Getting to UTF8 @ 20k TPS

Sup?

So, what's the problem?

- Anyone, anywhere can use your web site.
- Partially compatible character sets hit your software and database.
- Incoming data:
  - may not be character set tagged
  - may be mis-tagged
  - may be mixed or inconsistent character sets
  - could have embedded control sequences (Word…)
  - assumed character set is incorrect

Rut Ro!

- All your character string data is now suspect.
- Not certain *any* character string data is correct.
- Or even usable.
- Your database is running at 20k TPS.
- Can’t take downtime.
- Can’t tell how long to fix it.
- Not even sure you ***can*** fix it!

Get Smart!

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

http://www.joelonsoftware.com/articles/Unicode.html

Character Sets 101: Why You Care

- No characters in the computer
- Characters are mapped to numbers (bytes)
- This is called “character codes”
- A collection of character codes is a “code page”
- Code pages remapped the same bytes to different characters
- Everybody did it (IBM, HP, DEC, MS, ...)
- People using different code pages could not communicate
- Some countries have multiple code pages (Russia, China, etc.)

Unicode to the Rescue!

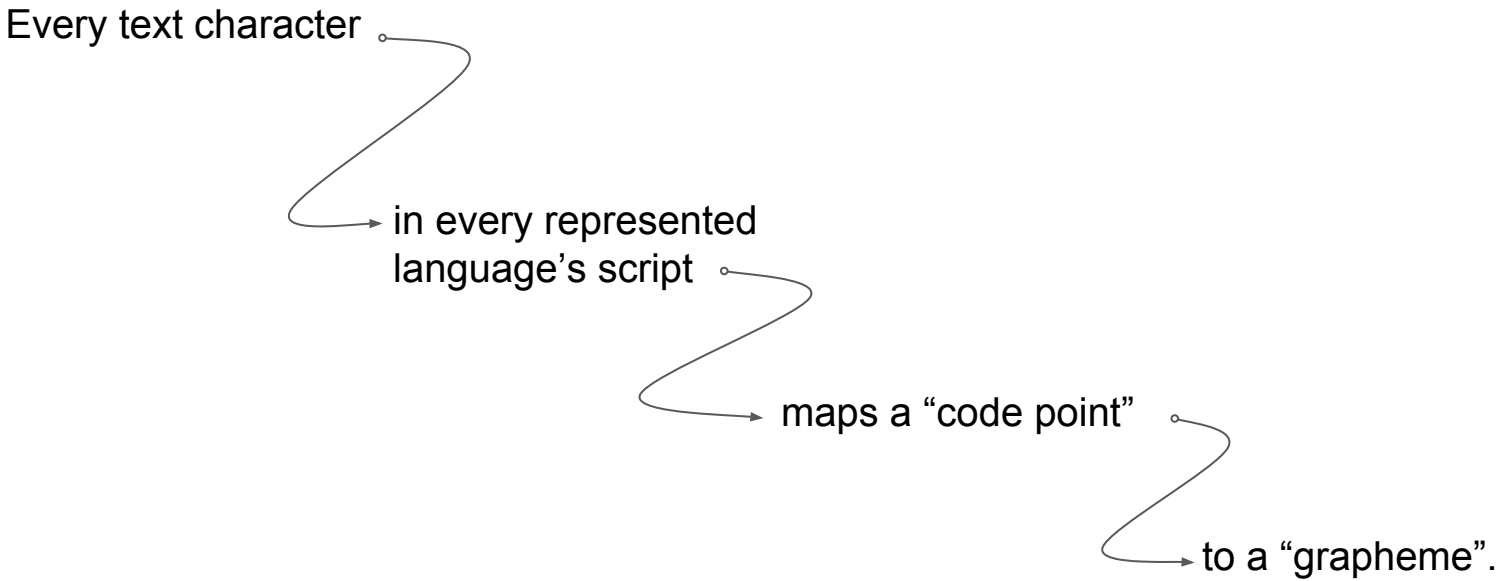
- Computing industry standard
- Maintained by the Unicode Consortium
- Covers most of the world's writing systems
- Even Esperanto! (check the database for Esperanto...)
- 8.0 is latest Unicode version
- 120,000+ characters
- 100+ scripts
- symbol sets (even emojies!)

Unicode 8.0, continued

Defines:

- reference code charts and data files
- encoding methods
- standard character encodings
- rules for normalization
- decomposition
- collation
- text rendering
- bidirectional display order

Unicode TL;DR



Through the Looking Glass

- “code point” - a number in a “code space”
- “code space” - a set of code points
- “grapheme” - smallest unit of a particular writing system
  - letters (a, b, c)
  - symbols ($ % @ & *)
  - punctuation (, . : ; ? !)
  - accent marks
- Unicode maps a code point to a grapheme.
- Graphemes are not glyphs!
- Glyphs are what you see on the screen
- Glyphs are mapped to fonts

Unicode Graphemes

- “U+” <two byte hex number>
- Grapheme is actually a *description* of the mapped character, not the character itself
- For instance:

Code Point	Grapheme
U+00E9	“LATIN SMALL LETTER E WITH ACUTE”

- And you see:

Unicode Mapping Methods

Two mapping methods:

- Unicode Transformation Format (UTF)
- Universal Character Set (UCS)

- Started out with good intentions
- Resulted in several mappings

The Road Back to Hell

- UTF-1 - Used 32 bit code points; non-performant on the machines of the time
- UTF-7 - 7 bit code points; ASCII only
- UCS-2 - fixed-width 16-bit encoding (2-byte Universal Character Set)
- UTF-16 - two-byte code points
- UTF-16LE - UTF16 for little endians
- UTF-16BE - UTF16 for big endians
- UTF-EBCDIC - Unicode for Big Iron
- UTF-32 - like UTF-16, but four byte
- UCS-4 - A superset of UTF-32

UTF8 For The Win!

The One True Encoding:

- 8-bit encoding
- Variable-width
- No byte-endianness funny business
- Backwards compatible with ASCII
- 1-4 byte encodings for Unicode

UTF8 Single Byte Encoding

- ASCII

UTF8 Multibyte Encoding

byte <= 127 ? stands_alone : (multibyte)

- First byte has two or more high-order 1s followed by a 0
- Continuation bytes all have '10' in the high-order position
- Each high-order 1 in a multibyte sequence's leading byte means "number of bytes in that sequence"
  - e.g. 0x11000000 => three byte sequence
  - the next two bytes are part of the same multi-byte sequence

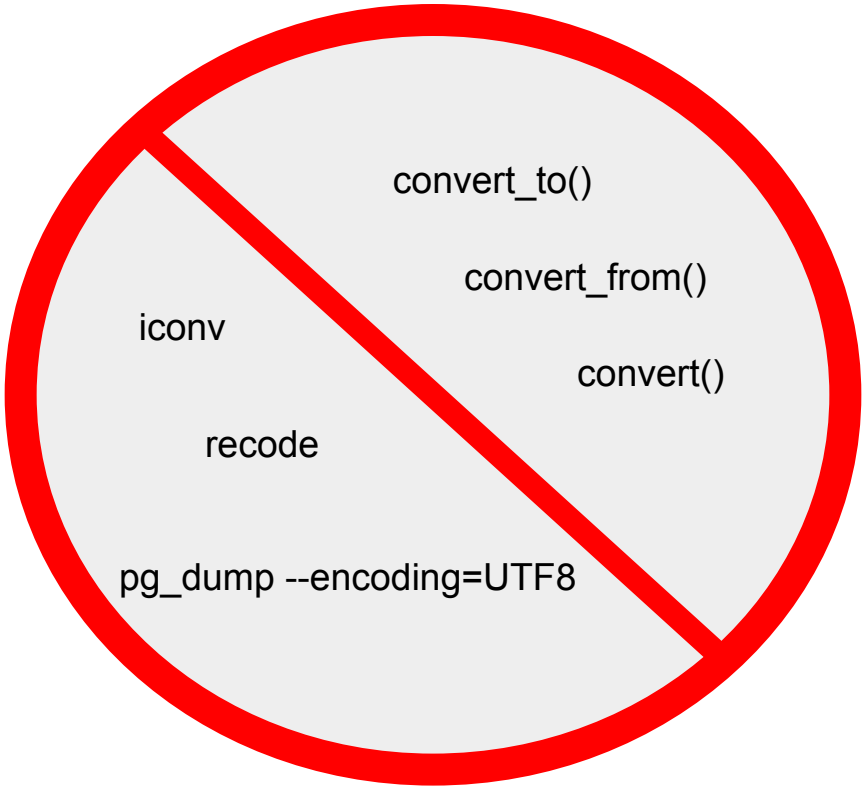
The Problem with SQL_ASCII

- **SQL_ASCII**:
  - extremely efficient
  - fast collation
  - fast sorts
  - other goodness
- ***BUT***:
  - Not ASCII
  - Not an encoding at all
  - It actually means “no encoding”
  - You: INSERT …
    PGSQL: “Sure, whatever.”

Then It Hits You...

- Char columns can hold any byte sequence!
- Separate columns in same row can have different encodings
- Different encodings in a single char column!!!
- Char indexes are subtly broken.
- Comparison functions can’t.
- ORDER BY doesn’t.

Up the Creek


convert_to()
convert_from()
iconv
convert()
recode
pg_dump --encoding=UTF8

Character Set Encoding, Detection & Identification

- Have to figure out the encoding on the fly.
- Must isolate char data and examine it independently.
- Identification is not guaranteed
- Some character set encodings overlap and look the same
  - ISO-8859-1 (Latin1)
  - Win1252 (Latin1, CP1252, Win1252)

The Good, The Bad and The Ugly

- **Good** - likelihood of a good guess goes up with the length of the character string
  - > 1kb is good
  - overlapping another encoding == no harm making a wrong guess

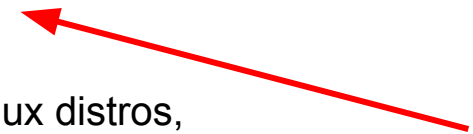
- **Bad** - confidence for shorter strings gets worse quickly

- **Ugly** - will never be deterministic
  - take a SWAG
  - hold your nose
  - rinse
  - repeat

Tools of the Trade

Useful Libraries:

- Mozilla charset detector - the great grandpappy
- Libchardet - open source Mozilla charset detector
  - works best for large documents
  - isn't designed for what we're doing
- International Components for Unicode
  - actively maintained by IBM
  - libraries and headers for most Linux distros,
  - yum and apt-get installs
  - source!
  - ICU license



Use this one!

