DB 2

06 – Buffering and Caching

Summer 2018

Torsten Grust
Universität Tübingen, Germany

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	≈ 100 ns	4 min	4–128 GB
SSD	50–150 µs	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: $|\text{block accesses}| \ll |\text{memory accesses}|$ 👍

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 ⇒ 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 PgSQL *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** (≡ set of typical queries submitted to the DBMS) contains repeated data references, close in time?

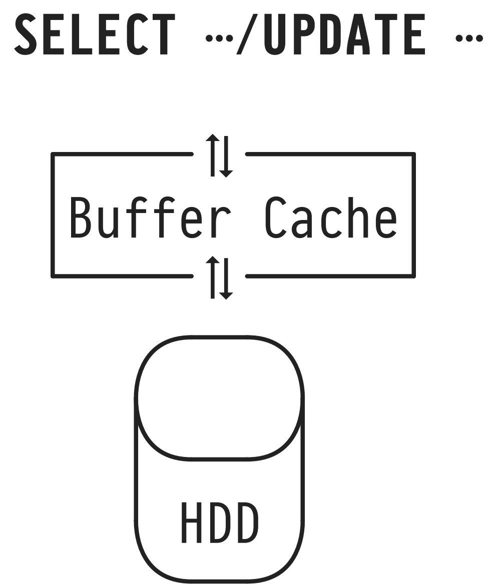
```
◷=t₀:    SELECT t.a, t.b FROM ternary AS t;
◷=t₀+Δt: SELECT s.a, s.c FROM ternary AS s;
 ⋮
```

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.



Show size of buffers, number of page entries in buffer:

```
db2=# show shared_buffers;
```

shared_buffers
128MB

◀ 128 MB / 8 KB per page: 16384 entries (minimum size: 16 entries ≡ 128 KB)

```
db2=# CREATE EXTENSION IF NOT EXISTS pg_buffercache;
db2=# SELECT COUNT(*) FROM pg_buffercache;
```

count
16384

```
db2=# set shared_buffers="128KB";  ◀ at heart of DB server, thus: edit postgresql.conf, then restart DB server
ERROR:  parameter "shared_buffers" cannot be changed without restarting the server
```

Buffer Cache Interface (API)

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

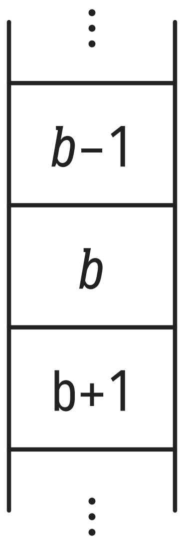
```
⟨b,m⟩ ← ReadBuffer(table, block);
  /* now may access 8 KB page starting at address m */
  ⋮
if (page at m has been written to)
 ⌊ MarkBufferDirty(b);
  ⋮
ReleaseBuffer(b);
  /* accesses to address m illegal from here on */
```

- 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 is current utility for the DBMS:

buffer pages



buffer descriptor

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

- 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;                              /* ❶ */
  | if (there is no such free v)
  | | v ← FindVictimBufferPage();                      /* ❷ */
  | | 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/altered):

```
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 ref_count(b) > 0, b is called **pinned**. If 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 view[3] 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│
│   ⋮                                                    │
```

Demonstrate dynamic buffer behavior using pg_buffercache.

❶ Create table ternary_100k(a,b,c)

```
DROP TABLE IF EXISTS ternary_100k;
CREATE TABLE ternary_100k (a int NOT NULL, b text NOT NULL, c float);
INSERT INTO ternary_100k(a, b, c)
  SELECT i,
         md5(i::text),
         log(i)
  FROM   generate_series(1, 100000, 1) AS i;
```

❷ ⚠ Restart PgSQL server to flush the buffer cache (STOP/START in Postgres.app menu)

