

Cosa hanno in comune?

- pg_stat_statements
- auto_explain
- pgstattuple
- uuid-oss
- pglogical

Sono tutte estensioni

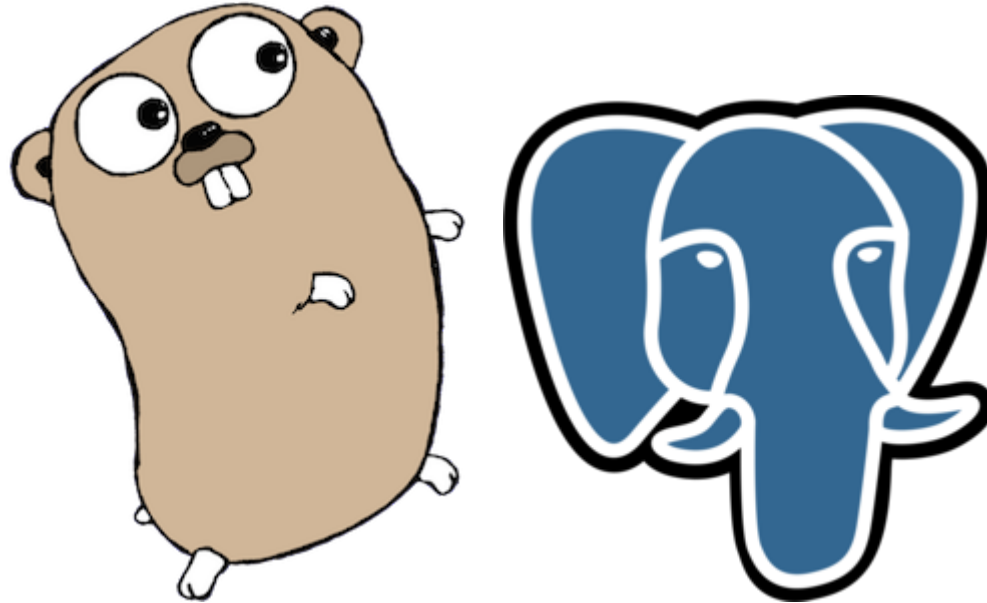
Scritte in C

Avete mai pensato di scrivere una estensione?



È più semplice del previsto e anche molto divertente!

Estendere PostgreSQL con Go



Leonardo Cecchi

 [@leonardo_cecchi](https://twitter.com/leonardo_cecchi)

 leonardo.cecchi@2ndquadrant.it

Architettura di PostgreSQL

- Applicazione Client (*frontend*)
- Server PostgreSQL (*backend - postmaster*) **estendibile**
- Processo PostgreSQL (*backend - per connessione*)
estendibile

Cosa si può fare con le estensioni?

- Nuove funzioni da chiamare da SQL
- Trigger
- Linguaggi procedurali (ad. es. PL/pgSQL)
- Sistemi di Logical Decoding

Estensioni di PostgreSQL

- Sono delle librerie condivise - *principalmente scritte in C sfruttando gli header di PostgreSQL*
- Hanno una specifica ABI - *vedi fmgr.h*
- Contengono funzioni richiamate da PostgreSQL
- Possono agganciarsi ai punti di estensione del motore
- Possono creare background workers

Calling Convention - Versione 0

```
#include "postgres.h"  
#include "fmgr.h"  
  
#ifdef PG_MODULE_MAGIC  
PG_MODULE_MAGIC;  
#endif  
  
int add_one(int arg)  
{  
    return arg + 1;  
}
```

Calling Convention - Versione 1

```
#include "postgres.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(add_one);

Datum add_one(PG_FUNCTION_ARGS)
{
    int32 arg = PG_GETARG_INT32(0);
    PG_RETURN_INT32(arg + 1);
}
```


Compilazione

```
$ gcc \  
  -shared # Libreria dinamica  
  -fPIC # Codice indipendente dalla posizione  
  -I$(pg_config --includedir-server)  
  -o /tmp/c_extension.so add_one.c
```

