Let's Build a Postgres Extension

Shaun Thomas

Principal Software Engineer

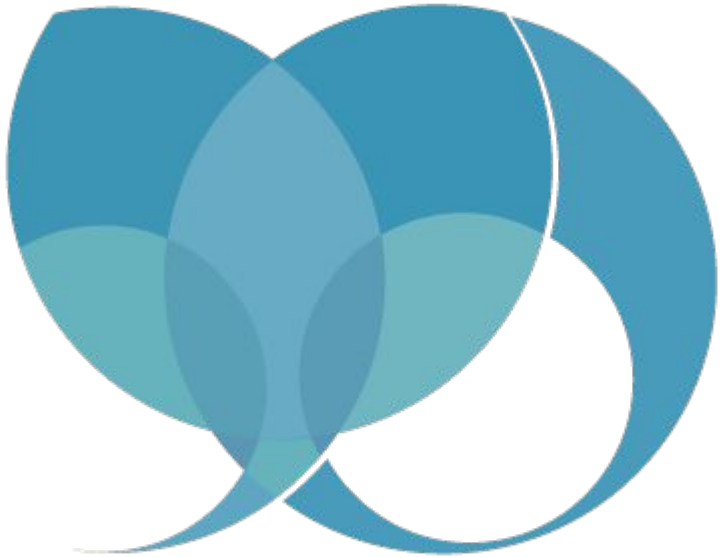
April 23, 2026

pgEdge®

Who are pgEdge?

- Distributed Postgres
- Active-Active clusters
- Postgres AI Tools
- Cloud Services
- Platform Automation
- Ultra High Availability

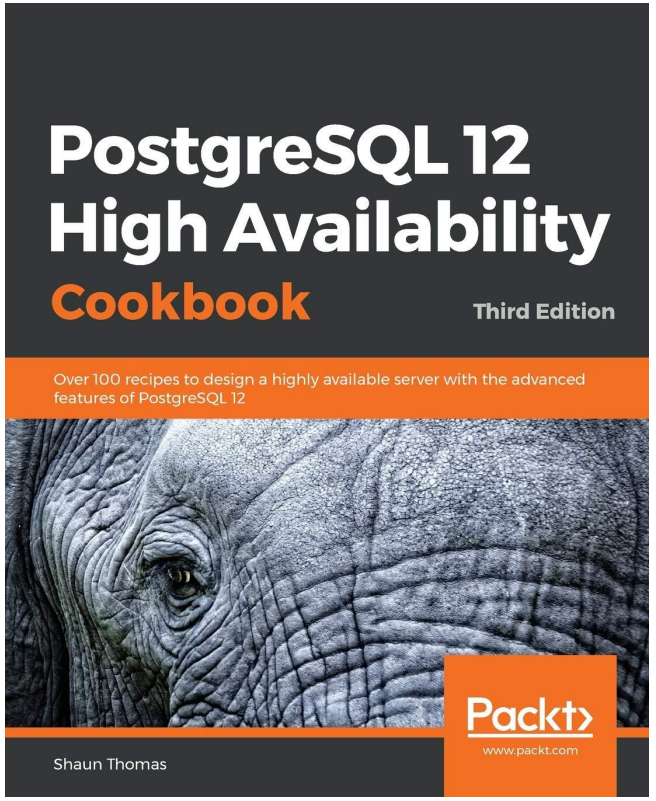
www.pgedge.com



Who am I?