❸ Check for relfilenode and # of pages for ternary_100k

```
SELECT c.relfilenode, c.relpages
FROM   pg_class AS c
WHERE  c.relname = 'ternary_100k';
```

relfilenode	relpages
88527	935

❹ Check that buffer cache holds no pages of ternary_100k

```
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88527;
```

bufferid	relblocknumber	isdirty	usagecount

❺ Scan all pages of ternary_100k: buffer cache misses for all pages

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT t.*
  FROM   ternary_100k AS t;
```

QUERY PLAN

Seq Scan on public.ternary_100k t (cost=0.00..1935.00 rows=100000 width=45) (actual time=0.030..27.838 rows=100000 loops=1)
  Output: a, b, c
  Buffers: shared read=935 ⬅ read ≡ I/O ≡ all buffers missed
Planning time: 0.467 ms

```
Execution time: 37.634 ms
```

❻ Check buffer cache for pages of ternary_100k: all pages in, not dirty, usagecount = 1

```
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88527;
```

bufferid	relblocknumber	isdirty	usagecount
269	0	f ⬅	1 ⬅
270	1	f	1
[...]			
1202	933	f	1
1203	934	f	1

(935 rows) ⬅

❼ Scan all pages of ternary_100k: buffer cache hits for all pages

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT t.*
  FROM   ternary_100k AS t;
```

QUERY PLAN
Seq Scan on public.ternary_100k t (cost=0.00..1935.00 rows=100000 width=45) (actual time=0.019..14.253 rows=100000 loops=1)
Output: a, b, c
Buffers: shared hit=935 ⬅ all buffers hit
Planning time: 0.064 ms
Execution time: 23.474 ms ⬅ may see a runtime improvement here

❽ Check buffer cache for pages of ternary_100k: all pages in, not dirty, usagecount = 2

```
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88527;
```

bufferid	relblocknumber	isdirty	usagecount
269	0	f ⬅	2 ⬅
270	1	f	2
[...]			

❾ Scan all pages of ternary_100k with a < 10: buffer cache hits for ALL pages (no index/ordering)

QUIZ: How many pages missed/hit? Usage counts?

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT t.*
  FROM   ternary_100k AS t
  WHERE  t.a < 100;
```

QUERY PLAN
Seq Scan on public.ternary_100k t (cost=0.00..2185.00 rows=94 width=45) (actual time=0.021..20.482 rows=99 loops=1)
Output: a, b, c
Filter: (t.a < 100) ⬅ filtering
Rows Removed by Filter: 99901 ⬅ (→ later in course)
Buffers: shared hit=935 ⬅ all buffers hit
Planning time: 0.225 ms
Execution time: 20.536 ms

🔟 Check buffer cache for pages of ternary_100k: all pages in, not dirty, usagecount = 3

```
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88527;
```

bufferid	relblocknumber	isdirty	usagecount
269	0	f ⬅	3 ⬅
270	1	f	3

[...]

1️⃣1️⃣ Now update a row in ternary_100k. START TRANSACTION such that we can observe things while they are in progress. ⚠️ NO TYPOS or the TX will abort!

Update row with a = 10: buffer cache hits for all pages, two pages dirty

```
START TRANSACTION;

EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  UPDATE ternary_100k
  SET    c = -1
  WHERE  a = 10;   ⬅  affected row on block 0 of ternary_100k
```

QUERY PLAN

```
Update on public.ternary_100k  (cost=0.00..2185.00 rows=1 width=51) (actual time=20.257..20.257 rows=0 loops=1)
  Buffers: shared hit=937 read=1 dirtied=2 ◀ the UPDATE has written to two pages
  ->  Seq Scan on public.ternary_100k  (cost=0.00..2185.00 rows=1 width=51) (actual time=0.019..19.886 rows=1 loops=1)
        Output: a, b, '-1'::double precision, ctid
        Filter: (ternary_100k.a = 10)
        Rows Removed by Filter: 99999
        Buffers: shared hit=935 ◀ all buffers looked at (any may contain a row with a = 10)
Planning time: 0.108 ms
Execution time: 20.650 ms
```

⓬ Check buffer cache for pages of ternary_100k: pages of old row and new row version dirty

