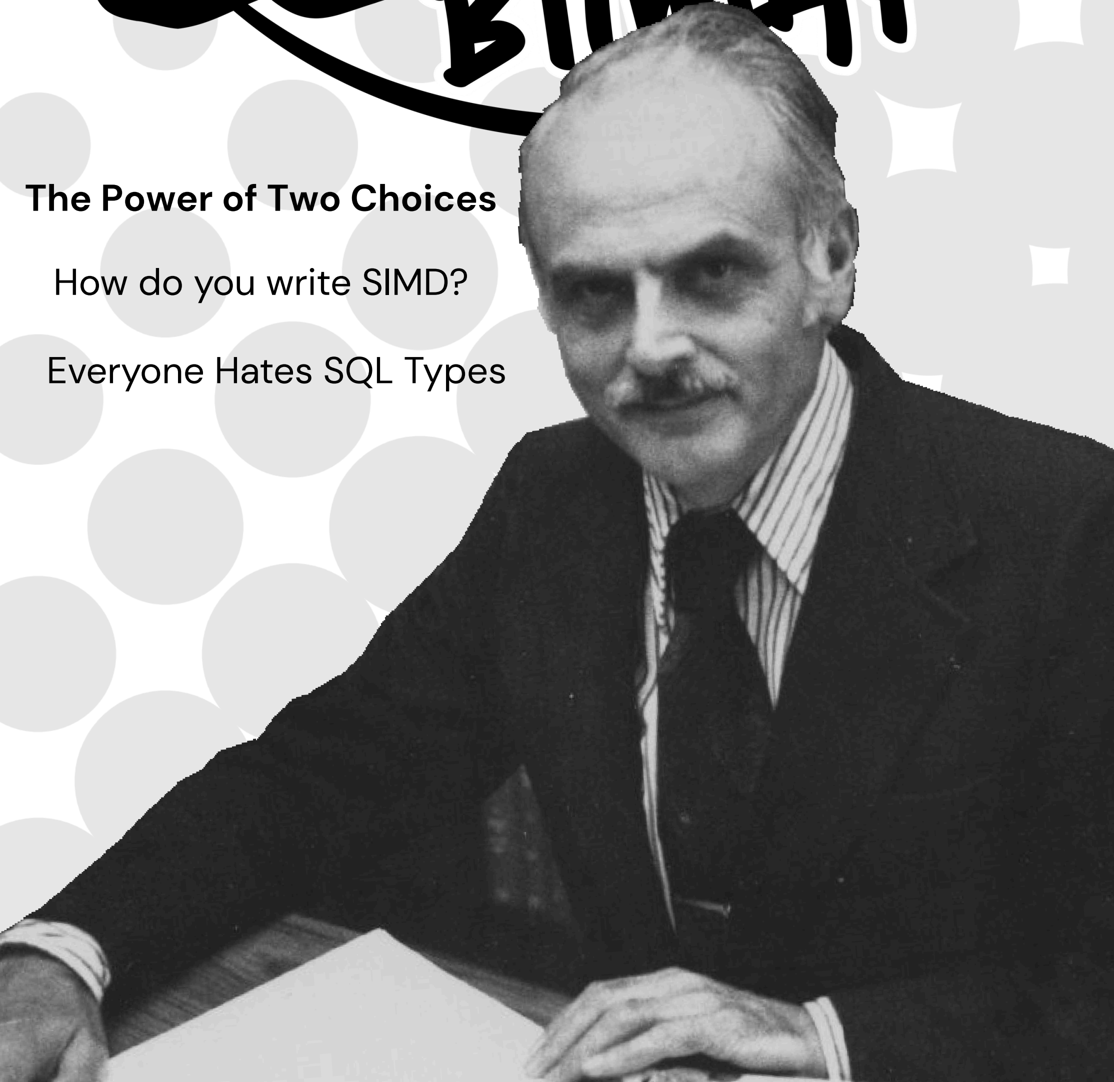
ISSUE #1 AUGUST 2025

NULL BITMAP

The Power of Two Choices

How do you write SIMD?

Everyone Hates SQL Types



IN THIS ISSUE

PHOTOGRAPHS BY CRISTIAN STAFIE

No syscalls?

No problem.

`io_uring.`

available in linux kernel versions 5.1 and higher. some syscalls are required in order to set up the ring buffer but afterwards io can proceed syscall-free. any security vulnerabilities are figments of your imagination and any benchmarks in which this loses to vendor-provided specific apis are null and void by virtue of cross-device limitations and standards. this advertisement not endorsed by jens axboe or facebook or the linux kernel or anyone else who could reasonably have an ownership claim over io_uring. please don't sue me

The perils of moral relativity: Type systems in relational models

Editor's note: As ardent defenders of free speech, we publish letters we receive as we receive them.

This is grand but true: type systems provide structure so that we may codify and share interpretations of the world. They provide the praxis of interpretation, through which we may reticulate the current state of the world, program by program, version by version.

