

Tabular Database Systems

③




Reading Data at the Speed of ~~Light~~ Memory

March 17, 2026

Torsten Grust
Universität Tübingen, Germany


1 | DBMSs Exploit Modern Computer Architecture¹

The internals of DBMSs are carefully engineered to exploit the performance features of modern computer architecture:

- **CPUs** (and their multi-threading capabilities),
- **main memory** (DRAM ) and its hierarchy of caches, and
- **secondary memory** (mass storage on SSDs  / rotating disks ).

Since database queries typically process millions of rows, the effect of even the tiniest performance tweaks/tricks played in the innermost loops of DBMS routines multiply.

Goal: Understand the performance spectrum for a simple “query.”

quick one-liner
shell script  hand-written
C program

¹ This chapter adapts and expands on a discussion found in Thomas Neumann's lecture [“Foundations in Data Engineering” \(TUM\)](#) .

A Simple Benchmark Query

1. Read the CSV file for TPC-H table `lineitem` (scale factor `SF = 1`: 6+ million rows \times 16 columns \approx 720 MB of data) and
2. sum the `quantity` integer values in the 5th column:

`lineitem.csv`

```
1|155190|7706|1|17|21168.23|0.04|0.02|N|0|1996-03-13|...NL
1|67310|7311|2|36|45983.16|0.09|0.06|N|0|1996-04-12|...NL
1|63700|3701|3|8|13309.60|0.10|0.02|N|0|1996-01-29|...NL
⋮
[6+ million more rows]
```

- Real TPC-H benchmark data and its queries are more complex but this suffices to demonstrate the effect of code optimizations.
- We will implement the query in awk, Python, C, and SQL.

2 : Performance Limits

What is the fastest query time we can hope for in principle?

- Torsten's current computer (🍏 MacBook Pro M2 Max, 2023):

Memory (📁 Secondary/📁 Primary)	Read Bandwidth	🕒 Query Time
(Ethernet)	2.5 GB/s	0.28s
📁 External USB-C SSD (2 TB)	800 MB/s	0.90s
📁 NVMe SSD (2 TB)	5 GB/s	0.14s
📁 DRAM (64 GB)	21 GB/s	0.03s

- **NB.**
 - **Query Time** based on I/O speed, ignores CPU cost (less significant for secondary mem, very significant for DRAM).
 - \Rightarrow We will *not* reach these limits. Let us try to get close.
- Understand how DuckDB achieves 0.002s for our query. 🐧☰

3 | Sum of Quantities ① — awk

- `awk`: interpreted text processing language popular on UNIX™.
 - Read input line by line, match each line against (regular) patterns in order, on a match invoke action `{...}` on line.

BEGIN	{ FS = " " }		delimiter in CSV is	#012
	sum = 0 }		match first line, reset sum	
	{ sum = sum + \$5 }		match any line, sum 5th column	
END	{ print sum }		match last line, output sum	

- Invoke the `awk` script, measure elapsed wall-clock time (s) 🕒:

```
$ time ./012-sum-quantity.awk lineitem.csv
153078795
      1.58 real          1.43 user          0.14 sys
```

Sum of Quantities ① — awk

- 🕒 Query time on Torsten's computer: $\approx 1.6\text{s}$:


Output of time	Measurement
real	elapsed wall-clock time 🕒 ($\approx \text{user} + \text{sys} + \Delta$)
user	time spent in application/library code
sys	time used by OS (system calls)

- The interpreted awk script cannot even keep up with secondary memory (SSD) read bandwidth:²
 - awk processes the CSV file with a throughput of 471 MB/s.
 - awk is **CPU-bound** for this query.
 - \Rightarrow The OS file system cache in DRAM does not help.

² Execution speed of awk variants vary greatly. We are using GNU awk ([gawk](#)) here. macOS awk is about 10 times slower for our benchmark query.

4 : Sum of Quantities ② — Python

- **Python**: established scripting/programming language, mainly follows an imperative paradigm.
 - Translates to bytecode, then interprets.

