DB 2

02 – Unary Table Storage

Summer 2018

Torsten Grust
Universität Tübingen, Germany

1 ┆ Q_1 — The Simplest SQL Probe Query

Let us send the very first **SQL probe** Q_1. It doesn't get much simpler than this:[1]

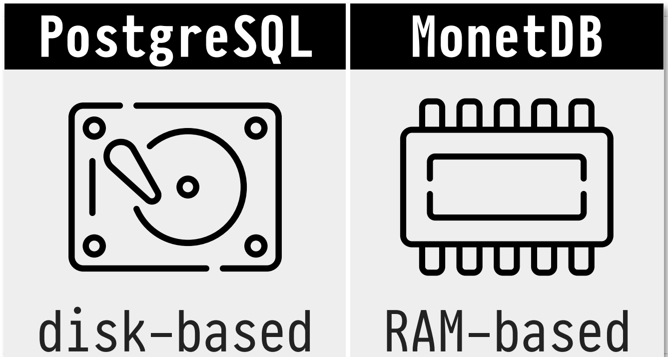
```
SELECT u.*            -- * ≡ access all columns of row u
FROM   unary AS u
```

Retrieve all rows (in some arbitrary order) and all columns of table unary. For now, we assume that unary has a single column of type int.

[1] In PostgreSQL, there is an equivalent even more compact form for Q_1: TABLE unary.

PostgreSQL vs. MonetDB

In the sequel, we use the marks below whenever we dive deep and discuss **material that is specific to a particular DBMS:**



- ⚠️ SQL syntax and semantics may (subtly) differ between both systems. This is a cruel fact of the current state of SQL and its implementations. Cope with it.

Aside: Populating Tables via generate_series()

Create and populate table unary as follows:

```sql
CREATE TABLE unary (a int);

INSERT INTO unary(a)
  SELECT i
  FROM   generate_series(1, 100, 1) AS i;
--
--                      start/end/step of sequence
```

- Table function generate_series(*s*,*e*,Δ) enumerates values[2] from *s* to *e* (inclusive) with step Δ (default Δ = 1).

[2] *s* and *e* both of type int, numeric, or timestamp (for the latter, Δ needs to have type interval).

Demonstrate generate_series():

```
db2=# SELECT i
      FROM   generate_series('now'::timestamp, 'now'::timestamp + '1 hour', '1 min') AS i;
```

SQL probe query Q1

```
-- Create unary table
db2=# DROP TABLE IF EXISTS unary;  -- clean up from earlier runs
db2=# CREATE TABLE unary (a int);

db2=# INSERT INTO unary(a)
        SELECT i
        FROM   generate_series(1, 100, 1) AS i;

-- Q1: Retrieve all rows (in arbirary oder) and all columns of table unary
db2=# SELECT u.*
      FROM   unary AS u;

-- Equivalent to Q1
db2=# TABLE unary;


db2=# EXPLAIN VERBOSE
      SELECT u.*
      FROM   unary AS u;

db2=# EXPLAIN (VERBOSE, ANALYZE)
      SELECT u.*
      FROM   unary AS u;
```

Using **EXPLAIN** on Q_1

Let us try to understand the evaluation of Q_1:

```
db2=# EXPLAIN VERBOSE
       SELECT u.*                  -- ⎱ Q1 as before
       FROM   unary AS u;          -- ⎰

 ┌──────────────────────────────────────────────────────────────┐
 │                          QUERY PLAN                          │
 ├──────────────────────────────────────────────────────────────┤
 │ Seq Scan on public.unary  (cost=0.00..2.00 rows=100 width=4) │
 │   Output: a                                                  │
 └──────────────────────────────────────────────────────────────┘
(2 rows)

db2=# █
```

Using EXPLAIN

Show the **query evaluation plan** for SQL query ‹Q›:

```
EXPLAIN ‹opt› ‹Q›
EXPLAIN (‹opt›, ‹opt›, ...) ‹Q›
```

‹opt› controls level of detail and mode of explanation:

‹opt›	Effect
VERBOSE	higher level of detail
ANALYZE	**evaluate** the query, then produce explanation
FORMAT {TEXT|JSON|XML}	output format (default: TEXT)
⋮	⋮

⚠️ Without ANALYZE, ‹Q› is *not* evaluated ⇒ output is based on the DBMS's **best guess** of how the plan will perform.

2 ┆ Sequential Scan (Seq Scan)

```
QUERY PLAN
Seq Scan on public.unary  (cost=0.00..2.00 rows=100 width=4)
  Output: a ◀──────────── type int ────────────┘
```

- **Seq Scan:** Sequentially scan the entire **heap file** of table unary, read rows in **some order**, emit all rows.

- Seq Scan returns rows in arbitrary order (*not*: insertion order) that may change from execution to execution. Meets bag semantics of the tabular data model (→ DB1).

