
CREATE TABLE Persons (
  ID    INTEGER,
  NAME  CHAR(10),
  SEX   CHAR(1)
);
```

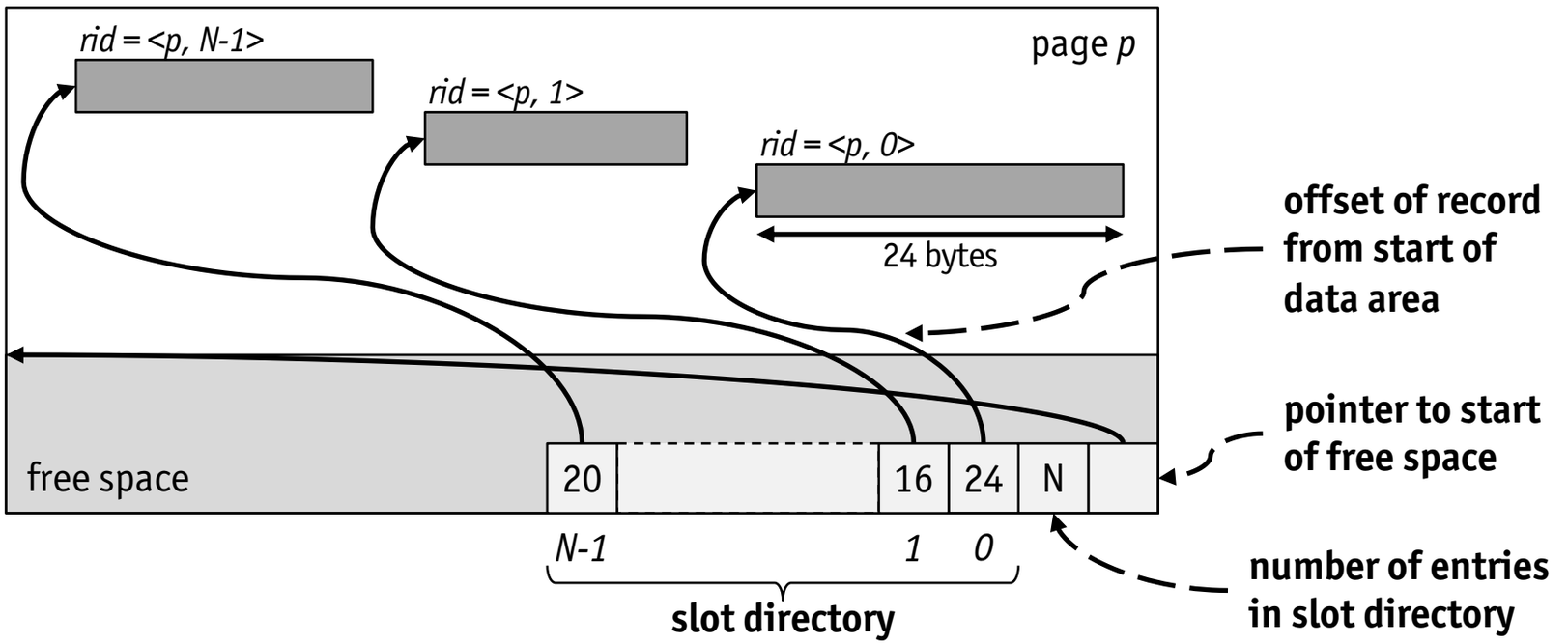
ID	NAME	SEX
2112	Walter____	M
~~3927~~	~~Jesse_____~~	~~M~~
9453	Skyler____	F



Variable-Length Records

- If records on a page are of varying size (e.g., SQL data type **`VARCHAR(`***n*`)`**), there can be **page fragmentation**
  - **`insertRecord(`***f, r*`)`** : needs to find an empty slot of size $\geq |r|$, such that the wasted space is **minimal**
  - compacting the remaining records to maintain a **contiguous area of free space** gets rid of holes produced by **`deleteRecord(`***f, rid*`)`**
- A solution is to maintain a **slot directory** on each page
  - similar free space management in a database heap file
  - contains entries <*offset, length*>, where *offset* is measured in bytes from the start of data page
  - **`deleteRecord(`***f*, <*p*, *n*>`)`** : set offset of directory entry *n* to –1 to indicate that entry can be reused subsequent **`insertRecord(`***f, r*`)`** calls, which hit page *p*

Variable-Length Records



Compaction of slot directory

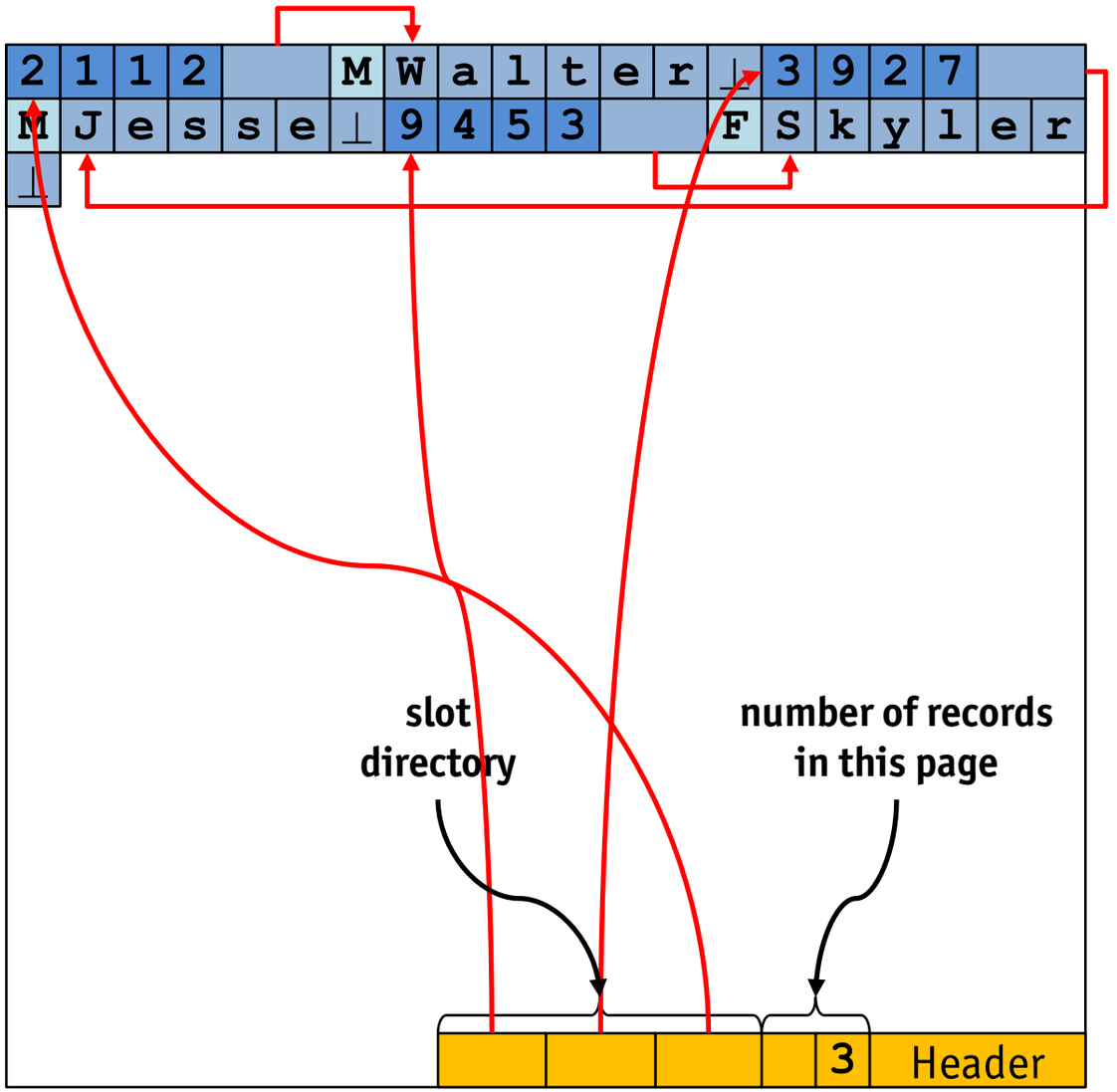
- **not allowed** in this scheme as this would again modify the rids of all records <*p*, *n'*>, for *n'* > *m*
- if insertion are much more common than deletions, the directory size will nevertheless be **close to the actual number of records** stored on the page
- **record compaction** (defragmentation) is performed, of course

Inserting Variable-Length Records

Creating a simple table in SQL

```
CREATE TABLE Persons (
  ID    INTEGER,
  NAME  VARCHAR(10),
  SEX   CHAR(1)
);
```

ID	NAME	SEX
2112	Walter⊥	M
3927	Jesse⊥	M
9453	Skyler⊥	F

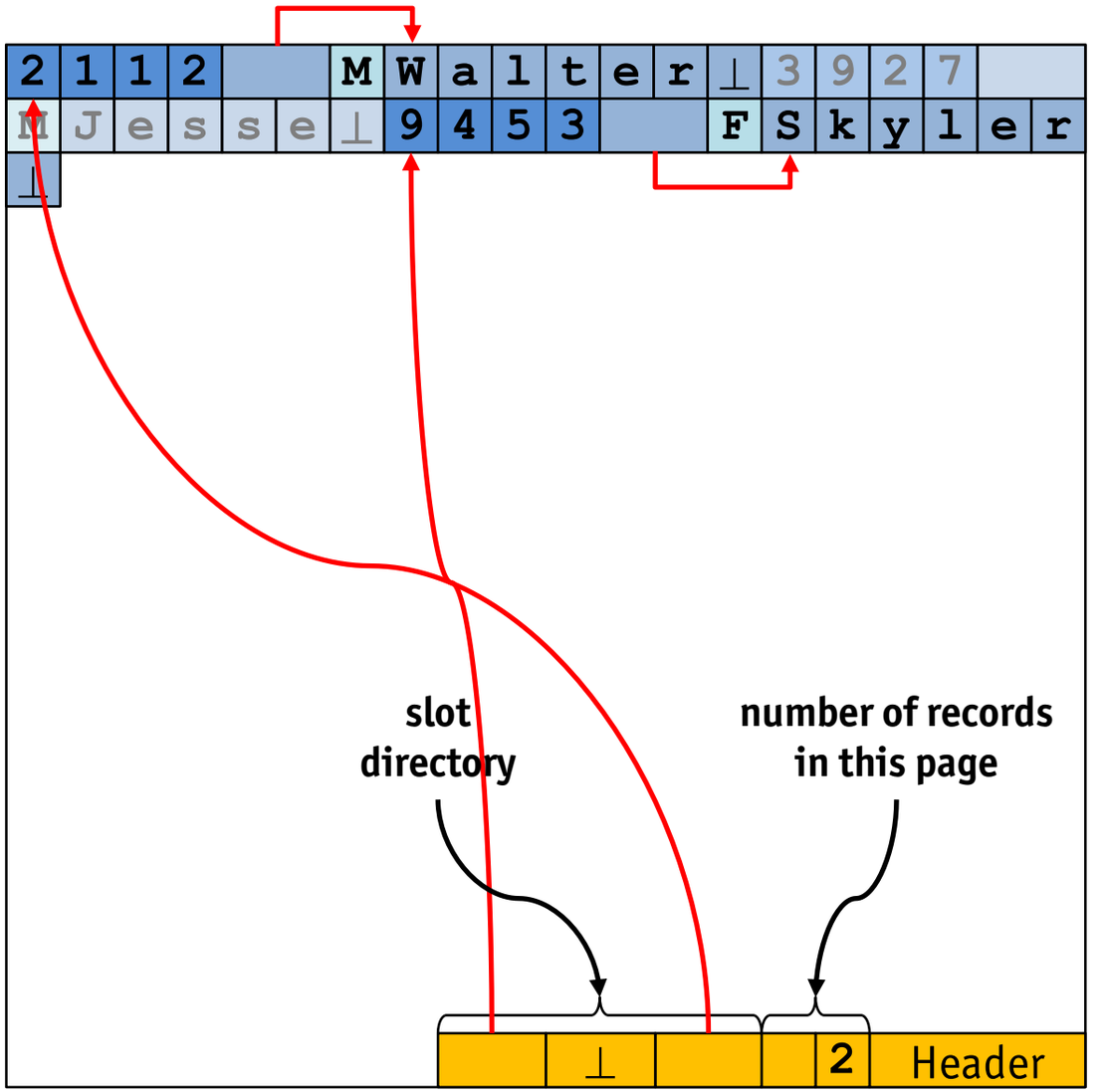


Deleting Variable-Length Records

Creating a simple table in SQL

```
CREATE TABLE Persons (
  ID    INTEGER,
  NAME  VARCHAR(10),
  SEX   CHAR(1)
);
```

ID	NAME	SEX
2112	Walter⊥	M
~~3927~~	~~Jesse⊥~~	~~M~~
9453	Skyler⊥	F

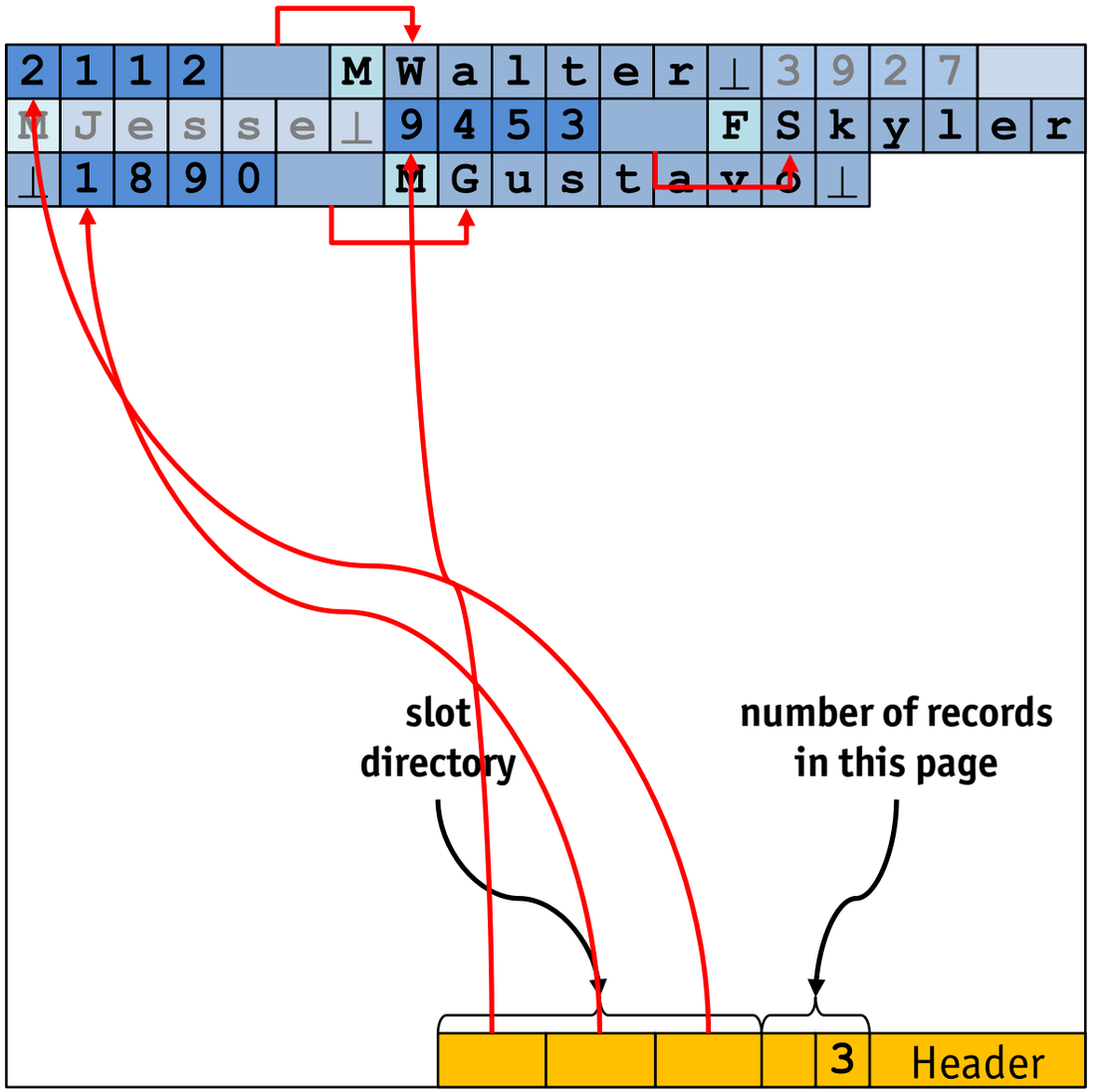


Inserting Variable-Length Records

Creating a simple table in SQL

```
CREATE TABLE Persons (
  ID    INTEGER,
  NAME  VARCHAR(10),
  SEX   CHAR(1)
);
```

ID	NAME	SEX
2112	Walter⊥	M
~~3927~~	~~Jesse⊥~~	~~M~~
9453	Skyler⊥	F
1890	Gustavo⊥	M



Record Formats

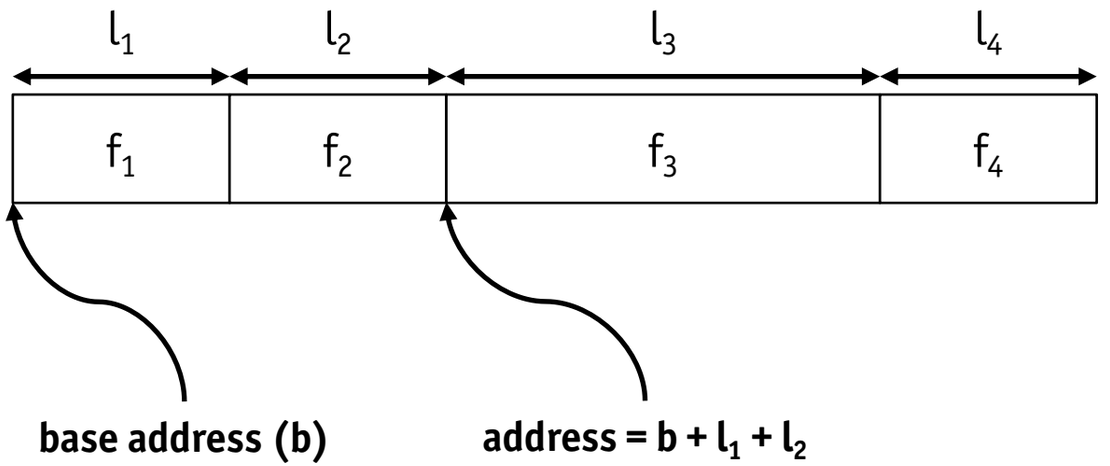
- Another focus change
  - **the road so far:** accessing records in a page
  - **now:** accessing fields (conceptually, attributes) in a record
- **Recall:** attributes are considered to be atomic in an RDBMS
- Record field type defines their **length**
  - **fixed-length**, e.g., `INTEGER`, `BIGINT`, `CHAR(`*n*`)`, `DATE`, ...
  - **variable-length**, e.g., `VARCHAR(`*n*`)`, `CLOB(`*n*`)`, ...
- On `CREATE TABLE`
