Extending postgresql with C

Manuel Kniep
co-founder & Developer
adjust.com

https://github.com/adjust/pg_c_dev

Postgresql

- The world’s most advanced open source database
- truly open source
- extrem liberal licence
- highly extensible

Extending Postgresql

- CREATE FUNCTION
- CREATE OPERATOR
- CREATE AGGREGATE
- CREATE TYPE
- CREATE FOREIGN DATA WRAPPER

Extending Postgresql - Languages

- SQL
- PL/pgSQL
- PL/Tcl
- PL/Perl
- PL/Python
- C

Extending Postgresql - Languages

WHY C ?

Extending Postgresql - Languages

FAST

Extending Postgresql - Languages

FUN

Fibonacci - SQL

```
CREATE FUNCTION fib(n int) RETURNS int
AS $$
    SELECT CASE n
        WHEN 0 THEN 0
        WHEN 1 THEN 1
        ELSE fib(n-1) + fib(n-2)
    END;
$$ LANGUAGE sql IMMUTABLE STRICT;
```

Fibonacci - plpgsql

```
CREATE FUNCTION fib(n int) RETURNS int
AS $$
    BEGIN
        CASE n
            WHEN 0 THEN RETURN 0;
            WHEN 1 THEN RETURN 1 ;
            ELSE RETURN fib(n-1) + fib(n-2);
        END CASE;
    END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

Fibonacci - C

```
CREATE FUNCTION fib(integer) RETURNS integer
AS 'fiblib','fib'
LANGUAGE C IMMUTABLE STRICT;
```

- c-interfacing function
- fiblib shared library
- fib exported function within the library

Fibonacci - C

```c
//fiblib.c
#include "postgres.h"
#include „fmgr.h"

PG_MODULE_MAGIC;

Datum       fib(PG_FUNCTION_ARGS);
static int fib_internal(int n);

PG_FUNCTION_INFO_V1(fib);
Datum fib(PG_FUNCTION_ARGS)
{
    int32 n = PG_GETARG_INT32(0);
    PG_RETURN_INT32(fib_internal(n));
}

static int
fib_internal(int n)
{
    switch (n)
    {
        case 0: return 0;
        case 1: return 1;
        default: return fib_internal(n - 1) + fib_internal(n - 2);
    }
}
```

Fibonacci - Makefile

Compile against exiting Postgres installation

```
MODULES = fiblib
PGXS   := $(shell pg_config --pgxs)
include $(PGXS)
```

- postgres ships with a portable build system
- builds your code against installed postgres
- compiles a shared library file
- shared library is looked up in pg_config —pkglibdir

Fibonacci - compiling

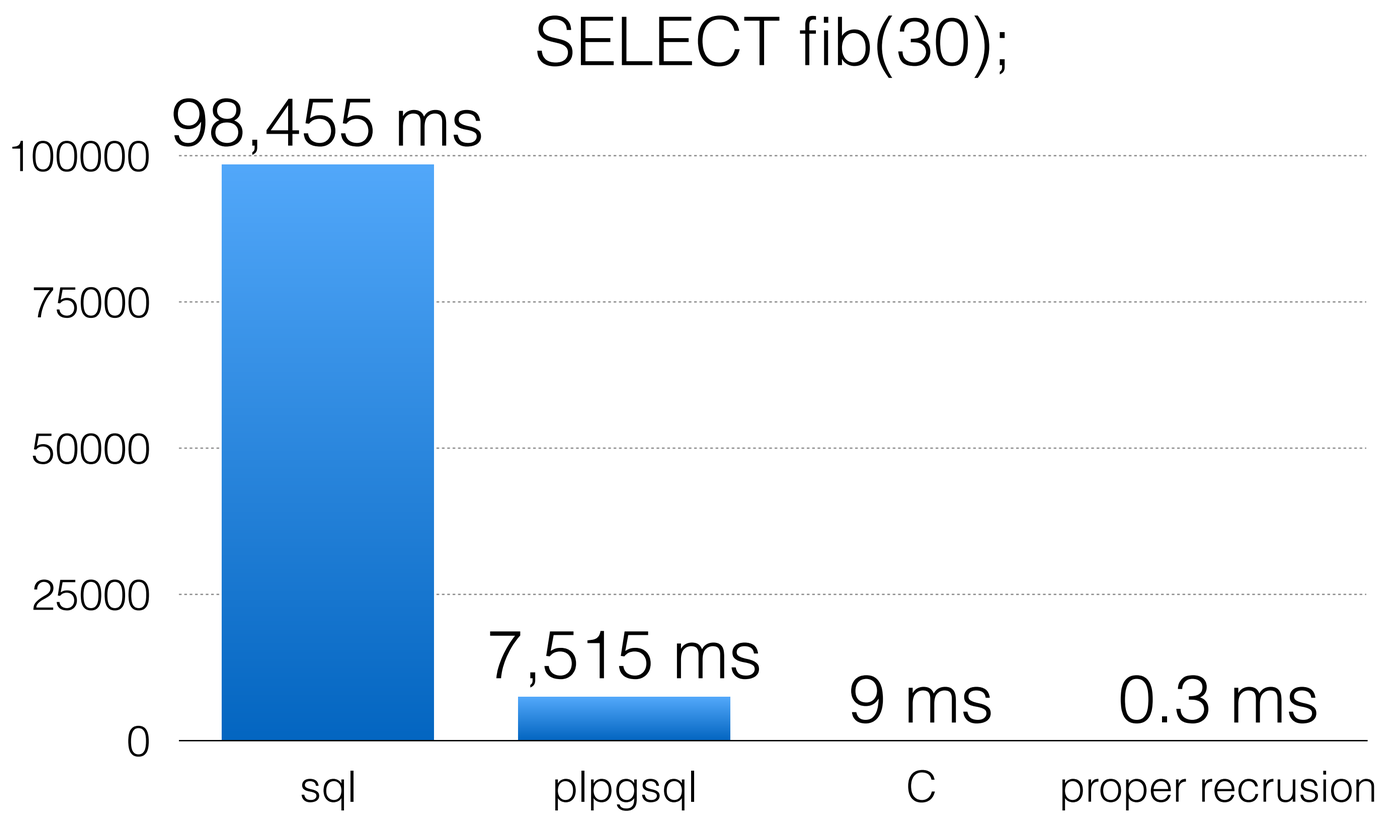
Compile against exiting Postgres installation