<pre>sum = 0 with open(sys.argv[1], 'r') as file: for line in file: sum = sum + int(line.split(' ')[4]) print(sum)</pre>		<pre>reset sum</pre>	 #013
		<pre>open file (reading) read line by line extract 5th col, cast to int, add output</pre>	

- 🕒 Query time on Torsten's computer: ≈ 2.75 s.
 - Python processes the CSV file with a throughput of 275 MB/s.
 - Python is **CPU-bound** for this query.

5 | Sum of Quantities ③ — C

- Switch to compiled programming language **C**. Start out with a direct transcription of the Python code:
 - Read CSV file line by line using `getline(3)`.³
 - Use `strchr(3)` to search for delimiter `'|'` in line (4×).
 - Convert string (up to next `'|'`) into integer using `atoi(3)`.

```
while (getline(&line, &linecap, file) > 0) {  
    delim = line;  
  
    for (column = 1; column < 5; column++) {  
        delim = strchr(delim, '|');  
        delim++;  
    }  
  
    sum = sum + atoi(delim);  
}
```

 #014

³ The (3) suffix in `getline(3)` refers to Section 3 of the UNIX™ manual which describes the function in the C Standard Library.

Sum of Quantities ③ — C

- 🕒 Query time on Torsten's computer: $\approx 0.5s$.
 - C processes the CSV file with a throughput of 1.5 GB/s.
 - Yet, C still is **CPU-bound** for this query. Getting closer to SSD read bandwidth, though.
- But where does time go?
 - **Profile** the running program, identify code portions consuming the most CPU time (UNIX™: [perf](#), macOS: [Instruments](#)).

C library routine	% of CPU time
(all other C code)	(16%)
getdelim (\equiv getline)	58%
atoi	14%
memchr (\equiv strchr)	12%

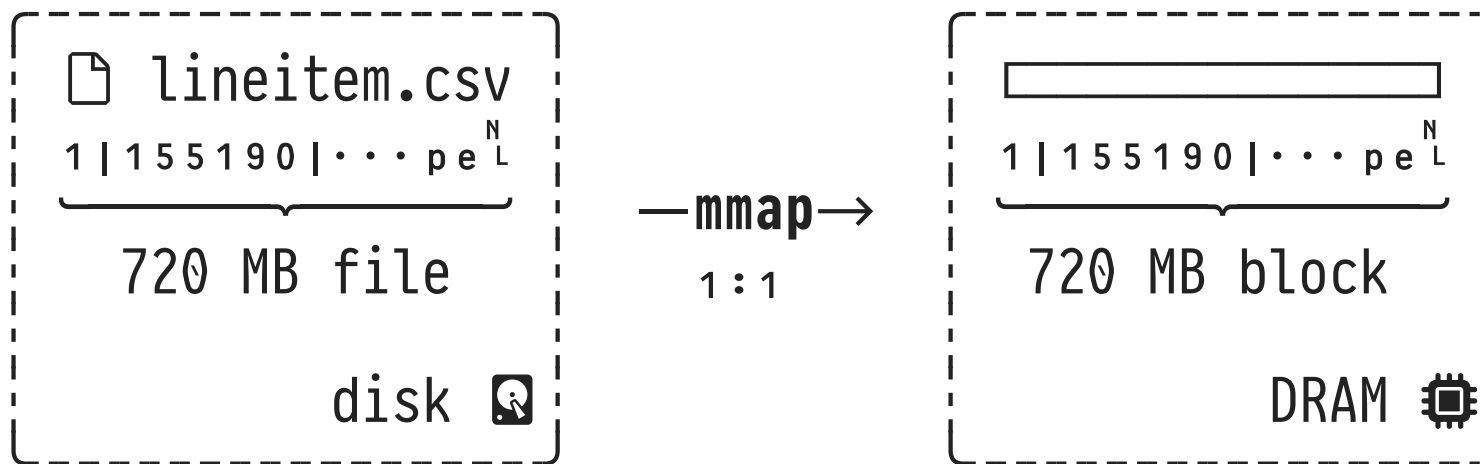
- Reading the CSV file line by line is too slow.
 - 💡 Read entire file at once, impose line structure ourselves.

6 : Sum of Quantities ④ — C with mmap(2)

Aim to *read the CSV file* into DRAM using a *single OS system call*:

- `mmap(2)` returns a pointer `data` to a contiguous block of memory that holds the contents of an entire disk file:

```
data = (char*)mmap(NULL, size, PROT_READ, MAP_SHARED, file, 0);
```



- If file size exceeds available DRAM, OS pages in file contents on demand.