  - DBMS computes field size information for the records of a file
  - thin information is then recorded in the **system catalog**

Record Field Sizes

The Real World

- **Microsoft SQL Server**
  - **exact numerics**
    `BIGINT` (8 bytes), **`INT`** (4 bytes), **`SMALLINT`** (2 bytes), **`TINYINT`** (1 byte)
  - **approximate numerics**
    `FLOAT(`*n***`)`** (4-8 bytes), **`REAL`** (4 bytes)
  - **date and time**
    `DATE` (3 bytes), **`TIME`** (5 bytes), **`DATETIME`** (8 bytes)
  - **character strings**
    `CHAR(`*n***`)`** (1-8,000 bytes), **`VARCHAR(`***n***`)`** (1-8,000 bytes, 2 GB for n = **`max`**), **`NTEXT`** (1-2,147,483,647 bytes)
  - **binary strings**
    `BINARY(`*n***`)`** (1-8000 bytes), **`VARBINARY(`***n***`)`** (1-8,000 bytes, 2 GB for n = **`max`**), **`IMAGE`** (1-2,147,483,647 bytes)

Fixed-Length Fields

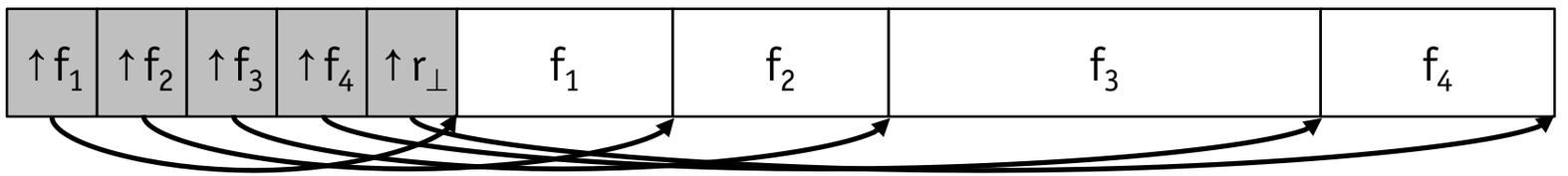


- Fixed-length record: each field has a **fixed length** and the number of fields is also **fixed**
  - fields can be stored **consecutively**
  - given the address of the record (b), the address of a particular field can be calculated using information about **lengths of preceding fields** (l_i)
  - this information is available from the DBMS **system catalog**

Variable-Length Fields

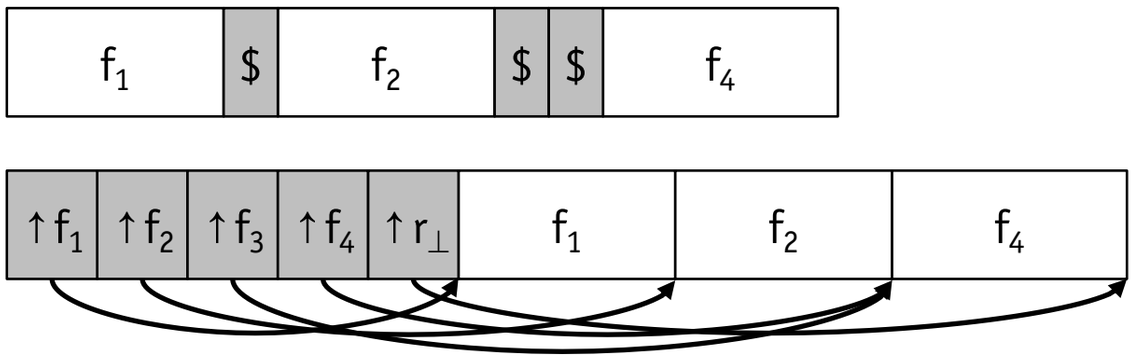
- Multiple variants exist to store records that contain variable-length fields
  1. use a special **delimiter symbol** ($) to separate record fields: accessing field f_n requires a scan over the bytes of fields $f_1 \ldots f_{n-1}$

  2. for a record of n fields, use an **array** of $n + 1$ offsets pointing into the record (the last array entry marks the end of field f_n)



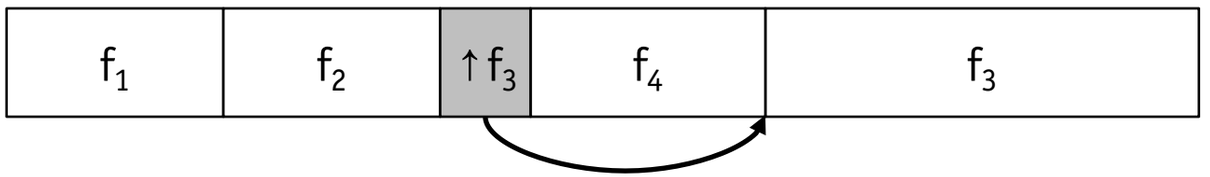
Delimiter vs. Array

- Array approach is typically superior
  - array overhead translates to **direct access** to any field
  - clean and compact way to deal with **null values** (`NULL` in SQL) by simply comparing pointers to **beginning** and **end** of field



Record Formats

- Another popular record format to support variable-length fields is a **combination** of the delimiter and array approach
  - variable-length fields are stored **at the end of the record**
  - fixed-length fields and pointers to variable-length fields are stored sequentially, starting **at the beginning of the record**



- Note that it may even make sense to use variable-length records to store fixed-length records
  - support for null values (see above)
  - schema evolution, i.e., adding or removing columns

Modifying Variable-Length Fields

Growing a record

Modifying a variable-length field may cause it to grow! How could the DBMS file manager cope with the following cases?