```
$ make install
clang -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Wendif-
labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasing -fwrapv -Wno-
unused-command-line-argument -O2  -I. -I./ -I/usr/local/Cellar/postgresql/10.1/include/
server -I/usr/local/Cellar/postgresql/10.1/include/internal  -I/usr/local/opt/openssl/
include -I/usr/local/opt/readline/include -I/Applications/Xcode.app/Contents/Developer/
Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.13.sdk/usr/include/libxml2   -c -o
fiblib.o fiblib.c
clang -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Wendif-
labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasing -fwrapv -Wno-
unused-command-line-argument -O2  -L/usr/local/lib -L/usr/local/opt/openssl/lib -L/usr/
local/opt/readline/lib  -Wl,-dead_strip_dylibs   -bundle -bundle_loader /usr/local/Cellar/
postgresql/10.1/bin/postgres -o fiblib.so fiblib.o
/bin/sh /usr/local/lib/postgresql/pgxs/src/makefiles/../../config/install-sh -c -d '/usr/
local/lib/postgresql'
/usr/bin/install -c -m 755  fiblib.so '/usr/local/lib/postgresql/'
```

Fibonacci - Performance

FAST

Fibonacci - Performance



Fibonacci - Performance

FUN

includes

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

PG_MODULE_MAGIC;

Datum      fib(PG_FUNCTION_ARGS);
static int fib_internal(int n);

PG_FUNCTION_INFO_V1(fib);
Datum fib(PG_FUNCTION_ARGS)
{
    int32 n, res;
    n   = PG_GETARG_INT32(0);
    res = fib_internal(n);
    PG_RETURN_INT32(res);
}
```

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

- postgres.h should be the first file included
- fmgr.h needed for sql interfacing functions

magic block

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

PG_MODULE_MAGIC;

Datum       fib(PG_FUNCTION_ARGS);
static int fib_internal(int n);

PG_FUNCTION_INFO_V1(fib);
Datum fib(PG_FUNCTION_ARGS)
{
    int32 n, res;
    n   = PG_GETARG_INT32(0);
    res = fib_internal(n);
    PG_RETURN_INT32(res);
}
```

- PG_MODULE_MAGIC needed to be called exactly once per module
- ensures compatibility with postgres version
- throws „missing magic block“ when omitted

Datum

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

PG_MODULE_MAGIC;

Datum       fib(PG_FUNCTION_ARGS);
static int fib_internal(int n);

PG_FUNCTION_INFO_V1(fib);
Datum fib(PG_FUNCTION_ARGS)
{
    int32 n, res;
    n   = PG_GETARG_INT32(0);
    res = fib_internal(n);
    PG_RETURN_INT32(res);
}
```

- void * like generic datatype
- all sql interfacing functions receive and return Datum type
- just a blob of data that need to be handled properly

Version 1 calling

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

PG_MODULE_MAGIC;

Datum       fib(PG_FUNCTION_ARGS);
static int fib_internal(int n);

PG_FUNCTION_INFO_V1(fib);
Datum fib(PG_FUNCTION_ARGS)
{
    int32 n, res;
    n   = PG_GETARG_INT32(0);
    res = fib_internal(n);
    PG_RETURN_INT32(res);
}
```

```c
PG_FUNCTION_INFO_V1(fib);
Datum
fib(PG_FUNCTION_ARGS)
```

- hides complexity of passing arguments and results
- mandatory for all sql interfacing functions
- PG_FUNCTION_INFO_V1 macro in addition to
- Datum func_name(PG_FUNCTION_ARGS)

Base Types

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

PG_MODULE_MAGIC;

Datum       fib(PG_FUNCTION_ARGS);
static int fib_internal(int n);

PG_FUNCTION_INFO_V1(fib);
Datum fib(PG_FUNCTION_ARGS)
{
    int32 n, res;
    n   = PG_GETARG_INT32(0);
    res = fib_internal(n);
    PG_RETURN_INT32(res);
}
```

- all postgres built in types are backed by c typedef
- pass by value, fixed-length (up to 4 or 8 bytes)
- pass by reference, fixed-length
- pass by reference, variable-length (with first 4 bytes indication length)

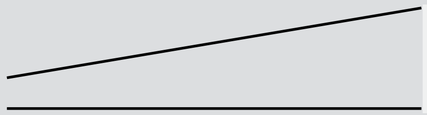
fetching data

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

PG_MODULE_MAGIC;

Datum       fib(PG_FUNCTION_ARGS);
static int fib_internal(int n);

PG_FUNCTION_INFO_V1(fib);
Datum fib(PG_FUNCTION_ARGS)
{
    int32 n, res;
    n   = PG_GETARG_INT32(0);
    res = fib_internal(n);
    PG_RETURN_INT32(res);
}
```

- each argument is fetched using a PG_GETARG_xxx() macro
- most of them defined in fmgr.h
- use a copy for pass by reference types when modifying inputdata!

returning data

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

PG_MODULE_MAGIC;

Datum       fib(PG_FUNCTION_ARGS);
static int fib_internal(int n);

PG_FUNCTION_INFO_V1(fib);
Datum fib(PG_FUNCTION_ARGS)
{
    int32 n, res;
    n   = PG_GETARG_INT32(0);
    res = fib_internal(n);
    PG_RETURN_INT32(res);
}
```



```c
PG_RETURN_INT32(res);
```