The challenge then is how you move between types, i.e. interoperatability. The question of how to share meaning. In short, we simply must impose an order on the relationships of these types. Bigger types are the natural successors to smaller types; their expressive domain forms a superset. Ordering types by domain forms a hierarchy that is simple to understand, intuitive, and creates frictionless communication within and between systems.

Polytrees are our best hope in the chaos of the world.

Woe betide, though, those who dabble in the profanity of cliques, for once entered, extrication is possible only through great conviction, forceful coercion, and extraordinary foresight. Such is the case with PostgreSQL's String type category.

Imagine, for instance, you are a busy computer science professor, and have outsourced some menial tasks to a young woman on campus, such as doing data entry for your last semester's students' reviews. Rather than using the existing `student_reviews` table in your database, she creates an entirely new table `student_review_spring_summer` with, of course, its own distinct schema.

I will spare you the gory details of these fateful decisions, but for simplicity's sake, I will say that she decides to model the `review` column, where students select a value of either `Poor`, `Average`, `Good`, or `Excellent`, as `TEXT`. I, having suffered the vagaries of the IT department's "optimizations" for many years had chosen `CHAR(9)`--this has the benefit of giving you string trimming during equality for free, which meant that I needed to do less cleaning of the sloppily maintained data.

Hers:

```
CREATE TABLE student_review_spring_summer
  (…, review TEXT)
```

Mine:

```
CREATE TABLE student_reviews
  (…, review CHAR(9) NOT NULL)
```

Now, as you all are database professionals yourself, you know that life's tasks distract you and sometimes you have to make things work in a pinch. Imagine, as well, that you don't realize what a wreck you've waded into until you begin preparing your tenure packet the day before its due date. Surely, with your sympathy, you can empathize with using PostgreSQL's type coercion system to help smooth out the wrinkles of this mess.

I also imagine you, like me, will be shocked by the bald-faced lie that PostgreSQL tells in its statement that `TEXT` is the String type category's preferred type--i.e. if the hierarchy of types is respected, any type in that category will be forcefully coerced into the superior type if that type is present. When I need to reconcile `TEXT` and `CHAR` columns, they will uniformly coerce to `TEXT`. With that assumption, I proudly submitted this query to the tenure committee demonstrating my student reviews (query simplified for legibility):

```
WITH all_reviews AS (
   SELECT review FROM student_reviews
   UNION ALL
   SELECT review FROM student_reviews_spring_summer
)
SELECT
   COUNT(*) AS total_reviews,
   ROUND(COUNT(*) FILTER (WHERE review_text NOT IN ('Poor',
'Needs Improvement')) * 100.0 / COUNT(*), 1) as
positive_review_percentage
FROM all_reviews;
```

Of note, we must submit the underlying data alongside our queries, so that the tenure committee can “check our maths.” So, friend, you can certainly understand my surprise when I received a letter which not only notified me that tenure was being withheld, but also suggested that I take a leave of absence. How could this have occurred?
After puzzling and poring over the PostgreSQL source code I have found that the term “algebra” in “relational algebra” is meaningless to Michael Stonebraker and Tom Lane! Two so-called “luminaries” in both relational databases and open source are charlatans, concerned with appeasing mewling JavaScript developers, rather than adhering to the mathematical principles software must be built on.

What motivates this invective? Here:

```
WITH all_reviews AS (
   SELECT review FROM student_reviews_spring_summer
   UNION ALL
   SELECT review FROM student_reviews
)
SELECT
   COUNT(*) AS total_reviews,
   ROUND(COUNT(*) FILTER (WHERE review_text NOT IN ('Poor',
'Needs Improvement')) * 100.0 / COUNT(*), 1) as
positive_review_percentage
FROM all_reviews;
```

```
 total_reviews | positive_review_percentage
---------------+---------------------------
           314|           84.1
```

Both eagle- and bleary-eyed readers will notice that I have simply re-ordered the relations in the `UNION`, which causes PostgreSQL to generate results of totally different types in the intermediate relations! In my original query, the `UNION` generated a type of `TEXT` simply because it came first. In this second, damnable query, the `CHAR` type comes first, generating a totally different set of equality semantics. This is not

```
 total_reviews | positive_review_percentage
---------------+---------------------------
           314|           69.1
```

commutative. It is not deterministic. It is the caprice of the weak-minded. And worse yet––`TEXT`, `CHAR`, and `VARCHAR` don't form some sort of rational hierarchy; they form a clique, able to be transmuted into one another willy-nilly! You can demonstrate this for yourself with the profane incantations below:

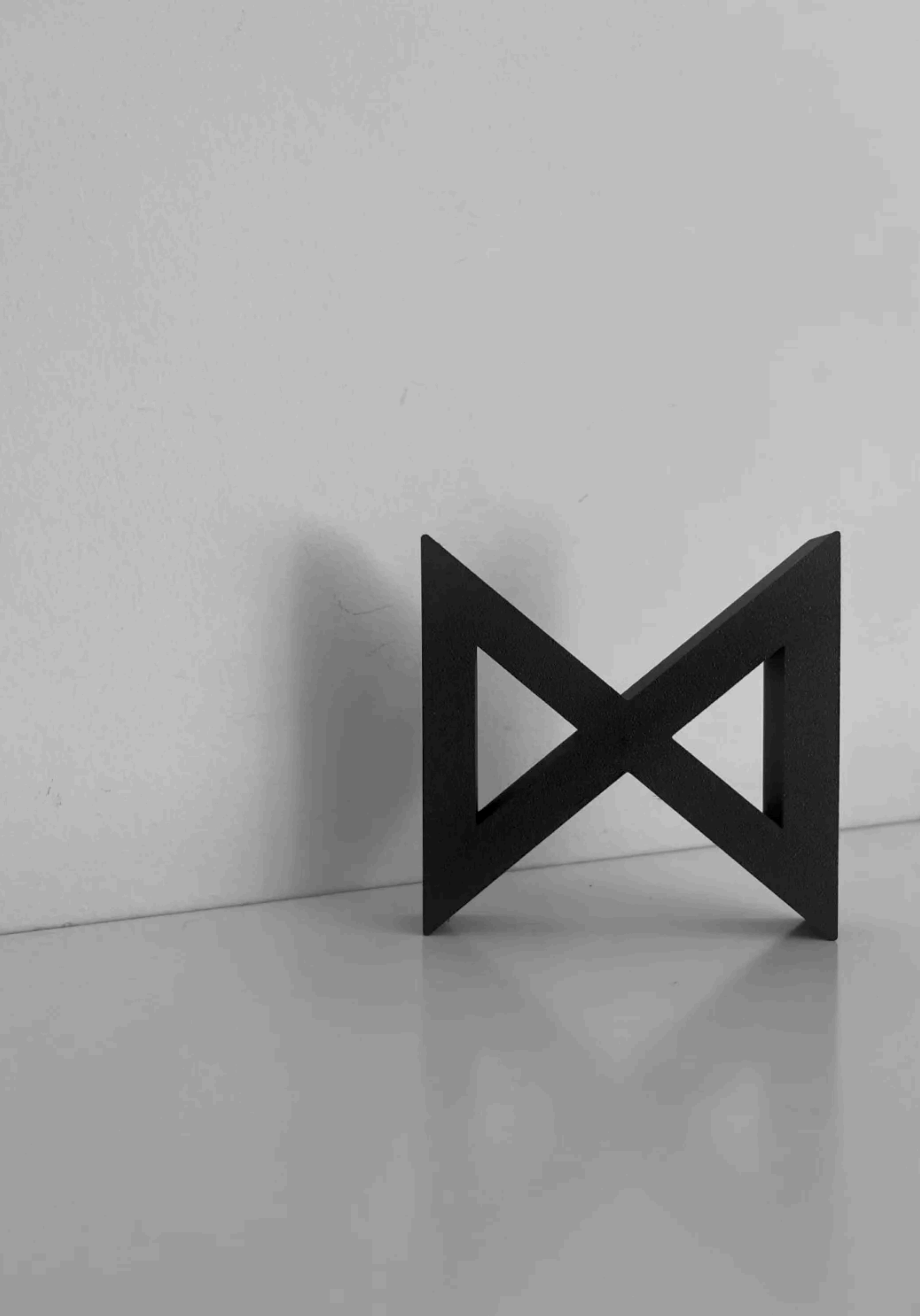
```
SELECT pg_typeof(
  COALESCE('abc'::TEXT,
           'abc'::CHAR));
SELECT pg_typeof(
  COALESCE('abc'::CHAR,
           'abc'::TEXT));
```

What benefit to us is commutativity (a core tenant of an algebra) if it isn't applied? Is mathematics simply optional to these people? They've boxed themselves into a corner where there are no good decisions to be made because they have treated all things as equals.

So––what compels me to pen this? Sympathy? Job offers? No––I need neither. After this dismal foray into academia, I plan to join my uncle's high-frequency trading firm––a task with tangible rewards, unlike the carrot dangled in front of educators to "help to shape future generations."

I see a worrying parallel between this kind of permissive attitude toward type coercion (which, you recall, violates a natural ordering of the relationships between data) and the permissive attitude I see emerging culture around us. I hope that the readers of this fine publication will clearly see the perils of this kind of “anything can be anything” thinking. For the world we’ve built to continue to work as it has, we need the kind of clear, well delineated order that polytrees provide––cliques be damned.

Arthur Pointsman



THE POWER OF TWO CHOICES

FORTE SHINKO

If you love computers as much as I do, then you probably love hashing equally so. Imagine that you have a hash function $X \to \{0, 1, 2, \ldots, N-1\}$, and suppose you hash N random elements of X. The best possible thing that could happen is that the N elements are sent to distinct values, but this simply won't happen if X was much larger than N. However, a more reasonable expectation is that there won't be too many such hash collisions, since for instance, it would also be very unlikely for half the elements to map to the same value.