Dichiarazione

```
# CREATE FUNCTION c_add_one(INTEGER)
  RETURNS INTEGER AS 'c_extension', 'add_one'
  LANGUAGE C STRICT;
CREATE FUNCTION

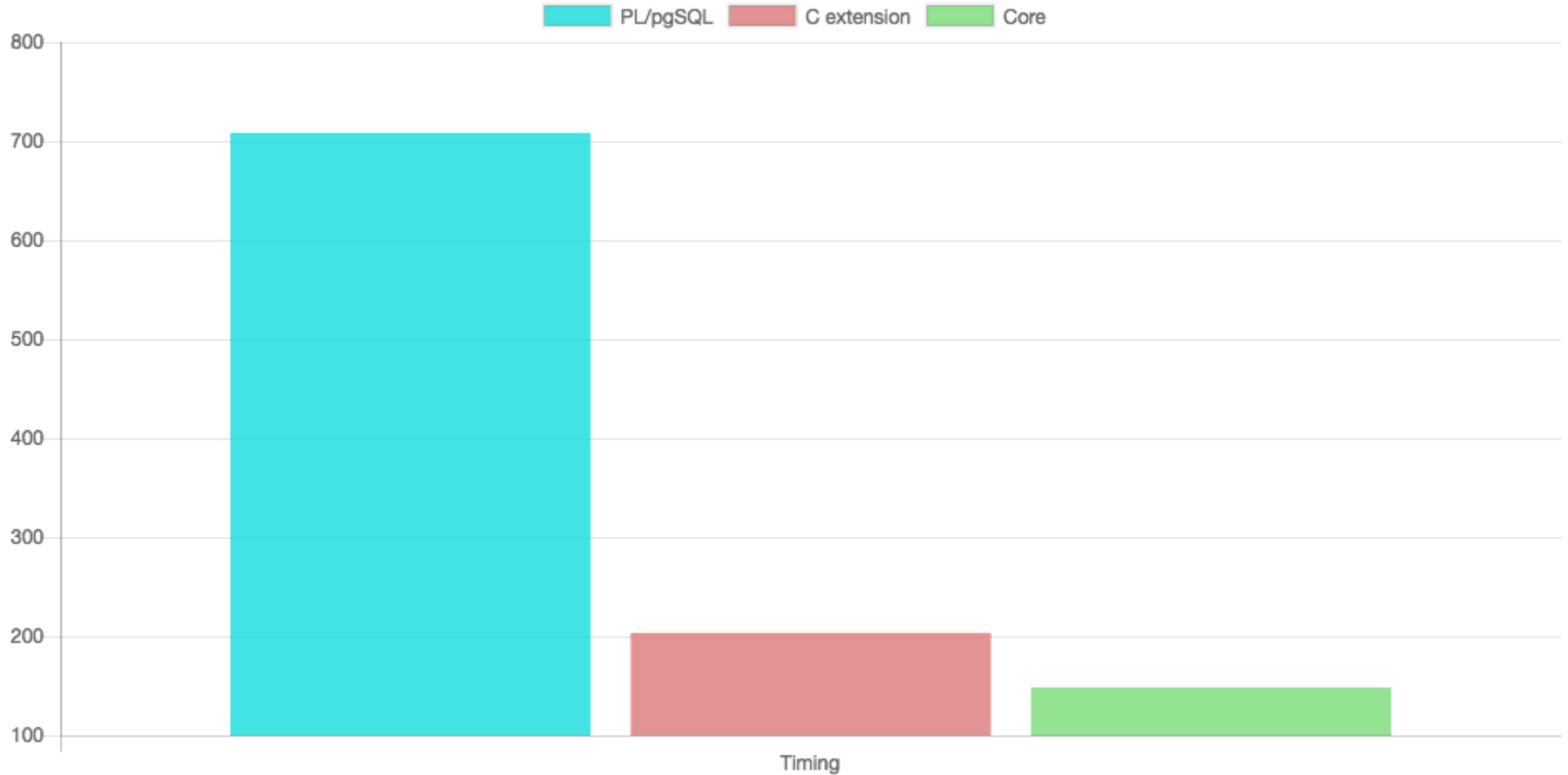
# SELECT c_add_one(34);
  c_add_one
-----
           35
(1 row)
```