1. **the updated record still fits on the page**

2. **the updated record does not fit on the page anymore**

3. **the updated record does not fit on any other page**

IBM DB2 Data Pages

The Real World

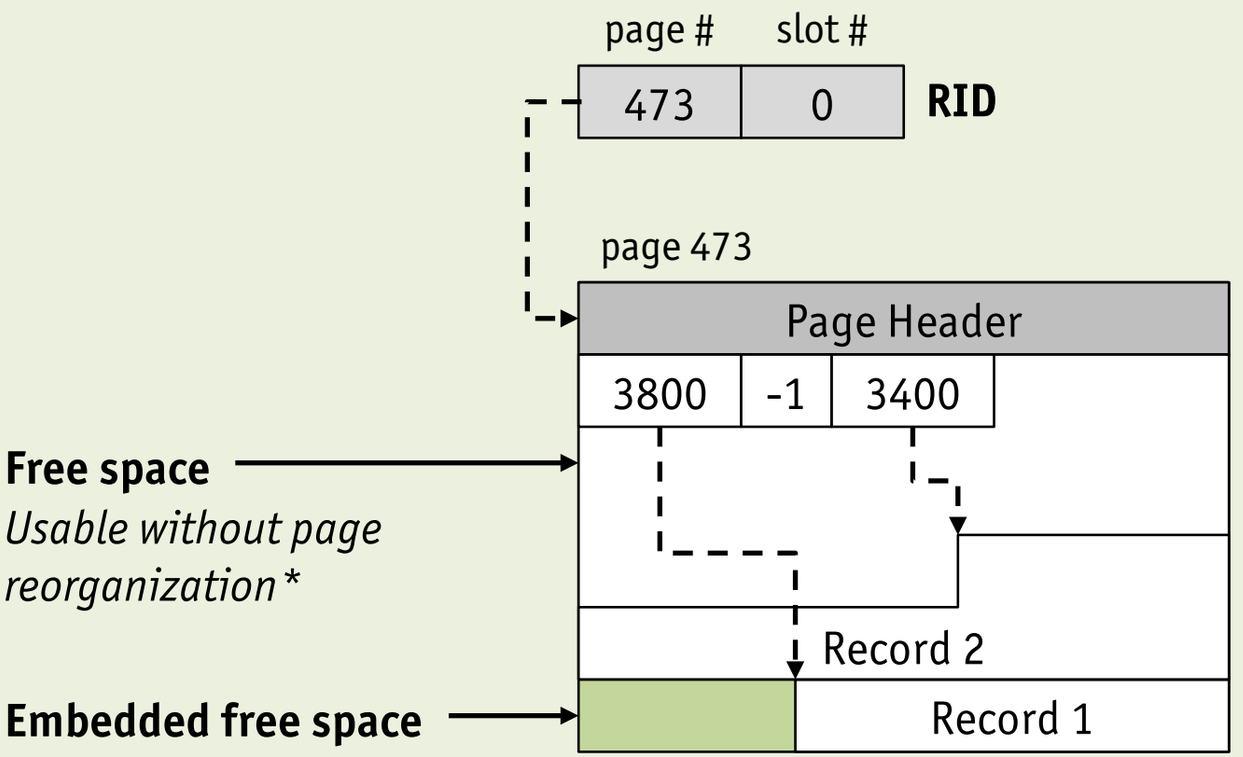
- Support for **4 kB**, **8 kB**, **16 kB**, and **32 kB data pages** in separate table spaces, buffer manager pages match in size
- **68 bytes** of database manager overhead per page, e.g., on a 4 kB page
  - **maximum user data:** 4,028 bytes
  - **maximum record size:** 4,005 bytes
- Records do **not** span pages
- **Maximum table size:** 512 GB (with 32 kB pages)
- **Maximum number of columns:** 1,012 (500 on a 4 kB page)
- **Maximum number of rows per page:** 255
- Columns of type **`LONG VARCHAR`**, **`CLOB`**, etc. maintained **outside** regular data pages, which contain descriptors only
- **Free space management:** first-fit order
  - free space map distributed on every 500th page in **free space control records**
  - records updated in-place if possible, otherwise uses **forwards**

Slide Credit: Torsten Grust, University of Tübingen, Germany

IBM DB2 Data Pages

The Real World

Data page and RID format



Supported page sizes:
4 kB, 8 kB, 16 kB, 32 kB

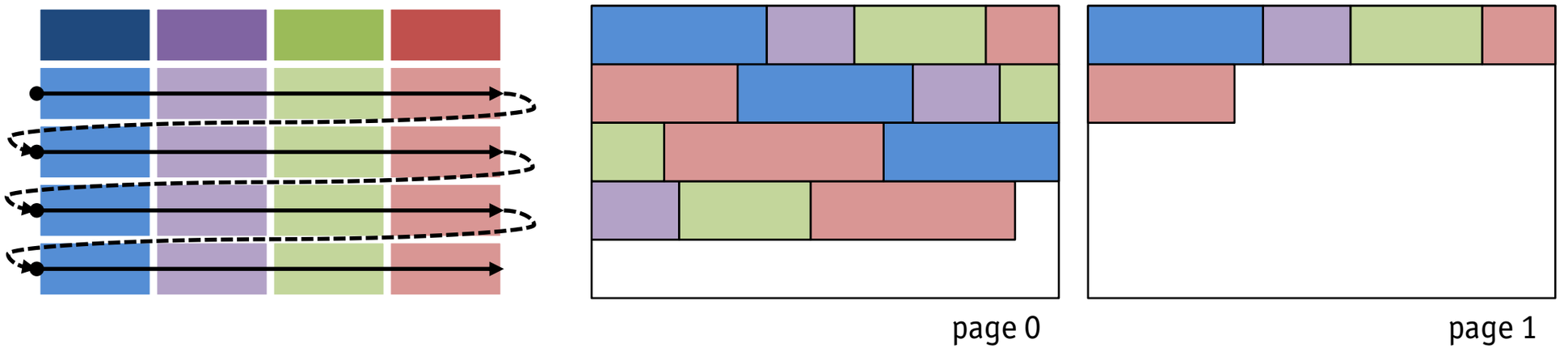
Set of table space creation. Each table space must be assigned a buffer pool with a matching page size.

* **Exception:** Any space reserved by an uncommitted **`DELETE`** is not usable.

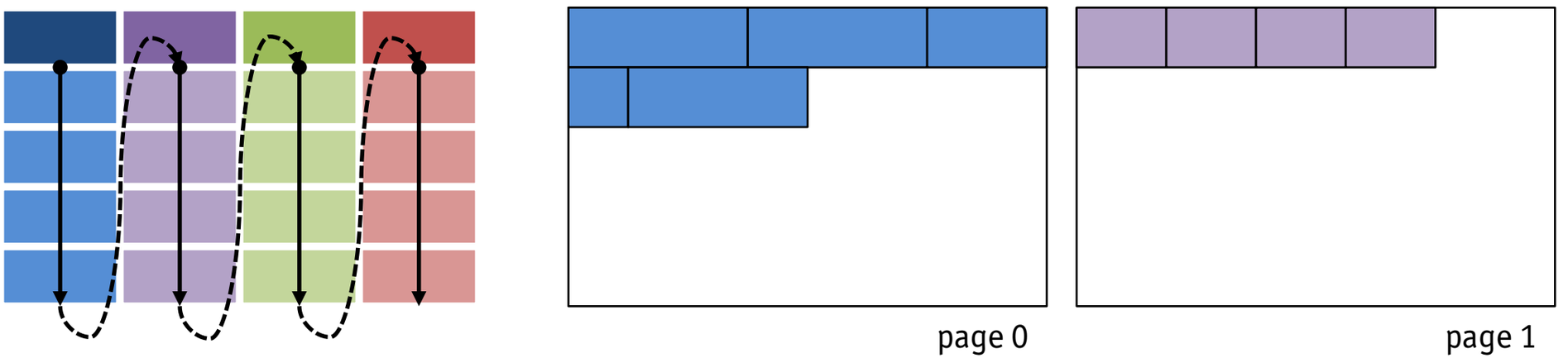
Figure based on: http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp

Alternative Page Layouts

- So far, we populated data pages in **row-wise** order



- We could also populate data pages in **column-wise** order



Slide Credit: Torsten Grust, University of Tübingen, Germany

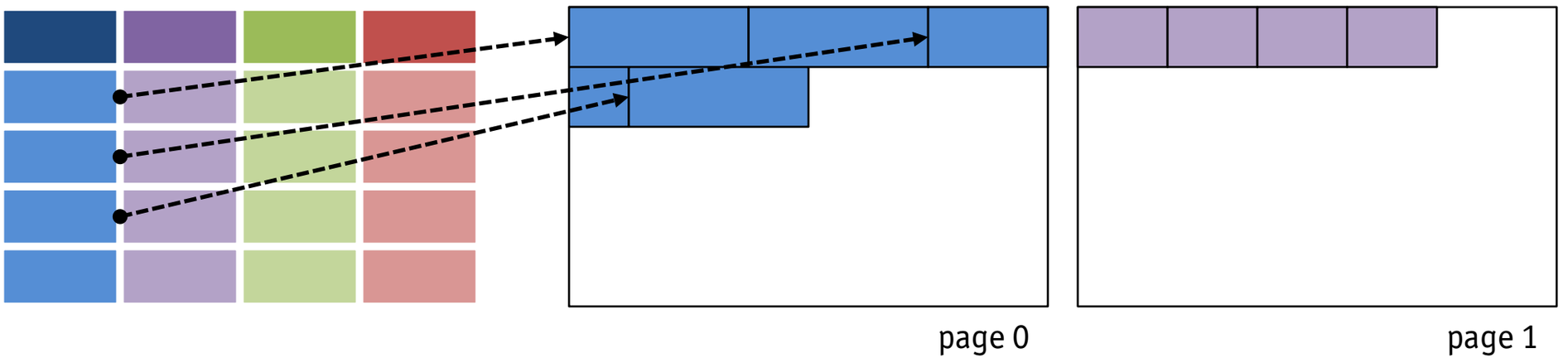
Alternative Page Layouts

- These two approaches are also know as **n-ary Storage Model (NSM)** and **Decomposition Storage Model (DSM)**
  - beneficial for different workload types, e.g., OLAP, OLTP, HTAP
  - suitable for narrow projections and in-memory database systems
  - different behavior with respect to compression
  - a.k.a. **row-store** and **column-store**

Slide Credit: Torsten Grust, University of Tübingen, Germany

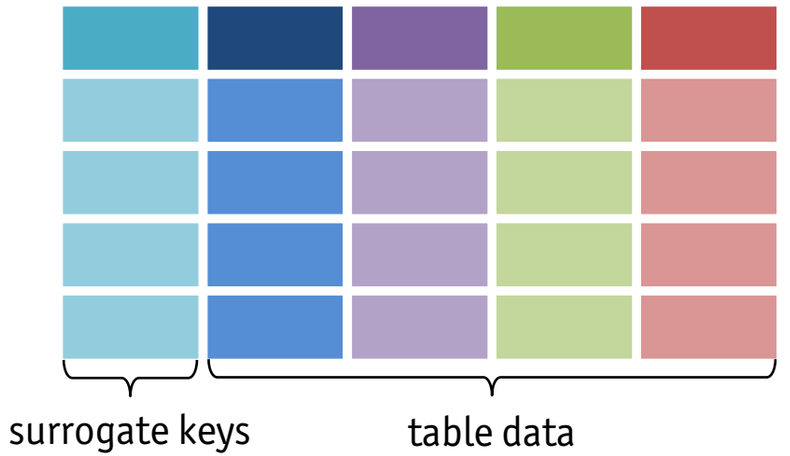
Column-major Layout

- Suppose you have a table in **column-wise** order
- How do you identify individual *tuples* or *values*?

