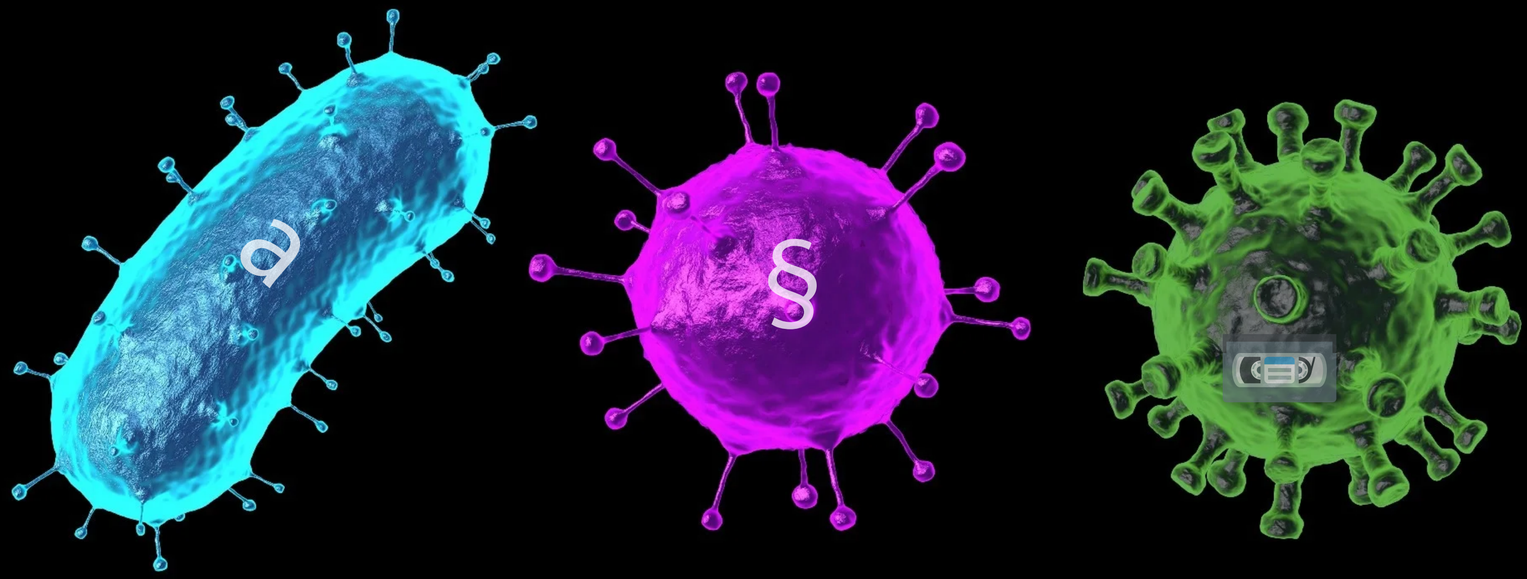

Language is a virus

or, PostgreSQL and Locales

Christophe Pettus, PGX
pgexperts.com

PGX
pgexperts.com

Hi.

- Christophe Pettus
- CEO, PGX Inc. (pgexperts.com)
- Database person for a long time.
- Here to talk about why language was a bad idea and probably shouldn’t have been invented in the first place.

PGX
pgexperts.com

Hi.

- Christophe Pettus
- CEO, PGX Inc. (pgexperts.com)
- Database person for a long time.
- ~~Here to talk about why language was a bad idea and probably shouldn't have been invented in the first place.~~
  - Just kidding.

PGX
pgexperts.com

Glyphs.

What is a “glyph”?

- “A glyph (/ɡlɪf/ GLIF) is any kind of purposeful mark.” — Wikipedia
- These are glyphs: A z 4 ü
- These are glyphs, too: ^ @ ! ‰
- So are these: 這些都是繁體中文的字形 and these: [illegible]
- Even these: 🏈 🐓 🔠 🛑

PGX
pgexperts.com

“Glyph” vs “Grapheme”

- The “glyph” is the mark itself.
  - These are three different glyphs: a  
- The “grapheme” is the underlying functional unit.
  - These are all the same grapheme: b  
- Unfortunately, computing uses “glyph” to mean “grapheme,” so we will too.