Quanto costa?

```
# CREATE FUNCTION p_add_one(x INTEGER) RETURNS INTEGER AS $$  
  BEGIN RETURN x + 1; END; $$ LANGUAGE 'plpgsql';  
  
# SELECT COUNT(p_add_one(x)) FROM generate_series(1,1000000) x;  
Time: 708.973 ms  
  
# SELECT COUNT(c_add_one(x)) FROM generate_series(1,1000000) x;  
Time: 203.153 ms  
  
# SELECT COUNT(x) FROM generate_series(1,1000000) x;  
Time: 149.234 ms
```

Performance

Lies, damned lies, and statistics



Enter go

- Garbage collection
- Codice nativo, gestito da un runtime
- Default: eseguibile statico

```
$ go install leonardoce.interfree.it/helloworld
$ ldd ./bin/helloworld
        not a dynamic executable
```

buildmode - Non solo compilazione statica

- *default*: Listed main packages are built into executables and listed non-main packages are built into .a files
- *c-shared*: Build the listed main package, plus all packages it imports, into a **C shared library**. The only callable symbols will be those functions exported using a **cgo //export comment**. Requires exactly one main package to be listed.

cgo

Enables the creation of Go packages that call C code.

Permette di inserire codice "C" nel codice "Go", facendo in modo che si possono chiamare l'un l'altro.

Esempio

```
package main

/*
#include <stdio.h>

void printInteger(int x) { printf("%i\n", x); }
*/
import "C"

func main() {
    quarantaDue := 42
    C.printInteger(C.int(quarantaDue))
}
```

cgo + buildmode c-shared

- possiamo creare librerie condivise
- il codice C può chiamare il codice Go
- è possibile utilizzare le macro di fmgr.h che servono per la ABI delle estensioni PostgreSQL

cgo, le regole del gioco

Go e C devono avere due spazi di memoria distinti, perché Go ha un garbage collector *preciso* e ha bisogno di conoscere la locazione di ogni puntatore alla memoria da lui gestita.

Memoria C e memoria Go

Il codice Go può passare un puntatore Go al C solamente se la memoria non contiene altri puntatori a memoria Go.

Il codice C non può copiare questa area, nemmeno temporaneamente

Il piano

- PostgreSQL chiama il codice C, rispettando tutte le ABI senza doverle riscrivere come codice Go
- Il codice C chiama la relativa funzione Go
- Il codice Go può comunque chiamare il codice C per utilizzare le primitive di PostgreSQL

Stub C

```
// prefisso estensione
PG_FUNCTION_INFO_V1(add_one);

int go_add_one(int);

Datum add_one(PG_FUNCTION_ARGS)
{
    int32 arg = PG_GETARG_INT32(0);
    int32 result = go_add_one(arg);
    PG_RETURN_INT32(result);
}
```

Codice estensione

```
package main
import "C"

func main() { }

//export go_add_one
func go_add_one(a C.int) C.int {
    return a+1
}
```

Compilazione

```
CGO_CFLAGS=-I$(pg_config --includedir-server) \  
go build -buildmode=c-shared \  
-o go_extension.so \  
leonardoce.interfree.it/go_extension
```

