

DB 2

06 – Buffering and Caching

Summer 2018

Torsten Grust
Universität Tübingen, Germany


1 | Memory: Fast, But Tiny vs. Slow, But Large

- Recall the enormous **latency gaps** between accesses to the (L2) CPU cache, RAM, and secondary storage (SSD/HDD):

Memory	Actual Latency ⌘	Human Scale 😬	Typical Size
CPU L2 cache	2.8 ns	7 s	$\frac{1}{4}$ -16 MB
RAM	\approx 100 ns	4 min	4-128 GB
SSD	50-150 μ s	1.5-4 days	$\frac{1}{2}$ -2 TB
HDD	1-10 ms	1-9 months	1-16 TB

- Facts: faster memory is significantly smaller. We will not be able to build cache-only systems.
 - The lion share of data will live in slow memory.
 - Only selected data fragments may reside in fast memory.
Which fragments shall we choose?

Spatial Locality

- In a DBMS (and most computing processes), **memory accesses are not random** but exhibit patterns of **spatial and/or temporal locality**:
1. **Spatial locality**: last memory access at address m , next access will be at address $m \pm \Delta m$ (Δm small).
- Often, $\Delta m \equiv$ machine word size: backward/forward scan of memory, i.e., iteration over an array.
 - Block I/O accesses and reads data at m and its vicinity:
|block accesses| \ll |memory accesses| 

Temporal Locality

2. **Temporal locality:** last memory access at m at time t , next access at m will be at time $t + \Delta t$ (Δt small).

Memory that is relevant now, will probably be relevant in the near future \Rightarrow DBMS **tracks frequency and recency of memory usage**. Uses both to decide whether to hold a page in fast memory.

Found on multiple levels (concerns PostgreSQL *and* MonetDB):

Slow Memory	Fast Memory	Fast/Slow Size ¹
SSD / HDD	RAM	$1/64$
RAM	CPU L2 cache	$1/65536$
CPU L2 cache	CPU L1 cache	$1/8$

¹ Specified for this  MacBook Pro (CPU Intel® Core i7, 32/256 KB L1/L2 cache, 16 GB RAM, 1 TB SSD).

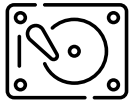
Q_5 (Set of Queries) — Temporal Locality of References

Can the DBMS benefit if the **query workload** (\equiv set of typical queries submitted to the DBMS) contains repeated data references, close in time?

```
 $\ominus=t_0$ :   SELECT t.a, t.b FROM ternary AS t;  
 $\ominus=t_0+\Delta t$ : SELECT s.a, s.c FROM ternary AS s;  
⋮
```

Set of referenced data pages overlap. We hope that I/O effort invested for earlier queries may benefit subsequent operations.

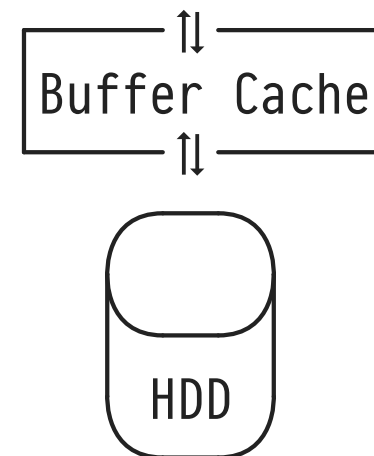
The Buffer Cache

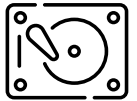


The DBMS sets aside a dedicated section of RAM — the **buffer cache** (or simply **buffer**) — to temporarily hold pages.

- *All* DBMS page accesses are performed using the buffer \Rightarrow can track page usage.
- $|\text{buffer}| \ll |\text{RAM}|$. In PostgreSQL, see config variable `shared_buffers` (defaults to 128MB). Good practice: buffer size \approx 25% of RAM.

SELECT .../UPDATE ...



