Writing & using Postgres Extensions

PostgreSQL Conference Europe 2013

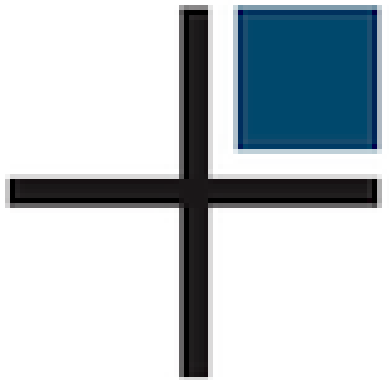
Dimitri Fontaine dimitri@2ndQuadrant.fr

October, 29 2013

Dimitri Fontaine

2ndQuadrant France
PostgreSQL Major Contributor

- `pgloader`, `prefix`, `skytools`, ...
- `apt.postgresql.org`
- `CREATE EXTENSION`
- `CREATE EVENT TRIGGER`
- MySQL migration tool, new `pgloader` version



Writing & using Postgres Extensions

Agenda

- How PostgreSQL extensibility works
- Things you can do with a PostgreSQL Extension
- The PostgreSQL indexing Framework
- **How to solve some practical use cases with existing extensions**
- Developping a new extension

PostgreSQL Extensibility

PostgreSQL is *highly* **extensible**

PostgreSQL Extensibility

```
select col1, col2 from table where col1 = 'something';
```

PostgreSQL Extensibility

```
SELECT col
  FROM table
 WHERE stamped > date 'today' - interval '1 day';
```

PostgreSQL Extensibility

```
select iprange, locid
  from geolite.blocks
 where iprange >>= '91.121.37.122';
```

```
          iprange           | locid
----------------------------+-------
 91.121.0.0-91.121.159.255 |    75
(1 row)

Time: 1.220 ms
```

PostgreSQL Extensibility: Operator Classes

SQL Operators are all dynamic and found in the catalogs

```
select amopopr::regoperator
  from pg_opclass c
  join pg_am am
    on am.oid = c.opcmethod
  join pg_amop amop
    on amop.amopfamily = c.opcfamily
 where opcintype = 'ip4r'::regtype
   and am.amname = 'gist';
```

```
    amopopr
---------------
 >>=(ip4r,ip4r)
 <<=(ip4r,ip4r)
 >>(ip4r,ip4r)
 <<(ip4r,ip4r)
 &&(ip4r,ip4r)
 =(ip4r,ip4r)
(6 rows)
```

PostgreSQL Extensibility

```
  select id, name, pos,
         pos <@> point(-6.25,53.34) as miles
    from pubnames
order by pos <-> point(-6.25,53.34)
   limit 10;
```

PostgreSQL is Extensible

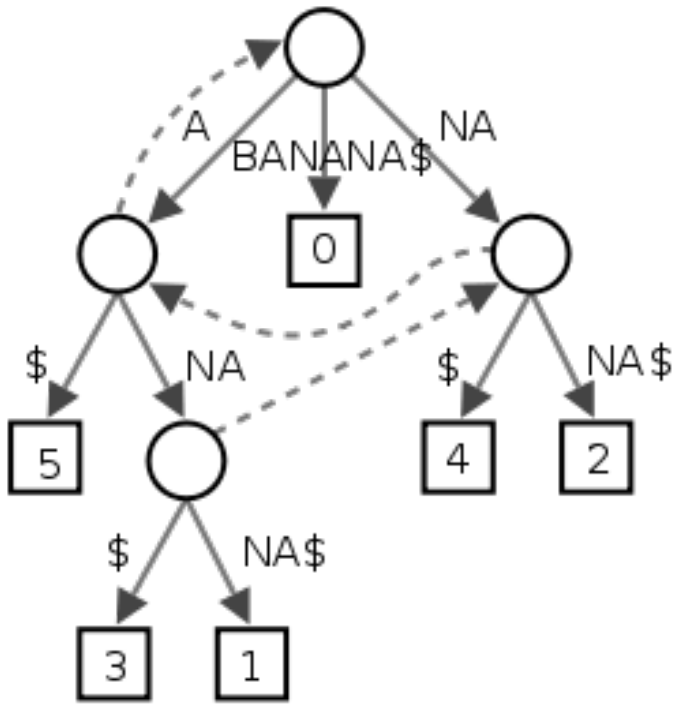
PostgreSQL plugins are data types and index support

