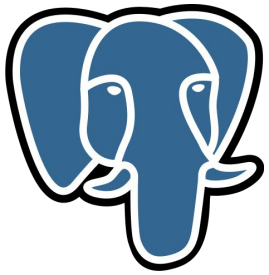
# Deep Dive into PostgreSQL Codebase for Beginner Contributors

**PostgreSQL Development Bootcamp**
**March 19, 2023**
**Anastasia Lubennikova, PostgreSQL Major Contributor, Neon, Inc.**

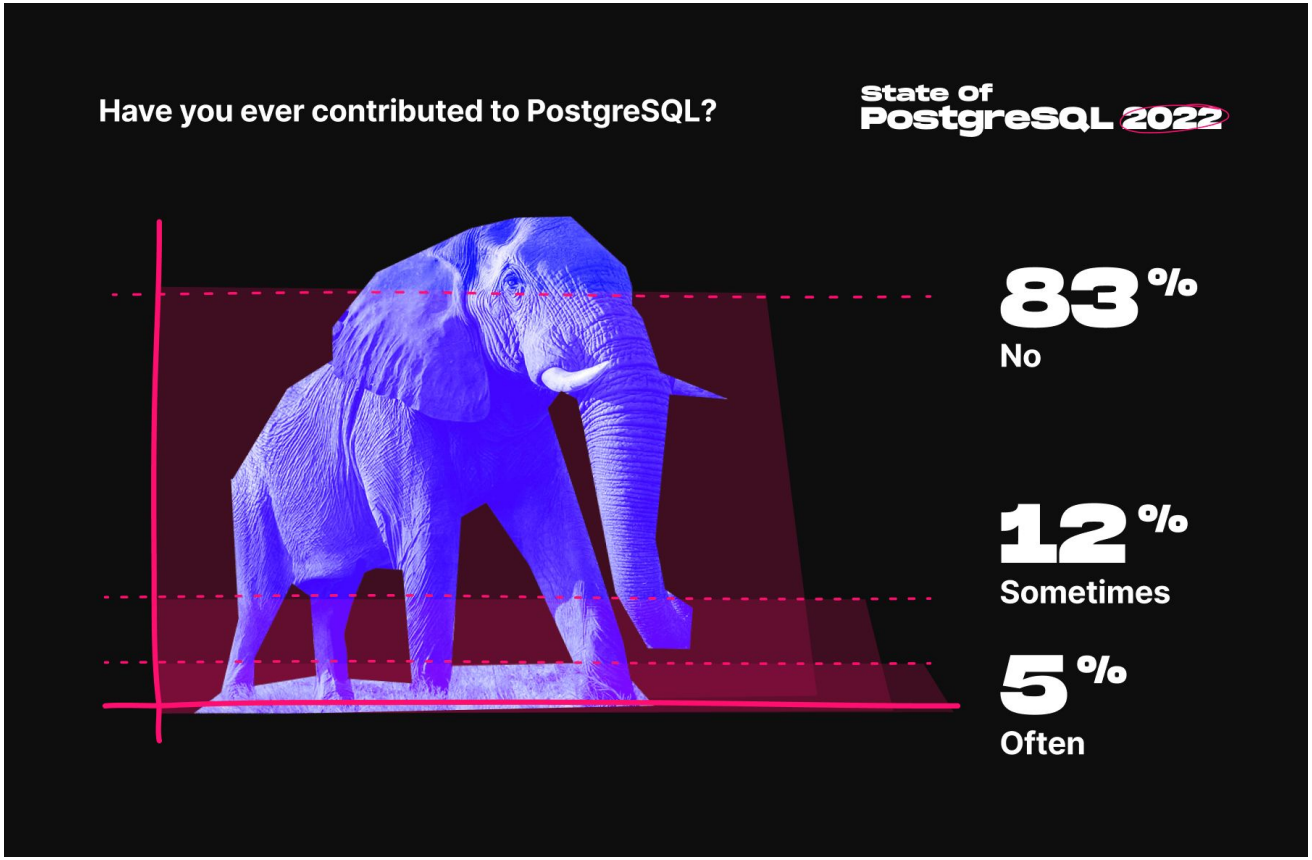
# About me


- Neon serverless database developer
- PostgreSQL Major Contributor
  - Index-only scan for GiST
  - Indexes with INCLUDE columns
  - B-tree compression
  - IF NOT EXISTS for CREATE SERVER
- pg_probackup developer
- GSoC (Google Summer Of Code) intern & mentor
- Educator & conference speaker
- ex-member of PostgreSQL Code of Conduct Committee



NEON

# About you




https://www.timescale.com/blog/state-of-postgresql-how-to-contribute-to-postgresql-and-the-community/

# What's on our list for today?

- PostgreSQL code overview
- PostgreSQL code specific
- Code style & development tips

# Disclamer

# About PostgreSQL

- Written in C
  - documentation uses SGML
- 1.3 million rows of code
- 30+ years of active development
- Global open source project
  - supported by international community and vendor companies
- BSD license
  - guarantees software will be available forever, including for proprietary use
- Not everything comes out-of-box
  - big ecosystem
- Internationalization and localization features

# PostgreSQL source code

```
    git clone https://git.postgresql.org/git/postgresql.git
--depth=1

    cd postgresql
```

There is a master and back branches and tags for releases

i.e.  REL_15_STABLE

      REL_15_2

# PostgreSQL Source Directory

| Directory | Description |
|---|---|
| config | Config system for driving the build |
| contrib | Source code for Contrib Modules, aka, Extensions |
| doc | Documentation (SGML) |
| src/backend | PostgreSQL Server ("Back-End") |
| src/bin | psql, pg_dump, initdb, pg_upgrade, etc ("Front-End") |
| src/common | Code common to the front and back ends |