- likewise data are returned with PG_RETURN_*xxx*() macros
- PG_RETURN_POINTER() for pass-by-ref types

Aggregates

Aggregates

```
CREATE AGGREGATE SUM(integer) (
  SFUNC = int4_sum,
  STYPE = bigint
);
```

- most simple form
- just as state function (SFUNC)
- and a state type (STYPE)

Aggregates

```
CREATE AGGREGATE AVG(integer) (
  SFUNC     = int4_avg_accum,
  STYPE     = bigint[],
  FINALFUNC = int8_avg,
  INITCOND  ='{0,0}'
);
```

- a state function
- state type as an array holding the 2 state vars (sum and count)
- a final function (sum / count)
- a state initialization

Analyzing logs

```
# SELECT * FROM logs;
 server_name |     created_at      |  response_time
-------------+---------------------+-----------------
 server_5    | 2017-04-10 06:37:49 | 00:00:00.025009
 server_3    | 2017-04-09 17:36:32 | 00:00:00.088386
 server_5    | 2017-01-29 01:08:34 | 00:00:00.035815
 server_3    | 2017-01-31 14:39:47 | 00:00:00.156583
 server_2    | 2017-03-11 22:14:11 | 00:00:00.091659
(5 rows)
```

- a pingdom like app
- measures response_time in irregular intervals
- calculate uptime fraction per server
- uptime where response_time < threshold

Analyzing logs

```
# SELECT server_name, UPTIME(…) FROM logs GROUP BY 1;
 server_name |          uptime
-------------+--------------------------
 server_1    | 0.5995732997866498 9332
 server_2    | 0.7496633748316874 1584
 server_3    | 0.9999984999992499 9962
 server_4    | 1.0000000000000000 0000
 server_5    | 1.0000000000000000 0000
(5 rows)
```

Aggregate feed with

- timestamp
- response_time
- threshold

Aggregation State

- last timestamp
- sum uptime
- sum downtime

Finally

- uptime / (uptime + down)

Analyzing logs

```
CREATE AGGREGATE UPTIME(timestamp, interval, interval)
(
    SFUNC     = uptime_sf,
    STYPE     = INT[],
    FINALFUNC = uptime_sf_final,
    INITCOND  = '{NULL,0,0}'
);
```

- SFUNC: state transition function
- STYPE: state type
- FINALFUNC: turn state type into aggregates return type
- INITCOND: initial value for state type

Analyzing logs - state function

```
CREATE OR REPLACE FUNCTION uptime_sf(state int[], current timestamp, response_time interval, threshold interval)
RETURNS int[]
as $$
SELECT CASE response_time < threshold
WHEN TRUE THEN --uptime case
    array [ EXTRACT(EPOCH FROM current)::int, state[2] + COALESCE(EXTRACT(EPOCH FROM current)::int - state[1], 0 ), state[3] ]
ELSE --downtime case
    CASE threshold IS NULL
    WHEN TRUE THEN
        state
    ELSE
        array [ EXTRACT(EPOCH FROM current)::int, state[2], state[3] + COALESCE(EXTRACT(EPOCH FROM current)::int - state[1], 0 ) ]
    END
END;
$$ LANGUGAE sql;
```

- beware of NULL values
- use int for internal state data

Analyzing logs - final function

```
CREATE OR REPLACE FUNCTION uptime_sf_final(state int[])
RETURNS numeric
as $$
    SELECT state[2]::numeric / NULLIF(state[2] + state[3],0);
$$ language sql;
```

- beware division by 0
- cast to numeric

Analyzing logs

```
CREATE AGGREGATE UPTIME(timestamp, interval, interval)
(
    SFUNC     = uptime_sf,
    STYPE     = internal,
    FINALFUNC = uptime_sf_final
);
```

- state type is now internal
- no initial state value

Analyzing logs - C interface

```
CREATE OR REPLACE FUNCTION uptime_sf(internal, timestamp, interval, interval)
RETURNS internal
AS 'uptime'
LANGUAGE C IMMUTABLE;

CREATE OR REPLACE FUNCTION uptime_sf_final(internal)
RETURNS numeric
AS 'uptime'
LANGUAGE C IMMUTABLE STRICT;
```

- sql interfacing function handling type internal
- uptime_sf is declared none strict now
- null handling becomes important in implementation

C function - includes

```c
#include "postgres.h"

#include "fmgr.h"
#include "utils/builtins.h"
#include "utils/timestamp.h"

PG_MODULE_MAGIC;

#ifdef HAVE_INT64_TIMESTAMP
#define USECS 1000000
#else
#define USECS 1
#endif

/*
 * Routines for UPTIME().  The transition datatype
 * is a three-element int4 array, holding last_unix_time, uptime_sum and downtime_sum.
 */
typedef struct UptimeAggState
{
    int32 last_epoch;
    int32 uptime;
    int32 downtime;
} UptimeAggState;
```

```c
#include "utils/builtins.h"
#include "utils/timestamp.h"
```

header to work with

- builtin types (here mostly from numeric.c)
- timestamps and intervals (also PG_GETARG macros)

