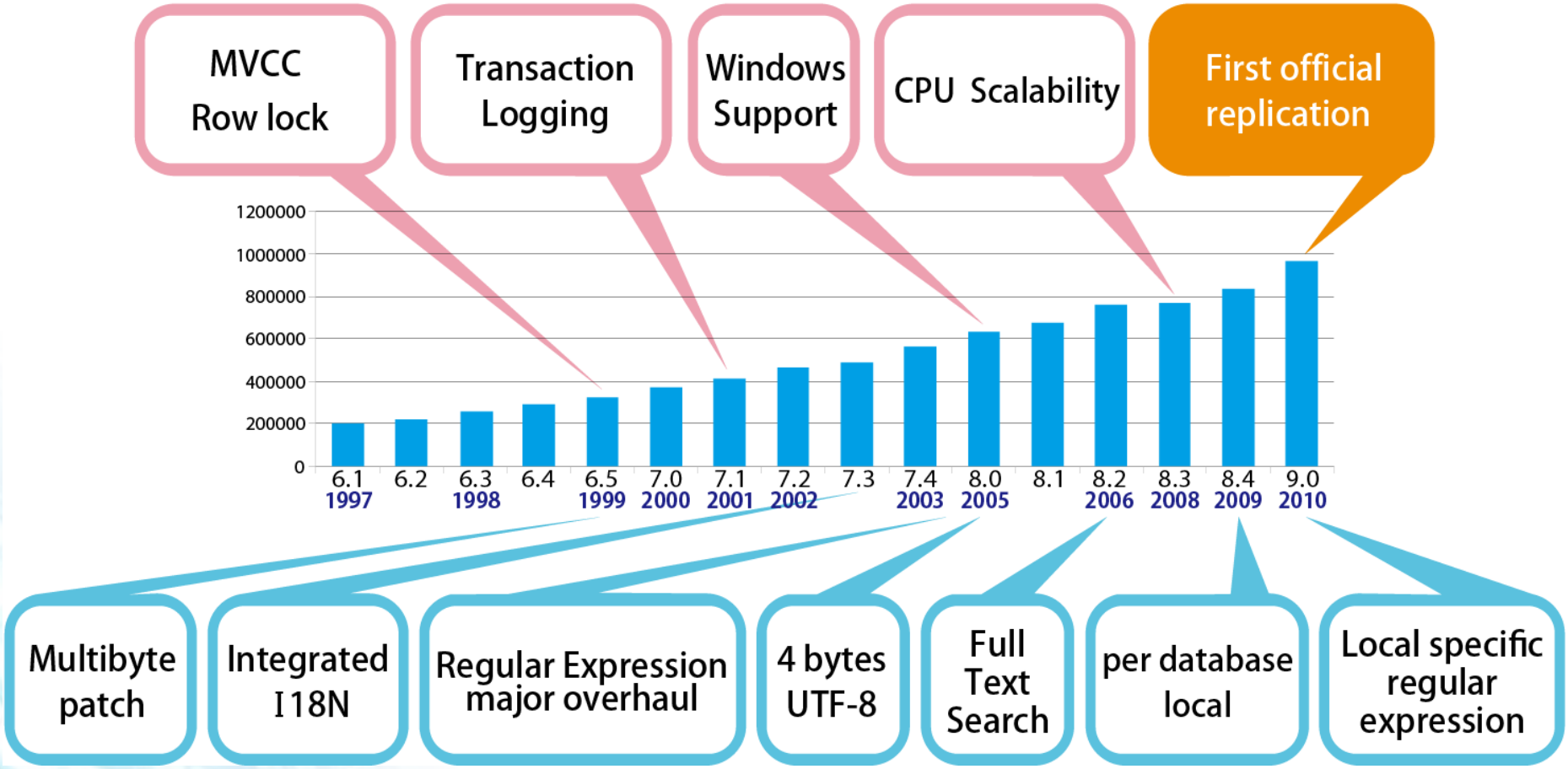
Road to a Cosmopolitan Database System: The History of internationalization of PostgreSQL

SRA OSS, Inc. Japan
Tatsuo Ishii

Who am I?

- Working for SRA OSS, Inc. Japan, an OSS service company
- One of the committers of the PostgreSQL project
- Developing PostgreSQL related OSS
  - Pgpool-II(http://pgfoundry.org/projects/pgpool/)

PostgreSQL and I18N History



The early days of PostgreSQL

- PostgreSQL 6.0, the first version, was born in 1996
  - No consideration for character sets except ASCII
  - Fortunately the code accepted 8-bit-set bytes
    - Most database operations are ok with Japanese Kanji characters as long as using EUC-JP
      - Data insertion, table names…
    - Except the regular expression query

What is the regular expression query in database anyway?