So, what's a "character"?

- A glyph more or less corresponds to what programmers call a "character."
- There are some exceptions: in particular, combining characters.
  - Is this one or two glyphs? ë Experts disagree.
  - So, Unicode supports both models.
  - This is a pain, but also beyond the scope of this talk. (Thankfully.)
- It is almost safe to use the terms interchangeably, so we'll do that, too.

We need computers to understand glyphs.

- Since computers only do numbers, we need to have a mapping between glyphs and numbers.
- The number assigned to a particular glyph is its code point.
- A complete set of code points is a character encoding.
- For a long time, that meant…

ASCII

American Standard Code for Information Interchange.

- Achieved its modern form in 1967.
- Assigned a seven bit code to the traditional Latin alphabet, Arabic numerals, a few commonly-used (at the time) symbols, and some control characters.
- Some of the greatest hits include:
  - `A ↔ 65 (0x41)`
  - `a ↔ 97 (0x61)`
  - `* ↔ 42 (0x2A)`

ASCII is pretty cool, actually.

- Upper case letter + 0x20 is the lower case letter.
- Encoding for a digit - 0x30 is the numeric value of the digit.
- And for a long time, ASCII was all we had.
  - OK, there was also EBCDIC, but we don't talk about that.
- This original character encoding is now usually called "7-bit ASCII."
  - Technically, "ASCII" is always 7 bits, but we need to distinguish because…

ASCII was instantly obsolete.

- This was designed to handle the typical 1967-era computer terminal.
- No way of encoding a lot of characters commonly used in non-English languages.
  - (and in English, for that matter.)
- And forget it if your language doesn’t primarily use the Latin alphabet.
- Everyone instantly ran off and started developing their own encoding.
  - Some used characters > 0x7F, some used multibyte characters…
  - Chaos reigned. (IANA documents 259 encodings, and there are plenty more.)

Unicode.

One encoding to rule them all.*

- First published in 1991.
- Attempts to assign a code point to every glyph used in human communication.
  - This is rather ambitious.
- Unicode code points run from U+0000 to U+10FFFF. (Practically, 2^{20} code points.)
- 7-bit ASCII has the first three bytes set to 0x00.
- The “Basic Multilingual Plane” (which includes most natural language glyphs) goes from U+0000 to U+FFFF.

* Tengwar is not yet formally a part of Unicode, although codepoints U+16080 to U+160FF have been reserved for it.

Is Unicode a character encoding?

- Wellllll… yes and no.
  - Unicode is very complex and we don’t have all week here.
- It does map glyphs to code points.
- But Unicode as such doesn’t specify how to store those inside the computer.
- Of course, you could store every single character as 24 bits.
  - But that would be kind of wasteful.

Enter the Unicode Transformation Formats.

- That's what the "UTF" in "UTF-8" stands for.
- These are clever mappings of Unicode into byte streams.
  - Yes, we're encoding an encoding. Computers are great!
- In UTF-8 (and UTF-16), characters are variable length.
- In UTF-8, 7-bit ASCII characters appear just as they normally do.
  - So, UTF-8 is a superset of 7-bit ASCII, which is super-convenient.

How UTF-8 Works.

- Scan along the character string.
- Any byte < U+0080 is interpreted as 7-bit ASCII.
- Otherwise, the Unicode 32-bit code point is represented by a 2-4 byte sequence.
  - ë is Unicode code point `U+00E9`, and is mapped to `0xC3 0xAB`.
  - 🐓 is Unicode code point `U+1F413` , and is mapped to `0xF0 0x9F 0x90 0x93`.

PGX
pgexperts.com

This breaks `strcmp`.