Sum of Quantities ④ — C with mmap(2)

- Cannot use `strchr()` to find '|' (next column) ❶.
- No `getline()`: need to locate '\n' (❷) on our own now:

```

column = 1;

while (data < end) {
    switch (*data) {
        case '|': column++; break;
        case '\n': ...
    }
    data++;

    if (column < 5)
        continue;

    sum = sum + atoi(data);

    column = 1;
    while (*data++ != '\n');
}

```

start in column 1

 #015

scan memory block, byte by byte

❶ found |: next column begins
error: line has too few columns

proceed through memory block

reached column 5 already?
no, keep scanning

convert to int (up to |, '\n'), add

next line starts with column 1
❷ skip rest of line until '\n'

Sum of Quantities ④ — C with mmap(2)

- 🕒 Query time on Torsten's computer:
 - Once the OS caches the file in DRAM, `mmap()` directly maps the file system cache into the program's address space.

OS file system cache	Query time 🕒	Throughput
cold	1.6s	471 MB/s
warm	0.42s	1.8 GB/s

- NB.** The C program's profile has changed:

C code fragment/function	% of CPU time
(all other C code)	(25.5%)
<code>atoi</code>	21.4%
<code>while (*data++ ≠ '\n')</code>	👎 53.1%

- Search for `\n` 📄 dominates. 💡 Use `strchr(data, '\n')` instead.
 - 🕒 Query time: 0.27s (throughput 2.8 GB/s).
 - How can `strchr()` be so efficient?


7 : Sum of Quantities ⑤ — C with mmap(2) + Block-Wise '\n' Search

Avoid byte-wise search for '\n'. Modern CPUs operate on 64-bit words.

-  Load **8 bytes (64 bits) at a time**, search for '\n' ('\n' = 0x0a) in this block. Advance pointer `data` in strides of 8 bytes.

```

/* HAS_NL(x): find '\n' in 64 bit-wide character block x */
#define HAS_ZERO(x) (((x) - 0x0101010101010101) &
                    ~(x) & 0x8080808080808080)
#define HAS_NL(x)   (HAS_ZERO(x ^ 0x0a0a0a0a0a0a0a0a))

HAS_NL(0x0a42440a6b637544) ≡ "Duck\nDB\n" reversed 
=      0x8000000800000000      (ARM CPUs: little endian)
      ↑      ↑
high bit set: found '\n' at offsets 4 and 7

```

- How do C macros `HAS_ZERO()` and `HAS_NL()` work?⁴

 #016

⁴ See the [Stanford Bit Twiddling Hacks](#)  (section “Determine if a word has a byte equal to n”) for a discussion of these C macros.

Sum of Quantities ⑤ — C with mmap(2) + Block-Wise `nl` Search

```
while (data < end) {  
    :  
    sum = sum + atoi(data);  
  
    column = 1;  
  
    block = (uint64_t*)data;  
    while (!(nl = HAS_NL(*block)))  
        block++; /* advance by one 8-byte-block (64 bits) */  
  
    data = (char*)block;  
    if (nl & 0x000000000000000080ULL) { data += 1; continue; }  
    if (nl & 0x00000000000000008000ULL) { data += 2; continue; }  
    if (nl & 0x0000000000000000800000ULL) { data += 3; continue; }  
    if (nl & 0x00000000008000000000ULL) { data += 4; continue; }  
    if (nl & 0x00000000800000000000ULL) { data += 5; continue; }  
    if (nl & 0x00000080000000000000ULL) { data += 6; continue; }  
    if (nl & 0x00008000000000000000ULL) { data += 7; continue; }  
    data += 8;  
}
```