```sql
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88527;
```

bufferid	relblocknumber	isdirty	usagecount	
269	0	t ◀	4	◀ block 0 of ternary_100k carries old version of row
270	1	f	4	
271	2	f	4	
[...]				
1203	934	t ◀	5	◀ block 934 has received the updated row
1209	0	f	1	◀ clean copy of block 0? (FIXME)

⓭ Check page contents of pages for old and new row version: see updated row slots

```sql
SELECT t_ctid, lp, lp_off, lp_len, t_xmin, t_xmax,
 (t_infomask::bit(16)  & b'0010000000000000')::int::bool AS "updated row?",
 (t_infomask2::bit(16) & b'0100000000000000')::int::bool AS "has been HOT updated?"
 FROM heap_page_items(get_raw_page('ternary_100k', 0));
```

t_ctid	lp	lp_off	lp_len	t_xmin	t_xmax	updated row?	has been HOT updated?	
(0,1)	1	8120	72	12763	0	f	f	
(0,2)	2	8048	72	12763	0	f	f	
(0,3)	3	7976	72	12763	0	f	f	
(0,4)	4	7904	72	12763	0	f	f	
(0,5)	5	7832	72	12763	0	f	f	
(0,6)	6	7760	72	12763	0	f	f	
(0,7)	7	7688	72	12763	0	f	f	
(0,8)	8	7616	72	12763	0	f	f	
(0,9)	9	7544	72	12763	0	f	f	
(934,63)	10	7472	72	12763	12766	f	f	◀ row has been updated (old, invisible, new version on page 934)
[...]								

```
SELECT t_ctid, lp, lp_off, lp_len, t_xmin, t_xmax,
 (t_infomask::bit(16)  & b'0010000000000000')::int::bool AS "updated row?",
 (t_infomask::bit(16)  & b'0000000100000000')::int::bool AS "updating TX committed?"
 FROM heap_page_items(get_raw_page('ternary_100k', 934));
```

t_ctid	lp	lp_off	lp_len	t_xmin	t_xmax	updated row?	updating TX committed?
(934,1)	1	8120	72	12763	0	f	t
(934,2)	2	8048	72	12763	0	f	t
(934,3)	3	7976	72	12763	0	f	t
[...]							
(934,61)	61	3800	72	12763	0	f	t
(934,62)	62	3728	72	12763	0	f	t
(934,63)	63	3656	72	12766	0	t ◀	f ◀

◀ new row version (updating TX has not committed yet)

```
COMMIT;
```

14 Check buffer cache for pages of ternary_100k: all pages in, all pages not dirty

```
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88527;
```

bufferid	relblocknumber	isdirty	usagecount	
269	0	f	5	◀ page has been accessed (for COMMIT)
270	1	f	4	
[...]				
1202	933	f	4	
1203	934	f	5	◀ page has been accessed (for COMMIT), should be 6 (but usage_count always ≤ BM_MAX_USAGE_COUNT ≡ 5)
1209	0	f	1	

15 Scan all pages of ternary_100k: buffer cache hits for all pages, but two pages dirty (after a read-only SCAN!?)

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT t.*
  FROM   ternary_100k AS t;
```

QUERY PLAN
Seq Scan on public.ternary_100k t (cost=0.00..1935.00 rows=100000 width=45) (actual time=0.038..16.616 rows=100000 loops=1)
Output: a, b, c

```
  Buffers: shared hit=935 dirtied=2 ◀ Why does a read-only scan dirty pages?
Planning time: 0.052 ms
Execution time: 24.687 ms
```

⓰ Check buffer cache for pages of ternary_100k: pages of old row and new row version dirty
- page with old row version: old row version marked as available for VACUUM
- page with new row version: bit xmin_committed of new row version is set (≡ updating TX has now committed)

```
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88527 AND b.isdirty;
```

bufferid	relblocknumber	isdirty	usagecount
269	0	t	5
1203	934	t	5

```
SELECT t_ctid, lp, lp_off, lp_len, t_xmin, t_xmax
FROM   heap_page_items(get_raw_page('ternary_100k', 0));
```

t_ctid	lp	lp_off	lp_len	t_xmin	t_xmax
(0,1)	1	8120	72	12763	0
(0,2)	2	8048	72	12763	0
(0,3)	3	7976	72	12763	0
(0,4)	4	7904	72	12763	0
(0,5)	5	7832	72	12763	0
(0,6)	6	7760	72	12763	0
(0,7)	7	7688	72	12763	0
(0,8)	8	7616	72	12763	0
(0,9)	9	7544	72	12763	0
□	10	0	0	□	□
(0,11)	11	7472	72	12763	0