- “Grep against database contents”
  - One of the powerfull features in PostgreSQL
  - SELECT * FROM foo WHERE **bar ~ '^some.*'**;
- Standard DBMS have just “LIKE” which is poor compairing with regular expression
  - SELECT * FROM foo WHERE bar LIKE 'some%';
  - Only '%' and '_' are allowed as wildcard
- Afterwards(1999) regular expression was defined as a part of the SQL standard

The implemetation

- Orginal code used Henry Spenser's regexp code. Modified it to allow wide characters
- Allowed character set is defined at the compile time
  - EUC_JP(1-3 bytes): Japanese Extended UNIX code
  - EUC_CN(1-3 bytes): Chinese Extended UNIX code
  - EUC_KR(1-2 bytes): Korean Extended UNIX code
  - EUC_TW(1-4 bytes): Taiwan Extended UNIX code
  - UNICODE(UCS-2)
  - Mule Internal: Multilingual editor used this code

Social aspect of the patches

- Most European/US commnuity memebers had no knowledge about I18N
- Need to explain what is “Kanji” and why Japanese need it over and over again
- The modification was carefully made not to be too “invasive” to the original code
- Do not make “Japanese only patch”. Key factor of the acceptance is benefit for all over the world

Anyway...

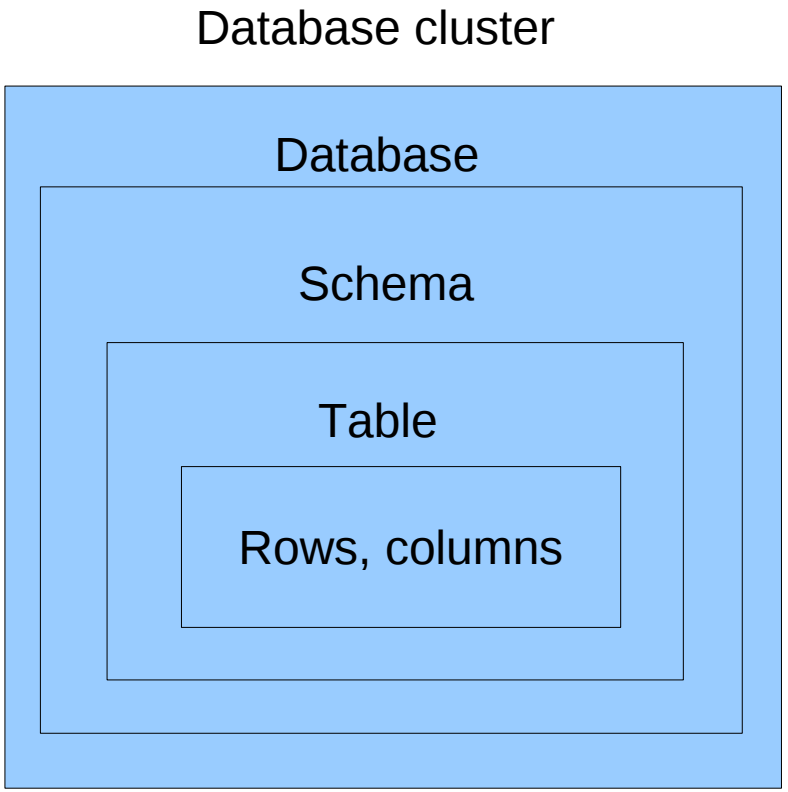
- The patch was accepted and released with PostgreSQL 6.3.1 in 1998
- Next year I became a committer of PostgreSQL

Toward true 18N

- After the first multibyte patches published several concerns were remained:
  - Support multiple characters sets, at least per database
  - “Neutral” to character set. No particular character set should not be “primary” or “dominant”
  - Transparent character set conversion between client and DB server
  - Performance is the top priority as usual

Support multiple characters sets

- Allow to choose character sets when creating “database”
- Database is chosen when a session is started
- All tables in a database have same character set