- Author
- Speaker
- Blogger
- Mentor
- Dev

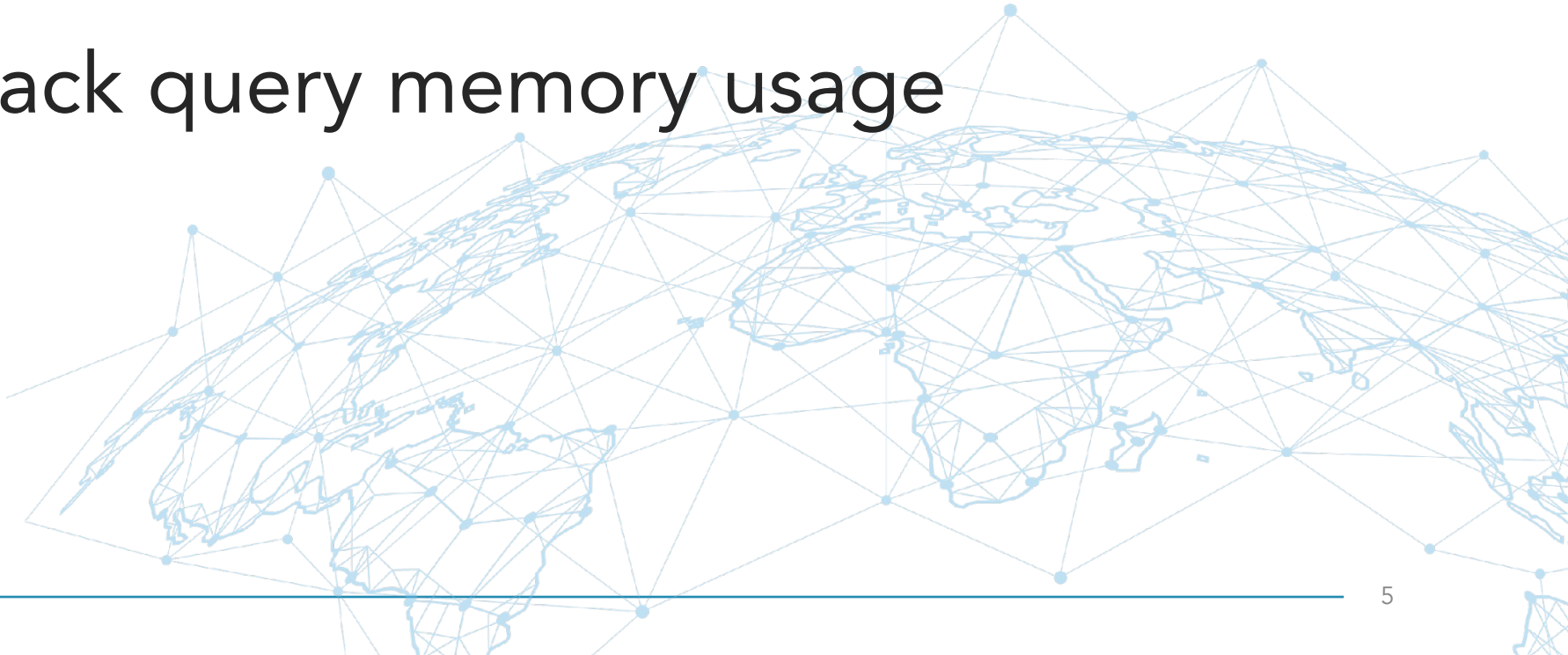
shaun.thomas@
pgedge.com



What's Our Agenda?

What we're Doing

Building an extension to track query memory usage

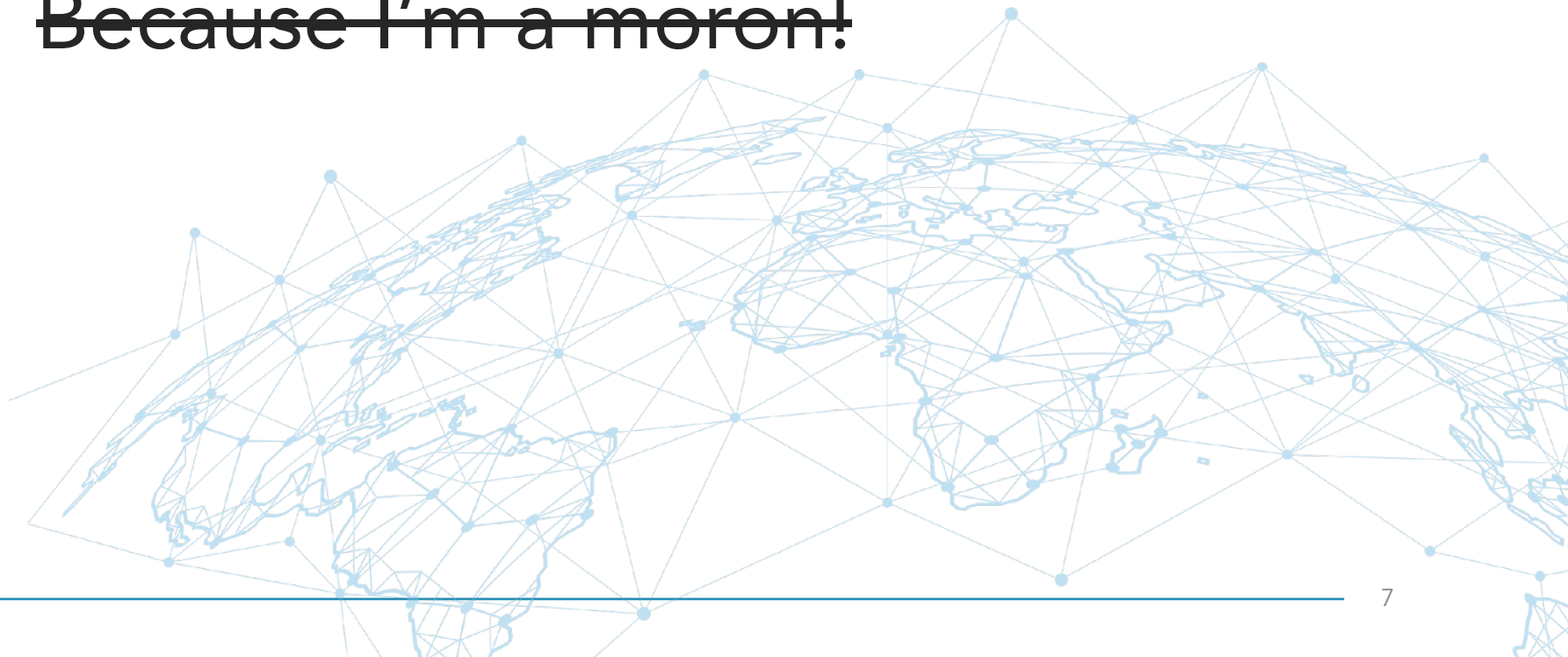


How are we Doing This?

By going on an adventure!

Why Should we Listen to You?

~~Because I'm a moron!~~



Why Should we Listen to You?

I learned by building this extension.

You can too.