Analyzing logs - C functions

```c
#include "postgres.h"

#include "fmgr.h"
#include "utils/builtins.h"
#include "utils/timestamp.h"

PG_MODULE_MAGIC;

#ifdef HAVE_INT64_TIMESTAMP
#define USECS 1000000
#else
#define USECS 1
#endif

/*
 * Routines for UPTIME().  The transition datatype
 * is a three-element int4 array, holding last_unix_time, uptime_sum and downtime_sum.
 */
typedef struct UptimeAggState
{
    int32 last_epoch;
    int32 uptime;
    int32 downtime;
} UptimeAggState;
```

```c
#ifdef HAVE_INT64_TIMESTAMP
#define USECS 1000000
#else
#define USECS 1
#endif
```

- platform specific handling of timestamps
- if postgres was build with 64 bit support timestamps are int64 microseconds
- double otherwise

Analyzing logs - C functions

```c
#include "postgres.h"

#include "fmgr.h"
#include "utils/builtins.h"
#include "utils/timestamp.h"

PG_MODULE_MAGIC;

#ifdef HAVE_INT64_TIMESTAMP
#define USECS 1000000
#else
#define USECS 1
#endif

/*
 * Routines for UPTIME().  The transition datatype
 * is a three-element int4 array, holding last_unix_time, uptime_sum and downtime_sum.
 */
typedef struct UptimeAggState
{
    int32 last_epoch;
    int32 uptime;
    int32 downtime;
} UptimeAggState;
```

```c
typedef struct UptimeAggState
{
    int32 last_epoch;
    int32 uptime;
    int32 downtime;
} UptimeAggState;
```

- internal state
- holding three 4 byte integers
- passed through state function

Analyzing logs - C functions

```
Datum uptime_sf(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    Timestamp       timestamp;
    int32           current_epoch;
    Interval *      response_time;
    Interval *      threshold;
    MemoryContext   agg_context;
    MemoryContext   old_context;
    bool            first_call = false;

    if (!AggCheckCallContext(fcinfo, &agg_context))
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED), errmsg("aggregate function called in non-aggregate context")));

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);

    /* Create the state data on the first call */
    if (state == NULL)
    {
        first_call  = true;
        old_context = MemoryContextSwitchTo(agg_context);

        state = (UptimeAggState *) palloc0(sizeof(UptimeAggState));
        MemoryContextSwitchTo(old_context);
    }

    if (PG_ARGISNULL(1) || PG_ARGISNULL(3))
        PG_RETURN_POINTER(state);
```

```
UptimeAggState *state;
Timestamp       timestamp;
int32           current_epoch;
Interval *      response_time;
Interval *      threshold;
```

- c types for the input values
- Timestamp -> typedef int64 Timestamp;
- Interval -> struct {time; day; month}

postgres Interval type

```
Datum uptime_sf(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    Timestamp       timestamp;
    int32           current_epoch;
    Interval *      response_time;
    Interval *      threshold;
    MemoryContext   agg_context;
    MemoryContext   old_context;
    bool            first_call = false;

    if (!AggCheckCallContext(fcinfo, &agg_context))
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED), errmsg("aggregate function called in non aggregate context")));

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);

    /* Create the state data on the first call */
    if (state == NULL)
    {
        first_call  = true;
        old_context = MemoryContextSwitchTo(agg_context);

        state = (UptimeAggState *) palloc0(sizeof(UptimeAggState));
        MemoryContextSwitchTo(old_context);
    }

    if (PG_ARGISNULL(1) || PG_ARGISNULL(3))
        PG_RETURN_POINTER(state);
```

```
typedef struct
{
  TimeOffset  time;
  int32       day;
  int32       month;
} Interval;
```

- Interval represents delta time
- time spanned might be unknown (e.g. 1 month)
- might even be timezone dependent (eg. daylight saving days)
- if only time is set we have a specific usec interval

postgres Memory Context

```
Datum uptime_sf(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    Timestamp       timestamp;
    int32           current_epoch;
    Interval *      response_time;
    Interval *      threshold;
    MemoryContext   agg_context;
    MemoryContext   old_context;
    bool            first_call = false;

    if (!AggCheckCallContext(fcinfo, &agg_context))
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED), errmsg("aggregate function called in non aggregate context")));

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);

    /* Create the state data on the first call */
    if (state == NULL)
    {
        first_call  = true;
        old_context = MemoryContextSwitchTo(agg_context);

        state = (UptimeAggState *) palloc0(sizeof(UptimeAggState));
        MemoryContextSwitchTo(old_context);
    }

    if (PG_ARGISNULL(1) || PG_ARGISNULL(3))
        PG_RETURN_POINTER(state);
```

```
MemoryContext   agg_context;
MemoryContext   old_context;
```

- Allocation set implementation for memory context (aset.c)
- group allocated pieces of memory, making it easier to manage lifecycle.
- organized in a tree, roughly matching the execution plans.
- minimize malloc calls/book-keeping, maximize memory reuse, and never really frees memory.
- further reading https://blog.pgaddict.com/posts/introduction-to-memory-contexts

postgres Memory Context

```
Datum uptime_sf(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    Timestamp       timestamp;
    int32           current_epoch;
    Interval *      response_time;
    Interval *      threshold;
    MemoryContext   agg_context;
    MemoryContext   old_context;
    bool            first_call = false;

    if (!AggCheckCallContext(fcinfo, &agg_context))
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED), errmsg("aggregate function called in non aggregate context")));

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);

    /* Create the state data on the first call */
    if (state == NULL)
    {
        first_call  = true;
        old_context = MemoryContextSwitchTo(agg_context);

        state = (UptimeAggState *) palloc0(sizeof(UptimeAggState));
        MemoryContextSwitchTo(old_context);
    }

    if (PG_ARGISNULL(1) || PG_ARGISNULL(3))
        PG_RETURN_POINTER(state);
```

```
if (!AggCheckCallContext(
        fcinfo, &agg_context))
```