- Let's compare `Z` and `ë`.
- `strcmp` says `Z < ë`, because `0x5A < 0xC3 0xAB`.
- This is wrong for every* language that uses the Latin alphabet.
- Note that the comparison becomes "out of sync," since the number of bytes != number of characters.

* I am certain there are some exceptions I don't know about.

We’re still kind of struggling with UTF-8.

- Decades of code assumed 1 character = 1 byte.
- The code reading UTF-8 needs to know that it is UTF-8 to display it properly.
  - We’ve all seen , Äù on a web page where “ should be.
- And how do we make sure that we are sorting in the right order?
- This brings us to…

Collations.

A collation is just a function.

- A collation takes two strings, and returns if one is greater than, less than, or equal to the other.
  - `f(string1, string2) → (<, =, >)`
- There are a lot of collations out there.
- Each collation is specific to a particular character encoding.
- But an encoding can have lots of collations.
  - UTF-8 on my laptop has 53 different collations.

Collations and languages.

- A collation is (generally) specific to a particular natural language.
- Natural languages can have more than one collation!
  - German sorts names (in a phone book) differently from words in a dictionary.
  - Danish has a different collation for Greenlandic Danish.
- Languages also have different ways of formatting numbers, dates, etc., etc.
- So, a way of bundling all those up was invented, and we called those bundles…

Locales.

A locale is just a bundle of code and supporting data.

- A locale provides a set of utility functions to handle different language- and region-specific formatting.
  - Number format setting
  - Character encoding
    - Some related utility functions (such as how to convert between cases in the encoding).
  - Date-time format setting
  - The string collation to use
  - Currency format setting
  - Paper size setting
- We’re interested in the character encoding and collation part.

PGX
pgexperts.com

POSIX Locales

- POSIX locales are the type of locales you usually see on *nx systems.
  - Windows locales are confusingly similar.
- The name of a typical POSIX locale looks like:

```
fr_BE.UTF-8
```

- `fr` is the language the locale applies to. (French)
- `BE` is the region of the local. (Belgium)
- `UTF-8` is the character encoding for the locale.
- If there's only one character encoding for a particular combination of language and territory, that part is often dropped.

C or POSIX Locale

- A big exception to the rules.
- This compares strings byte-by-byte, like our proud ancestors.
- This is really fast, but it breaks for any non-7-bit-ASCII characters (in UTF-8 or -16).
- It can even break for equality!
  - It's possible for `ë != ë` because of those pesky combining characters.
- Use with caution.

Farm to TABLE: Local(e) Providers.

- Locales are bundled together into libraries: locale providers.
- POSIX locales on nearly all *nx systems are provided by the library `libc`.
  - Which is usually the GNU version, `glibc`.
  - So much so that they are often called “glibc locales.”
- But there are others!
  - The International Components for Unicode (`icu`).
  - And as of version 17, PostgreSQL has its own built-in locale provider.

International Components for Unicode (`icu`)

- A large library of utility functions for Unicode.
- Includes a staggering number of locales.
  - Including a “build your own locale” system.
- You can have locales like `en-US-u-kn-ks-level2`
  - “US English, sort numeric strings as a single number, case insensitive.”
- Supported by PostgreSQL since version 10.

The secret weapon: `und-x-icu`

- A “good enough” collation for the whole Unicode code space.
- Not guaranteed to be correct for any particular language.
- But probably close enough for any non-specialized use.

Locales and PostgreSQL.

Databases need to know about locales.

- Databases use locales all over the place.
  - String comparison.
  - Building indexes.
  - Output functions.
- When you create a new database in PostgreSQL, you specify what locale should apply to it.
- Some things you can change on a table-by-table basis, but you can’t change the character encoding once the DB is created. This means you should…

PGX
pgexperts.com

Just use UTF-8.

There’s really no good reason to use anything else.

- Unless you know, for sure:
  - Your database will never accept any natural languages as data.
  - String comparison is going to be the bottleneck in the database.
- … UTF-8 is the correct choice.