First Things First

Things We'll Need

1. Postgres source
2. Build tools and libraries
3. A rough template
4. Luck

Getting the Postgres Source

Option 1: Packaging / Installer

https://www.postgresql.org/download/

Option 2: git (mirror)

https://github.com/postgres/postgres

Build Tools and Libraries

Debian-based systems:

```
sudo apt install build-essential \
     libreadline-dev zlib1g-dev flex bison libxml2-dev \
     libxslt-dev libssl-dev libxml2-utils xsltproc \
     ccache pkg-config libicu-dev
```

RHEL-based systems:

```
sudo yum install -y bison-devel readline-devel \
     zlib-devel openssl-devel wget ccache libicu-devel

sudo yum groupinstall -y 'Development Tools'
```

https://wiki.postgresql.org/wiki/Compile_and_Install_from_source_code

Basic Postgres Build

```
cd postgres
./configure --prefix=/custom
make -j8
make install
```

Basic Extension Build

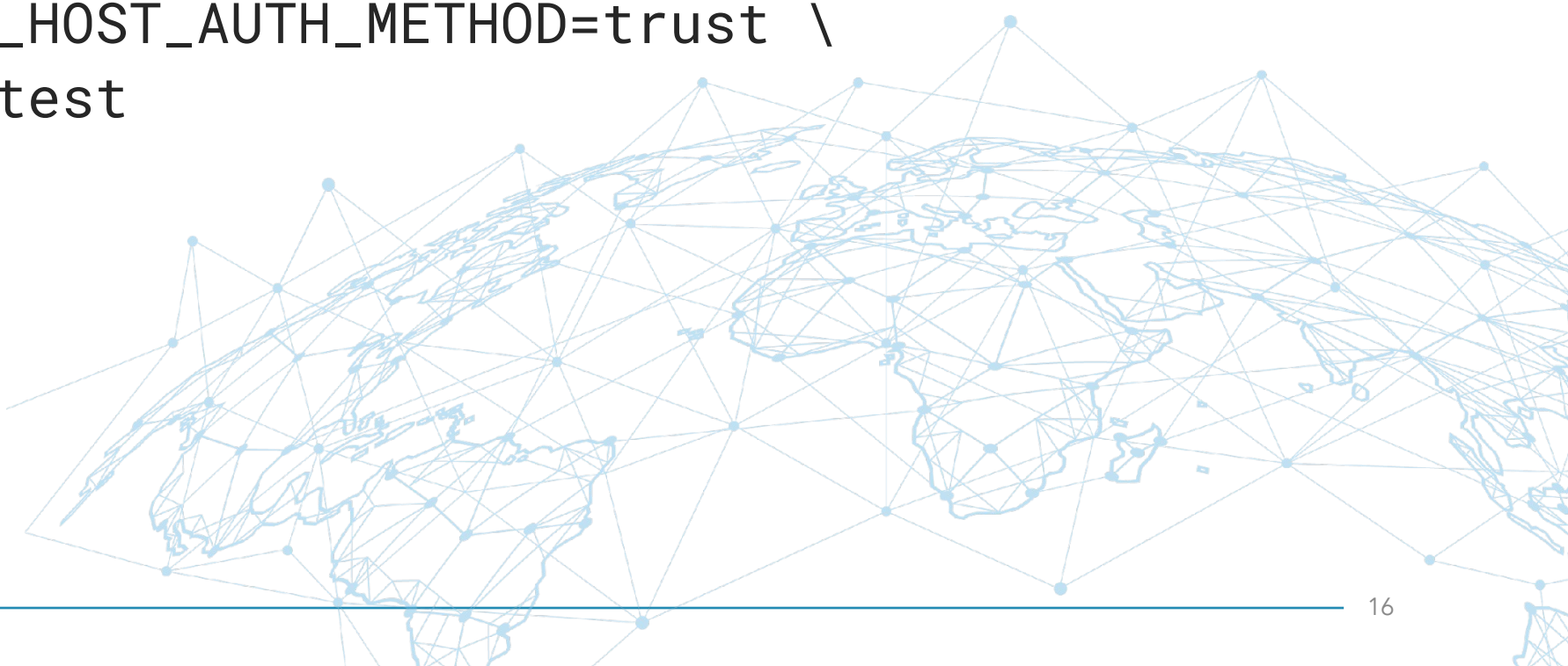
```
cd querymem
PATH=/my/postgres/bin:$PATH make install
make
make install
```

Consider Using Docker

```
FROM postgres:18

RUN apt-get -y update && apt-get -y upgrade && \
    apt-get -y install build-essential clang postgresql-server-dev-18 \
        libreadline-dev zlib1g-dev flex bison libxml2-dev \
        libxslt-dev libssl-dev libxml2-utils xsltproc \
        ccache pkg-config libicu-dev && \
    apt-get clean

COPY . /build

RUN cd /build && make && make install
```

Launching the Docker Env

```
cd querymem
docker build -t querymem:latest .

docker run -d --name=querymem \
    -e POSTGRES_HOST_AUTH_METHOD=trust \
    querymem:latest
```



The Makefile That Does Everything

```
MODULE_big = querymem

PGFILEDESC = "Extension to view query memory usage"
EXTENSION = querymem
DATA = querymem--1.0.sql
OBJS = querymem.o

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

The Extension Control File

```
# querymem.control
comment = 'PG Query Memory Extension'
default_version = '1.0'
module_pathname = '$libdir/querymem'
relocatable = true
```

A SQL Installer Stub

```
-- Postgres will execute this SQL file upon invoking
-- the "CREATE EXTENSION querymem" command.