To get a better sense of how many collisions there are, we'll work in a specific probabilistic model, namely the balls-and-bins model. In this model, there are N balls and N bins, and we throw each of the N balls uniformly at random into one of the N bins. At the end, each of the bins has a "load", which is the number of balls in it, and a reasonable measurement for the number of collisions is the "max load", which is the maximum load among all the bins. It turns out that for N balls and N bins, the expected max load is $O(\log N)$ (this isn't sharp).

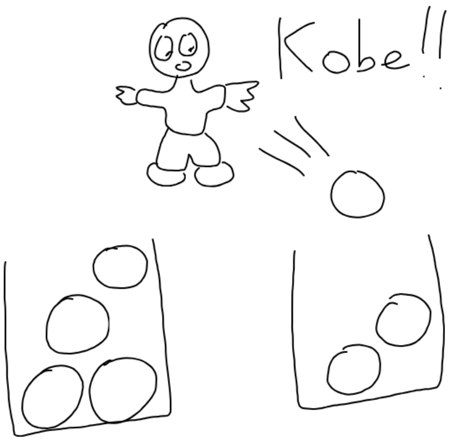
To get an idea of where log N comes from, note that the load of any given bin follows a binomial distribution approximating a Poisson process with rate 1. Since the Poisson distribution decays (at least) exponentially, a given bin will have probability at most 1/N of having load log N. Since there are N bins, in expectation there is at most one bin with load log N, indicating that the max load should also be around there.

The actual expected max load is $\Theta(\log N/\log\log N)$, which is basically still log N . To do better, we can employ the Power of Two Choices, introduced in the paper "Balanced Allocations" by Azar-Broder-Karlin-Upfal.

In our balls-and-bins model, the Power of Two Choices looks like the following: there are N balls and N bins, and for each ball, we choose TWO bins uniformly at random, and put the ball in the bin with the smaller load. Although it doesn't look like we've changed much, it turns out that this has a tremendous effect on the expected max load, dropping it all the way down to O(loglogN).

To see why, first note that at most half the bins can have ≥ 2 balls, since there are only N balls total. In particular, this means that the expected fraction of bins with ≥ 2 balls is at most 1/2. Now how about ≥ 3 balls? Well on a given ball's turn, how likely is it that it will increase the number of bins with ≥ 3 balls? For this to happen, the two bins it is presented with have to have 2 balls each. Since each individual bin has probability at most 1/2 of this happening, this means that the probability of increasing the number of bins with ≥ 3 balls is at most $(1/2)^2$ = 1/4. Summing this up over all the balls, we see that the expected fraction of bins with ≥ 3 balls is at most 1/4.

If we repeat this for the expected fraction of bins with ≥ 4 balls, we again see that the only way for a ball to make a new bin with ≥ 4 balls is to be presented with two bins of 3 balls each, which has probability $(1/4)^2$ = 1/16 of happening, showing that the expected fraction of bins with ≥ 4 balls is at most 1/16. In general, we see that the expected fraction of bins with ≥ k balls is at most $1/2^{(2^k)}$. So the expected fraction of bins with ≥ log log N balls is at most 1/N , which is at most one bin in expectation, so we should expect the max load to be around log log N.



THINKING IN OPLOGS

Evan Vigil-McClanahan

Jay Kreps called the log "real time data's unifying abstraction", and often you'll hear this statement extended to all distributed systems; they're all just a materialized view of a log of logical operations. It is mundane (even foundational!) to note it, but this notion can be extended to all programs of any complexity. But Turing's machines seem abstract, and might swim hazily in the memory with the other complexities of algorithms class, so let's restate it. A program takes a stream of inputs, and is fundamentally a state machine that turns those inputs into memory changes, IO operations, and etc. It's possible, if perhaps onerous, to pair each operation with the delta it causes within the system. One can use this for undo, or for time travel debugging.

I can hear you asking, "Why spell something so obvious out like this?"

It's worthwhile, because going through the effort involved in rethinking your system in these particular abstract terms comes with the strongest return available to you in programming. It helps with writing systems that are correct and in learning how to make more certain that they're correct. The oplog is a macroscope (an abstractoscope?), allowing you to zoom out from the details of how a particular operation is executed, and to start thinking about the interrelationship of those operations, their dependencies and costs. This view opens up new perspectives on API design. Do two endpoints logically belong to the same operation? Then they should probably be a single call. Two logical operations fused into a single call? Now you know why there are fifteen arguments.

It also opens up new frontiers in testing. Critically, interoperational dependencies are properties! Which means that stateful property based testing is now there, waiting for you to write a stateful test to permute its way to operational interleavings you could never have imagined, but your users will almost certainly try.