| src/fe_utils | Code useful for multiple front-end utilities |
| src/include | Header files for PG, mainly back-end |
| src/include/catalog | Definition of the PostgreSQL catalog tables (pg_catalog.* tables) |
| src/interfaces | Interfaces to PG, including libpq, ECPG |
| src/pl | Core Procedural Languages (plpgsql, plperl, plpython, tcl) |
| src/port | Platform-specific hacks |
| src/test | Regression tests |
| src/timezone | Timezone code from IANA |
| src/tools | Developer tools (including pgindent) |

https://www.postgresql.eu/events/fosdem2019/sessions/session/2282/slides/160/hackingpg.pdf

# PostgreSQL Backend Code (src/backend)

| Directory | Description |
|---|---|
| access | Methods for accessing different types of data (heap, btree indexes, gist/gin, etc). |
| bootstrap | Routines for running PostgreSQL in "bootstrap" mode (by initdb) |
| catalog | Routines used for modifying objects in the PG Catalog (pg_catalog.*) |
| commands | User-level DDL/SQL commands (CREATE/ALTER, VACUUM/ANALYZE, COPY, etc) |
| executor | Executor, runs queries after they have been planned/optimized |
| foreign | Handles Foreign Data Wrappers, user mappings, etc |
| jit | Provider independent Just-In-Time Compilation infrastructure |
| lib | Code useful for multiple back-end components |
| libpq | Backend code for talking the wire protocol |
| main | main(), determines how the backend PG process is starting and starts right subsystem |
| nodes | Generalized "Node" structure in PG and functions to copy, compare, etc |
| optimizer | Query optimizer, implements the costing system and generates a plan for the executor |
| parser | Lexer and Grammar, how PG understands the queries you send it |
| partitioning | Common code for declarative partitioning in PG |
| po | Translations of backend messages to other languages |

https://www.postgresql.eu/events/fosdem2019/sessions/session/2282/slides/160/hackingpg.pdf