 #017

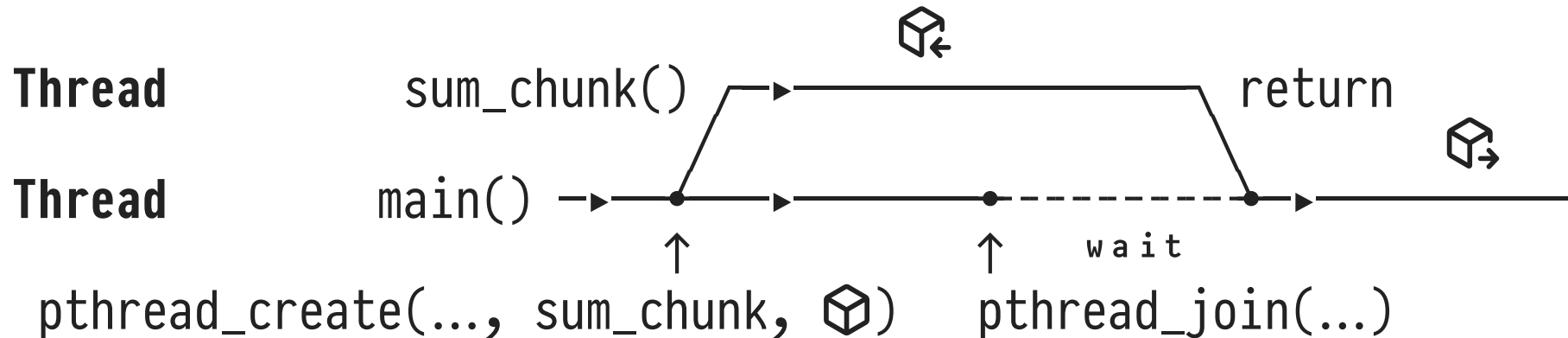
Sum of Quantities ⑤ — C with `mmap(2)` + Block-Wise `ℓ` Search

- 🕒 Query time on Torsten's computer (warm cache): ≈ 0.28 s.
 - C with `mmap()` and block-wise search for `ℓ` processes the CSV file with a throughput of 2.8 GB/s.
 - We match the performance of the built-in `strchr()` function.
- Our code definitely got more complex and fiddly.
 - Slowly getting an impression of how much careful performance engineering is required to build a DBMS kernel.

8 : Sum of Quantities ⑥ — C with mmap(2) + Threads

CPUs feature **multiple cores** that can execute code in parallel. The CPU (M2 Max) in Torsten's computer features $T = 12$ such cores.

- ? Split CSV file at \backslash boundaries into T **partitions (chunks)**.
- Spawn T **parallel threads**, each summing column 5 in one partition. Add thread-local partial sums to obtain overall sum.



- Threads use shared memory area ⑥ to exchange data (*e.g.*, thread ID, pointers to chunk start/end, thread-local sum).


Sum of Quantities ⑥ — C with mmap(2) + Threads

- C declaration of shared memory area :

```
struct chunk {
    pthread_t thread; /* thread ID */
    char      *data;  /* pointers to chunk start/end */
    char      *end;
    int       sum;    /* sum of partition */
};
typedef struct chunk chunk_t;
```



- Code for a thread (sums a chunk):

 #018

```
void *sum_chunk(void *arg)
{
    chunk_t *chunk = (chunk_t*) arg; /* this is the  */

    char *data = chunk->data; /* extract relevant arguments */
    char *end  = chunk->end;

    : /* sum chunk, just like sum-quantity-mmap.c did */

    chunk->sum = sum; /* put return value into  */
    return NULL; /* NULL ≡  */
}
```

Sum of Quantities ⑥ — C with mmap(2) + Threads

- 🕒 Query time on Torsten's computer ($T = 12$ threads spawned, warm cache): ≈ 0.04 s.
 - Jointly, the threads process the CSV file with a throughput of 18.8 GB/s. This approaches DRAM read bandwidth.
- Profile shows that each `sum_chunk()` + `main()` use about the same chunk summing time. 👍
 - Creating chunk sizes (and thus thread-local work) of roughly equal size has worked well.
 - Wait time after `pthread_join(...)` expected to be small.

Sum of Quantities — Summary so Far








Query Implementation	Query Time 🕒	Throughput
awk	1.60s	471 MB/s
Python	2.75s	275 MB/s
C with getline	0.50s	1.5 GB/s
C with mmap	0.27s	2.8 GB/s
C with mmap + block-wise scan	0.28s	2.8 GB/s
C with mmap + 12 threads	0.04s	18.8 GB/s

- Implementation language and techniques matter **a lot**.
 - 50+ years after the inception of the relational model, database query optimization is a lively field of research.
- Even your laptop can read and process multiple GB/s.
 - Here we saturate everything (but DRAM).
 - Do we always need “big iron” or server clusters? (🐧: “No!”)