/* SQL CODE TBA */
```

The Theory

The Problem With work_mem

It multiplies by the amount

of nodes in a query, per session

What we Want:

A tool that predicts maximum memory usage per query

Now What?

Here's What we Actually Need:

- The ability to parse a query
- A way to transform a parsed query into the expected plan
- A method to walk the plan
- The values of **`work_mem`** and **`hash_mem_multiplier`**
- A new function users can execute on any query

First Steps

Remember the Postgres Source?

We’re going to need it…

The User Docs are Great

Extending SQL

https://www.postgresql.org/docs/current/extend.html

C-Language Functions

https://www.postgresql.org/docs/current/xfunc-c.html

But they only really cover the basics.

The REAL Dev Docs

src/tutorial/README
src/pl/plperl/README
src/backend/replication/README
src/backend/lib/README
src/backend/regex/README
src/backend/statistics/README
src/backend/access/brin/README
src/backend/access/hash/README
src/backend/access/gist/README
src/backend/access/transam/README
src/backend/access/gin/README
src/backend/access/rmgrdesc/README
src/backend/access/nbtree/README
src/backend/access/spgist/README
src/backend/executor/README
src/backend/jit/README

src/backend/utils/misc/README
src/backend/utils/mmgr/README
src/backend/utils/mb/README
src/backend/utils/resowner/README
src/backend/utils/fmgr/README
src/backend/nodes/README
src/backend/snowball/README
src/backend/parser/README
src/backend/optimizer/plan/README
src/backend/optimizer/README
src/backend/storage/buffer/README
src/backend/storage/smgr/README
src/backend/storage/freespace/README
src/backend/storage/page/README
src/backend/storage/lmgr/README

src/common/unicode/README
src/tools/pg_bsd_indent/README
src/tools/ci/README
src/tools/pgindent/README
src/tools/pginclude/README
src/tools/ifaddrs/README
src/bin/pg_amcheck/README
src/bin/pgevent/README
src/interfaces/ecpg/test/connect/README
src/interfaces/libpq/README
src/interfaces/libpq-oauth/README
src/timezone/tznames/README
src/timezone/README
src/include/catalog/README
src/port/README

Other Good Options

/contrib

- Extensions written by the core devs.
- Great examples of working, updated code.

/src/include

- Struct definitions and function declarations.
- Extensive code comments as explanation.

The Initial Code

```
PG_MODULE_MAGIC;
PG_FUNCTION_INFO_V1(get_query_mem);

Datum get_query_mem(PG_FUNCTION_ARGS)
{
    char *query_arg = text_to_cstring(PG_GETARG_TEXT_PP(0));
    PG_RETURN_INT32(DatumGetInt32(0));
} // get_query_mem
```

The Initial SQL Wrapper

```
CREATE FUNCTION get_query_mem(text)
RETURNS INT
     AS 'MODULE_PATHNAME', 'get_query_mem'
     LANGUAGE C VOLATILE;
```

The Scavenger Hunt

How do we Parse Queries?

1. **`backend/parser/README`** says start at **`parser.c`**
2. **`raw_parser`** function is the first thing in `parser.c`
3. raw_parser returns a **`List`** of parse trees
4. A parse tree is just a **`RawStmt`** node
5. `RawStmt` is defined in **`include/nodes/parsenodes.h`**

How do we Work with Nodes?

1. **`backend/nodes/README`** explains how nodes work
2. **`include/nodes/pg_list.h`** defines helper macros
3. We only need the first node in the parse tree
4. Use **`linitial`** to “unwrap” the first `List` item

See Raw Parse Trees

```
Datum get_query_mem(PG_FUNCTION_ARGS)
{
    char *query_arg = text_to_cstring(PG_GETARG_TEXT_PP(0));

    List *parse_tree = raw_parser(query_arg, RAW_PARSE_DEFAULT);
    RawStmt *parsed = (RawStmt *)linitial(parse_tree);
    pprint(parsed);

    PG_RETURN_INT32(DatumGetInt32(0));
} // get_query_mem
```

What a Plan Comes Together

1. **`backend/optimizer/README`** says **`planner()`** does it all
2. `planner()` is declared in **`optimizer/optimizer.h`**
3. `planner()` takes a **`Query`**, not a `RawStmt`
4. Need to convert `RawStmt` into a `Query`
5. **`transformTopLevelStmt()`** in **`parser/analyze.h`** does this!
6. But `transformTopLevelStmt()` needs a **`ParseState`**
7. `ParseState` is defined in **`parser/parse_node.h`**
8. We can initialize `ParseState` with **`make_parsestate()`**

Checking the Plan

```
Datum get_query_mem(PG_FUNCTION_ARGS)
{
    ParseState  *pstate = make_parsestate(NULL);
    /* … */

    query = transformTopLevelStmt(pstate, parsed);
    plan = planner(query, query_arg, 0, NULL);
    pprint(plan);

    /* … */
} // get_query_mem
```

Figuring out the Plan

1. **`planner()`** returns a **`PlannedStmt`** struct
2. `PlannedStmt` is defined in **`nodes/plannodes.h`**
3. `PlannedStmt` has a **`planTree`** of type **`Plan`**
4. Also contains a `List` of **`subplans`**
5. Each `Plan` contains a **`lefttree`** and **`righttree`** `Plan` node
6. It's recursive function time!

Walking in the Trees

```c
typedef struct
{
    uint16_t node_count;
    double multiplier;
} WalkContext;

static void walk_plan(Plan *plan, WalkContext *ctx)
{
    if (plan == NULL) return;