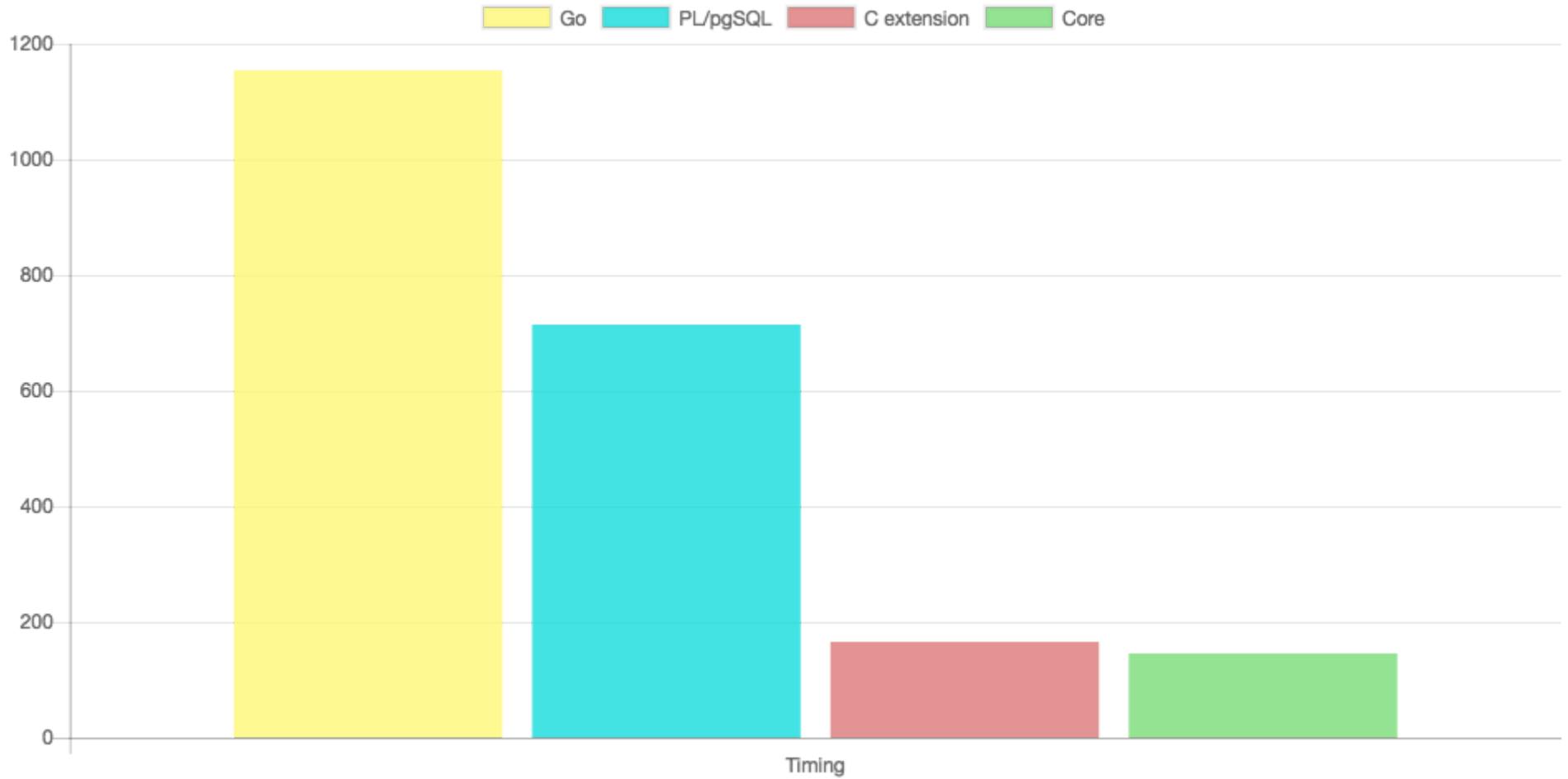

Quanto costa?

```
postgres=# SELECT COUNT(go_add_one(x))  
          FROM generate_series(1,1000000) x;  
Time: 1153.545 ms
```

```
postgres=# SELECT COUNT(p_add_one(x)) FROM ...  
Time: 714.234 ms
```

```
postgres=# SELECT COUNT(c_add_one(x)) FROM ...  
Time: 166.661 ms
```

Graficamente



Perchè?

Go, nonostante venga compilato in codice nativo, ha un runtime da inizializzare e da pulire.

- Garbage collector
- Thread sanitizer (se attivo)
- Gestione traceback (panic)

Praticamente

```
CGO_NO_SANITIZE_THREAD
int go_add_one(int p0)
{
    __SIZE_TYPE__ _cgo_ctxt = _cgo_wait_runtime_init_done();
    [...]
    _cgo_tsan_release();
    crosscall2(_cgoexp_952a6df9a21b_go_add_one, &stacktop, 16, _cgo_ctxt);
    _cgo_tsan_acquire();
    _cgo_release_context(_cgo_ctxt);
    return [...];
}
```