- check that function is called in aggregation context
- save aggregation context in agg_context for later use

postgres reporting errors

```
Datum uptime_sf(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    Timestamp       timestamp;
    int32           current_epoch;
    Interval *      response_time;
    Interval *      threshold;
    MemoryContext   agg_context;
    MemoryContext   old_context;
    bool            first_call = false;

    if (!AggCheckCallContext(fcinfo, &agg_context))
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED), errmsg("aggregate function called in non aggregate context")));

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);

    /* Create the state data on the first call */
    if (state == NULL)
    {
        first_call  = true;
        old_context = MemoryContextSwitchTo(agg_context);

        state = (UptimeAggState *) palloc0(sizeof(UptimeAggState));
        MemoryContextSwitchTo(old_context);
    }

    if (PG_ARGISNULL(1) || PG_ARGISNULL(3))
        PG_RETURN_POINTER(state);
```

```
ereport(ERROR,
  (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
   errmsg("aggregate function called in non aggregate context")));
```

- raising an error quits query execution with error message
- simple form elog(ERROR, "message")
- available severity level (ranging from DEBUG to PANIC)
- more details https://www.postgresql.org/docs/current/static/error-message-reporting.html

C functions - NULL handling

```
Datum uptime_sf(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    Timestamp       timestamp;
    int32           current_epoch;
    Interval *      response_time;
    Interval *      threshold;
    MemoryContext   agg_context;
    MemoryContext   old_context;
    bool            first_call = false;

    if (!AggCheckCallContext(fcinfo, &agg_context))
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED), errmsg("aggregate function called in non aggregate context")));

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);

    /* Create the state data on the first call */
    if (state == NULL)
    {
        first_call  = true;
        old_context = MemoryContextSwitchTo(agg_context);

        state = (UptimeAggState *) palloc0(sizeof(UptimeAggState));
        MemoryContextSwitchTo(old_context);
    }

    if (PG_ARGISNULL(1) || PG_ARGISNULL(3))
        PG_RETURN_POINTER(state);
```

```
state = PG_ARGISNULL(0) ? NULL :
    (UptimeAggState *) PG_GETARG_POINTER(0);
```

- on the first call (per group) the internal state is NULL
- never call PG_GETARG_xxx if passed data can be NULL
- non strict functions should always check null values using PG_ARGISNULL(x)

switching Memory Context

```
Datum uptime_sf(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    Timestamp       timestamp;
    int32           current_epoch;
    Interval *      response_time;
    Interval *      threshold;
    MemoryContext   agg_context;
    MemoryContext   old_context;
    bool            first_call = false;

    if (!AggCheckCallContext(fcinfo, &agg_context))
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED), errmsg("aggregate function called in non aggregate context")));

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);

    /* Create the state data on the first call */
    if (state == NULL)
    {
        first_call  = true;
        old_context = MemoryContextSwitchTo(agg_context);

        state = (UptimeAggState *) palloc0(sizeof(UptimeAggState));
        MemoryContextSwitchTo(old_context);
    }

    if (PG_ARGISNULL(1) || PG_ARGISNULL(3))
        PG_RETURN_POINTER(state);
```

```
first_call  = true;
old_context = MemoryContextSwitchTo(agg_context);

state = (UptimeAggState *) palloc0(sizeof(UptimeAggState));
MemoryContextSwitchTo(old_context);
```

- internal state needs to be allocated on first call
- to ensure availability on aggregation execution node, switch to correct Memory Context
- always use palloc to allocate memory
- palloc allocates memory in current memory context and thus doesn’t need to be freed
- use palloc0 to get clean memory (zeroed out)

C functions - NULL handling

```
Datum uptime_sf(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    Timestamp       timestamp;
    int32           current_epoch;
    Interval *      response_time;
    Interval *      threshold;
    MemoryContext   agg_context;
    MemoryContext   old_context;
    bool            first_call = false;

    if (!AggCheckCallContext(fcinfo, &agg_context))
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED), errmsg("aggregate function called in non aggregate context")));

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);

    /* Create the state data on the first call */
    if (state == NULL)
    {
        first_call  = true;
        old_context = MemoryContextSwitchTo(agg_context);

        state = (UptimeAggState *) palloc0(sizeof(UptimeAggState));
        MemoryContextSwitchTo(old_context);
    }

    if (PG_ARGISNULL(1) || PG_ARGISNULL(3))
        PG_RETURN_POINTER(state);
```

```
if (PG_ARGISNULL(1) || PG_ARGISNULL(3))
    PG_RETURN_POINTER(state);
```

- if timestamp or threshold is NULL we just skip
- return state using PG_RETURN_POINTER

C functions - get Arguments

```
timestamp     = PG_GETARG_TIMESTAMP(1);
threshold     = PG_GETARG_INTERVAL_P(3);
current_epoch = (int32)(timestamp / USECS);

// bail out for unspecific intervals
if (threshold->day != 0 || threshold->month != 0)
    elog(ERROR, "unspecific interval");

if (first_call)
{
    // no elapsed time available
    state->last_epoch = current_epoch;
    PG_RETURN_POINTER(state);
}

if (PG_ARGISNULL(2))
{
    // downtime case
    state->downtime += current_epoch - state->last_epoch;
    state->last_epoch = current_epoch;
}

response_time = PG_GETARG_INTERVAL_P(2);

if (response_time->day != 0 || response_time->month != 0)
    elog(ERROR, "unspecific interval");
```

```
timestamp     = PG_GETARG_TIMESTAMP(1);
threshold     = PG_GETARG_INTERVAL_P(3);
current_epoch = (int32)(timestamp / USECS);
```