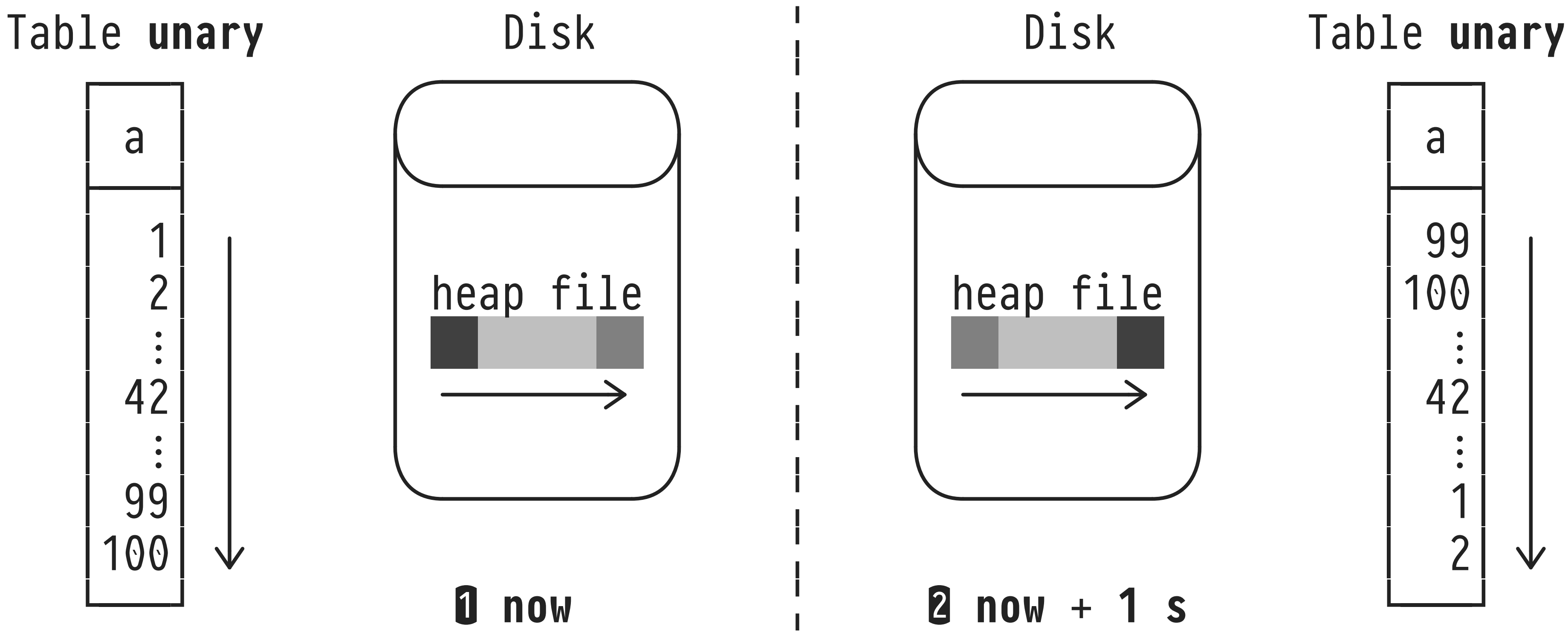
Heap Files

The rows of a table are stored in its **heap file**, a plain row container that can grow/shrink dynamically.

- Row insertion/deletion simple to implement and efficient, no complex file structure to maintain.
- Supports **sequential scan** across entire file.
- **No support for finding rows by column value** (no associative row access). If we need value-based row access, additional data maps (indexes) need to be created and maintained.

Heap Files and Sequential Scan

The DBMS may reorganize (e.g., compact or “vacuum”) a table's heap file at any time ⇒ no guaranteed row order:



Heap File ≡ OS File

Most DBMSs implement **heap files** in terms of **regular files on the operating system's file system** (also: raw storage).

- Files held in a DBMS-controlled directory. In PostgreSQL:

```
db2=# show data_directory;
```

data_directory
/Users/grust/Library/App⋯/Postgres/var-10

- DBMS enjoys OS FS services (e.g., backup, authorization).

```
db2=# show data_directory;
```

data_directory
/Users/grust/Library/Application Support/Postgres/var-10

```
(1 row)

$ cd '/Users/grust/Library/Application Support/Postgres/var-10'
$ cd base
$ ls
total 0
drwx------  295 grust  staff  10030 Oct  5 16:14 1
drwx------  295 grust  staff  10030 Oct  5 16:14 13266
drwx------  296 grust  staff  10064 Mar  5 09:59 13267
drwx------  295 grust  staff  10030 Nov 29 17:10 16385
drwx------  490 grust  staff  16660 Mar  5 09:58 16386
drwx------  391 grust  staff  13294 Mar  5 09:59 24576
drwx------  356 grust  staff  12104 Jan  3 10:28 50577
drwx------  297 grust  staff  10098 Mar  5 15:15 71857
drwx------    2 grust  staff     68 Feb 26 17:19 pgsql_tmp

db2=# SELECT oid, datname FROM pg_database;
```

oid	datname
13267	postgres
16385	grust
1	template1
13266	template0
16386	scratch
24576	lego
50577	provenance
71857	db2

```
(8 rows)

$ cd 71857
$ ls
112             13124           2604_vm         2618_vm         2685            2839            3456            3603_vm
113             13126           2605            2619            2686            2840            3456_fsm        3604
[...]

db2=# SELECT relfilenode, relname FROM pg_class ORDER BY relname DESC;
```

relfilenode	relname

```
│              │                                            │
├──────────────┼────────────────────────────────────────────┤
│        13194 │ views                                      │
│        13190 │ view_table_usage                           │
│        13186 │ view_routine_usage                         │
│        13182 │ view_column_usage                          │
│        13250 │ user_mappings                              │
│        13246 │ user_mapping_options                       │
│        13178 │ user_defined_types                         │
│        13171 │ usage_privileges                           │
│        71878 │ unary                                      │
│        13164 │ udt_privileges                             │
│        13160 │ triggers                                   │
[...]

$ l 71878
-rw-------  1 grust  staff  8192 Mar  5 15:15 71878
$ hexdump -C 71878
00000000  01 00 00 00 c0 a8 68 d3  00 00 00 00 a8 01 80 13  |......h.........|
00000010  00 20 04 20 00 00 00 00  e0 9f 38 00 c0 9f 38 00  |. . ......8...8.|
00000020  a0 9f 38 00 80 9f 38 00  60 9f 38 00 40 9f 38 00  |..8...8.`.8.@.8.|
00000030  20 9f 38 00 00 9f 38 00  e0 9e 38 00 c0 9e 38 00  | .8...8...8...8.|
00000040  a0 9e 38 00 80 9e 38 00  60 9e 38 00 40 9e 38 00  |..8...8.`.8.@.8.|
00000050  20 9e 38 00 00 9e 38 00  e0 9d 38 00 c0 9d 38 00  | .8...8...8...8.|
[...]

db2=# CREATE TABLE "unary'" (a text);
db2=# INSERT INTO "unary'" VALUES ('Yoda'), ('Han Solo'), ('Leia'), ('Luke');
db2=# TABLE "unary'";
┌──────────┐
│    a     │
├──────────┤
│ Yoda     │
│ Han Solo │
│ Leia     │
│ Luke     │
└──────────┘
db2=# SELECT relfilenode, relname FROM pg_class WHERE relname = 'unary''';
┌─────────────┬─────────┐
│ relfilenode │ relname │
├─────────────┼─────────┤
│       71881 │ unary'  │
└─────────────┴─────────┘

$ l 71881
-rw-------  1 grust  staff  8192 Mar  6 12:36 71881
$ hexdump -C 71881 | grep -C5 Yoda
00001fa0  00 00 00 00 00 00 00 00  03 00 01 00 02 09 18 00  |................|
```

```
00001fb0  0b 4c 65 69 61 00 00 00  cf 30 00 00 00 00 00 00  |.Leia....0......|
00001fc0  00 00 00 00 00 00 00 00  02 00 01 00 02 09 18 00  |................|
00001fd0  13 48 61 6e 20 53 6f 6c  6f 00 00 00 00 00 00 00  |.Han Solo.......|
00001fe0  cf 30 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.0..............|
00001ff0  01 00 01 00 02 09 18 00  0b 59 6f 64 61 00 00 00  |.........Yoda...|
00002000
```

Row IDs and Heap File Locations

Heap files do not support value-based access. We can still **directly locate a row** via its **row identifier (RID):**

- RIDs are **unique** within a table. Even if two rows r_1, r_2 agree on all column values (in a key-less table), we still have $\text{RID}(r_1) \neq \text{RID}(r_2)$.

- $\text{RID}(r)$ **encodes the location** of row r in its table's heap file. No sequential scan is required to access r.