    /* TODO: Count nodes */

    walk_plan(plan->lefttree, ctx);
    walk_plan(plan->righttree, ctx);
}
```

The Docs For work_mem Say

Sets the base maximum amount of memory to be used by a query operation (such as a sort or hash table) before writing to temporary disk files.

But then:

The memory limit for a hash table is computed by multiplying **`work_mem`** by **`hash_mem_multiplier`**.

Finding Hash Plan Nodes

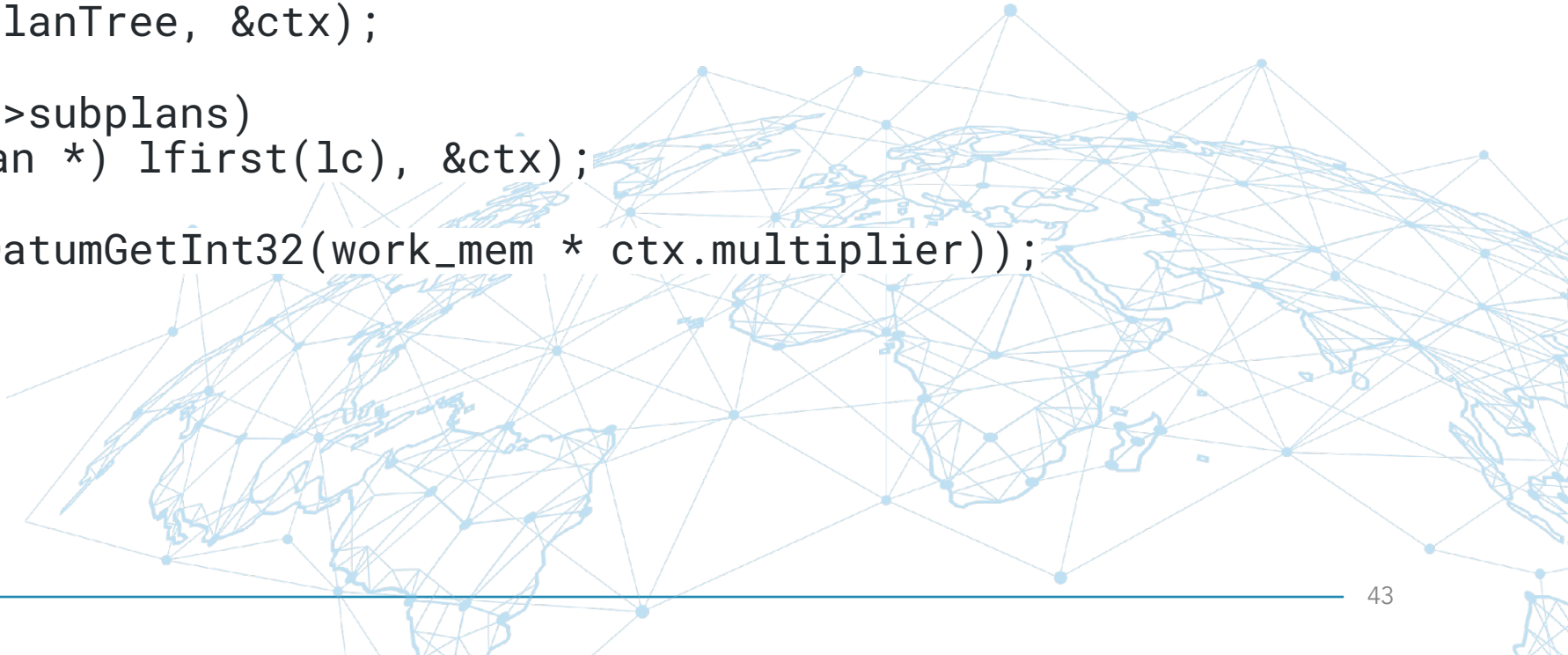
1. **`Plan`** nodes contain **`NodeTag type`** field
2. All `NodeTags` are defined in **`nodes/nodetags.h`**
3. Hash types include: **`T_HashPath`**, **`T_HashJoin`**, **`T_Hash`**, **`T_HashJoinState`**, and **`T_HashState`**
4. Everything else counts normally

The “Algorithm”

```
switch (nodeTag(plan))
{
    case T_Hash:
    case T_HashJoin:
    case T_HashJoinState:
    case T_HashPath:
    case T_HashState:
        ctx->node_count++;
        ctx->multiplier += hash_mem_multiplier;
        break;
    default:
        ctx->node_count++;
        ctx->multiplier += 1.0;
        break;
}
```

Integrating the Plan Traversal

```
Datum get_query_mem(PG_FUNCTION_ARGS)
{
    /* … */
    WalkContext  ctx = {0, 0.0};
    ListCell    *lc;

    /* … */
    walk_plan(plan->planTree, &ctx);

    foreach(lc, plan->subplans)
       walk_plan((Plan *) lfirst(lc), &ctx);

    PG_RETURN_INT32(DatumGetInt32(work_mem * ctx.multiplier));
} // get_query_mem
```



Does it Work?

Let's use Docker

```
cd querymem
docker build -t querymem:latest .

docker run -d --name=querymem \
    -e POSTGRES_HOST_AUTH_METHOD=trust \
    querymem:latest
```

Add Sample Data and Connect

```
docker exec -it -u postgres querymem \
       pgbench -i -s 10 postgres


docker exec -it -u postgres querymem psql
```

Create the Extension

```
CREATE EXTENSION querymem;
SET log_min_messages = NOTICE;
```

Try Something Simple