- get timestamp and threshold interval
- convert timestamp into epoch seconds
- note postgres epoch is Y2k

handling interval type

```
timestamp     = PG_GETARG_TIMESTAMP(1);
threshold     = PG_GETARG_INTERVAL_P(3);
current_epoch = (int32)(timestamp / USECS);

// bail out for unspecific intervals
if (threshold->day != 0 || threshold->month != 0)
    elog(ERROR, "unspecific interval");

if (first_call)
{
    // no elapsed time available
    state->last_epoch = current_epoch;
    PG_RETURN_POINTER(state);
}

if (PG_ARGISNULL(2))
{
    // downtime case
    state->downtime += current_epoch - state->last_epoch;
    state->last_epoch = current_epoch;
}

response_time = PG_GETARG_INTERVAL_P(2);

if (response_time->day != 0 || response_time->month != 0)
    elog(ERROR, "unspecific interval");
```

```
// bail out for unspecific intervals
if (threshold->day != 0 || threshold->month != 0)
    elog(ERROR, "unspecific interval");
```

- threshold needs to be defined as specific interval
- elog used as simple form for error reporting

C functions

```
timestamp     = PG_GETARG_TIMESTAMP(1);
threshold     = PG_GETARG_INTERVAL_P(3);
current_epoch = (int32)(timestamp / USECS);

// bail out for unspecific intervals
if (threshold->day != 0 || threshold->month != 0)
    elog(ERROR, "unspecific interval");

if (first_call)
{
    // no elapsed time available
    state->last_epoch = current_epoch;
    PG_RETURN_POINTER(state);
}

if (PG_ARGISNULL(2))
{
    // downtime case
    state->downtime += current_epoch - state->last_epoch;
    state->last_epoch = current_epoch;
}

response_time = PG_GETARG_INTERVAL_P(2);

if (response_time->day != 0 || response_time->month != 0)
    elog(ERROR, "unspecific interval");
```

- on the first call last_epoch is 0
- no span calculation possible
- just store current_epoch and return

C functions - NULL handling

```
timestamp     = PG_GETARG_TIMESTAMP(1);
threshold     = PG_GETARG_INTERVAL_P(3);
current_epoch = (int32)(timestamp / USECS);

// bail out for unspecific intervals
if (threshold->day != 0 || threshold->month != 0)
    elog(ERROR, "unspecific interval");

if (first_call)
{
    // no elapsed time available
    state->last_epoch = current_epoch;
    PG_RETURN_POINTER(state);
}

if (PG_ARGISNULL(2))
{
    // downtime case
    state->downtime += current_epoch - state->last_epoch;
    state->last_epoch = current_epoch;
    PG_RETURN_POINTER(state);
}

response_time = PG_GETARG_INTERVAL_P(2);

if (response_time->day != 0 || response_time->month != 0)
    elog(ERROR, "unspecific interval");
```

```
if (PG_ARGISNULL(2))
{
    // downtime case
    state->downtime += current_epoch - state->last_epoch;
    state->last_epoch = current_epoch;
    PG_RETURN_POINTER(state);
}
```

- no response_time is considered a down case

Analyzing logs - state function

```
    response_time = PG_GETARG_INTERVAL_P(2);

    if (response_time->day != 0 || response_time->month != 0)
        elog(ERROR, "unspecific interval");

    if (response_time->time < threshold->time)
    {
        // uptime case
        state->uptime += current_epoch - state->last_epoch;
        state->last_epoch = current_epoch;
    }
    else
    {
        // downtime case
        state->downtime += current_epoch - state->last_epoch;
        state->last_epoch = current_epoch;
    }

    PG_RETURN_POINTER(state);
}
```

Analyzing logs - final function

```
Datum uptime_sf_final(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    int32           sum;
    Datum           total_time;
    Datum           up_time;

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);
    if (state == NULL)
        PG_RETURN_NULL();

    sum        = state->uptime + state->downtime;
    total_time = DirectFunctionCall1(int4_numeric, Int32GetDatum(sum));

    if (sum == 0)
        PG_RETURN_DATUM(total_time);

    up_time = DirectFunctionCall1(int4_numeric, Int32GetDatum(state->uptime));

    PG_RETURN_DATUM(DirectFunctionCall2(numeric_div, up_time, total_time));
}
```

C functions - handling NULL

```
Datum uptime_sf_final(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    int32           sum;
    Datum           total_time;
    Datum           up_time;

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);
    if (state == NULL)
        PG_RETURN_NULL();

    sum        = state->uptime + state->downtime;
    total_time = DirectFunctionCall1(int4_numeric, Int32GetDatum(sum));

    if (sum == 0)
        PG_RETURN_DATUM(total_time);

    up_time = DirectFunctionCall1(int4_numeric, Int32GetDatum(state->uptime));

    PG_RETURN_DATUM(DirectFunctionCall2(numeric_div, up_time, total_time));
}
```

```
if (state == NULL)
    PG_RETURN_NULL();
```

- internal state can still be NULL (e.g. now rows)

C functions - calling other functions

```
Datum uptime_sf_final(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    int32           sum;
    Datum           total_time;
    Datum           up_time;

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);
    if (state == NULL)
        PG_RETURN_NULL();

    sum        = state->uptime + state->downtime;
    total_time = DirectFunctionCall1(int4_numeric, Int32GetDatum(sum));

    if (sum == 0)
        PG_RETURN_DATUM(total_time);

    up_time = DirectFunctionCall1(int4_numeric, Int32GetDatum(state->uptime));

    PG_RETURN_DATUM(DirectFunctionCall2(numeric_div, up_time, total_time));
}
```

```
total_time = DirectFunctionCall1(
                    int4_numeric,
                    Int32GetDatum(sum)
                );
```

cast int32 to numeric data type

- we can still call sql interfacing function from c
- DirectFunctionCall1(func_name, Datum)
- Int32GetDatum turns int32 into a Datum value