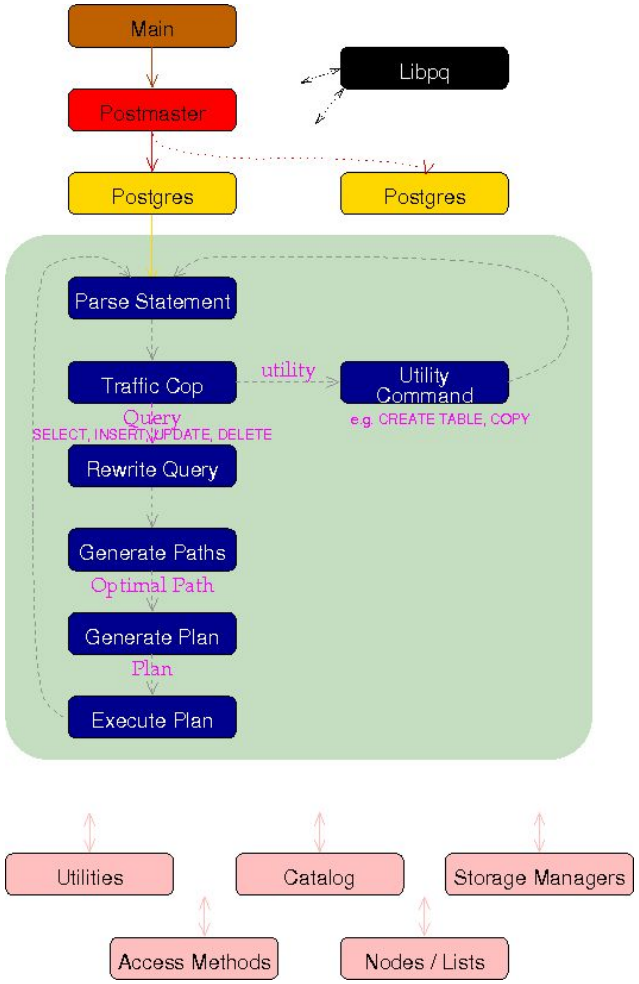
# PostgreSQL Backend Code (2)

| Directory | Description |
|---|---|
| port | Backend-specific platform-specific hacks |
| postmaster | The "main" PG process that always runs, answers requests, hands off connections |
| regex | Henry Spencer's regex library, also used by TCL, maintained more-or-less by PG now |
| replication | Backend components to support replication, shipping WAL logs, reading them in, etc |
| rewrite | Query rewrite engine, used with RULEs, also handles Row-Level Security |
| snowball | Snowball stemming, used with full-text search |
| statistics | Extended Statistics system (CREATE STATISTICS) |
| storage | Storage layer, handles most direct file i/o, support for large objects, etc |
| tcop | "Traffic Cop"- this is what gets the actual queries, runs them, etc |
| tsearch | Full-Text Search engine |
| utils | Various back-end utility components, cacheing system, memory manager, etc |

https://www.postgresql.eu/events/fosdem2019/sessions/session/2282/slides/160/hackingpg.pdf

# Backend flowchart

https://www.postgresql.org/developer/backend/



# PostgreSQL code specific

- Function calling convention
- Datum, Tuple, Oid, Nodes, NameData
- Memory management
- Lock management
- Logging and error handling
- Configuration management (GUC)
- Internal libraries

# Function Calling

C-Language Functions

src/include/fmgr.h

```
PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}
```

# Function Calling

```
/* Standard parameter list for fmgr-compatible functions */
#define PG_FUNCTION_ARGS   FunctionCallInfo fcinfo

/* Macros for fetching arguments of standard types */
#define PG_GETARG_DATUM(n) (fcinfo->arg[n])
#define PG_GETARG_INT32(n) DatumGetInt32(PG_GETARG_DATUM(n))
#define PG_NARGS() (fcinfo->nargs)
#define PG_ARGISNULL(n) (fcinfo->argnull[n])

/*  Macros for returning results of standard types */
#define PG_RETURN_DATUM(x) return (x)
#define PG_RETURN_VOID() return (Datum) 0
#define PG_RETURN_NULL()
```

