

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

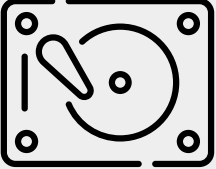
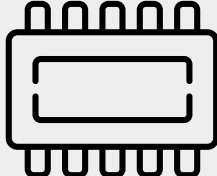
```
SELECT u.*           -- * ≡ access all columns of row u
FROM unary AS u
```


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**:

PostgreSQL	MonetDB
	
disk-based	RAM-based

-  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:

```
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 values² from `s` to `e` (inclusive) with step `Δ` (default `$\Delta = 1$`).

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

Using **EXPLAIN** on Q_1

Let us try to understand the evaluation of Q_1 :

```
db2=# EXPLAIN VERBOSE
      SELECT u.*           -- }  $Q_1$  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>**:

- 1 **EXPLAIN** **<opt>** **<Q>**
- 2 **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 \Rightarrow 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 (\rightarrow 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



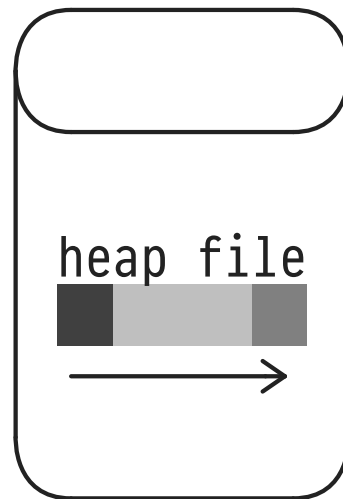
The DBMS may reorganize (e.g., compact or “vacuum”) a table's heap file at any time \Rightarrow no guaranteed row order:

Table unary

a
1
2
⋮
42
⋮
99
100

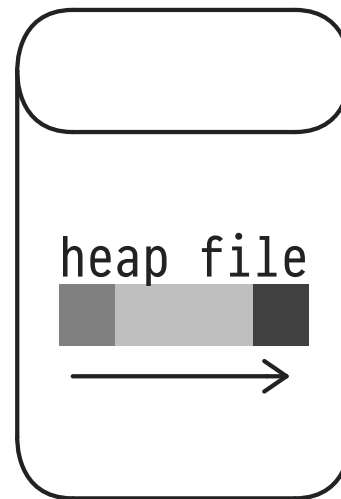


Disk



1 now


Disk



2 now + 1 s

Table unary

a
99
100
⋮
42
⋮
1
2



Heap File \equiv 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).

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 $RID(r_1) \neq RID(r_2)$.
 - $RID(r)$ **encodes the location** of row r in its table's heap file. No sequential scan is required to access r .
 - If r is updated, $RID(r)$ remains stable.
- ⚠ RIDs do *not* replace the relational key concept.³