9 : Interlude: SQL Aggregate Functions

SQL aggregates⁵ condense a bag of rows into a *single* value:

 vehicles

vehicle	kind	seats	wheels?
	car	5	true
	SUV	3	true
	bus	42	true
	bus	7	true
	bike	1	true
	tank	□	false
	cabrio	2	true

- Most aggregates ignore (= skip) NULL (□) values.
 - An optional **FILTER** clause can control value inclusion.
- Order-aware aggregates use clause **ORDER BY** to be deterministic:

`list(vehicle ORDER BY seats)`

`bool_and("wheels?")` ≡ false
`max(seats)` ≡ 42
`arg_max(kind, seats)` ≡ 'bus'
`count(vehicle)` ≡ 7

⁵ DuckDB's SQL dialect features an extensive list of [built-in aggregate functions](#) ↗.

Interlude: SQL Aggregate Functions

- Let us use annotations to explain SQL constructs (here: query clause **row cardinality**, **1 2 3**: clause processing order⁶):

<pre>SELECT <i>expr</i>,...,<i>expr</i> FROM <i>t</i> WHERE <i>p</i></pre>	<pre> 3 <i>m</i> 1 <i>n</i> (= <i>t</i> , cardinality of <i>t</i>) 2 <i>m</i> (≤ <i>n</i>) └── # rows returned by query clause</pre>
---	--

- SQL aggregates reduce row cardinality to 1:

<pre>SELECT <i>agg</i>,...,<i>agg</i> FROM <i>t</i> WHERE <i>p</i></pre>	<pre> 3 1 1 <i>n</i> (= <i>t</i>) 2 <i>m</i> (≤ <i>n</i>) └── # rows</pre>
---	---

- ⇒ **SELECT** clause *cannot mix* scalar *expr* and aggregates *agg*.

 #019

⁶ SQL clause processing order ≠ SQL syntactic order. DuckDB implements a “friendly SQL dialect”  to partly rectify this (e.g., allowing FROM-SELECT queries). We'll use such DuckDB-specific features sparingly.

10 : Sum of Quantities ⑦ — SQL

Aggregate function `sum()` is what we need to formulate a SQL variant of the benchmark query over the CSV file:⁷

```
D .timer on
D SELECT sum(l_quantity)
   FROM read_csv('lineitem.csv',
                 header = false,
                 names = [ ..., 'l_quantity', ... ])
```

#020

sum(l_quantity)
153078795



used CPU time

Run Time (s): real 0.448 user 3.322090 sys 0.133819

- 🕒 Query time on Torsten's computer: ~0.45s.
 - Overall CPU time is >3s: DuckDB uses **parallel processing**.

⁷ Command `.timer on` instructs the DuckDB CLI to report query times for all subsequent SQL commands. The DuckDB documentation contains a [complete list of such commands](#) ↗.

Interlude: SQL EXPLAIN

SQL DBMSs typically implement an **EXPLAIN facility**⁸ that exposes details of the system's **query evaluation plans**:

- Supports query and performance debugging.
- **Shows order of query operations** explicitly, making effects of query optimization visible (e.g., projection pushdown).
- **Measures query behavior during execution**, annotates plan with:
 - breakdown of how query operations use time,
 - # of rows processed,
 - size of intermediate results (in bytes), ...

```
D EXPLAIN
  query;
```

Do *not* run query. Show query plan and estimated row count.

```
D EXPLAIN ANALYZE
  query;
```