Character set and Encoding

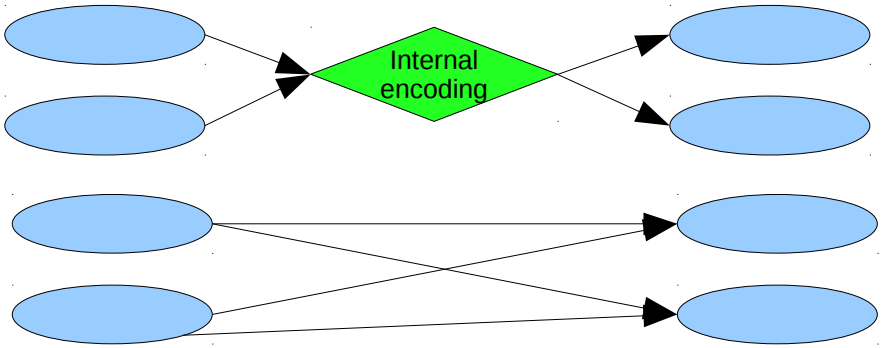
- Character set != Encoding
- A character set could have multiple encoding
- An encoding could have multiple character sets
- Encoding defines physical representation of particular character set thus precisely specifies character set
- PostgreSQL specifies encoding rather than character set

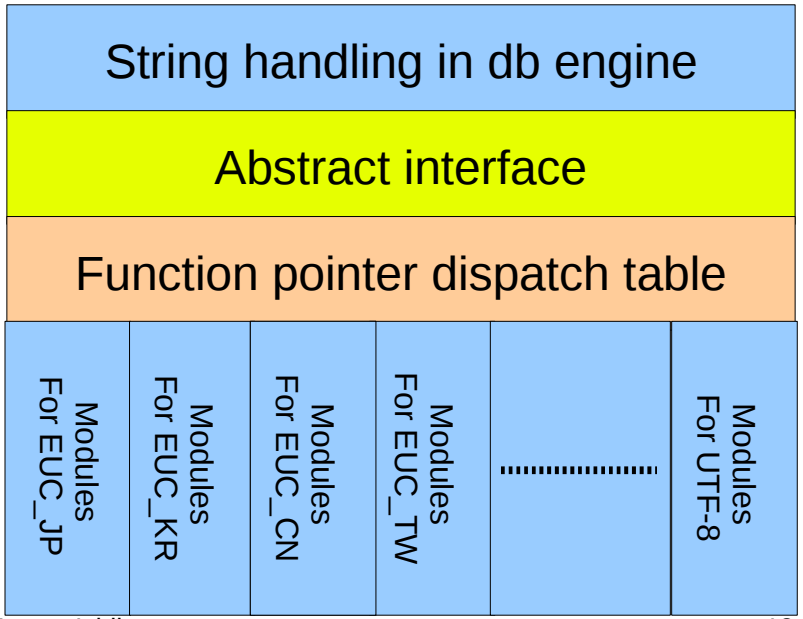
Avoid Unicode as an internal encoding

- Many OSS use Unicode as their internal encoding (intermediate representation) to make it simple. But this has several drawbacks:
  - Conversion from/to other encoding is slow because it needs to look up huge conversion tables
  - It's not a “loss less” conversion. Some characters might not be represented in Unicode

What do we do then?

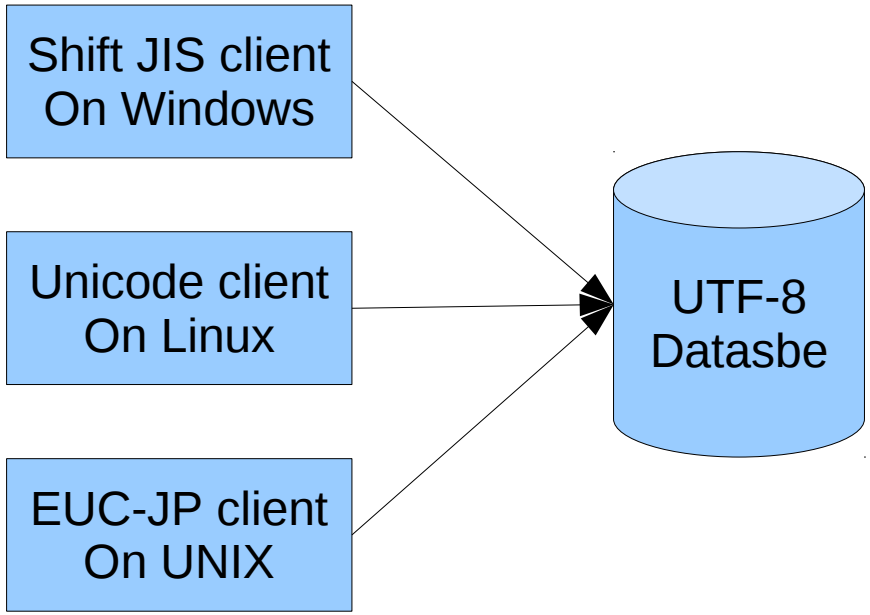
- Avoid “internal encoding”
- Have functions for all conversion pairs
- Create an abstraction layer to provide necessary operations for handling strings
- Conversion module dynamically loaded using PostgreSQL's “function manager”