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

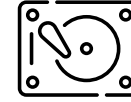
RIDs in PostgreSQL

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;
```

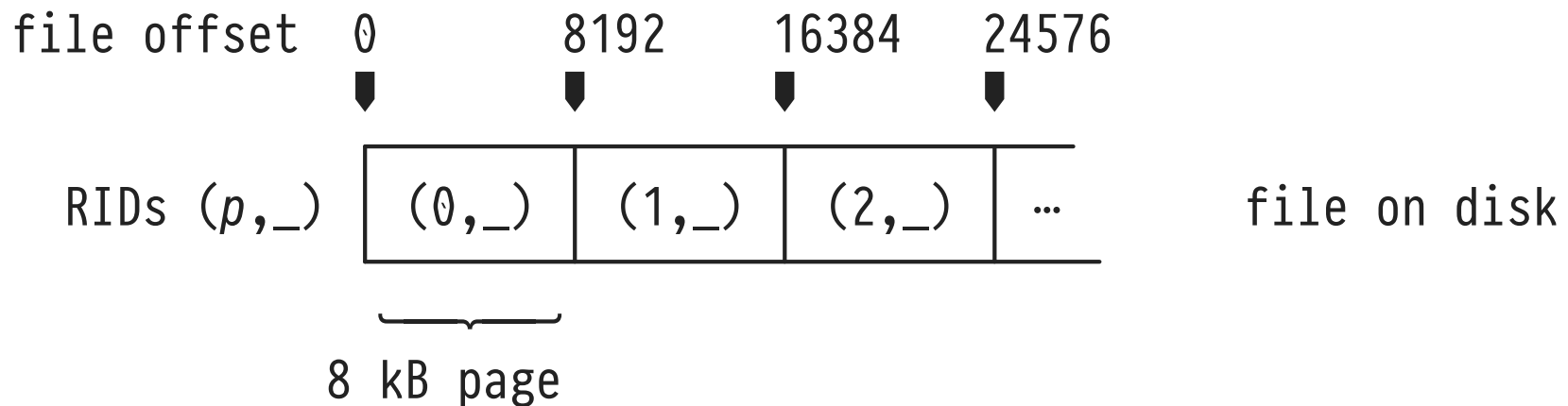
ctid	a
(0,1)	1
(0,2)	2
⋮	⋮
(1,1)	227
(1,2)	228
⋮	⋮
(4,95)	999
(4,96)	1000

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**.



Block I/O on Disk-Based Secondary Memory

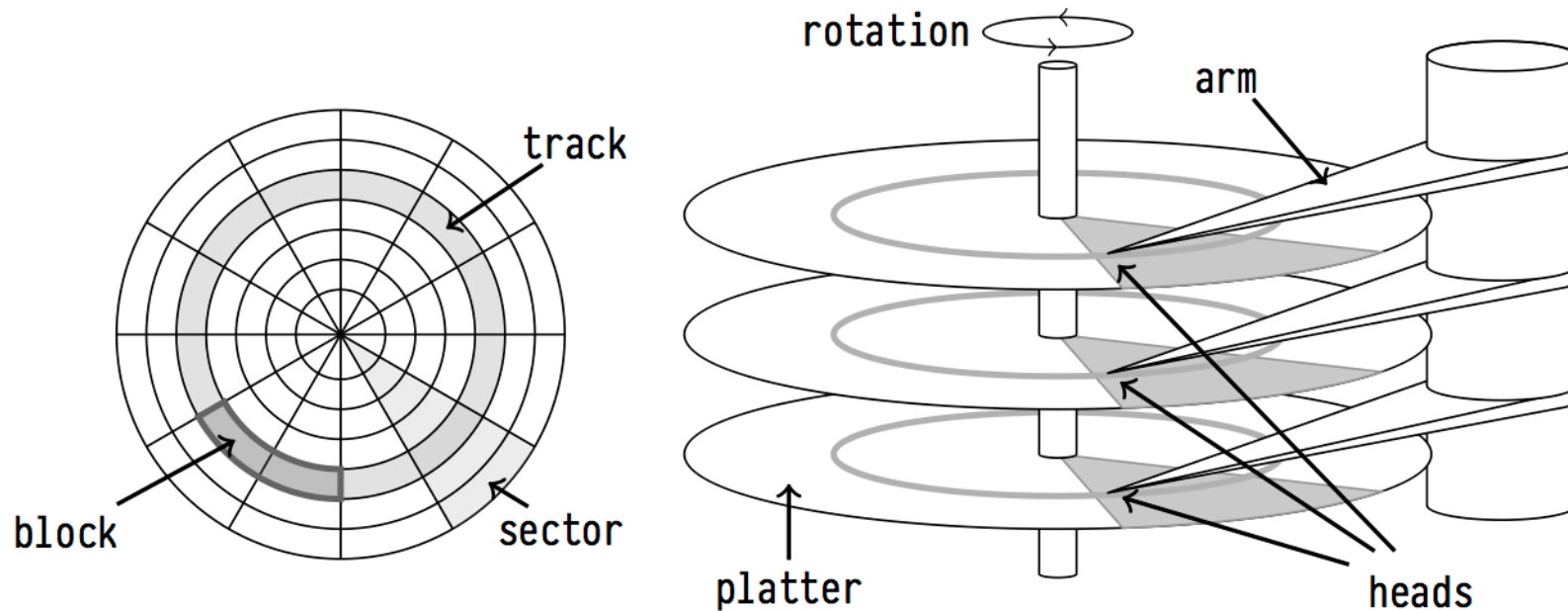
- Heap files are **read and written in units of 8 kB pages**.
 - Likewise, heap files grow/shrink by entire pages.
- This page-based access to heap files reflects the OS's mode of performing **disk input/output page-by-page**.
 - Terminology: DB  **page** \equiv **block**  OS
-  Any disk I/O operation 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:

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

- 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 : $\frac{1}{2} \times (1/15000 \text{ min}^{-1}) = 2 \text{ ms}$
 - Transfer time t_{tr} : $8 \text{ kB} / (163 \text{ MB/s}) = 0.0491 \text{ ms}$
 - $\Rightarrow t_s + t_r + t_{tr} = 3.4 \text{ ms} + 2 \text{ ms} + 0.05 \text{ ms} = \underline{5.45 \text{ ms}}$

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:
 - $1000 \times t_{tr} = 5.45 \text{ s}$ 🙄
- **Sequential access time** to 1000 adjacent blocks of 8 kB:
 - 512 kB per track: 1000 blocks will span 16 tracks
 - $\Rightarrow t_s + t_r + 1000 \times t_{tr} + 16 \times 0.2 \text{ ms} = \underline{58.6 \text{ ms}}$
- Once we need to read more than $58.6 \text{ ms} / 5450 \text{ ms} = 1.07\%$ of a file, we better **read the entire file sequentially**.

Solid State Disk Drives (SSDs)

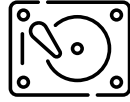


SSDs rely on non-volatile flash memory and contain no moving/electro-mechanical parts:

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

≈ 1.4 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
    └──────────┘
    ≈ 14.9 GB/s
```