- If r is updated, $\text{RID}(r)$ remains stable.
  ⚠️ RIDs do *not* replace the relational key concept.[3]

[3] But see comments on free space management and VACUUM later on.

```
db2=# TRUNCATE unary;
db2=# INSERT INTO unary(a)
        SELECT i
        FROM   generate_series(1, 1000, 1) AS i;
db2=# SELECT relfilenode, relname FROM pg_class WHERE relname = 'unary';
```

relfilenode	relname
71889	unary

```
$ l 71889
-rw-------  1 grust  staff  40960 Mar  6 13:04 71889        # 40960 bytes / 8192 bytes/page = 5 pages
```

RIDs in PostgreSQL

RIDs are considered DBMS-internal and thus withheld from users. PostgreSQL externalizes RIDs via **pseudo-column** **ctid**:

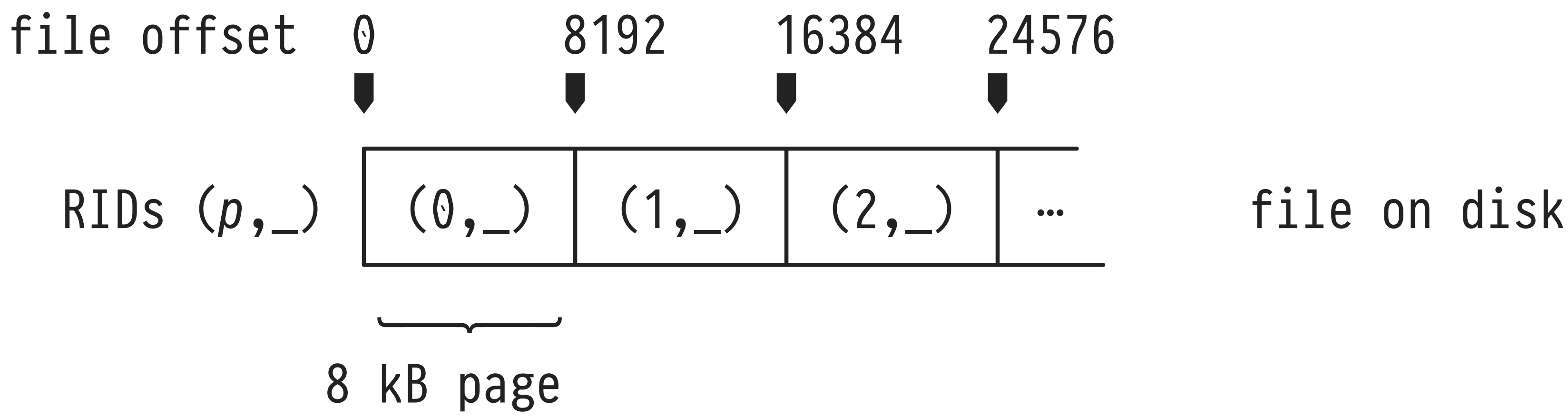
```
SELECT u.ctid, u.*
FROM   unary AS u;
```

ctid	a
(0,1)	1
(0,2)	2
⋮	⋮
(1,1)	227
(1,2)	228
⋮	⋮
(4,95)	999
(4,96)	1000

File Storage on Disk-Based Secondary Memory

A PostgreSQL RID is a pair **(‹page number›, ‹row slot›):**

- **Page number** *p* identifies a **contiguous block of bytes** in the file.
- **Page size** *B* is system-dependent and configurable. Typical values are in range 4–64 kB. PostgreSQL default: **8 kB.**



```
$ pg_controldata '/Users/grust/Library/Application Support/Postgres/var-10'
pg_control version number:            1002
Catalog version number:               201707211
Database system identifier:           6473429909854274770
[...]
Maximum data alignment:               8
Database block size:                  8192
Blocks per segment of large relation: 131072      131072 * 8 kB = 1 GB (split larger relations into multiple OS files)
[...]
Maximum length of identifiers:        64
Maximum columns in an index:          32
```

Block I/O on Disk-Based Secondary Memory 🖴

- Heap files are **read and written in units of 8 kB pages.**
  - Likewise, heap files grow/shrink by entire pages.

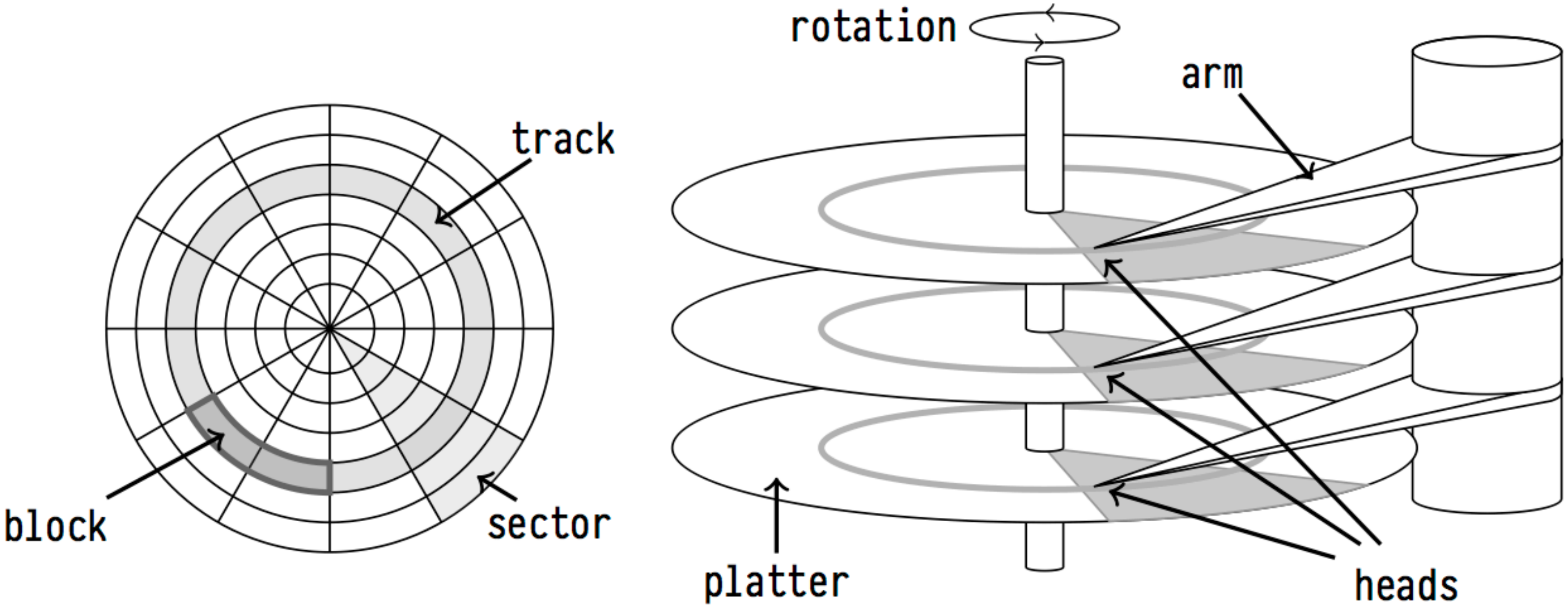
- This page-based access to heap files reflects the OS's mode of performing **disk input/output page-by-page.**
  - Terminology: DB🗨 **page** ≡ **block** 🗩OS

- ⚠️ Any disk I/O operation will read/write at least one block (of 8 kB). Disk I/O *never* moves individual bytes.

3 ┆ Rotating Magnetic Hard Disk Drives (HDDs)

Steadily rotating platters and read/write heads of a HDD

HDDs: Tracks, Sectors, Blocks



❶ **Seek** Stepper motor positions array of R/W heads over wanted **track**.

❷ **Rotate** Wait for wanted **sector** of blocks to rotate under R/W heads.

❸ **Transfer** Activate one head to read/write **block** data.

HDDs: Access Time

A HDD design that involves motors, mechanical parts, and thus inertia has severe implications on the **access time** t needed to read/write one block:

$$t = \underbrace{t_s}_{\text{seek time}} + \overbrace{t_r}^{\text{rotational delay}} + \underbrace{t_{tr}}_{\text{transfer time}}$$

- Amortize seek time and rotational delay by transferring one block at a time (**random block access**).
- Transfer a sequence of adjacent blocks: longer t_{tr} but, ideally, $t_s = t_r = 0$ ms (**sequential block access**).

HDDs: Random Block Access Time

Feature	
HDD layout	4 platters, 8 r/w heads
average data per track	512 kB
capacity	600 GB
rotational speed	15000 min^{-1}
average seek time (t_s)	3.4 ms
track-to-track seek time	0.2 ms
transfer rate	≈ 163 MB/s

Data Sheet Seagate Cheetah 15K.7 HDD

- **Random access time** t for a single 8 kB block:
  - Average rotational delay t_r: $\frac{1}{2} \times (1/15000\ \text{min}^{-1}) = 2$ ms
  - Transfer time t_{tr}: 8 kB / (163 MB/s) = 0.0491 ms
  - $\Rightarrow t_s + t_r + t_{tr}$ = 3.4 ms + 2 ms + 0.05 ms = <u>5.45 ms</u>

HDDs: Sequential Block Access Time

Feature	
⋮	⋮
average data per track	512 kB
track-to-track seek time	0.2 ms
⋮	⋮

Data Sheet Seagate Cheetah 15K.7 HDD

- **Random access time** for 1000 blocks of 8 kB:
  - $1000 \times t_{tr} = 5.45$ s 👎
- **Sequential access time** to 1000 adjacent blocks of 8 kB:
  - 512 kB per track: 1000 blocks will span 16 tracks
  - $\Rightarrow t_s + t_r + 1000 \times t_{tr} + 16 \times 0.2$ ms $=$ <u>58.6 ms</u>

- Once we need to read more than 58.6 ms / 5450 ms = 1.07% of a file, we better **read the entire file sequentially.**

Solid State Disk Drives (SSDs)

SSDs rely on non-volatile flash memory and contain no moving/electro-mechanical parts:

- Non-volatility (battery-powered DRAM or NAND memory cells) ensures data persistence even on power outage.
- **No seek time, no rotational delay** ($t_s = t_r = 0$ ms), no motor spin-up time, no R/W head array jitter.
- Admits **low-latency random read access** to large data blocks (typical: 128 kB), however slow random writes.[4]

[4] Groups of data blocks need to be erased, then can be written again. Memory cells wear out after 10^4 to 10^5 write cycles $\Rightarrow$ SSDs use wear-leveling to spread data evenly across the device memory.

SSDs: Access Time

Feature	
device memory	NAND flash
capacity	1 TB
block size	128 kB
transfer rate	≈ 1.8 GB/s

Data Sheet Apple AP1024J SanDisk SSD

- **Random access time** to 1000 blocks of 8 kB:
  - Transfer time t_{tr}: 128 kB / (1.8 GB/s) = 0.06 ms
  - $1000 \times t_{tr}$ = <u>60 ms</u>
- **Sequential access time** to 1000 adjacent blocks of 8 kB:
  - $\lceil (1000 \times 8\text{ kB}) / 128\text{ kB} \rceil \times t_{tr}$ = <u>3.75 ms</u>

⚠️ Sequential still beats random I/O (by a smaller margin).

SSDs: Still a Disk? Already like RAM? (1)

Both SSDs and DRAM provide $t_s = t_r = 0$ ms. How do they compare regarding t_{tr} (i.e., transfer speed)?

- **SSD transfer speed test** (write 4 GB of zeroes):

```
$ cd /tmp
$ time dd if=/dev/zero of=bitbucket bs=1024k count=4096
4096+0 records in
4096+0 records out
4294967296 bytes transferred in 2.825247 secs
```

$$\underbrace{\text{4294967296 bytes transferred in 2.825247 secs}}_{\approx\ 1.4\ \text{GB/s}}$$

SSDs: Still a Disk? Already like RAM? (2)

- **DRAM transfer speed test** (write 4 GB of 64-bit values):

  1. Allocate memory area of 8 MB (> $\sum$ L1–L3 cache sizes)
  2. Repeatedly scan the area, writing 64-bit by 64-bit:

```
$ cc -Wall -O2 transfer.c -o transfer
$ ./transfer
time: 267956µs
```

$\underbrace{\texttt{time: 267956µs}}_{\approx\ 14.9\ \text{GB/s}}$

- Still faster: use SIMD instructions (r/w up to 256 bits) and multiple CPU cores (but: bus bandwidth is limited).

See file live/transfer.c:

```
#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)

/* overall size of memory to scan (4 GB) */
#define MEMSIZE (4L * 1024 * 1024 * 1024)

/* size of scan area (8 MB, exceeds L1-L3 cache sizes) */
#define SCANSIZE (8 * 1024 * 1024)

/* scan the memory, do pseudo work */
void scan(int64_t *mem)
{
  for (size_t loop = 0; loop < MEMSIZE / SCANSIZE; loop = loop+1) {
    for (size_t i = 0; i < SCANSIZE / sizeof(size_t); i = i+1) {
      mem[i] = 42;
    }
  }
}

int main()
{
  int64_t *area;

  struct timeval t0, t1;
  unsigned long duration;

  /* allocate scan area */
  area = (int64_t *)malloc(SCANSIZE);
  assert(area);

  gettimeofday(&t0, NULL);
  scan(area);
  gettimeofday(&t1, NULL);

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

  return 0;
}
```

Cache sizes (output of ./bandwidth-mac64, see http://zsmith.co/bandwidth.html):

```
This is bandwidth version 1.5.1.
Copyright (C) 2005-2017 by Zack T Smith.

This software is covered by the GNU Public License.
It is provided AS-IS, use at your own risk.
See the file COPYING for more information.

CPU family: GenuineIntel
CPU features: MMX SSE SSE2 SSE3 SSSE3 SSE4.1 SSE4.2 AES AVX AVX2 XD Intel64

Cache 0: L1 data cache,        line size 64,  8-ways,    64 sets, size 32k
Cache 1: L1 instruction cache, line size 64,  8-ways,    64 sets, size 32k
Cache 2: L2 unified cache,     line size 64,  4-ways,  1024 sets, size 256k
Cache 3: L3 unified cache,     line size 64, 16-ways,  4096 sets, size 4096k
[...]
```

Heads-Up: System Latencies

During the entire course, be aware and recall the typical **latencies** ("wait times") of a contemporary system:

Operation	Actual Latency ⌛	Human Scale 😐
CPU cycle	0.4 ns	1 s
L1 cache access	0.9 ns	2 s
L2 cache access	2.8 ns	7 s
L3 cache access	28 ns	1 min
RAM access	≈ 100 ns	4 min
SSD I/O	50–150 μs	1.5–4 days
HDD I/O	1–10 ms	1–9 months
Internet roundtrip (DE ↔ US)	90 ms	7 years

System Latencies (at Human Scale)

Many DB design decisions become a lot clearer in this light.