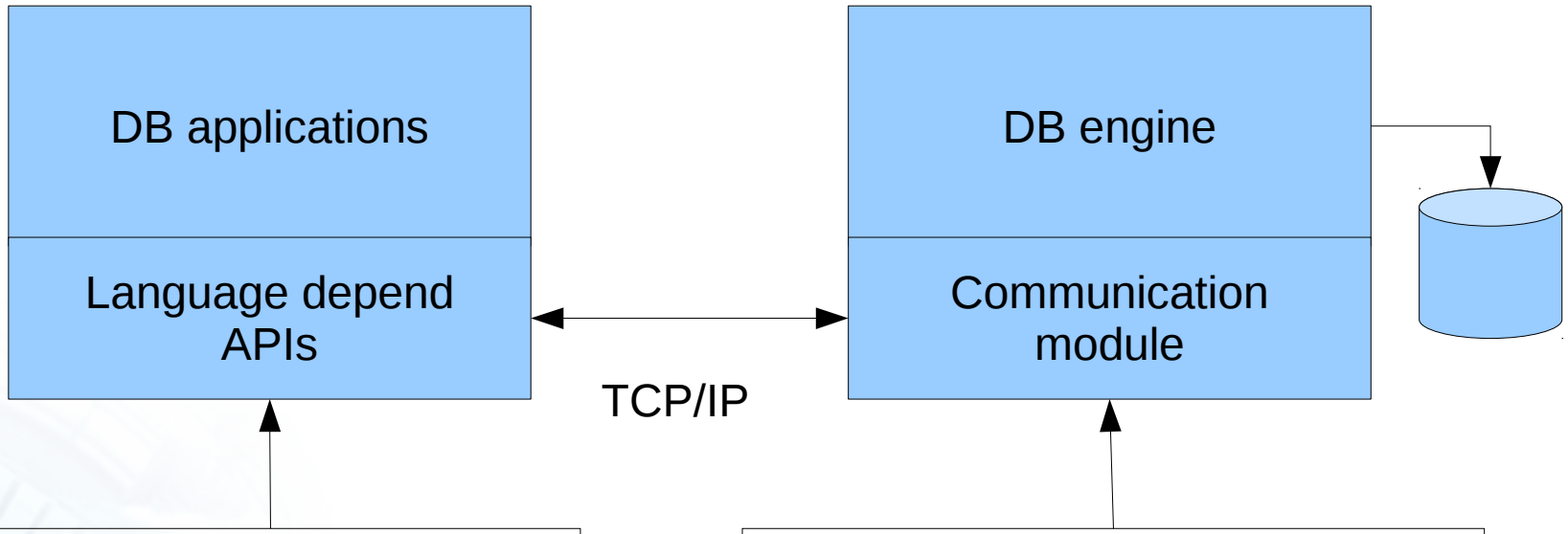


Use case of encoding conversion between client and DB server

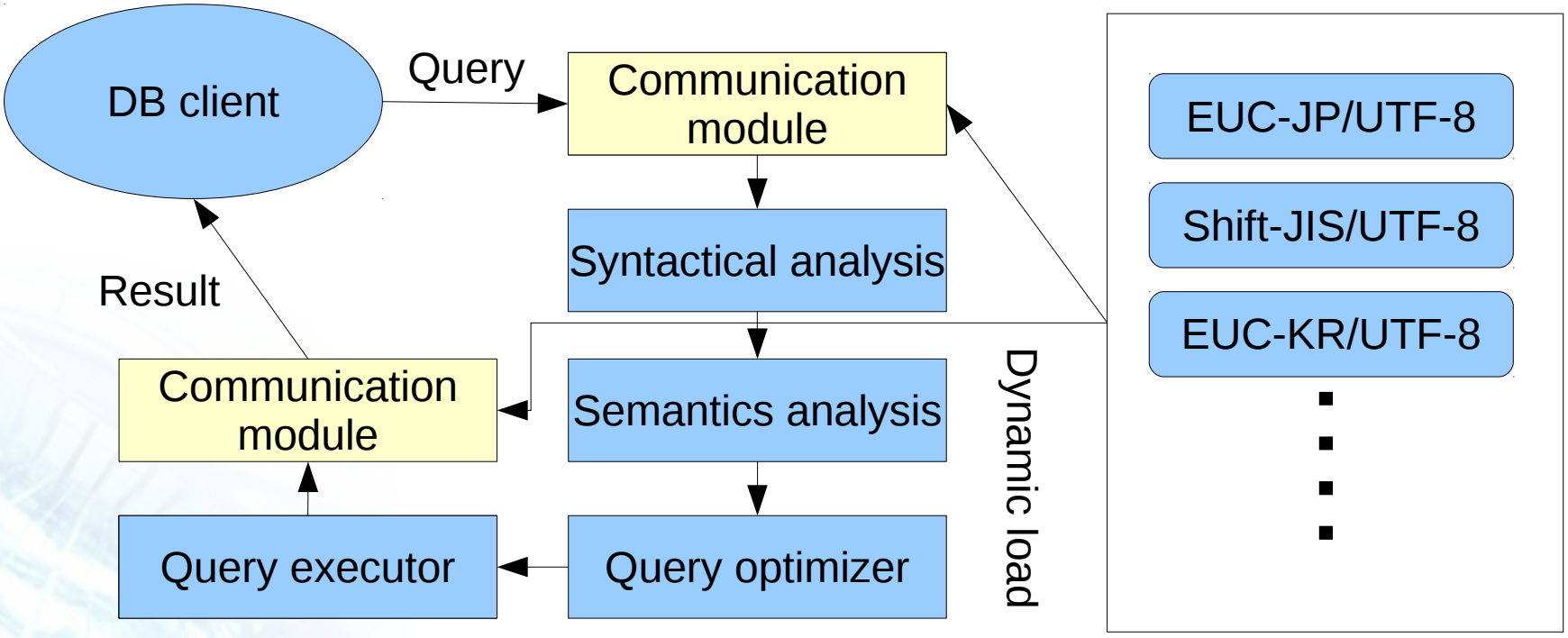
- A DB is shared by multiple clients(applications)
- They are possibly run on different platform, thus could use different encoding



Encoding conversion between client and DB server



The encoding conversion architecture



Encoding conversion summary

- Encoding conversion module could be big because of huge conversion tables. To avoid the process size growing, each module is loaded dynamically only when conversion needed
- If client and DB encoding is identical, no conversion occurs
- Easy to add new conversion module. 132 encoding conversion functions are registered so far

PostgreSQL 7.3 become I18N database(2002)

- After spending 5 years, finally PostgreSQL became I18N database
- Why 5 years?
  - Community members need time to understand the necessity of I18N
  - Lot's of technical challenges
  - I myself was lazy:-)

Next big progress – Full text search

- What is full text search in PostgreSQL?
  - Fast and scalable full text search in RDBMS
    - Up to several TB database possible
    - Full ACID complaint
    - Can be treated as ordinary data types
    - Can do search while updating database(concurrent index updating)
    - Small index size comparing with N-gram full text search system
    - Can have multiple language dictionaries