No system of thought is perfect. Log-systems do not always cleanly decompose into tractable sub-log-systems (although when they do, you've discovered a clean testing boundary). Invisible state dependencies can be teased out via careful thought, experimentation, and testing, but if left unknown can cause a great deal of misery and late-night debugging. And because the world is the world and computers are actual machines, nondeterminisic behavior in systems is impossible to fully stamp out. Regardless, this way of thinking about your programs and systems of programs is fruitful and important, and will lead to you growing as a writer of programs and a designer of systems and subsystems. You might even learn what transactions are for along the way!

[1]With titanic effort, one can even linearize and determinize the entire operating system and multithreaded programs to produce testing systems like Antithesis.

SIMD ATOI

ALEX MILLER

I’ve been learning about how to write SIMD code recently, and I’ve come to appreciate that there’s two distinct skills involved: knowing how to transform a given loop into one that has sufficient parallelism to apply SIMD, and knowing how to fit a parallel loop’s computation into the restricted set of available SIMD instructions. We’ll be looking at only the first part of that today: learning the pattern of how to transform one loop type into its parallel version.

Simple examples of SIMD optimizations tend to show off the speedups on problems that are already embarrassingly parallel. map is an easy function to parallelize: there’s no dependencies between loop iterations, so each element can be handled in parallel trivially. reduce with a commutative operation is similarly simple: repeatedly apply the operation over pairs of elements in parallel until there’s only one value left. Many problems fit the shape of a map() and a reduce(). Suppose we want to compute the numerical sum of an array of ASCII digits: this involves a map() to convert each to its integer representation, and then a reduce() to sum them together. And we can neatly fuse those in one loop:

scalar

```
int sum_ascii_digits(
    char* digits,
    size_t length)
{
  int sum = 0;
  for (int i = 0;
       i < length;
       i++) {
    sum += digits[i] - '0';
  }
  return sum;
}
```

SIMD

```
int sum_ascii_digits(
  char* digits,
  size_t length)
{
  // Process 4 digits at a
  // a time across 4 lanes.
  int sum1 = 0, sum2 = 0,
      sum3 = 0, sum4 = 0;
  int i = 0;
  for (; i+3 < length; i+=4)
  {
    sum1 = digits[i] - '0';
    sum2 = digits[i+1] - '0';
    sum3 = digits[i+2] - '0';
    sum4 = digits[i+3] - '0';
  }
  for (; i < length; i++) {
    sum1 += digits[i] - '0';
  }
  // Sum the lanes by adding
  // pairs in parallel.
  sum1 += sum3; sum2 += sum4;
  sum1 += sum2;
  return sum1;
}
```

Such transformations make a lot of sense in my head, because it's transforming an iteration over an array from one element at a time to many elements at a time, but the overall linear structure of the algorithm stays almost exactly the same. Any loop that's just a map() and reduce() of a commutative operation can be transformed in this fashion.

There's other patterns for SIMD solutions which I find fascinating because they require contorting the scalar solution of a linear scan into a completely different shape of an algorithm. We're going to be looking at the slightly more difficult problem of parsing integers instead. Given a string of 8 digits, I want to know what unsigned 64-bit int they represent. The scalar solution to this is a pretty simple linear scan:

```
uint64_t parse_int(
  char* digits) {
  uint64_t parsed = 0;
  for (int i = 0;
       i < 8;
       i++) {
    parsed = parsed * 10 +
      digits[i] - '0';
  }
  return parsed;
}
```

However, there's not a lot of opportunity for parallelism in this. Each iteration depends on applying an operation (multiply by 10) to the output of the previous iteration. If we fully multiply out

```
(((d[0] * 10 + d[1])*10 + d[2]) * 10 + d[3]) * 10 + ... + d[7]
```

We're left with

```
d[0] * 10^7 + d[1] * 10^6 + d[2] * 10^5 + ... + d[7] * 10^0
```

And we can write an algorithm that follows that style of computation instead, by building up the power of 10:

```
uint64_t parse_int2(
  char* digits) {
  uint64_t parsed = 0;
  uint64_t power_of_ten = 1;
  for (int i = 7;
       i > 0; i--) {
    parsed += (digits[i] -
      '0') * power_of_ten;
    power_of_ten *= 10;
  }
  return parsed;
}
```

However, this still leaves us computing O(digits) numbers of powers of 10, and each iteration still depends on the result of the previous iteration. But can go another step further. Extracting out a common factor of 10^4 from the first half of the elements, and then we're left with two very similar computations to perform, with one final multiply-and-add:

```
(d[0] * 10^3 + d[1] * 10^2 + ... + d[3] * 10^0) * 10^4 + (d[4] * 10^(3) + d[5] * 10^2 + ... + d[n] * 10^0)
```

This looks like a nice optimization, as it means we never need to compute a power of 10 higher than half the number of elements. But we can also apply the same extraction of common factors again and again:

```
((d[0] * 10^1 + d[1]) * 10^2 + (d[2] * 10^1 + d[3])) * 10^4 +
((d[4] * 10^1 + d[5]) * 10^2 +
 (d[6] * 10^1 + d[7]))
```