```
SELECT get_query_mem($$
  SELECT a.* FROM pgbench_accounts a
$$);

 get_query_mem
---------------
          4096
```

Add a Sort

```
SELECT get_query_mem($$
  SELECT a.* FROM pgbench_accounts a ORDER BY a.bid;
$$);

 get_query_mem
---------------
          8192
```

How About a Join?

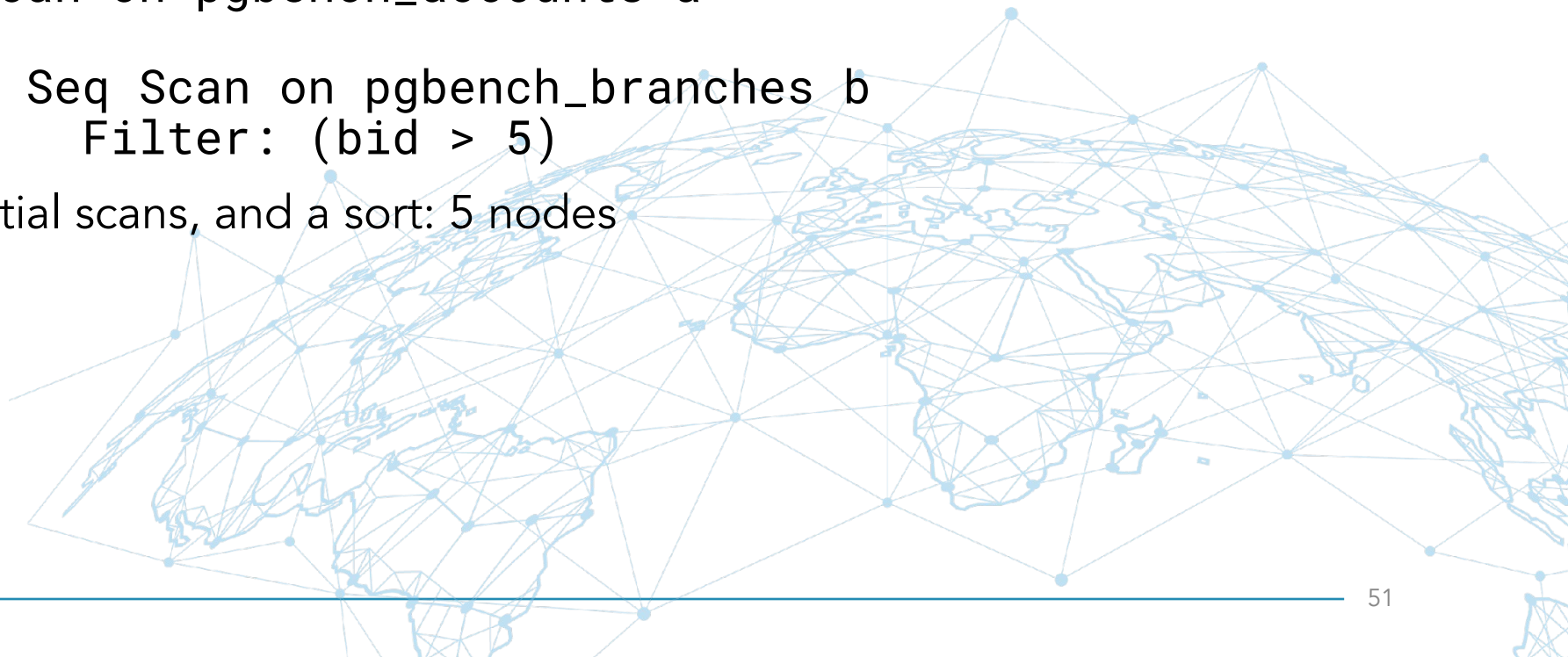
```
SELECT get_query_mem($$
  SELECT a.* FROM pgbench_accounts a
    JOIN pgbench_branches b ON (b.bid = a.bid)
   WHERE b.bid > 5 ORDER BY b.bid;
$$);

 get_query_mem
---------------
         28672
```

Why so High?

This is the query plan:

```
Sort
   Sort Key: a.bid
   ->  Hash Join
         Hash Cond: (a.bid = b.bid)
         ->  Seq Scan on pgbench_accounts a
         ->  Hash
               ->  Seq Scan on pgbench_branches b
                     Filter: (bid > 5)
```

Two hashes, two sequential scans, and a sort: 5 nodes



It Builds up Fast!

```
SELECT get_query_mem($$
  WITH bdata AS MATERIALIZED (SELECT * FROM pgbench_branches WHERE bid > 5)
  SELECT a.*
    FROM pgbench_accounts a
    JOIN bdata b ON (b.bid = a.bid)
   WHERE EXISTS (SELECT * FROM pgbench_tellers t
                  WHERE t.bid = b.bid)
   ORDER BY b.bid;
$$);