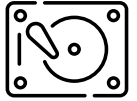


Buffer Cache Interface (API)

- Any database transaction properly “brackets” page accesses using `ReadBuffer()` and `ReleaseBuffer()` calls:

```
<math>b, m</math>  $\leftarrow$  ReadBuffer(table, block);  
  /* now may access 8 KB page starting at address m */  
  :  
if (page at m has been written to)  
  | MarkBufferDirty(b);  
  :  
ReleaseBuffer(b);  
  /* accesses to address m illegal from here on */
```

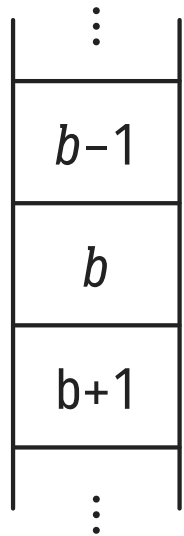
- Proper bracketing enables the DBMS to perform bookkeeping of buffer contents.



Buffer Page Bookkeeping

- Each page in the buffer is associated with meta data that reflects its current utility for the DBMS:

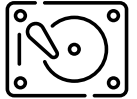
buffer pages



buffer descriptor

- $\text{ref_count}(b) \geq 0$: how often has this page been requested via `ReadBuffer()`?
- $\text{usage_count}(b) \geq 0$: how many r/w operations have been performed on this page?
- $\text{dirty?}(b)$: does memory on this page differ from its disk image?

- $\text{ref_count}(b)$ also commonly known as the *pin count* of b .

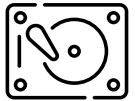


Reading a Buffer Page: Hit vs. Miss

```

ReadBuffer(table, block):
  if (a buffer page b already contains block of table)
    ref_count(b) ← ref_count(b) + 1;
    return <b, address of b's page>;      /* hit: no I/O */
  -----
  else                                     /* miss: I/O needed */
    v ← free buffer page;                 /* 1 */
    if (there is no such free v)
      v ← FindVictimBufferPage();        /* 2 */
      if dirty?(v)
        [ write page in v to disk block;
        read block from disk into page of v;
        ref_count(v) ← 1;
        dirty?(v) ← false;
      ]
    return <v, address of v's page>;

```



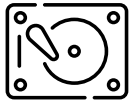
Clean vs. Dirty Buffer Pages

- Read-only transactions leave buffer pages **clean** — clean victim pages may simply be overwritten when replaced.
- Marking buffer page b **dirty** (i.e., written to/changed):

```
MarkBufferDirty( $b$ ):  
    dirty?( $b$ ) ← true;
```

- In regular intervals, the DBMS writes dirty buffer pages back (**checkpointing**) to match memory and disk contents.²
 - Checkpointing may lead to heavy I/O traffic.

² PostgreSQL: see config variable `checkpoint_timeout` (default: '5min'). SQL command `CHECKPOINT` forces immediate checkpointing.




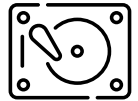
Releasing a Buffer Page

- Release buffer page b . If $\text{ref_count}(b) > 0$, b is called **pinned**. If $\text{ref_count}(b) = 0$, b is **unpinned**:

```
ReleaseBuffer( $b$ ):
```

```
    ref_count( $b$ ) ← ref_count( $b$ ) - 1;           /* no I/O */
```

-  **ReleaseBuffer()** does *not* write the page of b back to disk, even if b is unpinned and dirty. **Quiz:** Why?
- Any pinned buffer page is in active use by some transaction and thus may *never* be chosen as a victim for replacement.



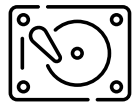
Inspect Dynamic Buffer Behavior

PostgreSQL offers extension `pg_buffercache`, providing a tabular view³ of the system's buffer cache descriptors:

```
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
[ WHERE b.relfilenode = <tbl> ]; -- focus on table <tbl> only
```

bufferid	relblocknumber	isdirty	usagecount
269	0	f	1
270	1	t	1
⋮	⋮	⋮	⋮

³ N.B.: This is only a tabular representation of the buffer descriptors. Internally, the buffer and its descriptors are implemented as C arrays.