# Datum usage

```
typedef uintptr_t Datum;
```

```
It's the backend-internal representation of a single value of any SQL
data type.  The code using the Datum has to know which type it is,
since the Datum itself doesn't contain that information.  Usually,
C code will work with a value in a "native" representation, and then
convert to or from Datum in order to pass the value through
data-type-independent interfaces.

                        regards, tom lane
```

https://www.postgresql.org/message-id/22559.1277304590%40sss.pgh.pa.us

# Datum

C-Language Functions

src/include/postgres.h

```
#define DatumGetCString(X) ((char *) DatumGetPointer(X))

#define CStringGetDatum(X) PointerGetDatum(X)
```

```
DatumGetInt32
Int32GetDatum
```

**PG_DETOAST_DATUM**

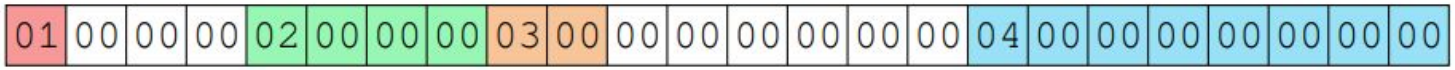
# Tuple

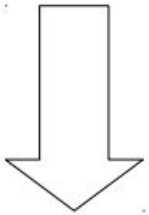
Returning Rows (Composite Types)

- Row data type
- Several Datums packed together
- `HeapTuple, IndexTuple`
- `TupleHeader`
- `TupleDesc`

# Tuple storage optimization

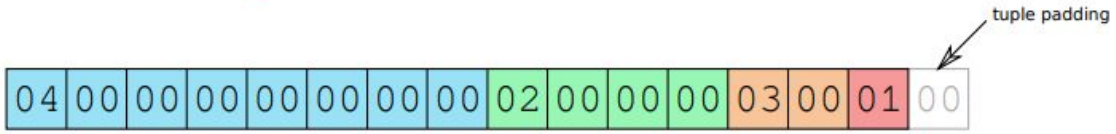
```
test=# create table test (a boolean, b int, c smallint, d bigint);
```



Reorder attributes in to get rid of unnecessary padding

```
test=# create table test (d bigint, b int, c smallint, a boolean);
```




[Tuple internals talk (2016)]

# Functions to work with tuple

```
#include "funcapi.h"

/* Build a tuple descriptor for our result type */
TypeFuncClass get_call_result_type(
       FunctionCallInfo fcinfo, Oid *resultTypeId,
       TupleDesc *resultTupleDesc)

TupleDesc BlessTupleDesc(TupleDesc tupdesc)

HeapTuple heap_form_tuple(
    TupleDesc tupdesc,
    Datum *values, bool *isnull)

HeapTupleGetDatum(HeapTuple tuple)
```

# Oid

```
/*
* Object ID is a fundamental type in Postgres.
*/
typedef unsigned int Oid;


#define FirstNormalObjectId      16384
```

Unique identifier in pg_catalog tables

# Object name length

- Object names have a 63 byte limit (by default, set at compile time)

#define NAMEDATALEN 64

```
/*
* Representation of a Name: effectively just a C string, but null-padded to
* exactly NAMEDATALEN bytes.  The use of a struct is historical.
*/
typedef struct nameData
{
   char        data[NAMEDATALEN];
} NameData;
typedef NameData *Name;
```

# Node

- PostgreSQL expression trees are made up of `Nodes`
- Each node has a type, plus appropriate data
- 'type' of a Node is stored in the Node, allowing `IsA()` function
- Nodes created using `makeNode(TYPE)`
- Used extensivly by the grammar, but also elsewhere
- To add a new Node type
  - Add to `include/nodes/nodes.h`
  - Create make / copy / equality funcs in `backend/nodes/`
- `To debug use elog_node_display`

https://www.postgresql.eu/events/fosdem2019/sessions/session/2282/slides/160/hackingpg.pdf

# Memory management

src/backend/utils/mmgr/README

- Memory allocation occurs within "contexts".
- `palloc(), palloc0(), pfree(), repalloc()`
- Context are organized into tree:
  - TopMemoryContext - for stuff that should live forever
  - TopTransactionContext
  - CurTransactionContext
  - CurrentMemoryContext - what palloc() will use
- Entire memory context is freed at the end of transaction / operation.