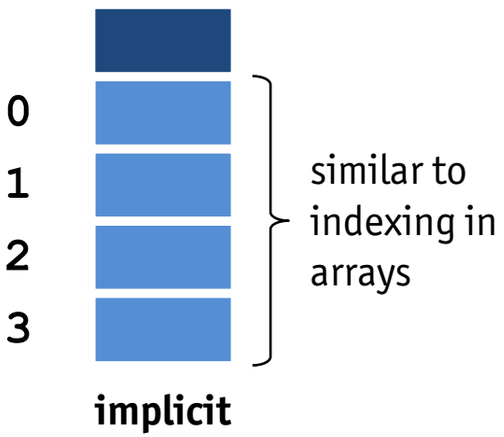


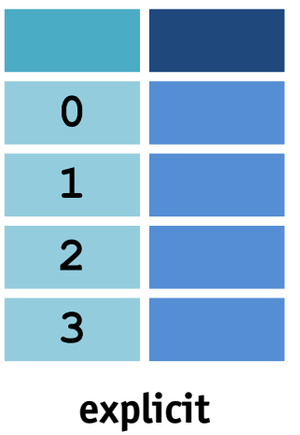
Surrogate Keys

- One solution is to use **surrogate keys**



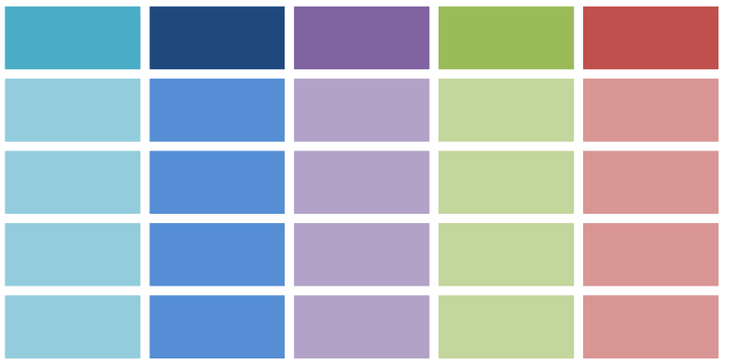
- Surrogate keys can be **implicit** or **explicit**

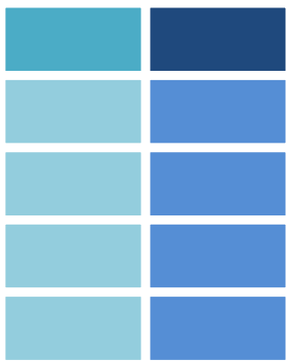


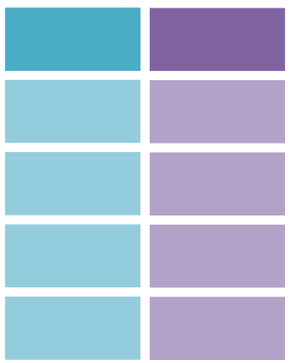


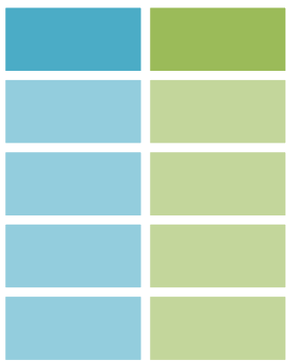
Binary Association Tables

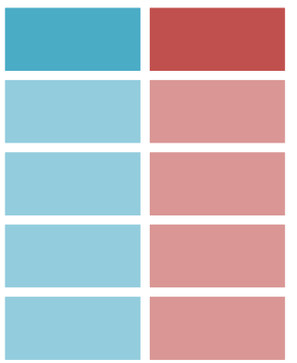
- Tables broken into individual columns are sometimes called **Binary Association Tables (BAT)**





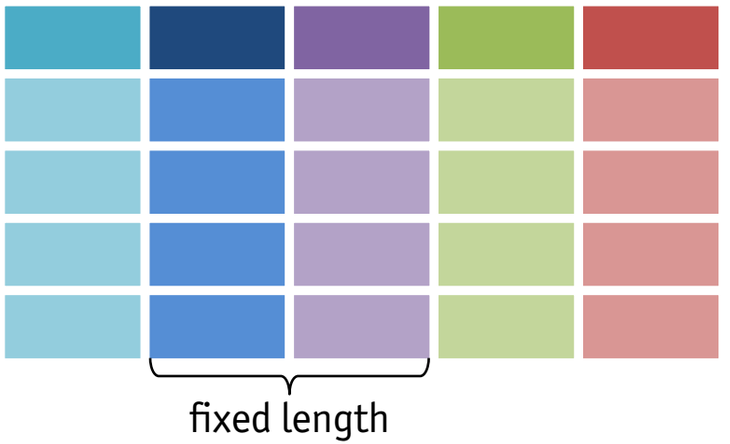






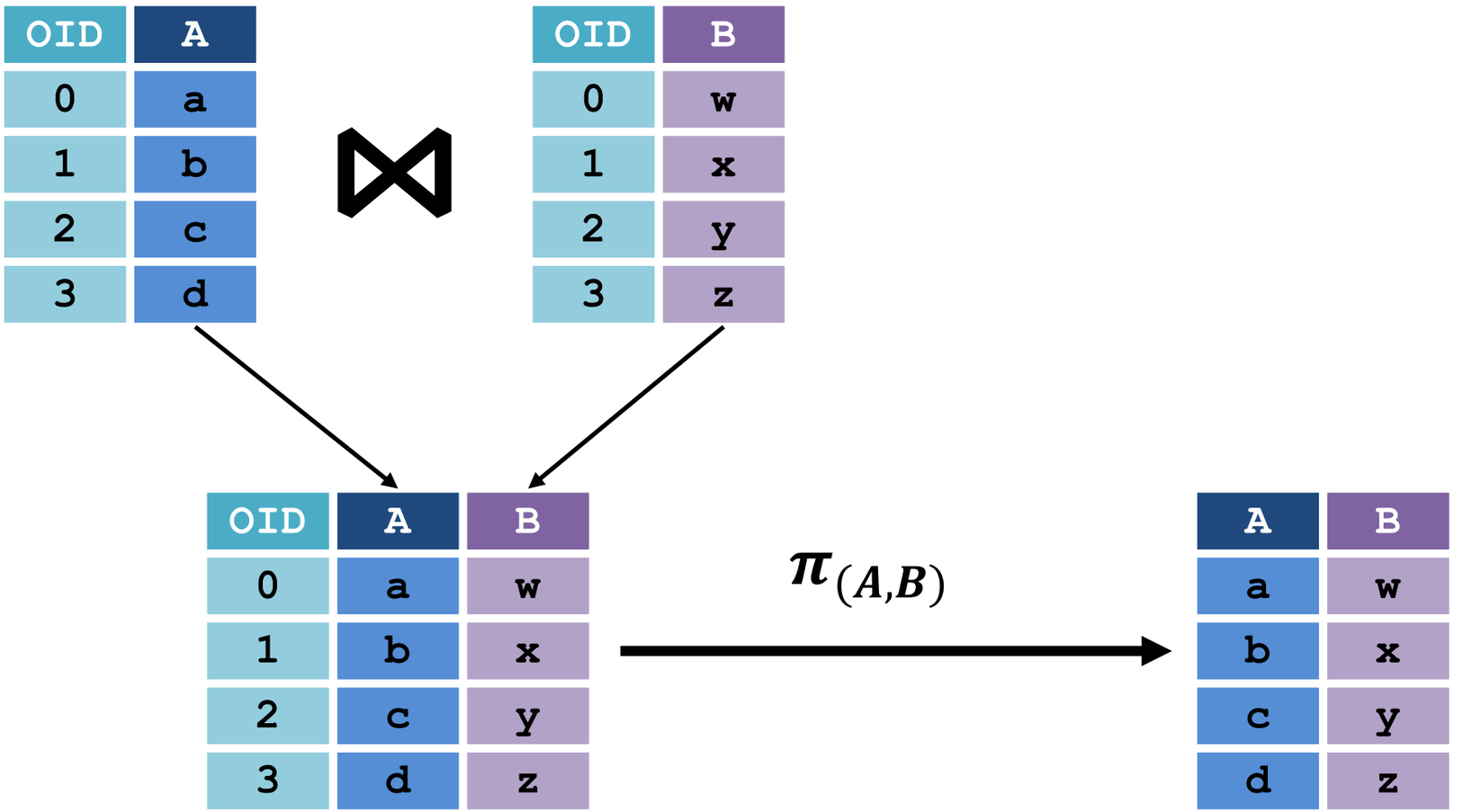
Binary Association Tables

- Fixed length data types can omit *storage* of surrogate keys
- Stored as densely packed array of values and mapped to memory



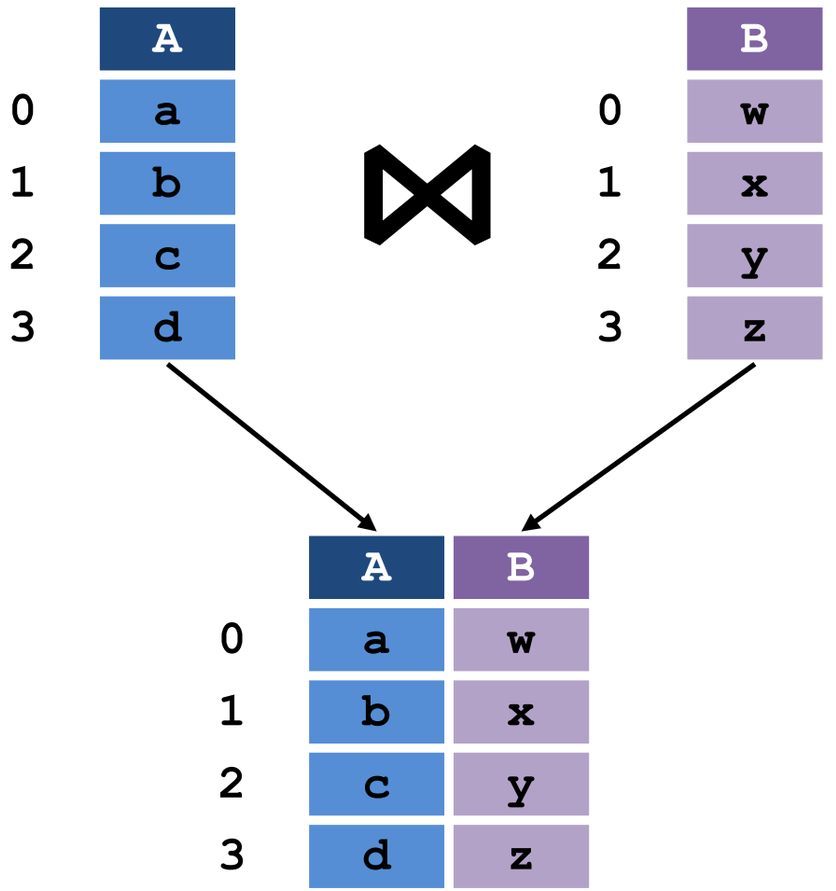
Tuple Reconstruction

- Reconstruct tuple from individual columns by surrogate keys



Tuple Reconstruction

- Reconstruct tuple from individual columns by surrogate keys
- Joins involving implicit surrogate keys use offset in column



Tuple Reconstruction

- Reconstruct tuple from individual columns by surrogate keys
- Joins involving implicit surrogate keys use offset in column
- When reconstruction should be done: **early**, **late**
  - **early:** only *access* operators need to know about columnar layout
  - **late:** every operator needs to support columnar layout

MonetDB: BATs & Late Reconstruction

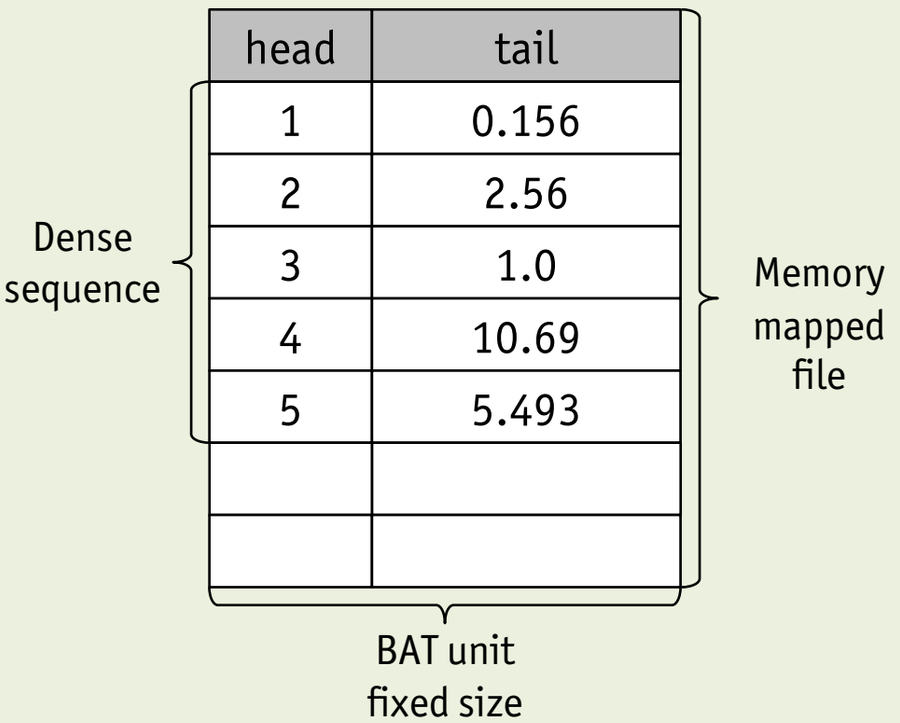
The "Real" World

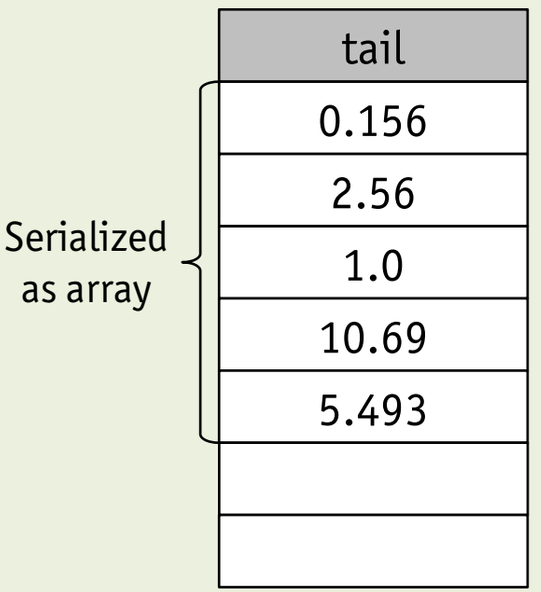
- MonetDB is a **column-store** using the **DSM** with **late** tuple reconstruction
- **BATs** for data storage and **BAT Algebra** for data processing