    - Created by Russian programers in 2006 (PostgreSQL 8.2)

Using full text search with Japanese

- Full text search assumes that each word in a sentence is separated by space. Unfortunately this is not the case for Japanese.
  - **今日は良い天気です**(Today is fine)
- Using “mecab” (a OSS morphological analysis program) we can convert it to:
  - **今日 は 良い 天気 です**
- There is a Japanese parser which can do this automatically(textsearch-ja)

Locale, locale...

- Why locale affects database?
  - The sort order. Example:
    - locale=C: ABCabc...
    - Locale=en_US: aAbBcC...
  - Some characters are identified to be equal in certain locale while they are not in other locale
  - Important for European languages
- How the the standard defines the locale handling
  - It defines “COLLATE” clause, which is close to LC_CTYPE and LC_COLLATE

What is the current status of locale handling?

- PostgreSQL 8.4(2009) introduced per database locale
- PostgreSQL 9.0(2010) allows to use case insensitive matching in regexp with UTF-8
- Per column COLLATE clause patches are proposed for next PostgreSQL version

Summary

- PostgreSQL started with English only database
- Non English speakers needed multibyte character sets and added them
- It evolved into I18N database management system without sacrificing performance which is the highest priority in database
- Implementing more flexible locale/collation handling is a future plan