 get_query_mem
---------------
         57344
```

What Else Can We Do?

Extra Feature Ideas

1. Log query memory usage
2. Block queries with excessively high estimates

What We'll Need

1. GUC: querymem.log_size
2. GUC: querymem.max_query_size
3. A Pre-execute hook

No Longer Pinocchio

Let's turn this extension

into a *real boy*

Starting with GUCs

- Extensions with GUCs must be in **`shared_preload_libraries`**
- We need a **`_PG_Init`** function
- Check **`utils/guc.h`** for:
  - registration functions
  - signal flags
  - unit definitions
- Lots of examples in the code

The _PG_init Shell

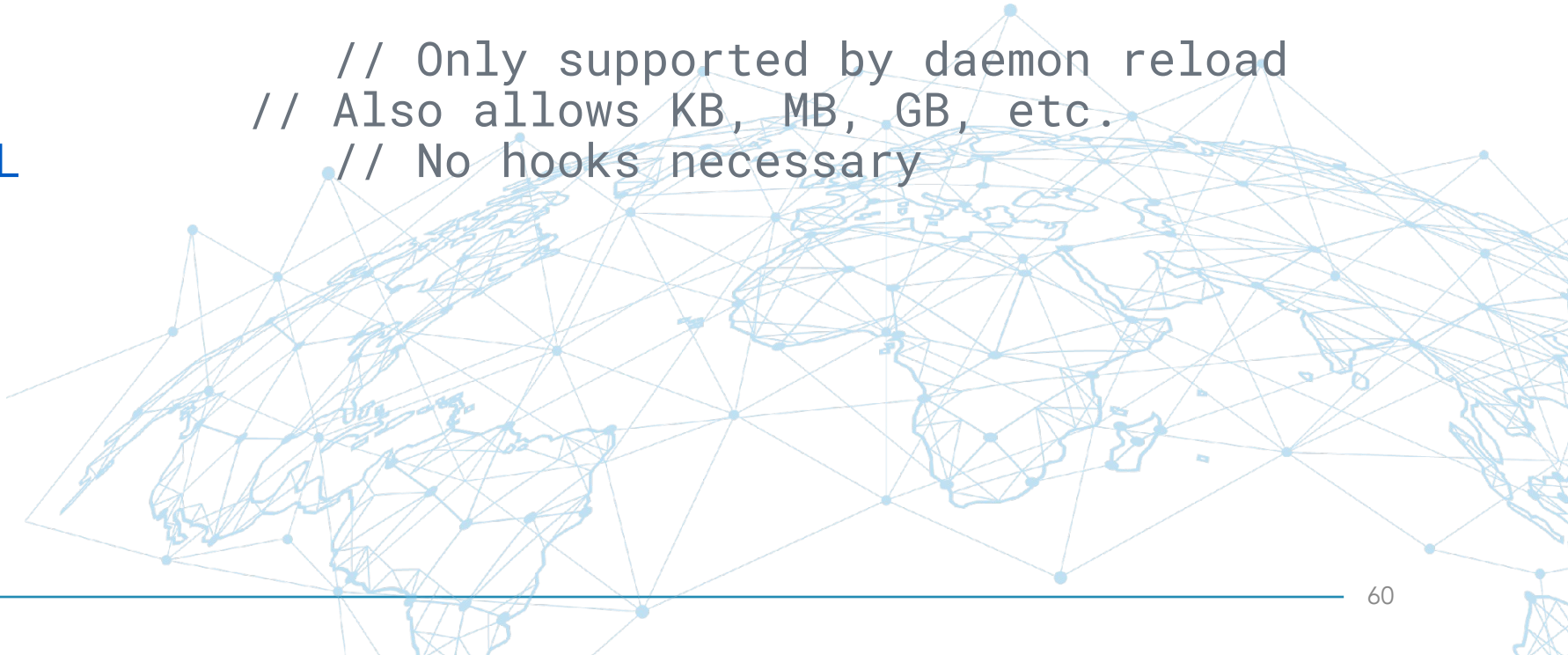
```
// 1. Define GUC global variables and pre-execution hook

void _PG_init(void)
{
    // 2. Define custom GUCs
    // 3. Reserve our extension's GUC prefix
    // 4. Set pre-execution hook

} // _PG_init
```

1. Define GUCs and Exec Hook

```
int querymem_log_size;
int querymem_max_query_size;

static ExecutorStart_hook_type next_ExecutorStart_hook = NULL;
```

2a. Define log_size GUC

```
DefineCustomIntVariable(
    "querymem.log_size",
    "Log any queries exceeding this amount of estimated memory.",
    "Default: 10x work_mem. Set to -1 to disable.",
    &querymem_log_size,
    work_mem * 10, work_mem,// Same limits as work_mem
    MAX_KILOBYTES,
    PGC_SIGHUP,                    // Only supported by daemon reload
    GUC_UNIT_KB,                // Also allows KB, MB, GB, etc.
    NULL, NULL, NULL               // No hooks necessary
);
```



2b. Define max_query_size GUC

```
DefineCustomIntVariable(
    "querymem.max_query_size",
    "Prevent execution of queries over this work_mem limit.",
    "Default: 100x work_mem. Set to -1 to disable.",
    &querymem_max_query_size,
    work_mem * 100, work_mem,  // Same limits as work_mem
    MAX_KILOBYTES,
    PGC_SIGHUP,                // Only supported by daemon reload
    GUC_UNIT_KB,             // Also allows KB, MB, GB, etc.
    NULL, NULL, NULL           // No hooks necessary
);
```

3. Reserve GUC prefix

```
// Prevent other extensions from using our prefix
MarkGUCPrefixReserved("querymem");
```

4. Set pre-execution hook

```
// Preserve the hook set by some other extension
if (ExecutorStart_hook)
    next_ExecutorStart_hook = ExecutorStart_hook;

// Replace it with our own hook
ExecutorStart_hook = querymem_ExecutorStart;
```

Important Hook Note!

Preserving hooks is only *encouraged*

not required!

Once Again For Emphasis!

There is no

`register_executor_hook()`

or similar built-in enforcement!

Need a Helper Estimation Function

```
int32_t estimate_from_plan(PlannedStmt *plan)
{
   ListCell    *lc;
   WalkContext ctx = {0, 0.0};

   walk_plan(plan->planTree, &ctx);

   foreach(lc, plan->subplans)
      walk_plan((Plan *) lfirst(lc), &ctx);

   return work_mem * ctx.multiplier;

} // estimate_from_plan
```

The Executor Hook Shell

```
static void
querymem_ExecutorStart(QueryDesc *queryDesc, int eflags)
{
    // 1. Get a plan estimate
    // 2. Abort any statement over the max query size
    // 3. Log any statement over the log size
    // 4. Call the next Executor hook