- **MonetDB Assemblee Language (MAL)** for programming MonetDB's **kernel** (or VM)
- Each BAT Algebra operator is **mapped** to a single MAL instruction
- Higher-level programming languages (e.g., SQL) are **translated** to MAL
- Each MAL instruction consumes BATs and produces BATs (with exceptions for DML)
- BATs are stored on disk and ...
  - ... accessed as **memory mapped files**
  - ... processed in **bulk**
- MonetDB optimizes case of fixed-length fields with densely packed surrogate keys

MonetDB: BAT format

The "Real" World

Binary Association Tables



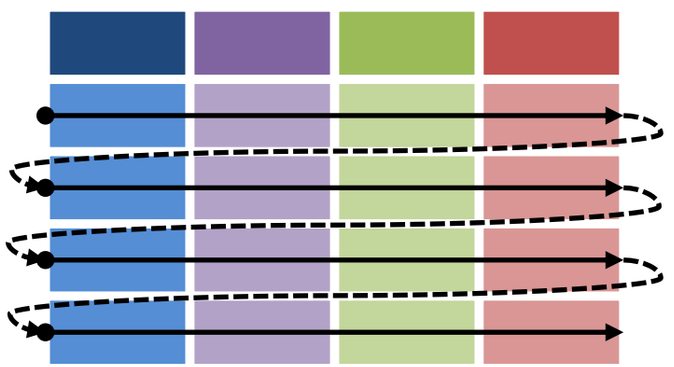


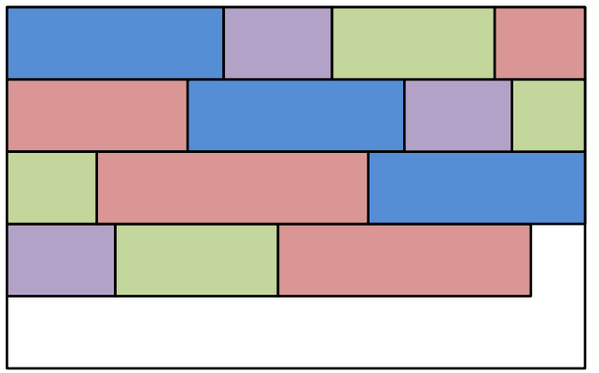
Head column:
For densely packed, fixed-length attributes, *head* is omitted.

Figures based on: The MonetDB Architecture, M. Kersten, CWI, Amsterdam

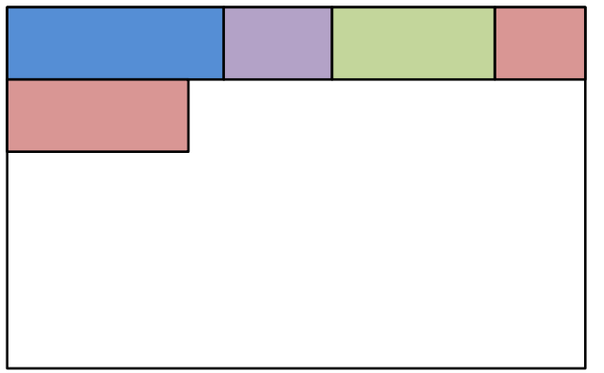
RECAP Alternative Page Layouts

- So far, we populated data pages in **row-wise** order



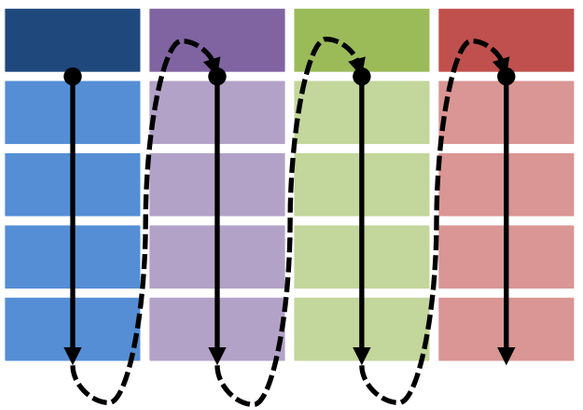


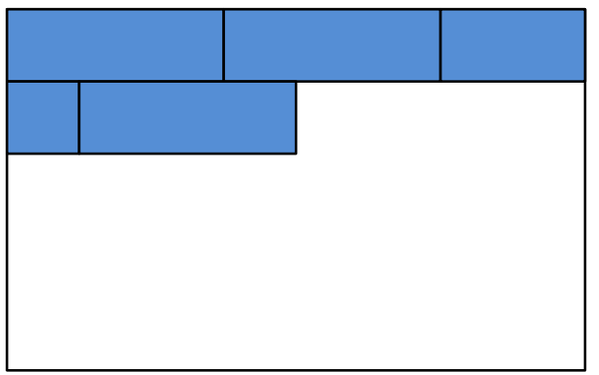
page 0



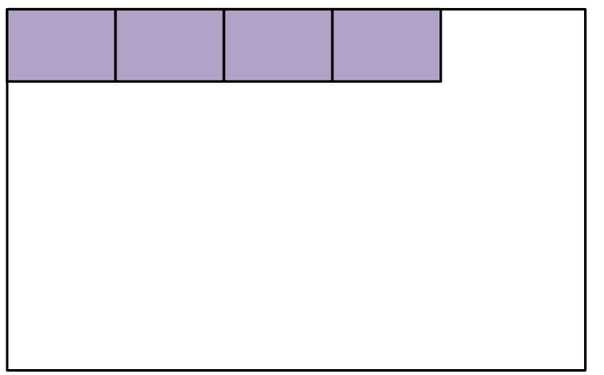
page 1

- We could also populate data pages in **column-wise** order





page 0



page 1

Slide Credit: Torsten Grust, University of Tübingen, Germany

Table Files

- **Table files** comprise **data** pages and **directory** pages
- Directory pages form a chain
- Data pages hold table data
- A **row-store** may organize a table file like this:

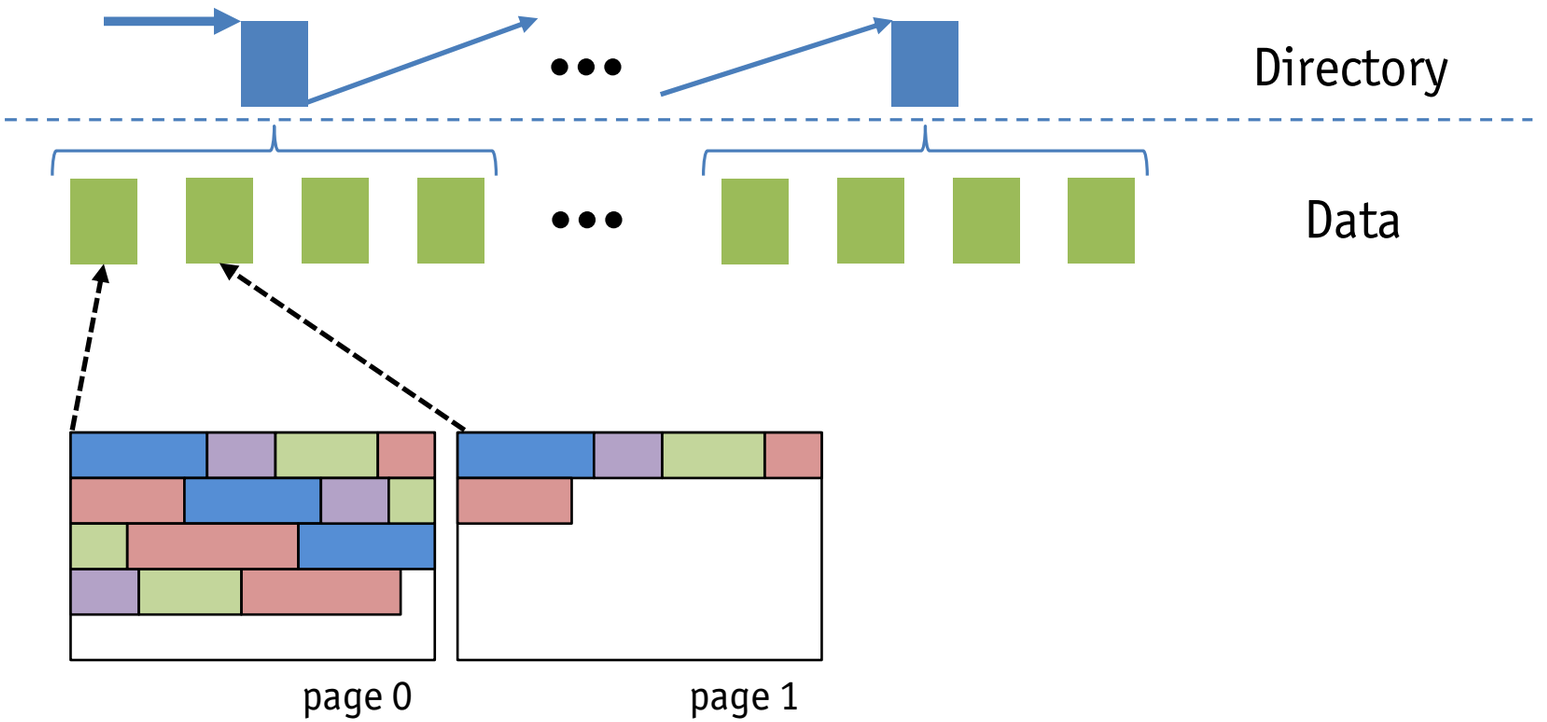


Table Files

- In a **column-store**, columns can be stored **independently**
- An implementation could use a ”mini-directory” for each column
- Free-slot information is then coded in a **free-slot bitmap**