# Locks

If you access a relation or relation's page,
you need to explicitly lock it and unlock after use

```
rel = relation_open(relid, AccessShareLock);
buf = ReadBufferExtended(rel, forknum, blkno, RBM_NORMAL, NULL);

LockBuffer(buf, BUFFER_LOCK_SHARE);
memcpy(raw_page_data, BufferGetPage(buf), BLCKSZ);
LockBuffer(buf, BUFFER_LOCK_UNLOCK);
ReleaseBuffer(buf);

relation_close(rel, AccessShareLock);
```

# Logging and Error reporting

```
ereport(ERROR,
(errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
 errmsg("cannot truncate a table referenced in a foreign key
constraint"),
  errdetail("Table \"%s\" references \"%s\".",
            relname2, relname),
  errhint("Truncate table \"%s\" at the same time, "
          "or use TRUNCATE ... CASCADE.",
          relname2)));
```

# Error handling

src/include/utils/elog.h

```
PG_TRY();
{
    ... code that might throw ereport(ERROR) ...
}
PG_CATCH();
{
    ... error handling code …
    PG_RE_THROW();
}
PG_END_TRY();
```

# Interruptions

src/include/miscadmin.h

```
/* Service interrupt, if one is pending and it's safe to
service it now */
#define CHECK_FOR_INTERRUPTS() \
do { \
    if (INTERRUPTS_PENDING_CONDITION()) \
        ProcessInterrupts(); \
} while(0)
```

Needed for pg_cancel_backend(pid) to work

# Assertions

- only work in builds with `--cassert-enabled`
- fatal exception
- help developers to read and use your code correctly
  - check invariants
  - check things that should never happen

```
Assert(some_ptr != NULL);

Assert(blkno != InvalidBlockNumber);
```

# GUC (grand unified configuration)

- config parameters
- set in `postgresql.conf` or via `SET` command
- Can be defined in extensions

# Useful GUCs

Developer options

```
    log_min_messages = DEBUG5

    log_error_verbosity = VERBOSE
```

```
postgres=# SET backtrace_functions = 'pg_strtoint32' ;

SET

postgres=# SELECT int 'foobar' ;

ERROR:  invalid input syntax for type integer: "foobar"

LINE 1: SELECT int 'foobar' ;
```

# Useful GUCs (2)

- Planner configuration

```
SET enable_indexscan=false;

SET enable_mergejoin=false;
```

- EXPLAIN

```
explain (analyze, buffers) table t;

explain (analyze, verbose) table t;

explain (analyze, format xml) table t;
```

# Internal libraries

src/backend/lib/README

- binaryheap.c - a binary heap
- bipartite_match.c - Hopcroft-Karp maximum cardinality algorithm for bipartite graphs
- bloomfilter.c - probabilistic, space-efficient set membership testing
- dshash.c - concurrent hash tables backed by dynamic shared memory areas
- hyperloglog.c - a streaming cardinality estimator
- ilist.c - single and double-linked lists
- integerset.c - a data structure for holding large set of integers
- knapsack.c - knapsack problem solver
- pairingheap.c - a pairing heap
- rbtree.c - a red-black tree
- stringinfo.c - an extensible string type

# Internal libraries (2)

src/backend/nodes/README

- bitmapset.c - Bitmapset support
- list.c - generic list support
- multibitmapset.c - List-of-Bitmapset support
- params.c - Param support
- tidbitmap.c - TIDBitmap support
- value.c - support for value nodes

- src/backend/utils/hash/dynahash.c - dynamic chained hash tables

- src/backend/storage/file/fd.c - Virtual file descriptor code
  - wrappers for open(), fopen(), opendir(), etc.