◀ row marked available for VACUUM

[...]

```
SELECT t_ctid, lp, lp_off, lp_len, t_xmin, t_xmax,
       (t_infomask::bit(16)  & b'0000000100000000')::int::bool AS "updating TX committed?"
FROM   heap_page_items(get_raw_page('ternary_100k', 934));
```

t_ctid	lp	lp_off	lp_len	t_xmin	t_xmax	updating TX committed?
(934,1)	1	8120	72	12763	0	t
(934,2)	2	8048	72	12763	0	t

[...]

(934,62)	62	3728	72	12763	0	t
(934,63)	63	3656	72	12766	0	t ◀

◀ updating TX has committed now (was f above)

17 After a forced CHECKPOINT, all buffers are synced with disk image

```
CHECKPOINT;

SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88555 AND b.isdirty;
```

bufferid	relblocknumber	isdirty	usagecount

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 if non-empty, remove its head slot v. Pick v (see ❶ 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 …).

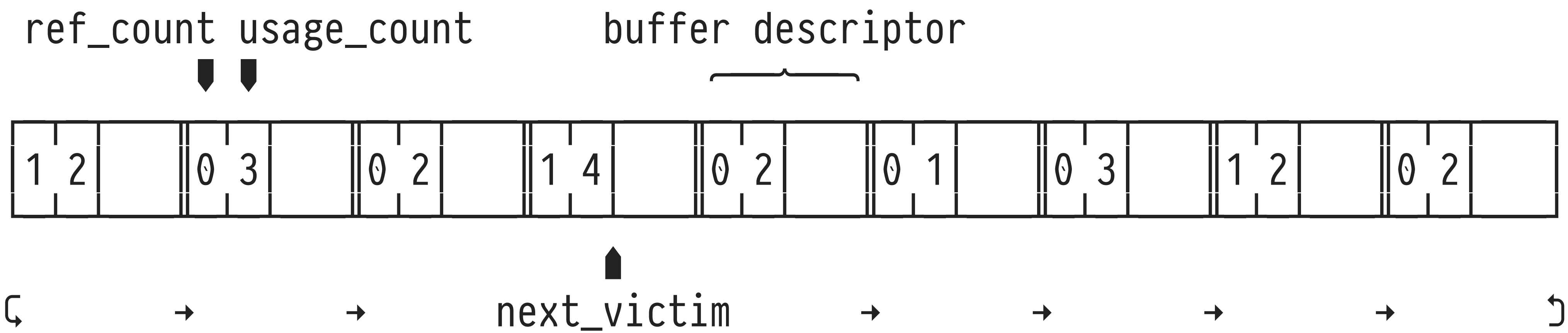
2. If free list is empty, use the **buffer replacement policy** to identify a **victim page** v. Pick v (see ❷).

A Replacement Policy: Clock Sweep (≈ 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:**