Inizializzazione runtime

```
uintptr_t _cgo_wait_runtime_init_done() {  
    void (*pfn)(struct context_arg*);  
  
    pthread_mutex_lock(&runtime_init_mu);  
    while (runtime_init_done == 0) {  
        pthread_cond_wait(&runtime_init_cond, &runtime_init_mu);  
    }  
    [...]  
    pfn = cgo_context_function;  
    pthread_mutex_unlock(&runtime_init_mu);  
    if (pfn != nil) { [...] }  
    return 0;  
}
```

crosscall2

```
TEXT crosscall2(SB), NOSPLIT, $0
SUBQ $0x118, SP
MOVQ BX, 0x18(SP)
MOVQ BP, 0x20(SP)
MOVQ R12, 0x28(SP)
MOVQ R13, 0x30(SP)
MOVQ R14, 0x38(SP)
MOVQ R15, 0x40(SP)

MOVQ SI, 0x0(SP)
MOVQ DX, 0x8(SP)
MOVQ CX, 0x10(SP)

CALL DI /* fn */

MOVQ 0x18(SP), BX
MOVQ 0x20(SP), BP
```

Tutte queste funzionalità costano 1150 ms per un milione di chiamate.

Attraversare i confini costa, **ma non troppo**

Da Go a PostgreSQL

La *Server Programming Interface* permette alle estensioni di avviare comandi SQL durante la loro esecuzione.

- `int SPI_connect(void)` *connette* connessione ai servizi degli SPI
- `int SPI_finish(void)` *disconnette* il contesto
- `int SPI_exec(const char * command, long count)` esegue una query SQL, eventualmente limitando i risultati ad un certo numero di righe

Sembra tutto semplice...

```
err := spi.Connect()
defer spi.Finish()

if err != nil {
    elog.Elog(elog.ERROR,
        "go_spi_example: SPI connection error")
    return
}

err = spi.Exec("INSERT INTO testtable (a) VALUES (1)", 0)
if err != nil {
    elog.Elog(elog.ERROR, "go_spi_example: SPI exec error")
    return
}
```

Attenzione

Note that if a command invoked via SPI fails, then control will **not be returned to your procedure**. Rather, the transaction or subtransaction in which your procedure executes will be rolled back. ([docs](#))

Stack

1. PostgreSQL Executor (C)
2. `go_test_spi` (C, generated by `cgo`)
3. `TestSpi` (Go)
4. `spi.Exec(...)` (Go)
5. `SPI_exec` (C, PostgreSQL)
6. `longjmp` in caso di errore, che riavvolge tutto al punto (1)

`cgo + longjmp/setjmp = HELL`

Perché?

```
CGO_NO_SANITIZE_THREAD
int go_add_one(int p0)
{
    __SIZE_TYPE__ _cgo_ctxt = _cgo_wait_runtime_init_done();
    [...]
    _cgo_tsan_release();
    crosscall2(_cgoexp_952a6df9a21b_go_add_one, &stacktop, 16, _cgo_ctxt);
    _cgo_tsan_acquire();
    _cgo_release_context(_cgo_ctxt);
    return [...];
}
```

La soluzione

```
int stub_SPI_exec(const char *query, long rows) {
    int spi_rc;
    BeginInternalSubTransaction(NULL);
    PG_TRY();
    {
        spi_rc = SPI_exec(query, rows);
        ReleaseCurrentSubTransaction();
    }
    PG_CATCH();
    {
        RollbackAndReleaseCurrentSubTransaction();
        return -1;
    }
    PG_END_TRY();
    return spi_rc;
}
```

Autonomous Subtransactions

Permettono di dividere la la transazione in blocchi, e di gestire gli errori di conseguenza

Autonomous Subtransactions != Autonomous Transactions

Autonomous Subtransactions ~~ Savepoints

Conclusione

- Integrare Go e PostgreSQL è possibile!
- Le macro di PostgreSQL sono così comode da giustificare l'uso di C nella nostra integrazione
- Passare da un universo all'altro costa, ma vale la pena

Domande?

Leonardo Cecchi

 [@leonardo_cecchi](https://twitter.com/leonardo_cecchi)

 leonardo.cecchi@2ndquadrant.it