Needle in a Haystack

Now you have to devise a way to inspect all the things, but:

- only check char-based columns
- large objects and bytea columns don’t count
  - unless you threw text in them
  - in which case they do count
  - so don’t do that
- can’t be sure which rows have been converted
- can’t be sure if converted rows haven’t been unconverted
- the schema might change on you!

To GitHub, Robin!

- pg-chardetect
  - Extension wrapping ICU
  - Detects character set of string-based column
  - Can convert to UTF8
  - Meant for use with triggers and trigger functions

- pg-utf8-transcoder
  - Transcodes PostgreSQL string data to UTF8

- pg-utf8-audit
  - Checks UTF8 compliance of character strings in a PostgreSQL database

Functions and Triggers and C, Oh My!

pg_chardetect pros:

- Can customize trigger functions
- Can toggle triggers
- Drop triggers when done
- It’s fast (~300k conversions/second, depending on schema)
- Transactional

```
BEGIN;
  YUCK!
ROLLBACK;
```

Functions and Triggers and C, Oh Bother!

pg_chardetect cons:

- Additional load
- Memory leaks
- Segfaults
- Database crash
- Incorrect encoding detection:
  - no warning
  - conversions might bork your data

Outside Looking In

pg-utf8-transcoder

Pros:

- Can run in parallel
- Crashes won’t take the database down
- You’ll sleep better at night

Cons:

- Not transactional
- Can still transcode data incorrectly
- Nothing prevents data from “reverting” to non-UTF8 data

Belt & Suspenders: Checking Your Work

pg-utf8-audit

- For checking the results of transcoding.
- Examines char data, checks if Python can decode it as UTF8
- If not, attempts to decode as Latin1, then re-encode as UTF8
- Writes UPDATE statements to file for non-UF8 data
- Ran it by restoring a database dump to its own machine (2.5 TB)
- GNU parallel is your friend:
    - 16 parallel jobs for 1300+ tables
    - ~ 6 hours

Restore, Check, Tweak, Repeat

- Must inspect each UPDATE statement for correctness
- Apply it to production database
- Wait for another backup cycle to complete
- Repeat entire restore/check/tweak cycle
- Not fast, but:
  - Sometimes you don’t want fast, you want thorough.
  - This is one of those times.

Leeky Data Bukkitz

- Apps put non-UTF8 data into the database
- Have to find all and fix all inputs of non-UTF8 data,
- Check periodic inputs (scripts, cron jobs, etc.)
- Maintenance scripts,
- Client programs that may do something like:

```
SET CLIENT_ENCODING TO LATIN1;
```

The Promised Land

- All data has been transcoded to UTF8 or deleted
- All verification runs verify UTF8 encoded data
- What is the next step?

Flipping the database encoding bit from SQL_ASCII to UTF8.

In the PostgreSQL system catalogs.

Database administration just got real!

Are We There Yet?

```
$ psql -d postgres \
 -c "select datname, encoding from pg_database where datname = 'template0' "

   datname  | encoding
------------+----------
 template0  | 0
(1 row)
```

Going for Broke

```
$ psql -c “update pg_database set encoding = 6 where datname = <mydb>”
$ pg_ctl restart -D <data directory>
$ pg_dump --encoding=UTF8 1> /dev/null 2> pg_dump.stderr
$ vacuumdb --analyze --verbose <mydb>
$ reindexdb --verbose --index=<char indexes> <mydb>
```

Use the Source, Luke

Download, build, and install pg_chardetect on your database host, then:

```
sudo su - postgres
dropdb test
createdb --encoding=SQL_ASCII --locale=C test
psql test -f $(pg_config --sharedir)/contrib/pg_chardetect.sql
cd test-data/
zcat test.dump_p.gz | psql test
psql test -f pg_chardetect_test.sql
```

The Source Awakens

Download, build and install pg-utf8-transcoder on your database host, then:

```
sudo su - postgres
dropdb test_transcoder
createdb --encoding=SQL_ASCII --locale=C test_transcoder
for f in $(ls -1 sql/*.sql); do psql test_transcoder -f $f; done
cd test-data/
zcat test.dump_p.gz | psql test_transcoder
transcoder --help
./run-test.sh
```

Happily Ever After? Not So Fast!

- Use best practices
- Backup your data!
- Keep history tables
- Try, try again on throwaway databases
- Take your time and be cautious to a fault
- Check, check & double check

You too can be a hero for doing the impossible!