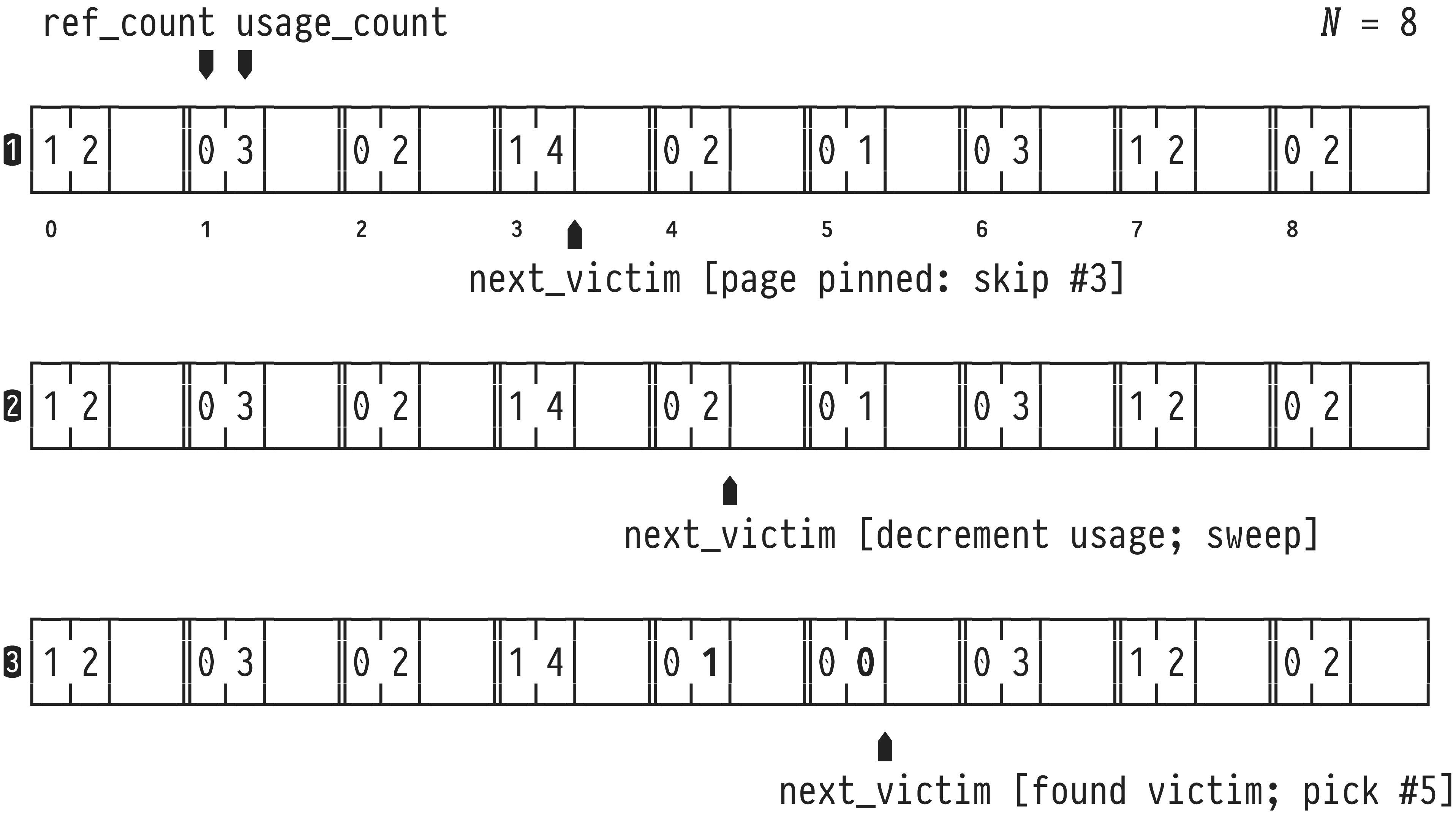
- Arrange buffer descriptors in a *circular* array (“clock”).
- Repeatedly “sweep” pointer next_victim through array:



```
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



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

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

1. See "*The LRU-k Page Replacement Algorithm for Database Disk Buffering*" (E.O'Neil et al.), SIGMOD 1993, Example 1.1.

   We *should* buffer the 100 index pages only since their probability of being re-referenced is 0.005 (once in each 200 page references). Probability of a page of R being re-referenced is only 0.00005 (once in each 20000 page refs). ⇒ Value of an index page is 100× the value of a page of R.

   With LRU, pages in the buffer will be the 100 most recently referenced ones. This will be 50 pages of I and 50 pages of R. :-(

2. The sequential scan will “swamp” the buffer with references for new/unseen pages (i.e. buffer misses) and will drive out pages needed by the other transactions — although the sequentially scanned pages are only used once. :-(

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:** ✎

Experiment: How does PostgreSQL react to sequential scan queries that would swamp the entire buffer?

❶ Reduce buffer size to simulate that buffer space is a scarce resource:

```
$ subl /Users/grust/Library/Application\ Support/Postgres/var-10/postgresql.conf

[...]
shared_buffers = 1MB        # FIXME ◀
#shared_buffers = 128MB     # min 128kB
          # (change requires restart)
[...]
```

❷ ⚠ Restart PgSQL server to flush the buffer cache (STOP/START in Postgres.app menu)

❸ Check resulting buffer size (128 slots, too few to hold table ternary_100k):

```
show shared_buffers;
```

shared_buffers
1MB

```
SELECT COUNT(*) FROM pg_buffercache;
```

count
128

❹ Check size of table ternary_100k (will swamp small buffer):

```
SELECT c.relfilenode, c.relpages
FROM   pg_class AS c
WHERE  c.relname = 'ternary_100k';
```

relfilenode	relpages
88555	935

```
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88555;
```

bufferid	relblocknumber	isdirty	usagecount

5 Perform sequential scan on ternary_100k:

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
  SELECT *
  FROM   ternary_100k;
```

QUERY PLAN

Seq Scan on public.ternary_100k (cost=0.00..1935.00 rows=100000 width=45) (actual time=0.297..23.086 rows=100000 loops=1)
  Output: a, b, c
  Buffers: shared read=935 ◀
Planning time: 2.391 ms
Execution time: 31.422 ms

6 Check buffer contents: only 16(!) pages have been used

```
SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
FROM   pg_buffercache AS b
WHERE  b.relfilenode = 88555;
```

bufferid	relblocknumber	isdirty	usagecount	
6	921	f	1	
9	922	f	1	
10	923	f	1	
11	924	f	1	
12	925	f	1	
15	926	f	1	
16	927	f	1	
67	928	f	1	
77	929	f	1	
81	930	f	1	
97	931	f	1	
100	932	f	1	
105	933	f	1	
106	➡ 934	f	1	ring buffer END
107	➡ 919	f	1	ring buffer START
112	920	f	1	

Ring buffer holds final pages of sequential scan (919..934).

7 ⇒ A new sequential scan will see 935-16 = 919 buffer misses:

```
EXPLAIN (VERBOSE, ANALYZE, BUFFERS)
```

```
  SELECT *
  FROM   ternary_100k;
```

QUERY PLAN
Seq Scan on public.ternary_100k (cost=0.00..1935.00 rows=100000 width=45) (actual time=0.066..20.673 rows=100000 loops=1)
Output: a, b, c
Buffers: shared hit=16 read=919
Planning time: 0.066 ms
Execution time: 30.785 ms

❽ Re-check for buffer contents: some pages may show a usage_count of 0 (⇒ potential victims)

```
 SELECT b.bufferid, b.relblocknumber, b.isdirty, b.usagecount
 FROM   pg_buffercache AS b
 WHERE  b.relfilenode = 88555;
```

[...]

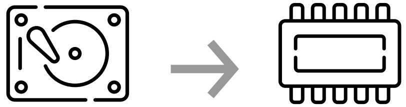
❾ ⚠ Reset shared_buffers in config file to original value (128 MB). Restart PgSQL server.