WARNING WARNING WARNING.

- If you use C encoding, the database will happily accept non-7-bit ASCII.
  - Bytes is bytes, right?
- And do absolutely nothing to interpret those characters.
- This means you could end up with mixed encodings in a single table.
  - There are lots of web clients out there that send text in an encoding other than UTF-8.
- This is a mess to fix if it happens.
- So, really, don’t use C encoding.

WARNING WARNING WARNING.

- You may see references to SQL_ASCII encoding.
- Under no circumstances use this encoding.
- Avert your eyes and pretend it does not exist.
- Even speaking of it is dangerous.

OK, but what locale should I use?

- That's a slightly more complicated question.
- Most Linux distros will try to shove `en_US.UTF-8` on you.
  - Which is fine, in a brown-and-green washer/dryer sort of way.
- There are other, better choices.
- Before we talk about that, we have to talk about…

The Doom That Came to PostgreSQL.

First, some thoughts about indexes.

- Indexes are really caches.
- They precalculate the results of a comparison, and store them in a structure that's fast to retrieve.
- The two hardest problems in computing are:
  - Naming things.
  - Cache invalidation.
  - Off-by-one errors.

Indexes on character strings depend on the collation function.

- Obviously.
- But what if the collation function changes?
- Suddenly, the “cache” is invalid.
- But if you don’t rebuild it…
  - … you have a corrupted index.

glibc 2.28

Buried in the release notes was this anodyne statement:

> The localization data for ISO 14651 is updated to match the 2016 Edition 4 release of the standard, this matches data provided by Unicode 9.0.0. This update introduces significant improvements to the collation of Unicode characters. [...] With the update many locales have been updated to take advantage of the new collation information. The new collation information has increased the size of the compiled locale archive or binary locales.

This caused all kinds of mayhem.

- Most non-C-locale indexes with non-ASCII characters were affected.
- Manifested in PostgreSQL as index corruption.
- This version was the one that shipped with:
  - Ubuntu 18.10 (cosmic)
  - RHEL/CentOS 8
  - Debian 10 (buster)

This caused problems with…

- Binary replication across the version boundary.
- System upgrades in place.
- Moving the data directory to a different system.
- Restoring a snapshot from before the boundary to a system that’s after.

Would using icu have fixed this?

- Well, it would have fixed this particular problem.
- But the icu collations change, too.
  - Although less frequently than glibc.
- So, what do we do?

Option 1: Monitor and rebuild indexes.

- As of version 10 (for icu) and version 13 (for glibc), PostgreSQL issues warnings if there is a version mismatch.
- These are emitted into the text logs.
- So check your text logs!
- If you see the warning, rebuild any indexes that are based on character strings.

Option 2: Use the built-in provider.

- As of version 17, PostgreSQL has a built-in locale provider.
- It has two collations:
  - C — Compares the strings byte-by-byte without caring about encoding. Produces bad results for UTF-8.
  - C.UTF-8 — Compares the numeric Unicode code points of a UTF-8 string. Not correct for (any) natural langauge, but much less whacky than C collation, and is fast.
- They both get Z < ë wrong, but C.UTF-8 gets it less wrong.

This is making my brain hurt.

Please, just let me run my database.

Here are some typical cases, with advice for each one.

Case 1:

“I want maximum speed and minimal headaches, I am running on PostgreSQL version 17 or higher, and it’s OK if collation is whacky for non-7-bit-ASCII characters.”

Case 1:

“I want maximum speed and minimal headaches, I am running on PostgreSQL version 17 or higher, and it’s OK if collation is whacky for non-7-bit-ASCII characters.”

Use the C.UTF-8 local from the built-in locale provider.

Case 2:

“I want maximum speed, I am running on a version of PostgreSQL before 17, and I swear that I will never, ever put a character in the DB that is not 7-bit ASCII.”

Case 2:

“I want maximum speed, I am running on a version of PostgreSQL before 17, and I swear that I will never, ever put a character in the DB that is not 7-bit ASCII.”