Which leaves us with a nice tree-shaped computation where we can perform all the multiply-and-adds that require 10^1, then compute 10^2 and perform all the multiply-and-adds which require that value, then compute 10^4 and … ad. nauseum. We only ever need to compute squares of 10!

```
Level 0: ['1', '2', '3', '4', '5', '6', '7', '8']
          |    |    |    |    |    |    |    |
          └──┬─┘    └──┬─┘    └──┬─┘    └──┬─┘
Level 1:     12        34        56        78
             | *10^2+  |         | *10^2+  |
             └────┬────┘         └────┬────┘
Level 2:        1234                5678
                  |      *10^4 +     |
                  └────────┬─────────┘
Level 3:                12345678
```

This idea can be translated back into C:

```c
uint64_t parse_int_tree(char* digits) {
    // For the sake of clarity, first
    // transform the digits into an
    // array of the integral values.
    uint64_t parsed[8] = {0};
    for (int i = 0; i < 8; i++) {
        parsed[i] = digits[i] - '0';
    }
    // Now follow the tree-shaped computation.
    // We multiply-and-add pairs of elements,
    // assigning the result back to the left-hand
    // side.
    uint64_t power_of_ten = 10;
    for (int level = 0; level < 3; level++) {
        for (int i = 0;
             i < 8; i = i + 2 << level) {
            parsed[i] = parsed[i] * power_of_ten
              + parsed[i+(1<<level)];
        }
        power_of_ten *= power_of_ten;
    }
    return parsed[0];
}
```

And now, we finally have a loop where each iteration doesn't depend on the previously resulting value. We've achieved parallelism!

From there, we could unroll our loops (#pragma clang loop unroll(full)!) to get a straight line of instructions to execute, but gcc/clang will do that for you already. In the true spirit of SIMD, we can further optimize this by packing the operations for multiple digits into one value. In SIMD land, you'll typically see this as a significant amount of masking and shifting. We mask to find each of the tens digits, we shift it to line up with the ones digits, perform the multiply-and-add, and then use a wider mask to do the same for hundreds and ten thousands. This is the SIMD-within-a-register (SWAR) technique:

```c
#include <endian.h>

uint64_t parse_int_swar(char* digits) {
  uint64_t digits_bytes = *(uint64_t*)digits;
  uint64_t digits_bcd = digits_bytes - 0x3030303030303030UL;
  // If the host is little endian, then loading it as a uint64_t
  // will mean the least significant byte is the most significant
  // digit, and it's mentally easier to think of it the other way.
  // This mental ease costs us one `bswap` instruction.
  digits_bcd = htobe64(digits_bcd);

  uint64_t tens_upper_mask = 0xFF00FF00FF00FF00UL;
  uint64_t tens_lower_mask = 0x00FF00FF00FF00FFUL;
  uint64_t level_one = ((digits_bcd & tens_upper_mask) >> 8) * 10 +
                       (digits_bcd & tens_lower_mask);

  uint64_t hundreds_upper_mask = 0xFFFF0000FFFF0000UL;
  uint64_t hundreds_lower_mask = 0x0000FFFF0000FFFFUL;
  uint64_t level_two = ((level_one & hundreds_upper_mask) >> 16) * 100 +
                       (level_one & hundreds_lower_mask);

  uint64_t tenK_upper_mask = 0xFFFFFFFF00000000UL;
  uint64_t tenK_lower_mask = 0x00000000FFFFFFFFUL;
  uint64_t level_three = ((level_two & tenK_upper_mask) >> 32) * 10000 +
                         (level_two & tenK_lower_mask);

  return level_three;
}
```

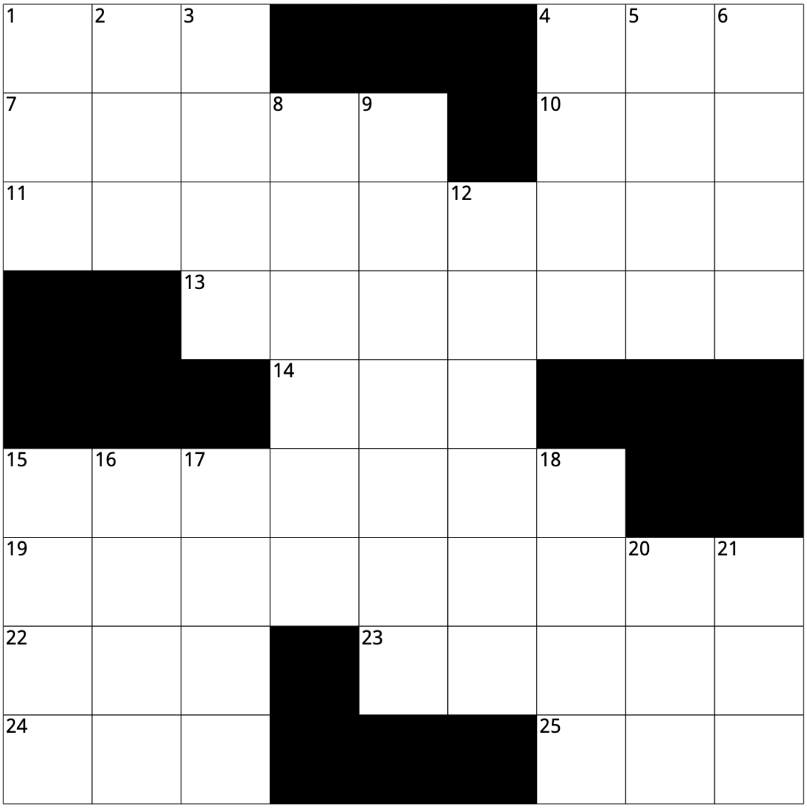
In general, any fold comprised of commutative operations can be computed in this fashion to unlock parallelism. SIMD-ifying code is easy when it's already embarrassingly parallel. The fun is in trying to find the right way to contort seemingly serial algorithms into parallel ones!