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

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.

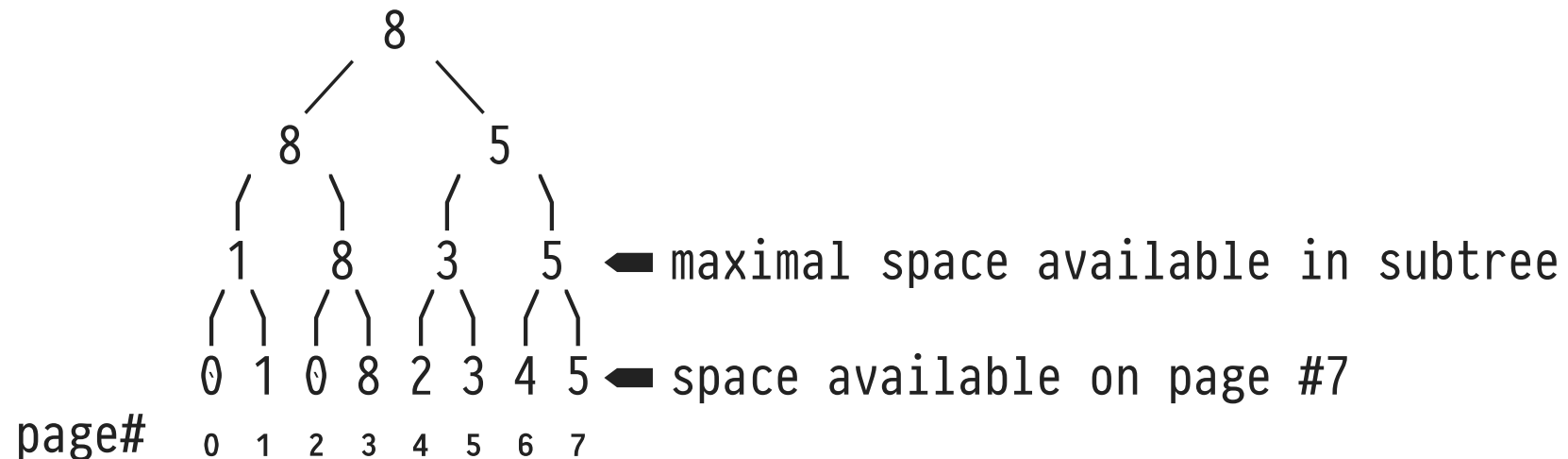
4 | Heap Files: Free Space Management

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

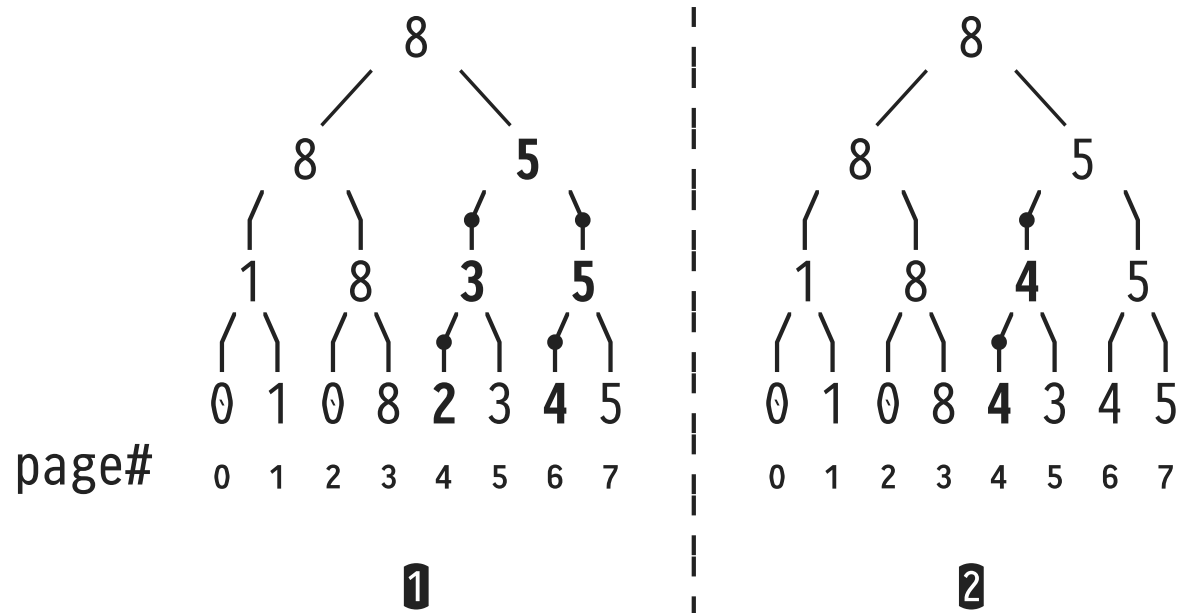
PostgreSQL maintains a **tree-shaped FSM** for each heap file:



- 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 $2 \uparrow 3 \uparrow 5 \downarrow 5 \downarrow 4$ along \nearrow).
- 2 Update page #4 to provide 4 available slots (traverses \nearrow , updates 3 to $\max(3, 4) = 4$, stops when $\max(4, 5) = 5$).

6 : Q_1 — The Simplest SQL Probe Query

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

```
SELECT u.*           -- *  $\equiv$  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 ?

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 `$\Delta = 1$`).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.*           -- \  $Q_1$  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> █
```

MonetDB Query Plan \equiv 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 \equiv 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
<code>bit</code>	<code>1:bit</code>	bit
<code>bte</code> , <code>sht</code> , <u><code>int</code></u> , <code>lng</code> , <code>hge</code>	<code>42:τ</code>	signed {8,16,32,64,128}-bit value
<code>oid</code>	<code>32@0</code>	32-bit row ID (\equiv table offset)
<u><code>flt</code></u> , <code>dbl</code>	<code>4.2</code>	{32,64}-bit floating point
<code>str</code>	<code>"42"</code>	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 underlined type.

Columns (BATs)

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

	head	tail	
densely ascending sequence of row IDs of type oid (row at offset i has oid $i@0$)	0@0	42	} scalars of type τ (\equiv int) (BAT “payload”)
	1@0	42	
	2@0	0	
	3@0	-1	
	4@0	nil	