} // querymem_ProcessUtility
```

1. Get a plan estimate

```
int32_t estimate = 0;

if (querymem_log_size > -1 || querymem_max_query_size > -1)
    if (!(eflags & EXEC_FLAG_EXPLAIN_ONLY))
        estimate = estimate_from_plan(queryDesc->plannedstmt);
```

2. Abort statements over max query size

```
// The ereport function automatically:
//    - aborts further action.
//    - logs the query for context.
if (querymem_max_query_size > -1 && estimate > querymem_max_query_size)
    ereport(ERROR,
        (errcode(ERRCODE_STATEMENT_TOO_COMPLEX),
         errmsg("This query may use too much work_mem (%dk).", estimate),
         errhint("Increase the querymem.max_query_size GUC to continue."))
    );
```

3. Log statements over the log size

```
// The elog function automatically:
//   - logs the query for context.
if (querymem_log_size > -1 && estimate > querymem_log_size)
   elog(LOG, "Estimated peak memory usage: %d kB", estimate);
```

4. Call the next Executor hook

```
// 1. ALWAYS call the next (or previous) hook!
// 2. Use the “standard” hook if there isn’t one.
// 3. A badly behaving extension can ruin it for EVERYONE!

if (next_ExecutorStart_hook)
    next_ExecutorStart_hook(queryDesc, eflags);
else
    standard_ExecutorStart(queryDesc, eflags);
```

Does it Work?

Setting the Stage

```
-- The defaults are too high to test
ALTER SYSTEM SET querymem.log_size = '8MB';
ALTER SYSTEM SET querymem.max_query_size = '20MB';
SELECT pg_reload_conf();
```

A Logged Query

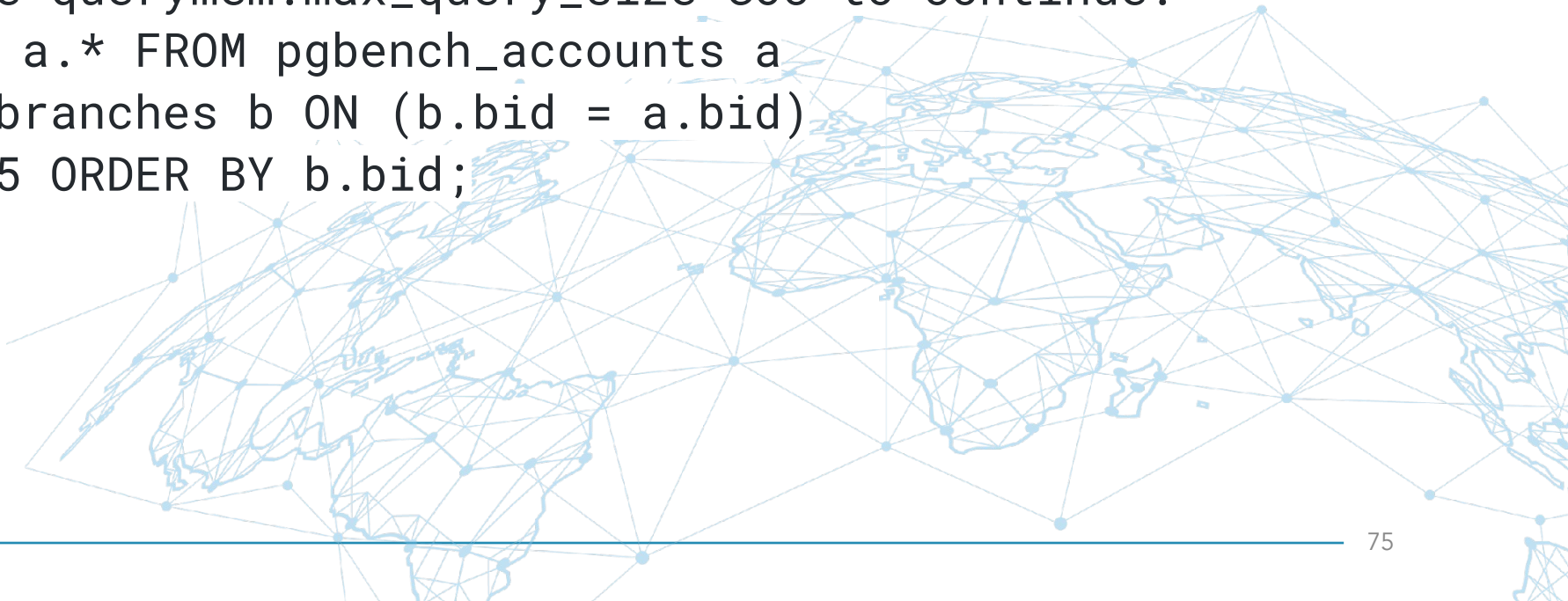
```
SELECT a.* FROM pgbench_accounts a ORDER BY a.bid;
```

```
LOG:  Estimated peak memory usage: 12288 kB
STATEMENT:  SELECT a.* FROM pgbench_accounts a ORDER BY a.bid;
```

A Blocked Query

```
SELECT a.* FROM pgbench_accounts a
  JOIN pgbench_branches b ON (b.bid = a.bid)
 WHERE b.bid > 5 ORDER BY b.bid;
```

```
ERROR:  This query may use too much work_mem (32768k).
HINT:  Increase the querymem.max_query_size GUC to continue.
STATEMENT:  SELECT a.* FROM pgbench_accounts a
     JOIN pgbench_branches b ON (b.bid = a.bid)
    WHERE b.bid > 5 ORDER BY b.bid;
```



Questions?

Grab the Code

github.com/pgEdge/querymem