Heap Files: Free Space Management

Row updates and deletions may lead to heap file pages that are not 100% filled. New records could fill such “holes.”

- DBMS maintains a **free space map** (FSM) for each heap file, recording the (approximate) number of bytes available on each 8 kB page.

- Required FSM operations:
  1. Given a row of *n* bytes, which page *p* (in the vicinity) has sufficient free space to hold the row?
  2. Free space on page *p* has been reduced/enlarged by *n* bytes. Update the FSM.

5 ¦ Heap Files: Free Space Management

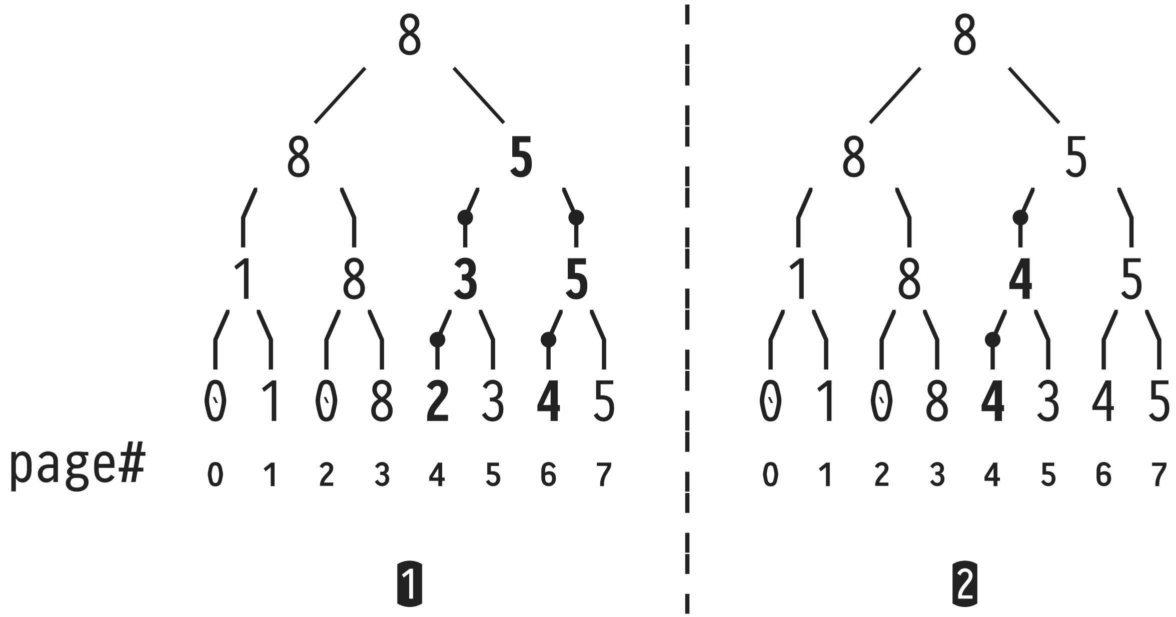
PostgreSQL maintains a **tree-shaped FSM** for each heap file:

```
                    8
                 /     \
              8           5
            /   \       /   \
           1     8     3     5     maximal space available in subtree
          / \   / \   / \   / \
         0   1 0   8 2   3 4   5   space available on page #7
page#    0   1 2   3 4   5 6   7
```

- Leaf nodes: space available in heap file page.[5]
- Inner nodes: maximal space found in this file (segment).

[5] PostgreSQL: space measured in 32 byte units (= 1/256 of a 8 kB page).

Heap Files: Free Space Management



1. Find a page with at least 4 available slots in the vicinity of page #4 (traverses **2↑3↑5↓5↓4** along).
2. Update page #4 to provide 4 available slots (traverses , updates **3** to max(3,4) = **4**, stops when max(4,5) = 5).

```
db2=# CREATE EXTENSION IF NOT EXISTS pg_freespacemap;

db2=# CREATE TABLE unary (a int);
db2=# INSERT INTO unary(a)
        SELECT i
        FROM   generate_series(1, 1000, 1) AS i;

db2=# show data_directory;
```

data_directory
/Users/grust/Library/Application Support/Postgres/var-10

```
db2=# SELECT oid, datname FROM pg_database WHERE datname = 'db2';
```

oid	datname
71857	db2

```
db2=# SELECT relfilenode, relname FROM pg_class WHERE relname = 'unary';
```

relfilenode	relname
80069	unary

```
$ cd '/Users/grust/Library/Application Support/Postgres/var-10/base'
$ cd 71857
$ ls -l 80069*
-rw-------  1 grust  staff  40960 Mar  8 09:50 80069
-rw-------  1 grust  staff  24576 Mar  8 09:50 80069_fsm ◀

db2=# VACUUM unary;
db2=# SELECT * FROM pg_freespace('unary');
```

blkno	avail
0	0
1	0
2	0
3	0
4	4704

◀ pages tightly packed, residual space on last heap file page

```
db2=# DELETE FROM unary AS u WHERE u.a  BETWEEN 400 AND 500;
```

```
db2=# VACUUM unary;
db2=# SELECT * FROM pg_freespace('unary');
┌───────┬───────┐
│ blkno │ avail │
├───────┼───────┤
│     0 │     0 │
│     1 │  1696 │ ⬅ available space in the middle of the heap file
│     2 │  1536 │
│     3 │     0 │
│     4 │  4704 │
└───────┴───────┘

db2=# INSERT INTO unary(a) VALUES (-1); ⬅ insert easy to detect sentinel
db2=# VACUUM unary;
db2=# SELECT * FROM pg_freespace('unary');
┌───────┬───────┐
│ blkno │ avail │
├───────┼───────┤
│     0 │     0 │
│     1 │  1664 │ ⬅ space on page #1 reduced (from 1696; 32 byte units)
│     2 │  1536 │
│     3 │     0 │
│     4 │  4704 │
└───────┴───────┘

db2=# TABLE unary;
┌─────┐
│  a  │
├─────┤
│   1 │
│   2 │
│   3 │
[...]
│ 398 │
│ 399 │
│  -1 │ ⬅ ⚠ table order (= heap file order) ≠ insertion order
│ 501 │
│ 502 │
[...]

db2=# VACUUM (VERBOSE, FULL) unary; ⬅ VACUUM FULL compactly rewrites heap file
db2=# SELECT * FROM pg_freespace('unary');
┌───────┬───────┐
│ blkno │ avail │
├───────┼───────┤
│     0 │     0 │
```

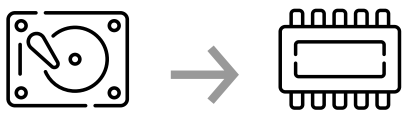
```
     1 |     0
     2 |     0
     3 |     0

db2=# SELECT relfilenode, relname FROM pg_class WHERE relname = 'unary';
 relfilenode | relname
-------------+--------
       80072 | unary    ⟵ new OS heap file (heap file rewritten, OS can reclaim file 80069)
```

Recall our very first **SQL probe** Q_1:

```
SELECT u.*          -- * ≡ access all columns of row s
FROM   unary AS u
```

Retrieve all rows (in some arbitrary order) and all columns of table unary. For now, the table has a **single column** of type int.