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

Using MAL to Process SQL

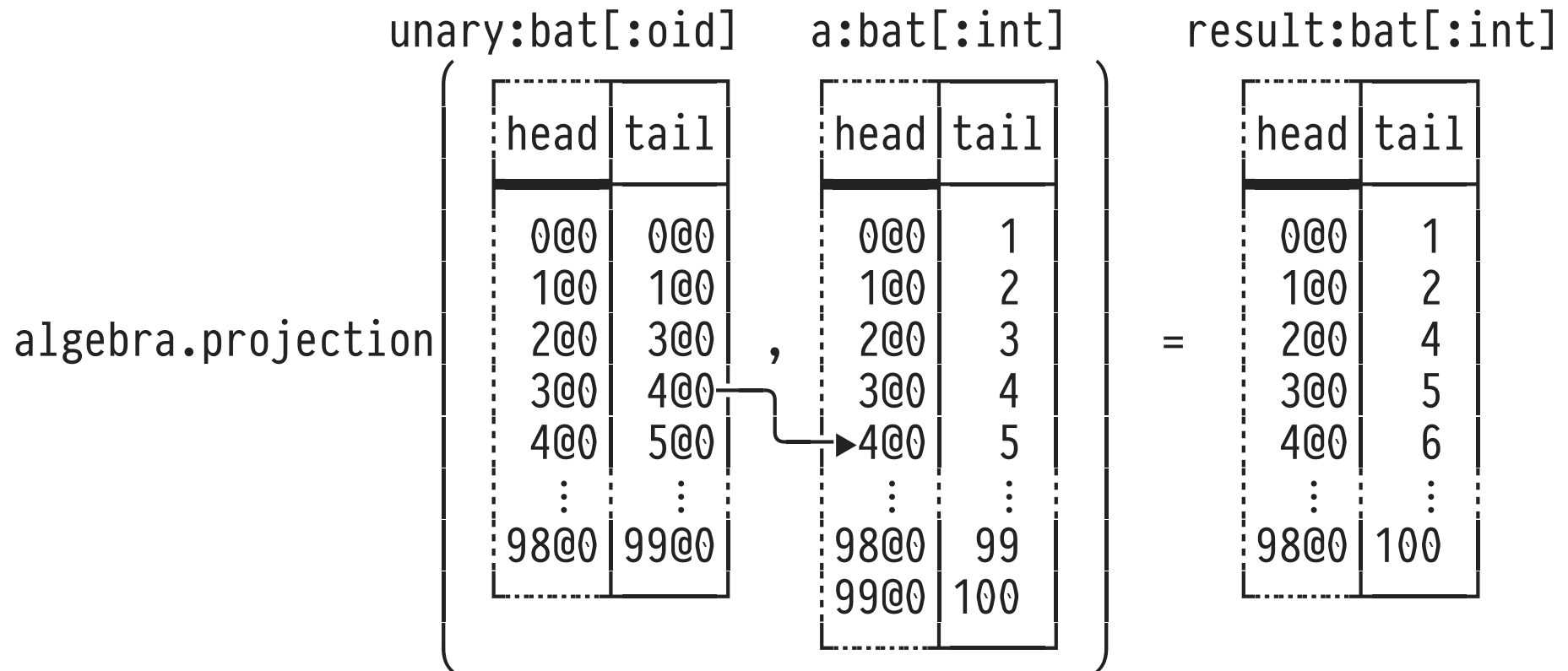
MAL program for Q_1 , shortened and formatted:

```
⋮  
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

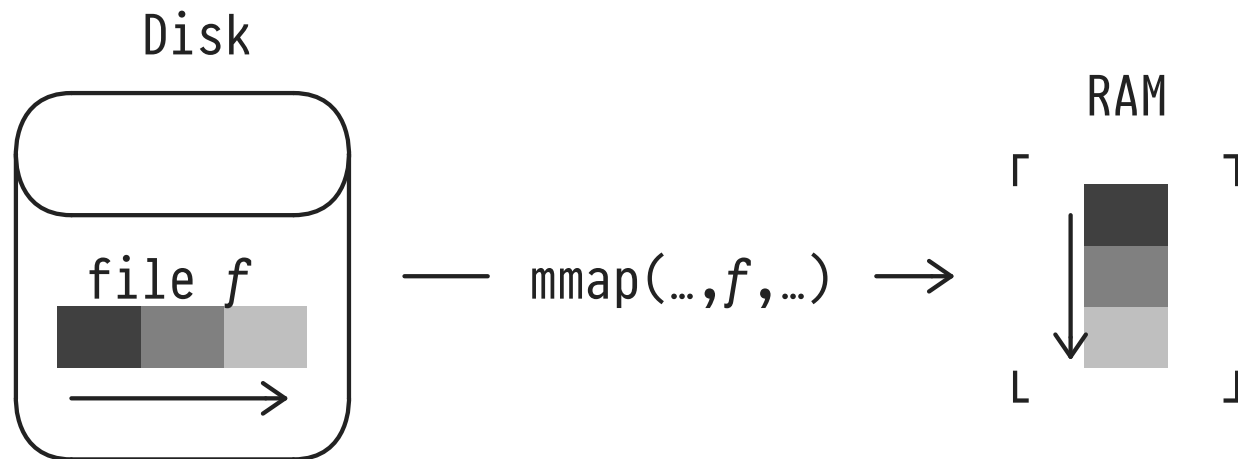
Assume that the row with $a = 3$ (oid $2@0$) has been deleted (BAT `unary` reflects this update, thus no $2@0$ in its tail):



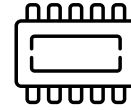
7 | MonetDB: A Main-Memory DBMS

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)`ed⁸ 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

```
[...]
```

> █

Fixed-Width Tail Columns and Row Offsets

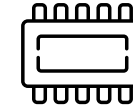
- Each tail column entry in a MonetDB BAT of type `bat[: τ]` is of **fixed width** (e.g., $\tau \equiv \text{int}$: 4 bytes).
- Runtime representation of **tail column as a C array**, say `a`. Access entry with oid `i@0` simply via

`a[i - hseqbase]`

effective address: `a + (i - hseqbase) × size of τ`

- \Rightarrow 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:

