July 16, 2024

Why German Strings are Everywhere

Many data processing systems have adapted our custom string format. Find out what makes it so special and why it is so relevant to them.

Lukas Vogel

German Strings

Strings are conceptually very simple: It's essentially just a sequence of characters, right? Why, then, does every programming language have their own slightly different string implementation? It turns out that there is a lot more to a string than "just a sequence of characters"[1].

We're no different and built our own custom string type that is highly optimized for data processing. Even though we didn't expect it when we first wrote about it in our inaugural Umbra research paper, a lot of new systems adopted our format. They are now implemented in DuckDB, Apache Arrow, Polars, and Facebook Velox.

In this blog post, we'd like to tell you more about the advantages of our so-called "German Strings" and the tradeoffs we made.

But first, let's take a step back and take a look at how strings are commonly implemented.

Contents

How C does it

How C++ does it

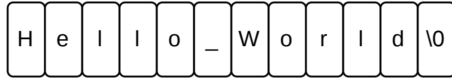
Can we do better?

- Most strings are short
- Strings aren't usually changed all that often
- We usually only look at a small subset of the string

German Strings

How C does it

In C, strings are just a sequence of bytes with the vague promise that a `\0` byte will terminate the string at some point.

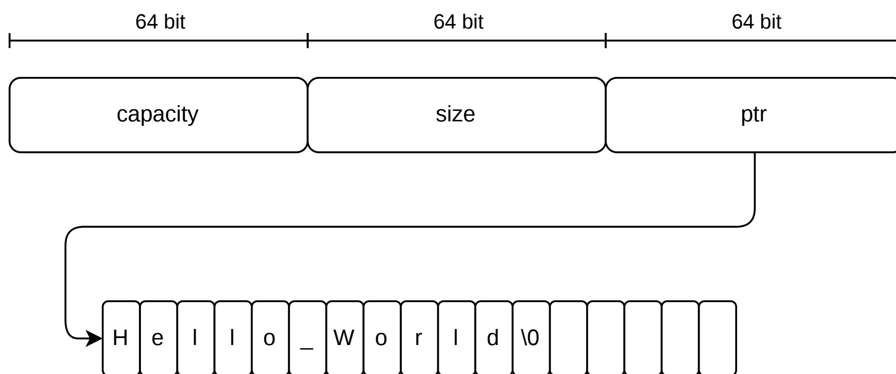


This is a very simple model conceptually, but very cumbersome in practice:

- What if your string is not terminated? If you're not careful, you can read beyond the intended end of the string, a huge security problem!
- Just calculating the length of the string forces you to iterate over the whole thing.
- What if we want to extend the string? We have to allocate new memory, move it there and free the old location all by ourselves.

How C++ does it

C++ exposes much nicer strings through its standard library. The C++ standard doesn't enforce a specific implementation, but here's how `libc++` does it:



Each string object stores its size (already better than C!), a pointer to the actual payload, and the capacity of the buffer in which the data is stored. You're free to append to the string "for free" as long as the resulting string is still shorter than the buffer capacity and the string will take care of allocating a larger buffer and freeing the old one when it grows too much: `std::string` s are *mutable*.

This string implementation also allows for the very important "short string optimization": A short enough string can be stored "in place", i.e., we set a specific bit in the `capacity` field and the remainder of `capacity` as well as `size` and `ptr` become the string itself. This way we save on allocating a buffer and a pointer dereference each time we access the string. An optimization that's impossible in Rust, by the way ;).

If you're interested in a more in-depth overview, take a look at Raymond Chen's very detailed blog post about various `std::string` implementations

German String

Short string representation

Long string representation

Distinguishing between representations

Conclusion

C++, especially with short string optimization, already does quite well. However, if you know your use case, it turns out that you can do much better[2].

While building CedarDB, we made the following observations:

Most strings are short

Despite being able to store arbitrary amounts of text, most people store fairly short and predictable data in their strings (as reported by Vogelsgesang et al. in their fantastic "Get Real" paper).