- How does **MonetDB** cope with Q_1?

Starting MonetDB (mserver5) in directory MonetDB/. Connect to database scratch. See MonetDB/README.md on how to setup database scratch.

```
$ mserver5 --dbpath=(pwd)/data/scratch --set monet_vault_key=(pwd)/data/scratch/.vaultkey
# MonetDB 5 server v11.27.13 "Jul2017-SP4"
# Serving database 'scratch', using 4 threads
# Compiled for x86_64-apple-darwin16.7.0/64bit with 128bit integers
# Found 16.000 GiB available main-memory.
# Copyright (c) 1993 - July 2008 CWI.
# Copyright (c) August 2008 - 2018 MonetDB B.V., all rights reserved
# Visit https://www.monetdb.org/ for further information
# Listening for connection requests on mapi:monetdb://127.0.0.1:50000/
# MonetDB/SQL module loaded
>
```

Switch to different terminal:

```
$ mclient -d scratch
Welcome to mclient, the MonetDB/SQL interactive terminal (Jul2017-SP4)
Database: MonetDB v11.27.13 (Jul2017-SP4), 'scratch'
Type \q to quit, \? for a list of available commands
auto commit mode: on
sql>
```

Aside: Populating Tables via generate_series()

One way to create and populate table unary in MonetDB:

```
CREATE TABLE unary (a int);

INSERT INTO unary(a)
  SELECT value  --  ◀ fixed column name
  FROM   generate_series(1, 101, CAST(1 AS int));
--                       ▲    ▲          ▲
--                    start/end+1   / step of sequence
```

- Table function generate_series(s,e,Δ) enumerates values from s to e (exclusive) with step Δ (default Δ = 1).[6]

[6] Consider the CAST as an oddity (bug?) of MonetDB's function overloading.

Using EXPLAIN on Q_1

Evaluate Q_1 in MonetDB's SQL REPL, mclient:

```
sql> EXPLAIN
       SELECT u.*                  -- ⎫ Q₁ as before
       FROM   unary AS u;          -- ⎭

+----------------------------------------------------…+
| mal                                                   |
+====================================================…+
| function user.s44_1():void;                           |
|     X_1:void := querylog.define("explain select u.… |
:                           :                           :
| #total                    actions=23 time=315 usec    |
+----------------------------------------------------…+
sql> █
```

```
sql>DROP TABLE IF EXISTS unary;
sql>CREATE TABLE unary (a int);

sql>INSERT INTO unary(a) SELECT value FROM generate_series(1, 101);

sql>EXPLAIN SELECT u.* FROM unary AS u;
+----------------------------------------------------------------------------------------------------+
| mal                                                                                                |
+====================================================================================================+
| function user.s44_1():void;                                                                        |
|     X_1:void := querylog.define("explain select u.* from unary as u;", "default_pipe", 21:int);   |
|     X_19 := bat.new(nil:str);                                                                      |
|     X_25 := bat.new(nil:int);                                                                      |
|     X_23 := bat.new(nil:int);                                                                      |
[...]
```

MonetDB Query Plan ≡ MAL Program

```
 ⋮
X_4              := sql.mvc();
C_5:bat[:oid]    := sql.tid(X_4, "sys", "unary");
X_8:bat[:int]    := sql.bind(X_4, "sys", "unary", "a", 0:int);
X_17             := algebra.projection(C_5, X_8);
 ⋮
```

- Queries are compiled into (mostly) linear **MonetDB Assembly Language (MAL)** programs.
  - Program ≡ sequence of assignment statements: <var> := <expression>. Any <var> assigned only once.
- The **MonetDB kernel** implements a **MAL virtual machine (VM)**.

MAL: Scalar Data Types (Atoms)

Once assigned, a MAL variable has a fixed defined **type**:

- **Scalar data types (atoms):**

Scalar Type τ	Literal[7]	Domain
bit	1:bit	bit
bte, sht, <u>int</u>, lng, hge	42:τ	signed {8,16,32,64,128}-bit value
oid	42@0	32-bit row ID (≡ table offset)
<u>flt</u>, dbl	4.2	{32,64}-bit floating point
str	"42"	variable-length UTF-8 string

- Each type τ comes with a constant nil:τ (“*undefined*”, cf. SQL's NULL).

[7] Polymorphic literals without explicit type cast :τ are implicitly assigned the <u>underlined</u> type.

Columns (BATs)

MonetDB implements a *single* collection type bat[:τ], the **Binary Association Tables (BATs)** of values of type τ:

head	tail
0@0	42
1@0	42
2@0	0
3@0	-1
4@0	nil

densely ascending sequence of row IDs of type oid (row at offset *i* has oid *i*@0)

scalars of type τ (≡ int) (BAT “payload”)

- **Head:** store sequence base 0@0 only (“virtual oids”, void)
- **Tail:** one **ordered column** (or vector) of data

Use mserver5 prompt to create and populate a BAT:

```
$ mserver5 --dbpath=(pwd)/data/scratch --set monet_vault_key=(pwd)/data/scratch/.vaultkey
[...]
>t := bat.new(nil:int);    ⬅ BAT creation

>bat.append(t, 42);        ⬅ populate BAT
>bat.append(t, 42);
>bat.append(t, 0);
>bat.append(t, -1);
>bat.append(t, nil:int);
>io.print(t);
#--------------------------#
# h t  # name
# void  int  # type
#--------------------------#
[ 0@0,  42  ]
[ 1@0,  42  ]
[ 2@0,  0  ]
[ 3@0,  -1  ]
[ 4@0,  nil  ]

>v := algebra.fetch(t, 3@0);   ⬅ access BAT by oid (≡ row offset), v has type int
>io.print(v);
[ -1 ]

>bat.replace(t, 4@0, 2);   ⬅ update BAT by oid (≡ row offset)
>io.print(t);
#--------------------------#
# h t  # name
# void  int  # type
#--------------------------#
[ 0@0,  42  ]
[ 1@0,  42  ]
[ 2@0,  0  ]
[ 3@0,  -1  ]
[ 4@0,  2  ]

>t1 := algebra.slice(t, 1@0, 3@0);  ⬅ positional BAT slicing
>io.print(t1);
#--------------------------#
# h t  # name
# void  int  # type
#--------------------------#
[ 1@0,  42  ]   ⬅ head column doesn't have to start at 0@0
[ 2@0,  0  ]
[ 3@0,  -1  ]
```

```
>b := bat.getSequenceBase(t);    get BAT head seqbase
>io.print(b);
[ 0@0 ]
>b := bat.getSequenceBase(t1);
>io.print(b);
[ 1@0 ]

>bat.delete(t, 1@0);    deleting a row leaves no "holes": move last row into deleted offset
>io.print(t);
#--------------------------#
# h t  # name
# void  int  # type
#--------------------------#
[ 0@0,  42  ]
[ 1@0,  2  ]    moved!
[ 2@0,  0  ]
[ 3@0,  -1  ]
```