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 $\frac{1}{4}$ × 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 $\leqslant \frac{1}{8}$ × 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** (≡ set of typical queries submitted to the DBMS) contains repeated data references, close in time?

```
◷=t0:    SELECT t.a, t.b FROM ternary AS t;
◷=t0+Δt: 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 ⇒ 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 ⇒ MMDBMS relies on **built-in CPU data cache replacement policies.**

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

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

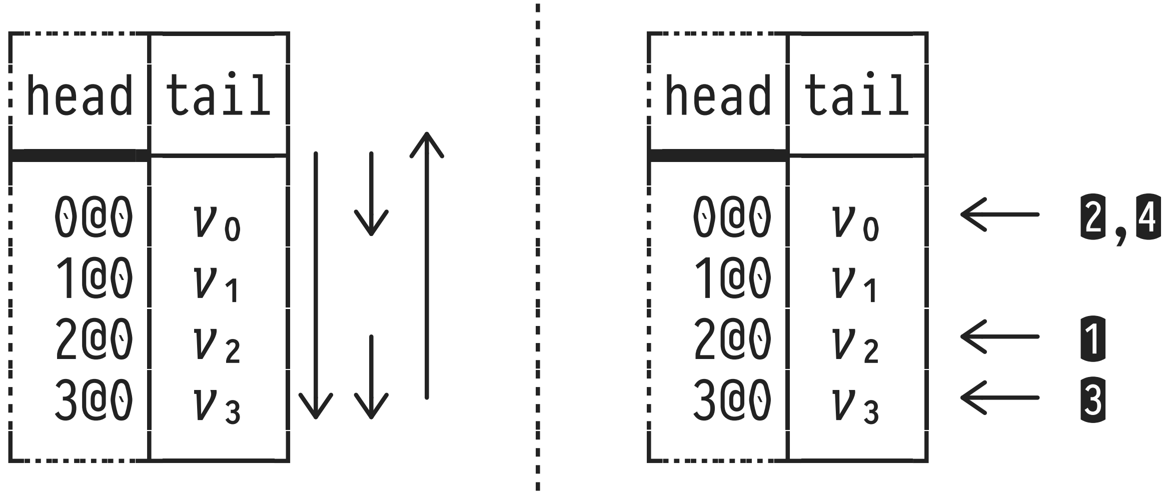
Excerpt of System Latencies (at Human Scale)

- 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



- 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.

Demonstrate effects of automatic/explicit memory prefetching while scanning the elements of an `int` data vector with 16M elements.

1. Linear scan (automatic prefetching)
2. Random bounce-around (unpredictable, no prefetching)
3. Random bounce w/ explicit prefetching (simulates knowledge about future needed addresses by using a buffer `locations[]` of upcoming access locations)

See C file `mat/prefetch.c`, compile via `cc -O2 prefetch.c -o prefetch`. Sample output:

```
$ ./prefetch
time (linear): 4288µs (sum = 75497460)
time (bounce): 431747µs (sum = 75515095)
time (bounce with prefetch): 175740µs (sum = 75515095)
```

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys time.h="">
#include <stdint.h>

#define MICROSECS(t) (1000000 * (t).tv_sec + (t).tv_usec)

/* process int vector of 16M elements >> L2 cache of 256 kB */
#define SIZE (16 * 1024 * 1024)

/* prefetch how many iterations ahead? */
#define LOOKAHEAD 128


/* linearly scan the vector, add elements
   (no manual prefetching, but CPU will detect the linear memory access
   pattern and automatically issue prefetching operations) */
int linear(int *vector)
{
  int sum = 0;

  for (int i = 0; i < SIZE; i = i + 1) {
    sum = sum + vector[i];
  }

  return sum;
}

/* randomly bounce around the vector (no manual or CPU prefetching) */
int bounce(int *vector)
{
  int sum = 0;
```

```
  /* initialize deterministic random number sequence */
  srand(42);

  for (int i = 0; i < SIZE; i = i + 1) {
    sum = sum + vector[rand() % SIZE];
  }

  return sum;
}

/* randomly bounce around the vector, but explicitly prefetch the address
   needed in LOOKAHEAD iterations from now: hide memory access latency */
int prefetching_bounce(int *vector)
{
  int sum = 0;
  int locations[LOOKAHEAD];

  /* initialize deterministic random number sequence */
  srand(42);

  /* prime the buffer of prefetching addresses neeed in future iterations
     (simulates that we know about our future memory access pattern) */
  for (int l = 0; l < LOOKAHEAD; l = l + 1)
    locations[l] = rand() % SIZE;   /* can also prefetch these — but makes no measurable difference */

  for (int i = 0, l = 0; i < SIZE; i = i + 1) {
      sum = sum + vector[locations[l]];

      locations[l] = rand() % SIZE;
      /* prefetch memory needed in LOOKAHEAD iterations from now */
      __builtin_prefetch(&vector[locations[l]]);
      l = (l + 1) % LOOKAHEAD;
  }

  return sum;
}

int main()
{
  int *vector, sum;
  struct timeval t0, t1;
  unsigned long duration;

  vector = malloc(SIZE * sizeof(int));
  assert(vector);
  for (int i = 0; i < SIZE; i = i + 1)
```

```
    vector[i] = i % 10;

  /* ❶ linear scan */
  gettimeofday(&t0, NULL);
  sum = linear(vector);
  gettimeofday(&t1, NULL);

  duration = MICROSECS(t1) - MICROSECS(t0);
  printf("time (linear): %luµs (sum = %d)\n", duration, sum);

  /* ❷ bounce, no prefetch */
  gettimeofday(&t0, NULL);
  sum = bounce(vector);
  gettimeofday(&t1, NULL);

  duration = MICROSECS(t1) - MICROSECS(t0);
  printf("time (bounce): %luµs (sum = %d)\n", duration, sum);

  /* ❸ bounce with prefetch */
  gettimeofday(&t0, NULL);
  sum = prefetching_bounce(vector);
  gettimeofday(&t1, NULL);

  duration = MICROSECS(t1) - MICROSECS(t0);
  printf("time (bounce with prefetch): %luµs (sum = %d)\n", duration, sum);

  return 0;
}
```