Examples of such short strings are:

- ISO country codes (`USA` , `DEU` , `GBR`), 3 characters
- IATA airport codes (`LHR` , `MUC`), 3 characters
- Enums (`MALE/FEMALE/NONBINARY` , `YES/NO/MAYBE`), usually less than 10 characters
- ISBNs (`0465062881`), 10 or 13 digits

We definitely want to optimize for such short strings wherever possible.

Strings aren't usually changed all that often

Most data is only written once, but read many times. The `libc++` approach of reserving 64 bits per string just to store the capacity for the off chance that someone wants to extend the string seems kind of wasteful when string sizes don't change all that often.

Also, simultaneously accessing and modifying a string concurrently can lead to data races if we don't use expensive locking techniques or think very carefully about our application.

For these two reasons, we'd like to have *immutable* strings whenever we can get away with it.

We usually only look at a small subset of the string

Take a look at the following to SQL query:

```
select * from messages where starts_with(content, 'http');
```

We only want to look at the first four characters of each string. It seems kind of wasteful to always dereference a pointer just to compare the first four characters.

Let’s look at another query:

```
select sum(p.amount)
from purchases p, books b
where p.ISBN = b.ISBN and b.title = 'Gödel, Escher, Bach: An Eternal Golden Braid';
```

Here, we need to compare all ISBNs with each other and all book titles with a string constant. While we obviously need to compare the entire string to make sure we have a match, most books probably won’t be called “Gödel, Escher, Bach: An Eternal Golden Braid” (how many strings starting with “Gö” do you know, dear non-German reader?). If a character at the beginning of the string already differs, we can rule it out without further checking the rest of the string.

German Strings

To solve these problems, Umbra, the research predecessor of CedarDB, invented what Andy Pavlo now affectionately (we assume ;)) calls “German-style strings”.

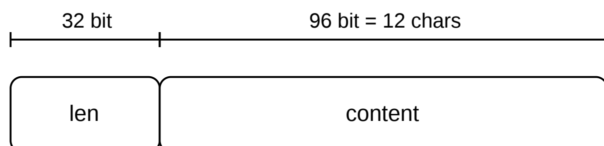
Let’s dig in: The Anatomy of a German String

The first big change is that each string is represented by a single 128-bit `struct`. Besides the obvious advantage of saving a third of the overhead over `std::string` by dropping the `capacity` field, it also allows us to pass strings in function calls via two registers instead of putting them on the stack.

This `struct` contains one of the following two representations:

Short string representation

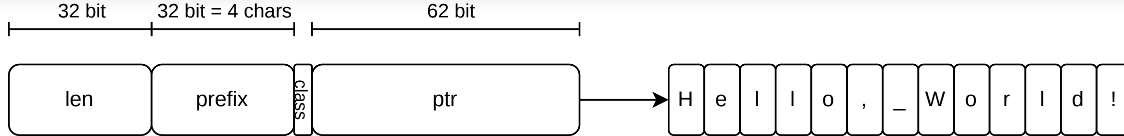
Here’s the memory layout for short strings:



As long as the string to be stored is 12 or fewer characters, we store the content directly in place.

Accessing the content itself, or just a prefix, is easy: Just start reading directly after the `len` field, no pointer dereference needed!

Long string representation



Like C++ strings, we also store a `len` field and a pointer to the data itself, but with some twists:

Length

To squeeze the whole string into 128 bits, we shorten the length field to 32 bits. This limits the string size to 4 GiB, a tradeoff we were willing to make for our use case.

Prefix

Right after the length we also store a four character prefix. This prefix speeds up operations like equality/inequality comparisons, lexicographic sorting, or prefix comparisons enormously, because we save a pointer dereference.

Pointer

The pointer points to a memory region exactly the size of the string. No buffer with some remaining `capacity` here!

The obvious advantage is that we save the 64 bits for the capacity field and can tightly pack payloads of different strings without gaps.