Using MAL to Process SQL

MAL program for Q_1, shortened and formatted:

```
   ⋮
❶ sql                 := sql.mvc();
❷ unary :bat[:oid] := sql.tid( sql, "sys", "unary");
❸ a     :bat[:int] := sql.bind(sql, "sys", "unary", "a",…);
❹ result:bat[:int] := algebra.projection(unary, a);
   ⋮
```

❶ Get database catalog handle (also: TX management).

❷ Get IDs of all **currently visible** rows in table unary.

❸ Get all values in column a of table unary.

❹ Compute result column of all visible a values.

Using MAL to Process SQL

Assume that the row with a = 3 (oid 2@0) has been deleted (BAT unary reflects this update, thus no 2@0 in its tail):

algebra.projection(unary:bat[:oid] , a:bat[:int]) = result:bat[:int]

unary:bat[:oid]

head	tail
0@0	0@0
1@0	1@0
2@0	3@0
3@0	4@0
4@0	5@0
⋮	⋮
98@0	99@0

a:bat[:int]

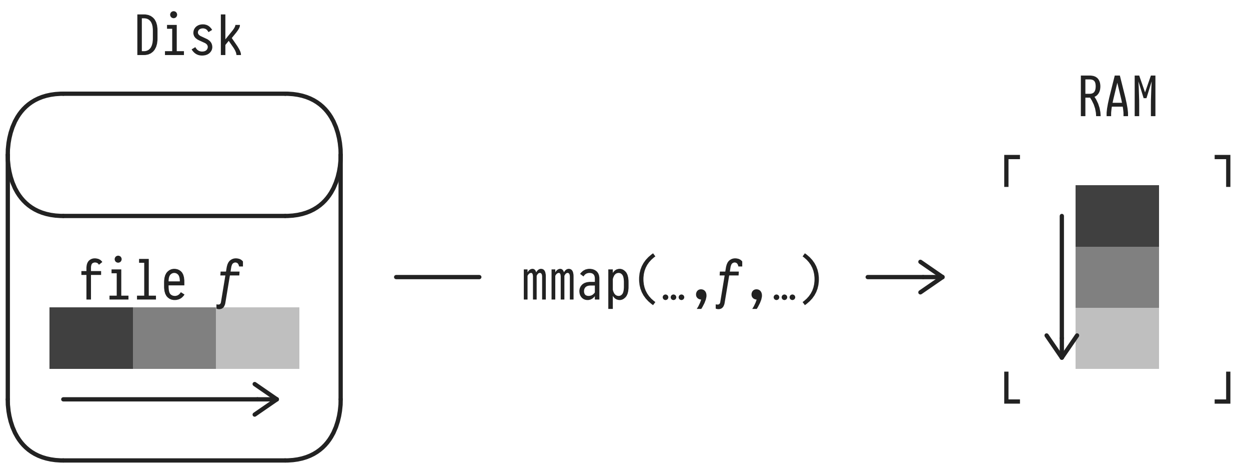
head	tail
0@0	1
1@0	2
2@0	3
3@0	4
4@0	5
⋮	⋮
98@0	99
99@0	100

result:bat[:int]

head	tail
0@0	1
1@0	2
2@0	4
3@0	5
4@0	6
⋮	⋮
98@0	100

All BATs are processed as in-memory arrays of fixed-width elements (atoms).

- **Transient** BATs exist in RAM only.
- **Persistent** BATs live on disk and are mmap(2)ed into RAM:



UNIX mmap(2): Map Files into Memory

```
MMAP(2)                       BSD System Calls Manual                       MMAP(2)

NAME
  mmap -- allocate memory, or map files or devices into memory

LIBRARY
  Standard C Library (libc, -lc)

SYNOPSIS
  #include <sys/mman.h>

  void *
  mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);

DESCRIPTION
  The mmap() system call causes the pages starting at addr and continuing for at most len bytes to be mapped from the
  object described by fd, starting at byte offset offset. [...]
```

- The contents of file fd are **mapped 1:1 into contiguous memory.** No conversion or transformation takes place — cf. this with PostgreSQL's row storage (later).
- OS implements virtual memory: can map even huge files.

Peeking into a MonetDB BAT

Use MAL builtin function bat.info() to collect details about the BAT for column unary(a) of 100 32-bit ints:

```
> a := sql.bind(sql, "sys", "unary", "a", …);
> (i1,i2) := bat.info(a);
> io.print(i1,i2);
# void  str str  # type
#--------------------------#
[...]
[ 7@0,  "tail",           "int"            ]
[ 8@0,  "batPersistence","persistent"     ]  persistent BAT
[ 33@0, "tail.free",      "400"            ]  size on disk
[ 37@0, "tail.filename", "17/1703.tail" ]  OS file
[...]
> 
```

Use mserver5 console or mclient REPL and UNIX shell to peek into the BAT for column unary(a):

```
module sql;   ⬅ only in mclient (do not use in mserver5 console)
sql.init();
sql := sql.mvc();
a:bat[:int] := sql.bind(sql, "sys", "unary", "a", 0:int);
io.print(a);
(i1,i2) := bat.info(a);
io.print(i1,i2);
#--------------------------#
# t t t  # name
# void  str str  # type
#--------------------------#
[...]
[ 4@0,  "batCount", "100"  ]
[ 5@0,  "batCapacity",  "1024"  ]
[ 6@0,  "head", "void"  ]   ⬅
[ 7@0,  "tail", "int"  ]   ⬅
[ 8@0,  "batPersistence", "persistent"  ]   ⬅
[ 13@0, "hseqbase", "0@0"  ]   ⬅
[ 30@0, "batCopiedtodisk",  "1"  ]
[ 33@0, "tail.free",  "400"  ]   ⬅ tail file size in bytes
[ 34@0, "tail.size",  "4096"  ]
[ 35@0, "tail.storage", "malloced"  ]
[ 37@0, "tail.filename",  "17/1703.tail"  ]   ⬅
[...]

$ cd MonetDB/data/scratch/bat/17
$ ls -l 1703.tail
-rw-r--r--  1 grust  staff  400 Mar  8 16:47 1703.tail
                             ⬆
                     = 100 × 4 (MonetDB type int) bytes

# scan/print tail file contents
see C program live/mmap.c (⚠ edit to fill in tail file name)
```

Fixed-Width Tail Columns and Row Offsets

- Each tail column entry in a MonetDB BAT of type `bat[:τ]` is of **fixed width** (e.g., $\tau \equiv$ `int`: 4 bytes).