# Internal libraries (3)

src/port/

- pg_usleep
- pg_strcasecmp
- pg_strong_random

# Code style

PostgreSQL Coding Conventions

src/tools/editors/

- 4-space tabs
- Line length 80 symbols
- Comments
  - C-style comments only

    ```
    /* like this */
    ```

  - on their own lines
  - Describe why, not what or how
  - Lots of them

# Reuse existing code

- It has already been tested
  - most likely all corner cases and errors are handled
  - and it is covered by regression tests
- It is already portable
- You have less to write
  - and less to maintain

- Read comments to ensure that you use API correctly
  - There can be some assumptions about who is responsible for locking data, freeing memory, etc..

# Development

- Coding
- Debugging
- Testing
  - regression tests
  - concurrency tests
  - performance tests
  - backward compatibility
- Code style
- Documentation

# Sources

https://db-engines.com/ Knowledge Base of DBMS

https://dbdb.io/ Database of Databases

Video lecture courses:
CMU Database Systems
CMU Advanced Database Systems

https://www.vldb.org/conference.html Scientific conference VLDB

https://summerofcode.withgoogle.com/ Google Summer Of Code internship

# Sources. PostgreSQL

https://www.postgresql.org/ PostgreSQL website

https://www.postgresql.org/docs/ PostgreSQL documentation

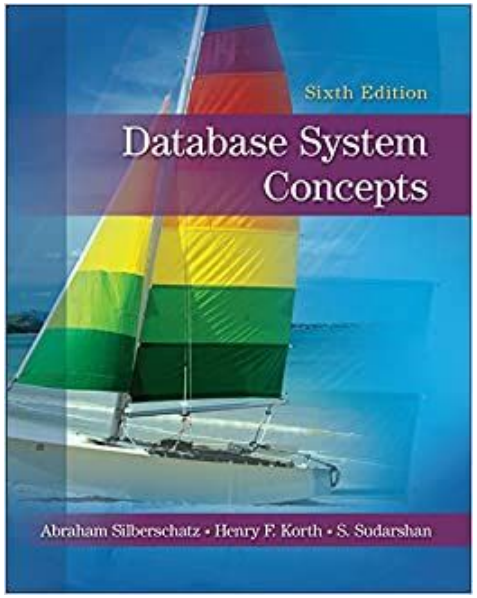
https://wiki.postgresql.org/wiki/Main_Page PostgreSQL Wiki

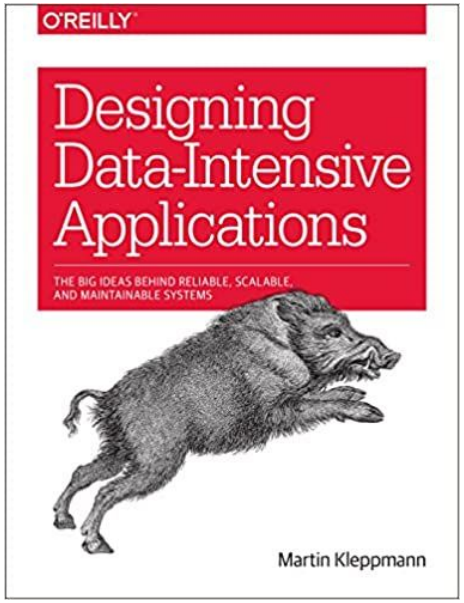
http://www.interdb.jp/pg/index.html The Internals of PostgreSQL

https://www.youtube.com/c/PgconOrg Developer Conference talk videos

# Books

https://www.db-book.com/ Database System Concepts
https://dataintensive.net/ Designing Data-Intensive Applications



# Thank you for attention!
# Questions?