Predictable (Sequential) File Access

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

- Use madvise(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 */
  madvise(map, size, MADV_SEQUENTIAL);
  ```

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

- PostgreSQL routine `PrefetchBuffer()`: see `src/backend/storage/buffer/bufmgr.c`
- PostgreSQL's `pg_prewarm` extension: https://www.postgresql.org/docs/10/static/pgprewarm.html

Demonstrate the effect of `madvise(2)` when informing the OS, that a 4.3 GB `mmap()`ed file will be read sequentially.

- Compile via `cc -O2 madvise.c -o madvise`
- Run via `sudo purge; and ./madvise`
- Can use `Activity Monitor.app` to control the size of *Cached Files* (bottom display in *Memory* tab)

See file `mat/madvise.c`, (de-)activate `madvise()` via `#define ADVISE`:

```
#include <sys mman.h="">
#include <sys stat.h="">
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include <sys time.h="">

#define MICROSECS(t) (1000000 * (t).tv_sec + (t).tv_usec)


/* compile via cc -O2 madvise.c -o madvise
   use `sudo purge' to clear OS buffer cache */

/*            | no madvise()  | with madvise()
 * cold cache | 17038847µs    | 15819001µs
 */

#define ADVISE 1
/* file has 4627922661 bytes ≅ 4.3 GB */
#define PATH "/Users/grust/Music/iTunes/iTunes Music/Movies/01 The LEGO Batman Movie (1080p HD).m4v"

/* scan the file, do pseudo work */
int scan(char *m, off_t size)
{
  int sum = 0;

  for (off_t i = 0; i < size; i = i+1) {
    sum = sum + *m;
    m = m + 1;
  }

  return sum;
}
```

```c
int main()
{
  int sum;
  int fd;
  off_t size;
  void *map;
  struct stat status;

  struct timeval t0, t1;
  unsigned long duration;

  /* open file and determine its size in bytes */
  fd = open(PATH, O_RDONLY);
  assert(fd >= 0);
  assert(stat(PATH, &status) == 0);
  size = status.st_size;

  /* map file into memory */
  map = mmap(NULL, size, PROT_READ, MAP_SHARED, fd, 0);
  assert(map != MAP_FAILED);

  close(fd);

#if ADVISE
  /* advise the OS that file access will be sequential */
  assert(madvise(map, size, MADV_SEQUENTIAL) >= 0);
#endif

  gettimeofday(&t0, NULL);
  sum = scan(map, size);
  gettimeofday(&t1, NULL);

  duration = MICROSECS(t1) - MICROSECS(t0);
  printf("time: %luµs (sum = %d)\n", duration, sum);

  return 0;
}
```