Then you have my permission to use the C/POSIX locale.

Case 3:

"I'm willing to trade off a little bit of speed so that my collations are reasonable across languages, or I am on a version before 17 and do want to use non-7-bit-ASCII characters."

Case 3:

“I’m willing to trade off a little bit of speed so that my collations are reasonable across languages, or I am on a version before 17 and do want to use non-7-bit-ASCII characters.”

Use the ICU `und-x-icu` locale.

Case 4:

“I need my collation to be spot-on correct for one particular language, but that will be the only language in the database.”

Case 4:

“I need my collation to be spot-on correct for one particular language, but that will be the only language in the database.”

Use the `libc` or `icu` locale specific to your language requirements. (Probably `icu`, see later.)

Case 5:

“My database will have different languages in it, and I’d like collation to be spot-on correct for those languages.”

PGX
pgexperts.com

Case 5:

“My database will have different languages in it, and I’d like collation to be spot-on correct for those languages.”

Create the database as `und-x-icu`, and then put each language in its own table, or column within the table, specifying the appropriate collation for each.

PGX
pgexperts.com

Case 6:

"I am on PostgreSQL 17 or higher, and I never, ever want to worry about my locale provider library changing on me."

PGX
pgexperts.com

Case 6:

“I am on PostgreSQL 17 or higher, and I never, ever want to worry about my locale provider library changing on me.”

Use the `C.UTF-8` locale from the built-in locale provider.

PGX
pgexperts.com

Case 7:

“I am on a version of PostgreSQL earlier than 17, and I never, ever want to worry about my locale provider library changing on me.”

Case 7:

"I am on a version of PostgreSQL earlier than 17, and I never, ever want to worry about my locale provider library changing on me."

Then you are stuck using C/POSIX locale.

Case 8:

"I'm using SQL_ASCII."

Case 7:

“I’m using SQL_ASCII.”

You are a bad person and should feel bad (not really, but come up with a plan to move off of it).

PGX
pgexperts.com

Case 9:

"I'm need to use either `libc` or `icu`, but I am terrified that they will change underneath me and break things."

Case 9:

“I’m need to use either `libc` or `icu`, but I am terrified that they will change underneath me and break things.”

There are some steps to take.

… they are:

- Upgrade to PostgreSQL version 17.
- Never, ever run a primary and a binary replica on two different versions of the OS.
- Never, ever restore a binary snapshot onto a different version of the OS.
- When something underlying PostgreSQL changes (new OS, etc.), be sure to respond at once to the WARNING that is generated when a locale provider library changes by doing:
  - ALTER COLLATION … REFRESH VERSION;
  - REINDEX DATABASE;

Using icu: CREATE COLLATION

- **Using icu after database creation requires an additional step: CREATE COLLATION.**
  - `CREATE COLLATION german_phonebook (provider = icu, locale = 'de-u-co-phonebk');`
  - `CREATE TABLE "das Buch" ( … ) COLLATION german_phonebook;`

Using icu: CREATE DATABASE

- You can specify icu locales directly in CREATE DATABASE (as of v15):
  - `CREATE DATABASE mydb locale_provider=icu icu_locale='und-x-icu' encoding='UTF8' template=template0;`
- ‘template0’ is required because the usual template database (template1) already has an encoding and locale, and you can’t change that.

Speed?

One million records, SELECT * FROM … ORDER BY…

Provider	Locale	Time
builtin	C.UTF-8	486ms
libc	POSIX	470ms
icu	und-x-icu	765ms
icu	de-AT	770ms
libc	de_AT	3645ms

Conclusions.

Locale stuff is hard.

- But manageable.
- Make a conscious choice of what locale to use.
  - Don’t just take what the system throws at you.
- `icu` is under-appreciated: it’s very powerful, very fast, and changes less often than `glibc`.
- If you think you won’t have non-7-bit-ASCII characters in your database, you’re probably wrong.

Thank you!

Questions?


PGX
pgexperts.com