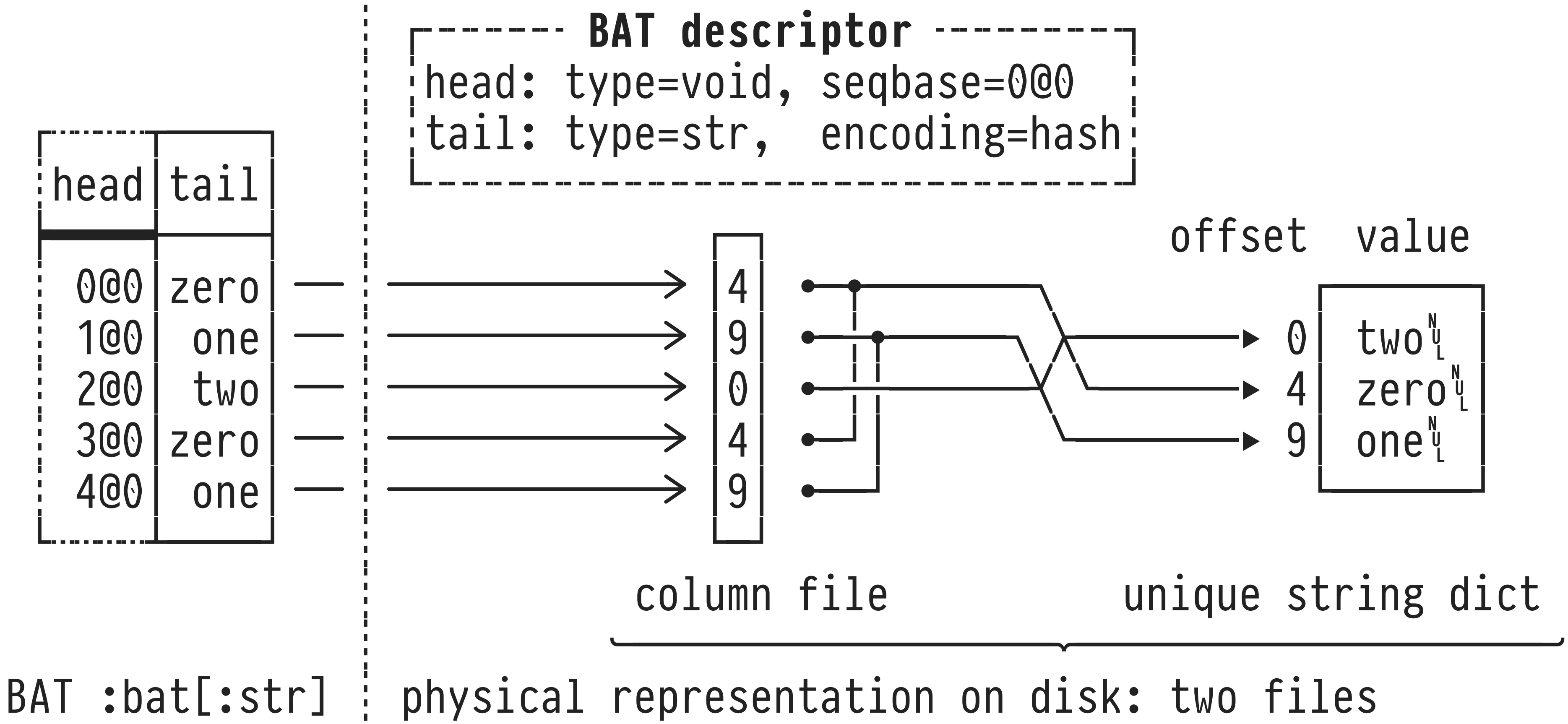
- Runtime representation of **tail column as a C array**, say `a`. Access entry with oid i@0 simply via

$$\underbrace{\texttt{a[i - hseqbase]}}_{\text{effective address: a + (i - hseqbase)} \times \text{size of } \tau}$$

- ⇒ BAT processing routines (like `algebra.projection()`) implemented as (tight) loops over C arrays. 🚀

Variable-Width Tail Columns: Dictionary Files

Use fixed-width tail column and separate hashed dictionary:



Generate table unary' from original unary table. Create tail column of type str.

In mclient SQL console:

```
CREATE TABLE "unary'" (a text);
INSERT INTO "unary'"(a)
  SELECT s.w
  FROM   unary AS u, (VALUES (0, 'zero'),
                             (1, 'one'),
                             (2, 'two'),
                             (3, 'three'),
                             (4, 'four')) AS s(n, w)
  WHERE u.a % 5 = s.n;
```

At MAL prompt:

```
module sql;   ⬅ only in mclient (do not use in mserver5 console)
sql.init();
sql := sql.mvc();
a:bat[:str] := sql.bind(sql, "sys", "unary'", "a", 0:int);
io.print(a);
(i1,i2) := bat.info(a);
io.print(i1,i2);
#--------------------------#
# t t t  # name
# void  str str  # type
#--------------------------#
[ 6@0,  "head", "void"  ]
[ 7@0,  "tail", "str"  ]   ⬅   variable-sized atoms in tail
[ 8@0,  "batPersistence", "persistent"  ]
[ 13@0, "hseqbase", "0@0"  ]
[ 30@0, "batCopiedtodisk",  "1"  ]
[ 33@0, "tail.free",  "100"  ]   ⬅ size of tail file
[ 34@0, "tail.size",  "1024"  ]
[ 37@0, "tail.filename",  "23/2300.tail"  ]
[ 39@0, "theap.free", "8269"  ]   ⬅ size of heap file, dictionary starts @ offset 8192
[ 40@0, "theap.size", "16384"  ]   ⬅
[ 43@0, "theap.filename", "23/2300.theap"  ]   ⬅

$ cd MonetDB/data/scratch/bat/23
$ ls -l
-rw-r--r--  1 grust  staff   100 Mar 10 18:24 2300.tail
                              ⬆
                          100 × 1 byte (offsets into string dictionary)
-rw-r--r--  1 grust  staff  8269 Mar 10 18:24 2300.theap

$ hexdump -C 2300.theap
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
[...]
*     ⬇  dictionary starts at offset 0x2000 = 8192
00002000  00 00 00 00 00 00 00 00  6f 6e 65 00 be 7f 00 00  |........one.....|
00002010  00 00 00 00 00 00 00 00  74 77 6f 00 00 00 00 00  |........two.....|
00002020  00 00 00 00 00 00 00 00  74 68 72 65 65 00 00 00  |........three...|
00002030  00 00 00 00 00 00 00 00  66 6f 75 72 00 00 00 00  |........four....|
00002040  00 00 00 00 00 00 00 00  7a 65 72 6f 00           |........zero.|
0000204d

# scan/print tail and heap file contents
see C program live/heap.c
```