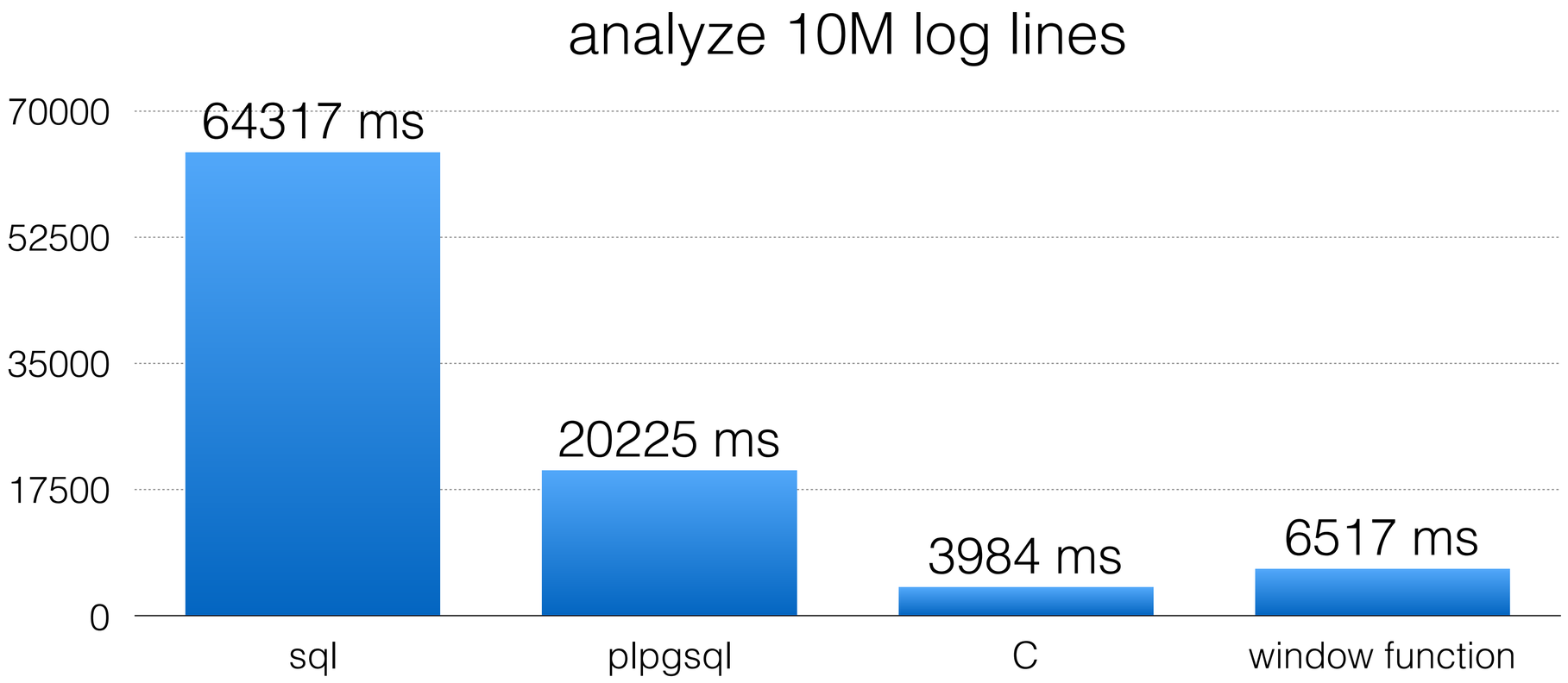
Analyzing logs - final function

```
Datum uptime_sf_final(PG_FUNCTION_ARGS)
{
    UptimeAggState *state;
    int32           sum;
    Datum           total_time;
    Datum           up_time;

    state = PG_ARGISNULL(0) ? NULL : (UptimeAggState *) PG_GETARG_POINTER(0);
    if (state == NULL)
        PG_RETURN_NULL();

    sum        = state->uptime + state->downtime;
    total_time = DirectFunctionCall1(int4_numeric, Int32GetDatum(sum));

    if (sum == 0)
        PG_RETURN_DATUM(total_time);

    up_time = DirectFunctionCall1(int4_numeric, Int32GetDatum(state->uptime));

    PG_RETURN_DATUM(DirectFunctionCall2(numeric_div, up_time, total_time));
}
```

```
PG_RETURN_DATUM(
    DirectFunctionCall2(
        numeric_div,
        up_time,
        total_time
    )
);
```

- return value is a Datum type
- we use build in numeric_div
- we determine that pg_getdef

Analyzing logs - Performance



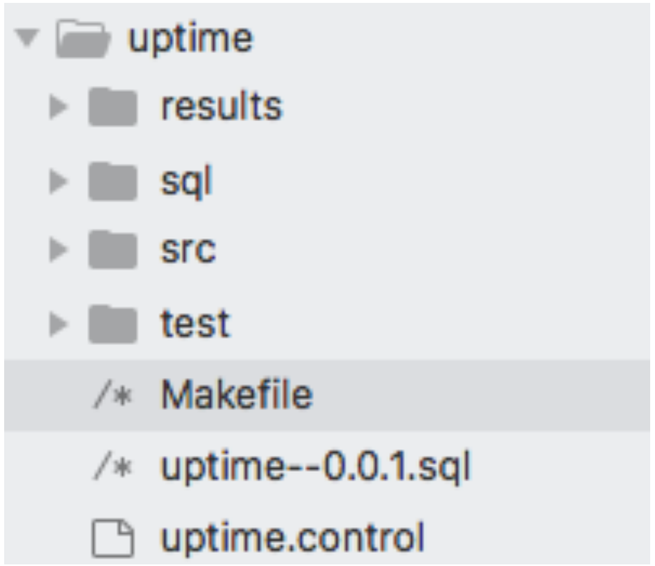
But Window Functions

```
-- window functions can do the trick as well
SELECT server_name, (SUM(span) FILTER(WHERE response_time < '300 ms'))::numeric / SUM(span)
FROM(
    SELECT server_name, created_at, response_time,
        EXTRACT(
            EPOCH FROM created_at - lag(created_at) OVER (
                PARTITION BY server_name ORDER BY created_at
                RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
            ))::int as span
    FROM logs
)t GROUP BY server_name;

-- but are complex compared to using an aggregate
SELECT server_name, UPTIME(created_at, response_time, '300 ms' ORDER BY created_at)
FROM logs GROUP BY server_name;
```