Since the data we're pointing to is now immutable, we can also read it without acquiring a read lock, since the data is guaranteed to never change as long as the string lives.

We need to be aware of the downside as well: Appending data to a string is now a relatively expensive operation, since a new buffer must be allocated and the payload must be copied. However, this is not a big problem in our use case, since database systems rarely update data in-place anyway.

Storage Class

While developing our string concept, we noticed that developers have different requirements for the lifetime of their strings depending on where they use them. We call these "storage classes", which you can specify when creating a string. A string can be `persistent` , `transient` , or `temporary` . To encode this storage class, we steal two bits from the pointer.

Let's start with the cases you might already know from your favorite programming language:

- `temporary` strings behave like their C++ counterparts: When constructing a

- `persistent` strings behave like string constants: They remain valid forever. All short strings are always persistent strings, because you can just pass them via the stack or CPU registers. Long strings can also be persistent, for example, when referencing C++ string literals. The memory that holds the data of the string literal is automatically allocated statically when your program starts and is never deallocated, so referencing the string data is always valid.

But there is a third pattern: The amount of data we need to store in a database system is often larger than RAM, and so some of it is swapped out to disk. With conventional strings, if we loaded a page containing a string from disk, we would have to

- first load the page into memory,
- and then initialize a new string which internally copies the data to the newly initialized memory.

This process copies the string data twice. This is wasteful, since many times we only want to access the string once. For example, consider the following query:

```
select * from books where starts_with(title,'Tutorial')
```

If we filter all books for matches, most strings won't qualify. We'll never need to show them to the person issuing the query, so why copy them if we don't need to access them later?

We would like to have a string that is very cheap to construct and points to a region of memory that is *currently* valid, but may become invalid later without the string having control over it.

This is where `transient` strings come in. They point to data that is currently valid, but may become invalid later, e.g., when we swap out the page on which the payload is stored to disk after we've released the lock on the page.

Creating them has virtually no overhead: They simply point to an externally managed memory location. No memory allocation or data copying is required during construction! When you access a transient string, the string itself won't know whether the data it points to is still valid, so you as a programmer need to ensure that every transient string you use is actually still valid. So if you need to access it later, you need to copy it to memory that you control. If you do not need it later on, we just accessed a string without having to do any expensive initialization!

Distinguishing between representations

How do we know if we're dealing with a short string or a long string? It's actually quite simple! If its size is 12 characters or less, it is a short string. Since our strings

If we just want to access the prefix, we don't even need to check whether the string is long or short. In either case, bits 32-63 will be the first four characters.

Conclusion

German strings have many advantages: You get great performance due to space savings and reduced allocations and data movement. Because data is always treated as immutable, it is also much easier to parallelize your string handling code. Thanks to its concept of storage classes, you can tightly manage the lifetime of your strings, trading performance for ease of use when necessary.

Of course, nothing comes without its challenges: German strings require you to think more deeply about your application: What is the lifetime of my string? Can I get away with a `transient` string, or do I have to copy it? Will my strings be updated often? Am I okay with immutable strings?

If you're okay with asking yourself these questions, you can benefit a lot from German Strings, even if you're not building a database.

Do you want to harness the power of German Strings? Don't be shy and join our waitlist to be among the first to get access to CedarDB!

Join our waitlist!

1. Not to speak of the *contents* of such a character sequence. What even is a character? Say hello to `ä`, `ö` and `(╯°□°)╯︵ ┻━┻`. But that's for another blog post... ↩
2. Disclaimer: Some of the observations here are made through the lens of people building a high performance database system. However, all of the approaches are applicable to strings in all kinds of software. ↩

‹ Working with JSON and Graphs in CedarDB

A Deep Dive into German Strings ›

Supported by

s Projekt LunaBase wird im Rahmen des EXIST-Programms durch das Bundesministerium für Wirtschaft und Klimaschutz und den Europäischen Sozialfonds geförd