- *All information has to be stored in pages!*

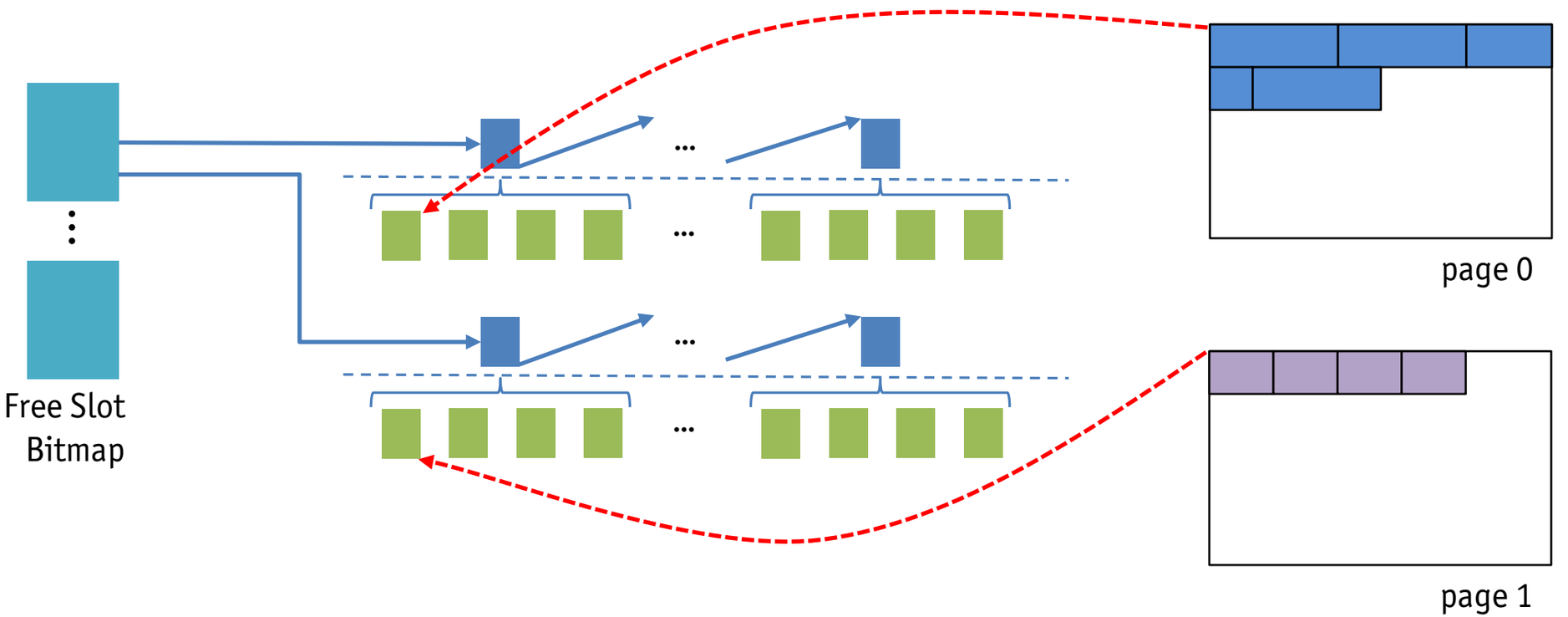
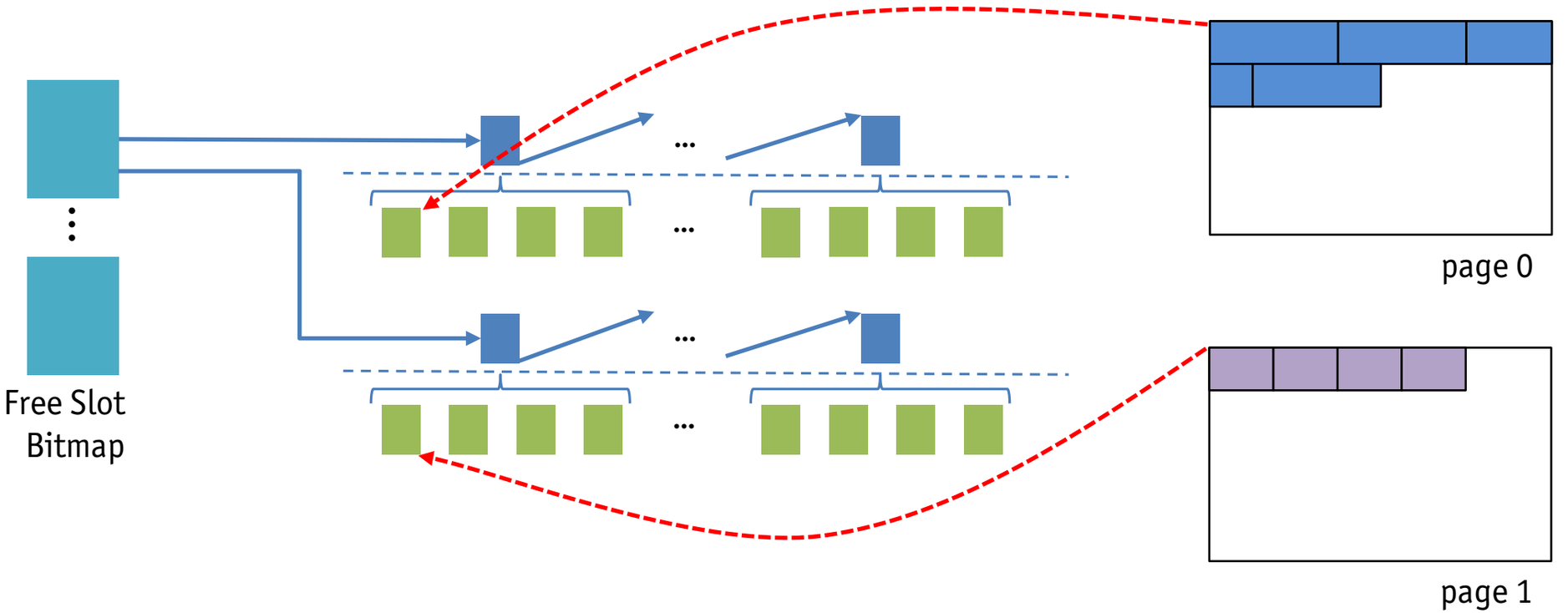


Table Files



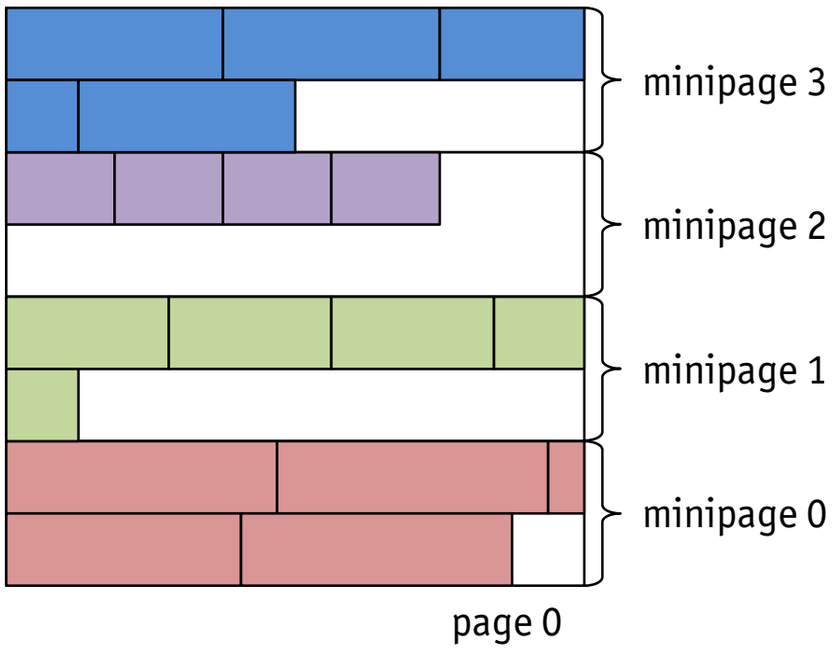
- ⊕ Fully **sequential** scans along every column
- ⊖ Scanning multiple columns may exhibit **skip-sequential** or even **random** access
- ⊖ Updates/Inserts/Deletes need to access **multiple** locations

Benefits of Page Layouts

- Different layouts optimize for different workloads
- **Row-major** layout favors singular access to whole tuples
- **Column-major** layout favors table scans accessing only one/few columns
- Single whole tuples are needed in insert/update/delete operations (OLTP workload)
- Scans/aggregations access whole tables with one/few columns (OLAP workload)
- Optimally, we would want the best of both worlds: **hybrid/adaptive layouts** try to solve this

Alternative Page Layouts

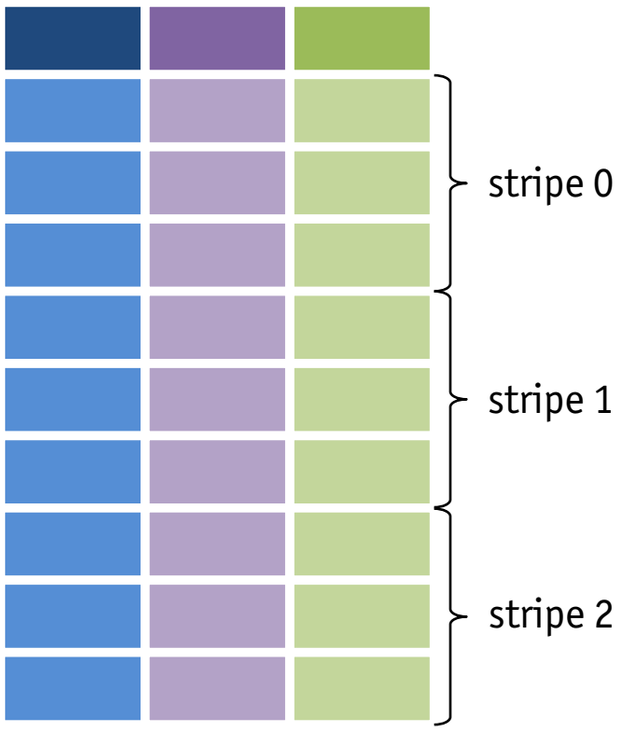
- A hybrid approach is the **Partition Attributes Across (PAX)** layout
  - divide each page into **minipages**
  - group attributes into them
- On-disk performance of NSM
- Cache friendliness/compression characteristics of DSM

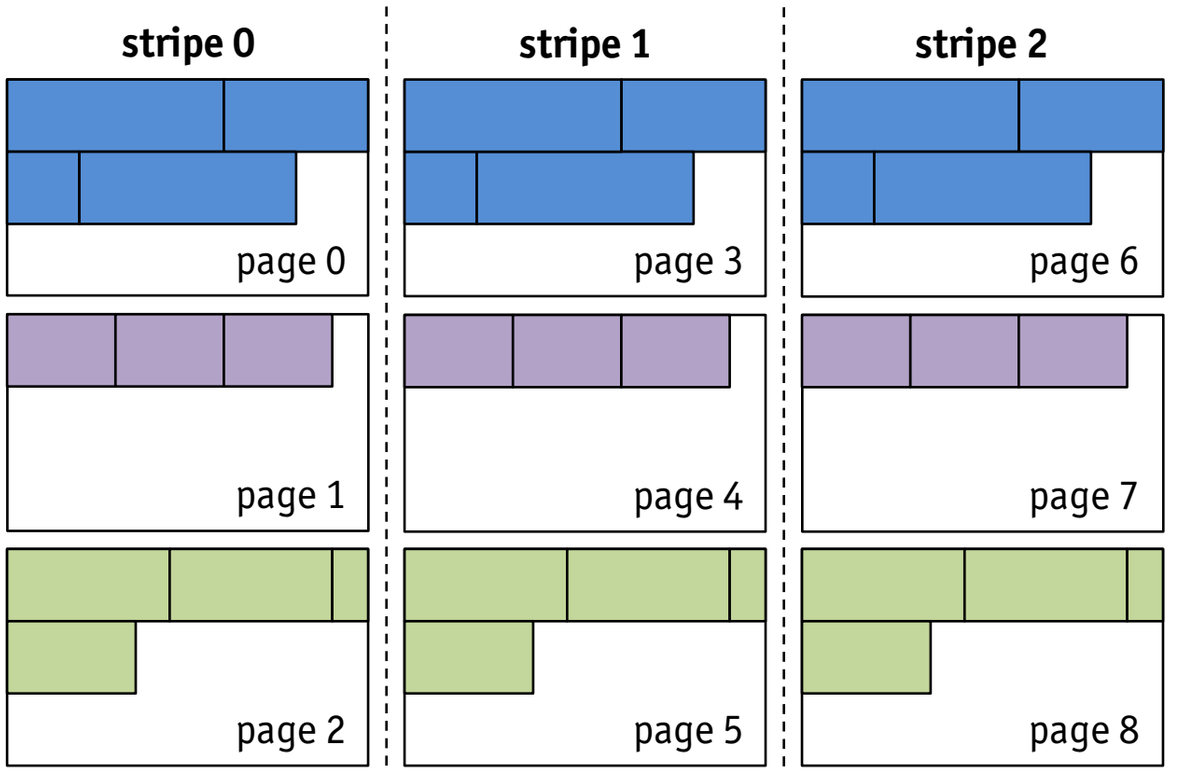


Slide Credit: Torsten Grust, University of Tübingen, Germany

Alternative Page Layouts

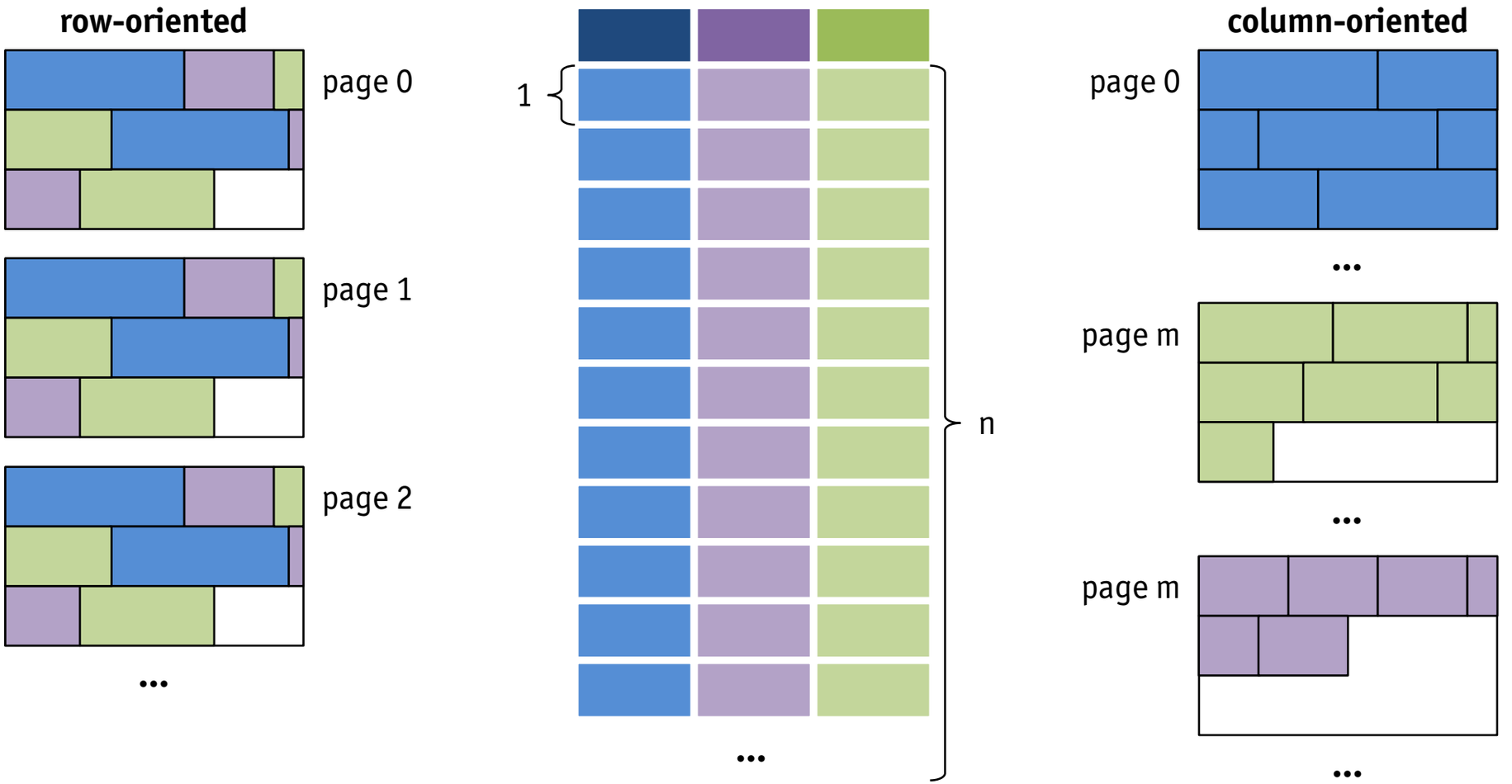
- Another hybrid approach is the **Optimized Row Columnar (ORC)** file format used by *cstore_fdw* for PostgreSQL
- ORC partitions records twice
  - **horizontally** into stripes
  - **vertically** into columns





Alternative Page Layouts

- Stripe size parameter can be used to **trade-off** between row-oriented and column-oriented storage layouts



Alternative Page Layouts

- ”Flexible Storage Model” (FSM) is a hybrid/adaptive layout
- Abstract physical storage from logical storage in **tile groups**
- Data is conceptually divided into two partitions
  - **Hot data:** Inserted/updated tuples in a **row-major** layout
  - **Cold data:** Read-mostly tuples in a **column-major** layout

Alternative Page Layouts

- **Physical tiles** (tile groups) are stored on disk

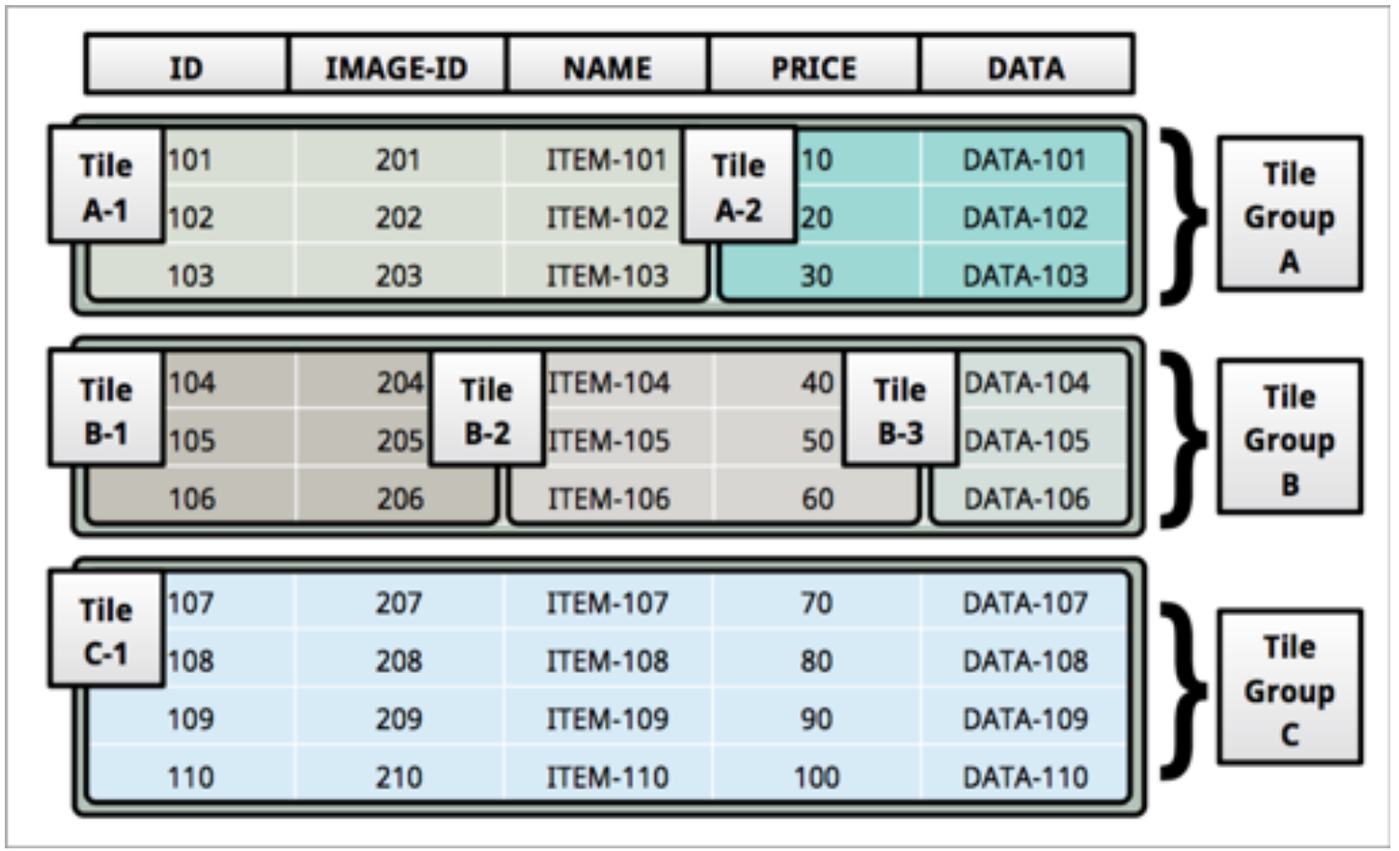


Figure Credit: Joy Arulraj, Carnegie Mellon University

Alternative Page Layouts

- **Layout transparency** through logical tiles
- **Logical tile algebra** with boundary operators to handle tiles and inter-operate with traditional operators/storage manager

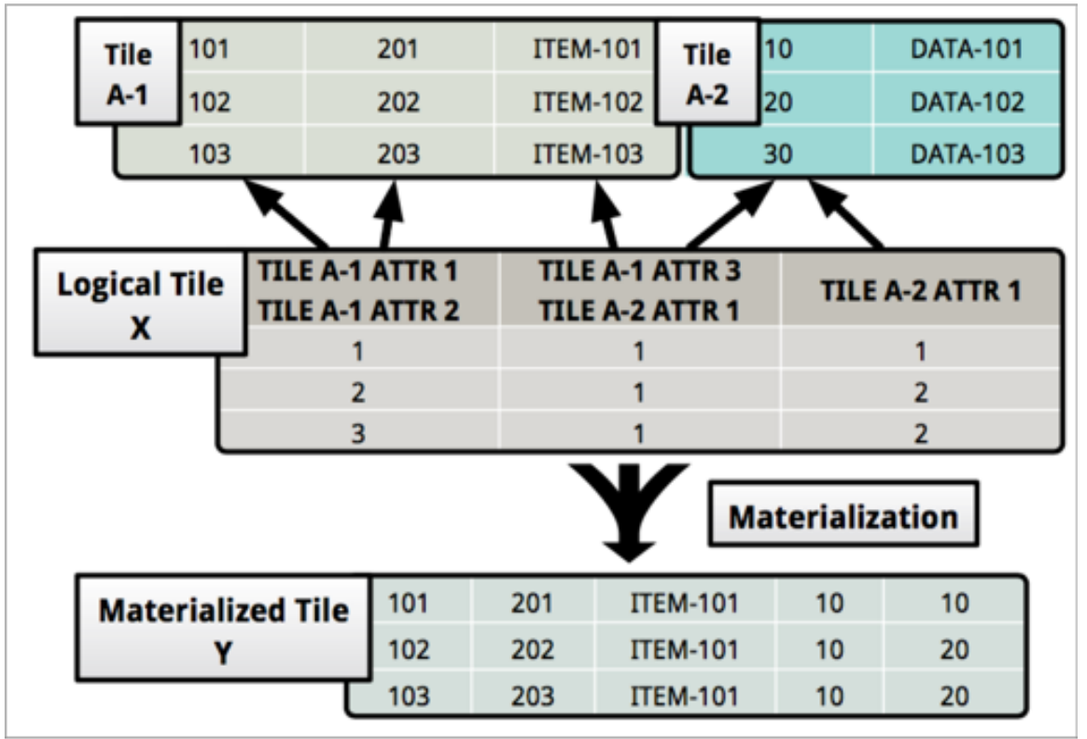


Figure Credit: Joy Arulraj, Carnegie Mellon University

Addressing Schemes

- Criteria for "good" record ids (*rids*)
  - given an *rid*, it ideally takes **no more than one page I/O operation** to get the record itself
  - *rids* should **stable** under all circumstances, e.g., when a record is being **moved within a page** or **across pages**

Why are these goals important to achieve?

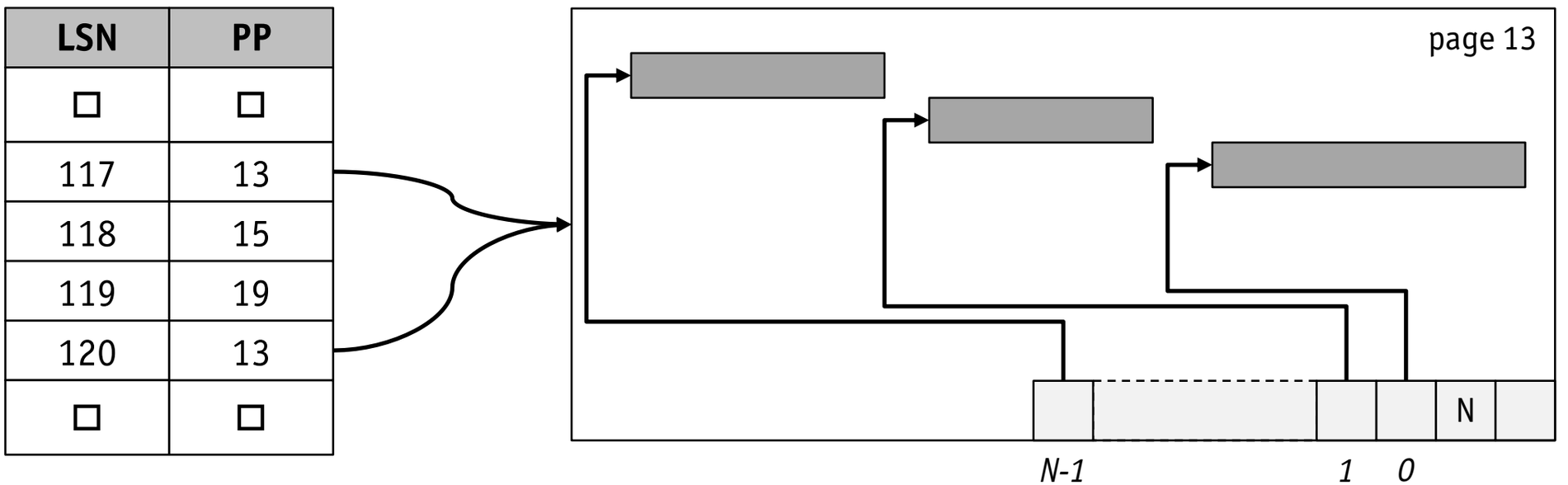
These goals are conflicting. Explain!

Direct Addressing

- **Relative Byte Address (RBA):** think of a disk file as a persistent virtual address space and use byte-offset as *rid*
  - ⊕ very efficient access to pages and records within pages
  - ⊖ no stability at all w.r.t. to moving records
- **Page Pointers (PP):** use disk page numbers as *rid*
  - ⊕ very efficient access to page, locating records within a page is also cheap (in-memory operation)
  - ⊖ stable w.r.t. to moving record within a page, but not when moving records across pages

Indirect Addressing

- **Logical Sequence Numbers (LSN):** assign logical numbers to records and use address translation table to map LSN to PP (or even RBA)
  - ⊕ full stability w.r.t. all relocations of records
  - ⊖ additional I/O operation to translation table (often in the buffer)

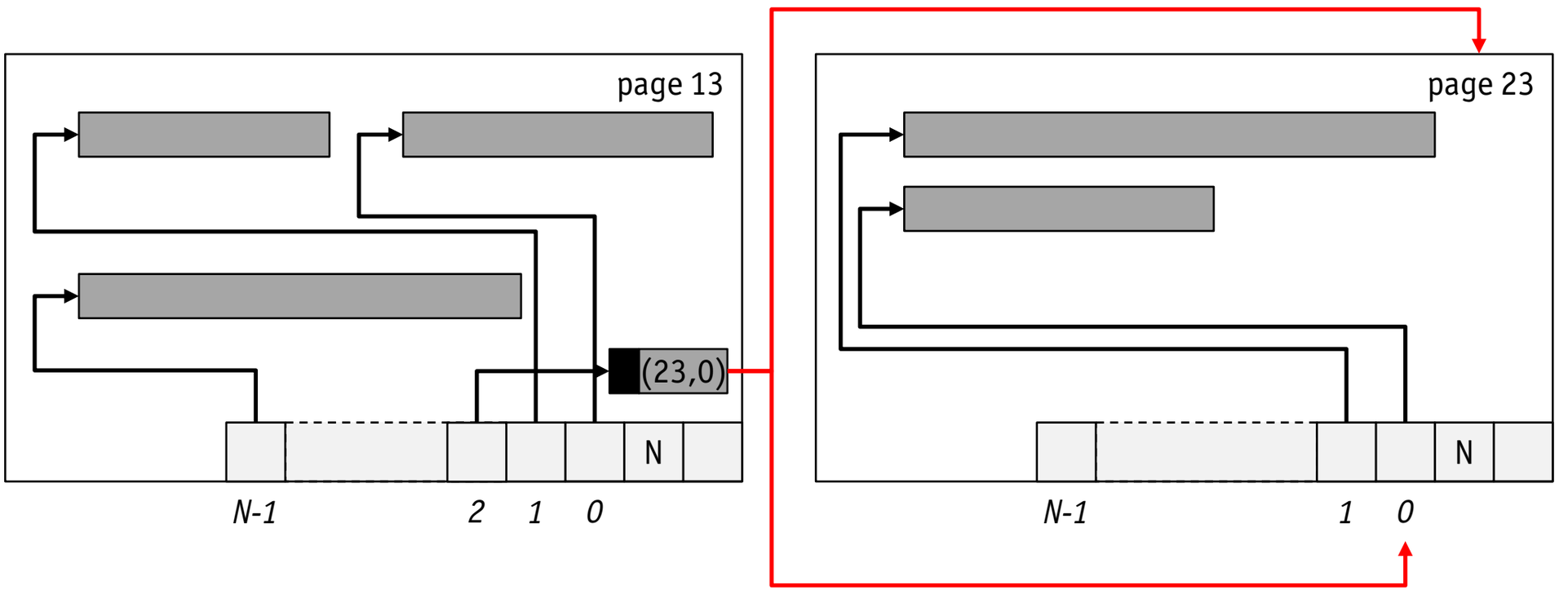


Fancy Indirect Addressing

- **LSN with Probable Page Pointers (LSN/PPP):** try to avoid extra I/O operations by adding a "probable" PP (PPP) to LSN, where PPP is PP at the time of insertion into database (if record is moved across pages, PPP is **not** updated)
  - ⊕ full stability w.r.t. all record relocations and PPP can save extra I/O operation, if still correct
  - ⊖ two additional page I/O operations if PPP is no longer valid: need to read "old" page to notice that record has moved, then need to read translation table to lookup new page number

Recall Initial Addressing Scheme

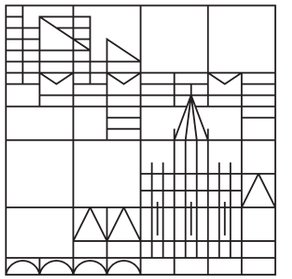
- Tuple Identifier (TID): use <*pageNo*, *slotNo*> pair as *rid*
  - *slotNo* is an index in a page-local offset array
  - this guarantees stability w.r.t. relocation within a page
  - to guarantee stability w.r.t. relocation across pages, leave a **forwarding address** on original page



Tuple Identifier Addressing Scheme

But what happens when the record is moved again?

- Discussion
  - ⊕ full stability w.r.t. all relocations of records, no extra I/O operations due to indirection
  - ⊖ only one additional page I/O operation in case of forward pointer on original page
- Therefore, most DBMS use this addressing scheme!



TO BE CONTINUED...