So, what did our optimizations achieve?

```
Benchmark                Time             CPU   Iterations
-----------------------------------------------------------
parse_int1           0.426 ns        0.425 ns   1667920355
parse_int2           0.421 ns        0.420 ns   1665745819
parse_int_tree       0.484 ns        0.483 ns   1483969012
parse_int_swar       0.421 ns        0.420 ns   1666246273
```

Nothing! But it sure was fun!

If you're interested in more of this, highload.fun gives a nice framework and set of challenges for trying to get practice at applying SIMD to real problems. What we've looked at is only a small portion of the first "parsing integers" challenge.



ACROSS

1 Hosp. scan
4 STDIN and STDOUT, say (abbrev.)
7 Indian yogurt drink
10 Mine find
11 One eschewing internal data
13 Style with straight black bangs
14 Internal availability target
15 Treelike cactus
19 Dangerous DDL
22 Japanese honorific
23 Be
24 Purple yam
25 OCaml cousin

DOWN

1 World Series org.
2 Knock
3 Archipelago part
4 Places for discussion
5 Three, to TUM folk
6 Oracle
8 Does a particular aggregation
9 Run at SERIALIZABLE
12 Chest
15 SoCal school
16 Spirited horse
17 Left
18 Kimono sashes
20 Structure for write-heavy workloads (abbrev.)
21 Proc. to load a warehouse

GETTING REAL WITH SQLITE

MAX FORBES

Dear NULL BITMAP,

I was naive, but you can understand why I picked it. Its social media profile was glowing: Litestream, D1, The Library of Congress. Everyone on the orange website loved it. Couldn't stop talking about its exhaustive test suite.

Of course, I'm talking about choosing server-side SQLite.

At first, the relationship went well. Great communication: exhaustive docs, if a bit old school (raw HTML). But it all nearly fell apart when I couldn't get a straight answer about a double.

SQLIte's Four Data Type Abstractions — When making a new column, SQLite always walks the type through the same four abstractions (below).

The only choice I make is to write **(1) the declared type**. I can put anything I want there, like NUMERIC or DATETIME or even FLAMING_TROUSERS.

SQLite brings **(2) its type affinities**. When I say BIGINT, it hears INTEGER… but it also hears INTEGER if I say CHARINT or CHICKEN_ INTERRUPTION.

Then we have **(3) the storage classes**. It only has five: NULL, INTEGER, REAL, TEXT, BLOB. Every declared type gets mapped to one of these five with the type affinity rules. SQLite says all I should worry about are storage classes, that I shouldn't think any further. But I know about its…

(4) Datatypes. It "makes a big difference on disk," it says. "The INTEGER storage class, for example, includes 7 different integer datatypes of different lengths." (I count 8?) But when I ask: well how do I know what you're choosing? Can I ask you about them in SQL? Even if I learn one of your secret

Declared type	Type affinity	Storage class	Datatype
BIGINT	match "INT"	INTEGER	BE 16-bit twos-c int

PRAGMAs? It just says no.
What confounds me is that even if a column storage class is REAL, a value in there can be stored as a REAL, TEXT, or BLOB. And it can be different for each value.

Double Trouble
SQLite told me upfront: I can't handle 47.49. In great detail:
"If you provide a "price" value of 47.49, that number will be represented in binary64 as: 6683623321994527 × 2-47 Which works out to be: 47.4900000000000001989519 66012828052043914794921875 [...] It is a mathematical limitation inherent in the design of floating-point numbers."
I believed it. From them on, I avoided REALs.
I always used Python to talk to SQLAlchemy, and SQLAlchemy to talk to SQLite. Both Python and SQLAlchemy know about Decimals. They tell me, yes, we'll keep it exact, we'll store it as text. Don't worry about, say, summing your per-token teensy LLM costs. It'll be accurate.

But SQLite—it won't talk about Decimals. SQLAlchemy gives SQLite a Decimal, and SQLite throws a tantrum — stores it as a REAL, tells me all guarantees are out the window. I had to write my own extension to make sure Decimals survive as TEXT.