- we probably don’t want to explain window functions to our application developers
- most ORMs doesn’t support them
- developers will use sql strings - and end up with sql injections

shipping as Extension

- ship sql code together with c library
- just call `CREATE EXTENSION uptime`
- include tests
- version your code `ALTER EXTENSION UPDATE TO 'new version';`



- regression test results
- sql code
- c source code
- test files
- generic Makefile
- versioned sql extension
- extension control file

extension control file

```
# uptime extension
comment = 'Server uptime analytics'
default_version = '0.0.1'
relocatable = true
requires = ''
```

- CREATE EXTENSION relies on a .control file named after the extension
- can set different parameters (at least default version should be set)
- see https://www.postgresql.org/docs/current/static/extend-extensions.html

generic Makefile

```
EXTENSION    = uptime
EXTVERSION   = $(shell grep default_version $(EXTENSION).control | \
               sed -e "s/default_version[[:space:]]*=[[:space:]]*'\([^']*\)'/\1/")
PG_CONFIG   ?= pg_config
DATA         = $(wildcard *--*.sql)
PGXS        := $(shell $(PG_CONFIG) --pgxs)
MODULE_big   = $(EXTENSION)
OBJS         = $(patsubst %.c,%.o,$(wildcard src/*.c))
TESTS        = $(wildcard test/sql/*.sql)
REGRESS      = $(patsubst test/sql/%.sql,%,$(TESTS))
REGRESS_OPTS = --inputdir=test --load-language=plpgsql --load-extension= $(EXTENSION)
SQLSRC       = $(wildcard sql/*.sql)
include $(PGXS)

all: $(EXTENSION)--$(EXTVERSION).sql

$(EXTENSION)--$(EXTVERSION).sql: $(SQLSRC)
  echo "-- complain if script is sourced in psql, rather than via CREATE EXTENSION" > $@
  echo "\echo Use \"CREATE EXTENSION ${EXTENSION}\" to load this file. \quit" >> $@
  echo "" >> $@
  cat $^ >> $@
```

regression tests

```
TESTS         = $(wildcard test/sql/*.sql)
REGRESS       = $(patsubst test/sql/%.sql,%,$(TESTS))
REGRESS_OPTS = --inputdir=test \
               --load-language=plpgsql \
               --load-extension=$(EXTENSION)
```

- build system adds an installcheck target
- calls pg_regress
- test cases placed in sql/test_case.sql
- expected output in expected/test_case.out
- both are located under test/
- simply diffs output

how to go from here?

- explore the postgres source code
- investigate how existing objects are implemented
- explore system catalogs
  - pg_proc
  - pg_aggregate,
  - pg_operator
  - pg_type
- use pgAdmin to brows catalog
- look at pg_getdef

pg_getdef - explore Functions

```
SELECT * FROM get_func('numeric_div');
-[ RECORD 1 ]--------+-------------------------
Name                 | numeric_div
Result data type     | numeric
Argument data types  | numeric, numeric
Type                 | normal
Volatility           | immutable
Language             | internal
Source code          | numeric_div
Description          | implementation of / operator
```

https://github.com/adjust/pg_c_dev

pg_getdef - explore Operators

```
SELECT * FROM get_op('/','integer', 'integer');
-[ RECORD 1 ]------------------------
oprkind    | both
source     | int4div(integer,integer)
leftarg    | integer
rightarg   | integer
commutator | 0
negator    | 0
restrict   | -
join       | -
merges     | false
hashes     | false
```

https://github.com/adjust/pg_c_dev

pg_getdef - explore Aggregates

```
SELECT * FROM get_agg('avg','integer');
-[ RECORD 1 ]----+-------------------
sfunc            | int4_avg_accum
finalfunc        | int8_avg
stype            | bigint[]
combinefunc      | int4_avg_combine
serialfunc       | -
deserialfunc     | -
msfunc           | int4_avg_accum
minvfunc         | int4_avg_accum_inv
mfinalfunc       | int8_avg
mstype           | bigint[]
initcond         | {0,0}
```

https://github.com/adjust/pg_c_dev

Code samples and slides

https://github.com/adjust/pg_c_dev

further reading

- Intro into Writing PostgreSQL Functions in C from 2007
  https://linuxgazette.net/139/peterson.html
- Intro into writing postgres extension in four parts
  http://big-elephants.com/2015-10/writing-postgres-extensions-part-i/
- Overview about memory contextx
  https://blog.pgaddict.com/posts/introduction-to-memory-contexts
- C-Language Functions
  https://www.postgresql.org/docs/current/static/xfunc-c.html
- Extension Building Infrastructure
  https://www.postgresql.org/docs/current/static/extend-pgxs.html
- Packaging Related Objects into an Extension
  https://www.postgresql.org/docs/current/static/extend-extensions.html

If you had fun…
If you want more…

We are hiring

Code samples and slides

https://github.com/adjust/pg_c_dev