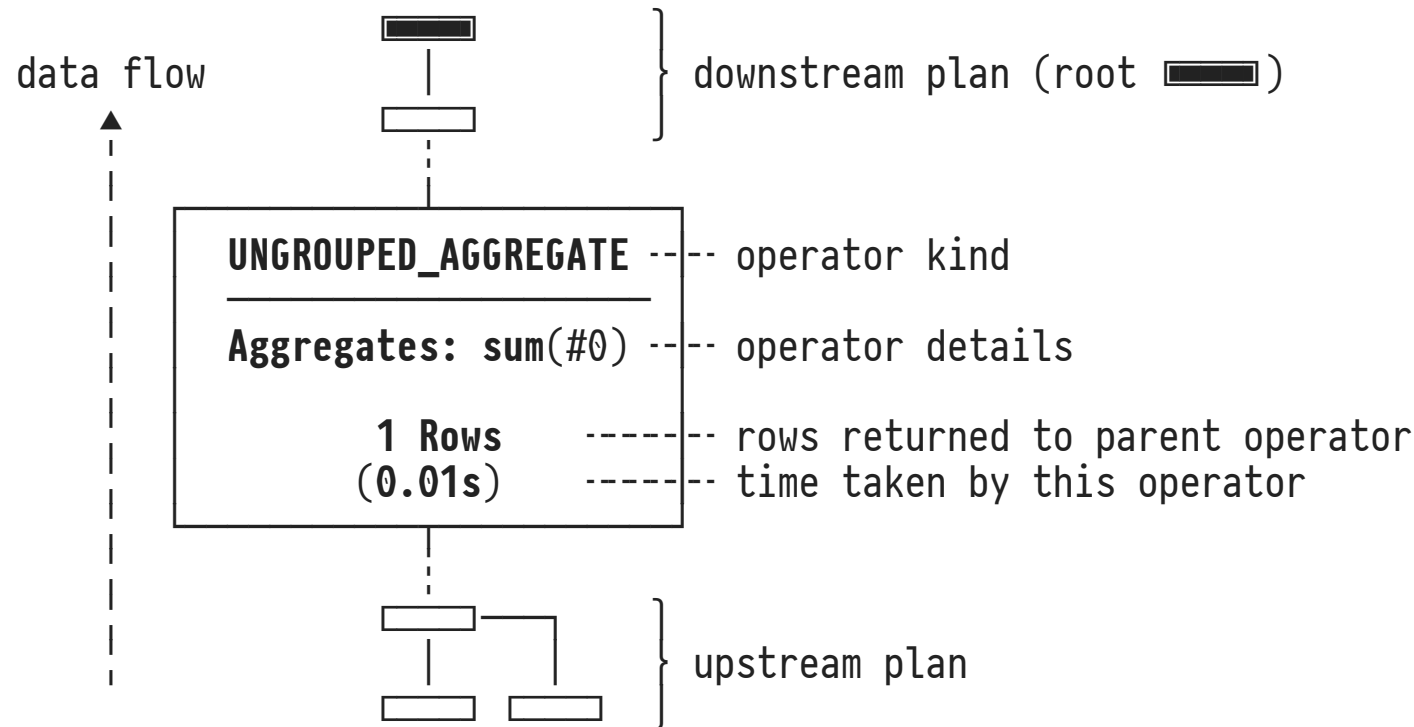
Actually run query. Measure times/rows, annotate plan.

⁸ DuckDB's **EXPLAIN facility** is extensive. Can see even more plan details via **EXPLAIN (FORMAT json) query**.

Interlude: SQL EXPLAIN

Query evaluation plans visualize bottom-up **data flow**:⁹


- Data sources (e.g., `TABLE_SCAN`) reside at the leaves.
- Query result is produced by the root (top-most operator).



⁹ U Tübingen has developed the [DuckDB Execution Plan Visualizer](#) which can render and inspect plans in the web browser (use `EXPLAIN (ANALYZE, FORMAT json) query` to produce plans that the visualizer can render).

Sum of Quantities 7 — SQL

```
D SELECT SUM(l_quantity)
   FROM   lineitem;
```

 #020

sum(l_quantity)
153078795

```
Run Time (s): real 0.002 user 0.007288 sys 0.000294
```

A 🕒 query time of 0.002s for the benchmark indicates that

- the DBMS uses multiple cores (threads) to evaluate SQL queries,
- the query plan does *not* scan—or skip over—all 16 columns of table `lineitem` (projection pushdown focuses on `l_quantity`),
- column values are *not* stored as text and thus do *not* have to be parsed again and again, and that
- the data for table `lineitem` does reside in DRAM (not in secondary storage).