A Secret is Spilled
One night, I decided to actually test SQLite on it. Can it really not do Decimals? Will 47.49 truly bring it to its knees?

```
sqlite> CREATE TABLE foo (id
INTEGER, value REAL);
sqlite> INSERT INTO foo VALUES
(1, 47.49);
sqlite> SELECT * FROM foo;
1|47.49
```

I couldn't believe it. Were the docs lying to me the whole time? I flailed:

```
SELECT printf("%.50f", value) as
reveal_yourself from foo;
47.490000000000000000000000000000
00000000000000000000
```

I called Claude. We probed:

```
sqlite> SELECT id, value,
typeof(value) as storage_class,
length(value) as byte_length,
hex(value) as hex_representation,
quote(value) as quoted_value FROM
foo;
1|47.49|real|5|34372E3439|47.49
```

Aha, yelled Claude! It's a liar! IEEE 754 doubles are 8 bytes, but it's only using 5! It's using a compressed storage format.

Even check the hex—5 pairs (34 37 2E 34 39) becomes 4 7 . 4 9.

Floating Telephone

But Claude is one of those friends who is stupid and jumps to conclusions.

I looked up length(x) in the exhaustive docs, and I found (emphasis mine): “For a string value X, the length(X) function **returns the number of Unicode code points (not bytes)** in input string X.” I think length() was just operating on the string.

I called my friend ChatGPT. Also stupid, but gossip triangulates truth. ChatGPT told me to use these decimal extensions:

```
sqlite> SELECT decimal(value)
FROM foo;
47.4900000000000019895196601282
8052043914794921875
sqlite> select
ieee754_mantissa(value),
ieee754_exponent(value) FROM
foo;
6683623321994527|-47
```

It was ChatGPT’s turn to scream: Aha! Now that proves it! But fool me once, can’t get fooled again. Couldn’t it be that these functions, would just do the same thing on any old post-hoc value? Just like length(x)?

```
sqlite> select decimal(47.49);
47.4900000000000019895196601282
8052043914794921875
```

I say this means nothing was proved. We must go deeper.

Look at the bits

Finally I called you, NULL BITMAP, and you told me to diff the bits. You told me: make two similar databases, xxd ‘em (dump the raw bytes), and diff ‘em. I tried this with 47.49 and another random number 17.48, and found:

```
… 07 40 47 BE B8 51 EB 85 1F 00 …
… 07 40 31 7A E1 47 AE 14 7B 00 …
```

Just 7 bytes were different. After some sleuthing, I could understand it: the previous 07 is the record’s “serial type,” indicating the next 8 bytes are a big-endian 64-bit float. The spec: IEEE 754. In common parlance: a double. This means the 40 is part of each number. Now we can decode the first one, which should be 47.49.

Remember old IEEE 754:

```
  Sign (S)      =  1 bit
  Exponent (E) = 11 bits
  Fraction (F) = 52 bits
  ----------------------
                 64 bits
```

```
      40       47       BE       B8       51       EB       85       1F
01000000 01000111 10111110 10111000 01010001 11101011 10000101 00011111
```

```
    0 10000000100 0111101111101011100001010001111010111000010100011111
    S           E                                                    F
```

The sign (S) is 0, positive. The exponent (E) we read straight off as an int, 1028, then per-spec subtract 1023, to get 5.

F is weird. A binary fraction which gets added to 1. Going by bit, we sum:

- The first bit (0) is 2^(-1) or ½
- The next bit (1) is 2^(-2) or ¼
- The next bit (1) is 2^(-3) or ⅛
- … and so on.

A quick Python script later (where I had to use proper Decimals AGAIN else everything got rounded) gave me this fraction in decimal:

0.484062500000000062172489 3790087662637233734130859375

Final formula: S * 2^E * 1.F. And it just so turns out that 1 * 2^5 * 1.484062500000000062172489 3790087662637233734130859375 = 47.49000000000000198951966012828052043914794921875000 00. Just what it said on the tin.[1]

So it turns out the REAL was a double whole time, SQLite was just doing some nice float printing that I don't know how to inspect or interrogate or disable.

What did I learn?

Looking at the raw bytes and the spec was a bit liberating. There aren't actually that many datatypes: NULL, 8 int lengths, 1 double, raw BLOB, and TEXT (likely UTF-8). It's all big-endian, and it's all really truly right there, on disk.

At the same time, both the CLI and Python's interfaces rounded the floats, which made it hard to trust what was happening under the hood. This confirms using Decimal as an application-level abstraction on top of SQLite's TEXT was a sound path.

Maybe this whole thing is why I keep going back to text files. They don't keep any secrets.

Yours,
Missing Floats

[1]Their representation, 6683623321994527 × 2^(-47), is from writing the binary fraction as a 6683623321994527 times 2^(-52), which combines with our 2^5 exponent to get 2^(-47).