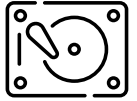
EXPLAIN: Buffer Hits and Misses

EXPLAIN can be instructed to show whether the DBMS experienced **buffer hits or misses** during query evaluation:

```
db2=# EXPLAIN (ANALYZE, BUFFERS, <opt>, ...) <Q>
```

QUERY PLAN

```
⋮  
Buffers: shared read=m ◀ I/O needed ≡ miss  
⋮  
Buffers: shared hit=h ◀ page found in buffer, no I/O  
⋮
```

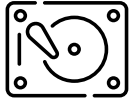


2 : Picking a Free Buffer Page

After a buffer miss, **pick a buffer slot** that will hold the new to-be-loaded page from disk:

1. If the **free list** of buffer slots is non-empty, remove its head slot v . Pick v (see ① in `ReadBuffer()`).
Buffer slot appended to free list when
 - database server (and buffer manager) starts up, or
 - a table or an entire database is dropped (`DROP ...`).
2. If free list is empty, use the **buffer replacement policy** to identify a **victim page** v . Pick v (see ②).

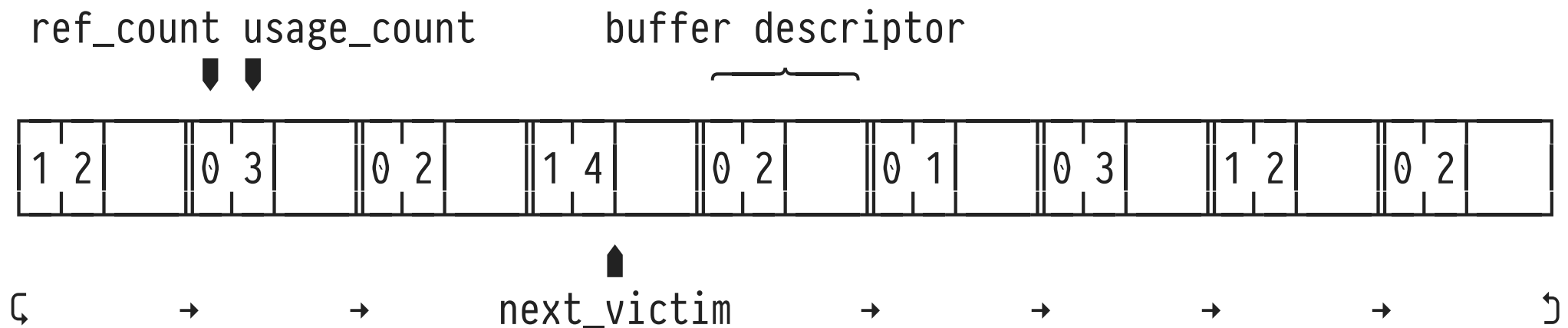
A Replacement Policy: Clock Sweep (\approx LRU)

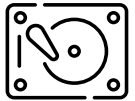


Heuristic: The **least recently used (LRU)** page is a good victim v to pick. We assume v remains unused from now on.

One implementation of LRU: **Clock Sweep:**

- Arrange buffer descriptors in a *circular* array (“clock”).
- Repeatedly “sweep” pointer `next_victim` through array:

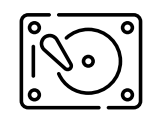




Clock Sweep

```
FindVictimPage():
  try ← 0;
  while (try < N)          /* one full round w/o progress? */
  |
  | v ← buffers[next_victim];
  | if (ref_count(v) = 0)  /* unpinned? */
  | | usage_count(v) ← usage_count(v) - 1;
  | | if (usage_count(v) = 0) /* unpopular page? */
  | | | return v;          /* victim found */
  | | try ← 0;
  | else
  | | | try ← try + 1;
  | | next_victim ← (next_victim + 1) % N; /* skip/sweep */
  return out-of-buffer-space-↵;
```

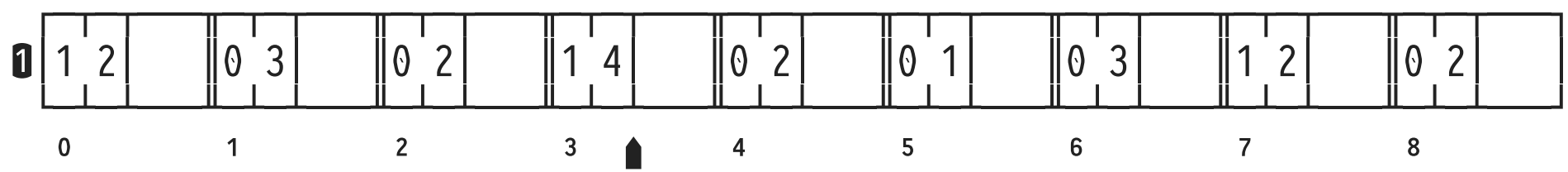
- **N.B.:** `usage_count()` of pages may increase asynchronously.



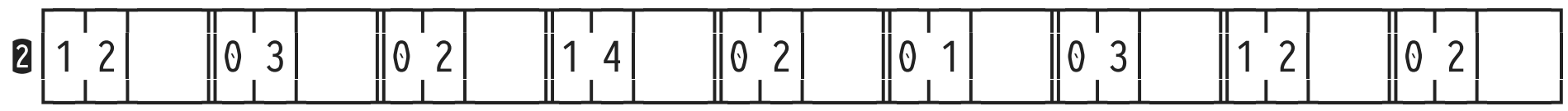
Clock Sweep: Example

ref_count usage_count

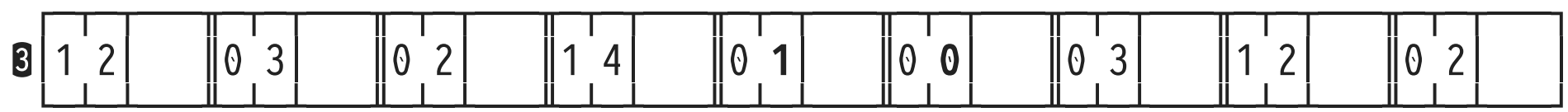
N = 8



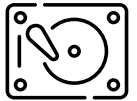
next_victim [page pinned: skip #3]



next_victim [decrement usage; sweep]



next_victim [found victim; pick #5]



Challenges for LRU

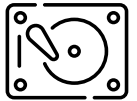
LRU is a **heuristic** and may fail in specific scenarios.

Consider:

1. Assume a 100-page index I with pages I_k and a table R with 10000 pages R_j . We repeatedly use I to look up rows in R . The **page access pattern** will be $I_1, R_1, I_2, R_2, I_3, R_3, \dots$

Q: How will an LRU buffer of 100 slots operate? **A:** 

2. Transactions T_1, T_2, \dots access a small fragment of the database. Transaction T_0 performs a *sequential scan* of a large table (think `SELECT * FROM wide_100M`).

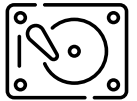


More Page Replacement Heuristics

Other heuristics have been proposed to account for DBMS-specific page reference patterns:

- **LRU- k** : Like LRU, but consider the time passed between the k latest references to a page (typically, $k = 2$).
- **MRU** (most recently used): Replace the page that has been used just now.
- **Random**: Pick a victim randomly. (Straightforward implementation 👍.)

Q: What are the rationales behind these policies? **A:** 🖋️



Variants of LRU: Ring Buffering

PostgreSQL: To protect (small or busy) buffers from being “swamped” by large sequential scans, adopt a **ring buffering strategy**:

- SQL commands that may swamp the buffer:
 - `SELECT ... FROM T` (if T larger than $\frac{1}{4} \times \text{shared_buffers}$ pages),
 - `COPY T FROM ...`, `CREATE TABLE T AS Q`, `ALTER TABLE T ...`,
 - `VACUUM`.
- If command may swamp the buffer:
 1. Use **ring buffer** of size $\leq \frac{1}{8} \times \text{shared_buffers}$ pages.
 2. Release ring buffer immediately after use.

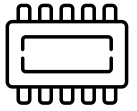
3 : Q_5 (Set of Queries) — Temporal Locality of References



How will a main-memory-based DBMS benefit if the **query workload** (\equiv set of typical queries submitted to the DBMS) contains repeated data references, close in time?

```
 $\ominus=t_0$ : SELECT t.a, t.b FROM ternary AS t;  
 $\ominus=t_0+\Delta t$ : SELECT s.a, s.c FROM ternary AS s;  
:
```

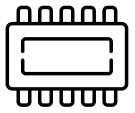
After the first query, the vectors for columns **a**, **b** are located in RAM or even the CPU cache. An additional DBMS-maintained buffer cache will *not* add value.



MMDBMS: Do *Not* Reimplement Caching


MMDBMS typically rely on the cache hierarchy already maintained by the underlying system:

- Recall: We use the OS' `mmap(2)` to map BATs from disk files into RAM \Rightarrow MMDBMS relies on the **OS file system buffer** to cache mapped file contents.
- Contents of RAM addresses accessed recently are found in the CPU's L3/L2/L1 cache hierarchy \Rightarrow MMDBMS relies on **built-in CPU data cache replacement policies**.




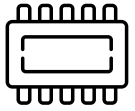
4 : “Memory: The New Disk”

Recall: It makes a significant difference whether accessed memory is present in the CPU data cache or only in RAM:

Operation	Actual Latency \times	Human Scale 
CPU cycle	0.4 ns	1 s
L2 cache access	2.8 ns	7 s
RAM access	\approx 100 ns	4 min

Excerpt of System Latencies (at Human Scale)

- Impact on MMDBMS implementation strategies:
 - When CPU has moved data from RAM into its cache, **make the best use of all that data**: data vectors / BATs. 
 - If possible, use **simple memory access patterns** such that CPU can **predict** which addresses are needed next.

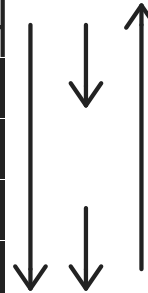


Predictable Memory Access

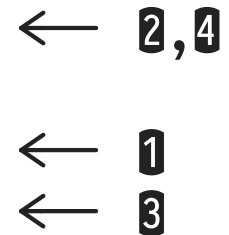
Predictable access patterns:

- forward scans
(possibly with skips)
- backward scans

head	tail
0@0	v_0
1@0	v_1
2@0	v_2
3@0	v_3

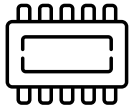


head	tail
0@0	v_0
1@0	v_1
2@0	v_2
3@0	v_3



- Predictable: CPU issues **asynchronous memory prefetch operations** to preload data cache and hide memory latency.
- Unpredictable: DBMS code adds explicit **software prefetch instructions**⁴ for addresses needed in the future.

⁴ No-ops with side effect on CPU cache, e.g. [prefetcht1](#), loads data into L2 cache on Intel® Core i7.



Predictable (Sequential) File Access

Likewise, there is limited support to **inform the OS file system buffer** that future block references will be regular:

- Use `advise(2)` to tune the OS's prefetching and caching strategy: “read/writes will be sequential (random)”, “blocks will definitely (not) be needed”, etc.:

```
/* map file into memory */  
map = mmap(NULL, size, PROT_READ, MAP_SHARED, fd, 0);  
/* advise the OS that file access will be sequential */  
advise(map, size, MADV_SEQUENTIAL);
```

- **N.B.:** PostgreSQL asynchronously prefetches buffer pages via `PrefetchBuffer()`. Also see extension `pg_prewarm`.