- Data Type
- Input/Output functions
- Casts
- Operator Classes

PostgreSQL is Extensible

PostgreSQL support several kind of indexes

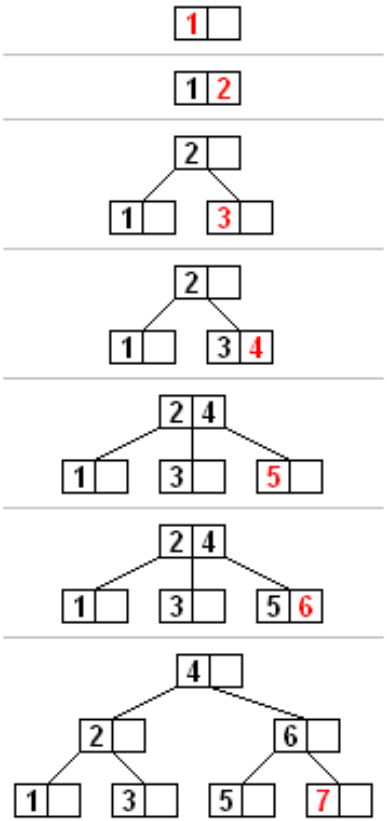
- BTree, binary tree
- GiST, Generalized Search Tree
- SP-GiST, Space Partitioned GiST
- GIN, Generalized Inverted Index



Binary Tree

Btree, the default index type

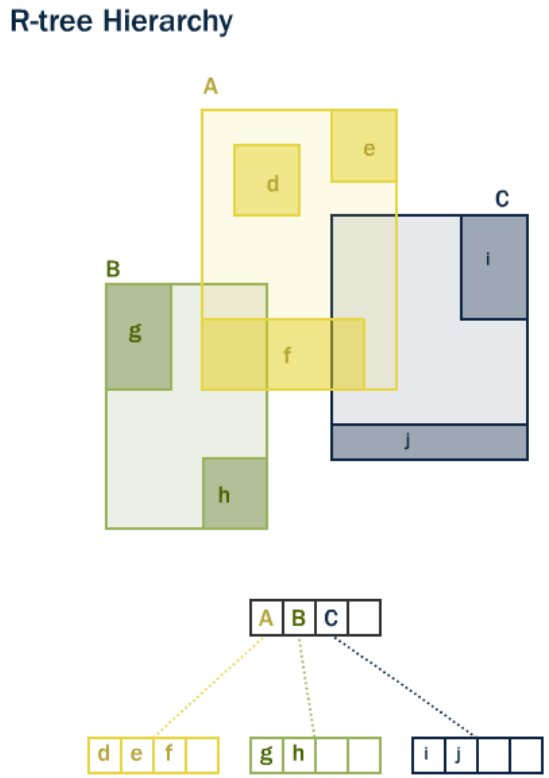
- Built for speed
- *unique* concurrency tricks
- Balanced
- support function: `cmp`
- operators: `<=` `<` `=` `>` `>=`



Generalized Index Search Tree

GiST or the Indexing API

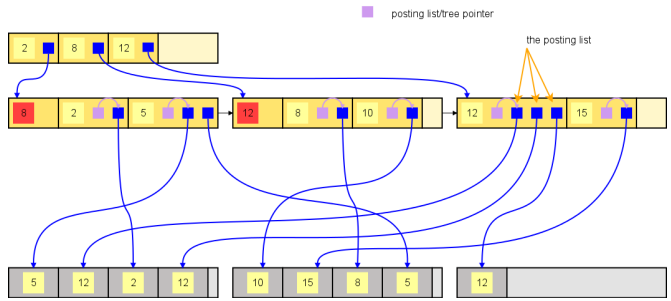
- Built for comfort
- Balanced
- API: `consistent`, `same`, `union`
- API: `penalty`, `picksplit`
- API: `compress`, `decompress`
- operators: `@>` `<@` `&&` `@@` `=` `&<` `&>` `<<|` ...



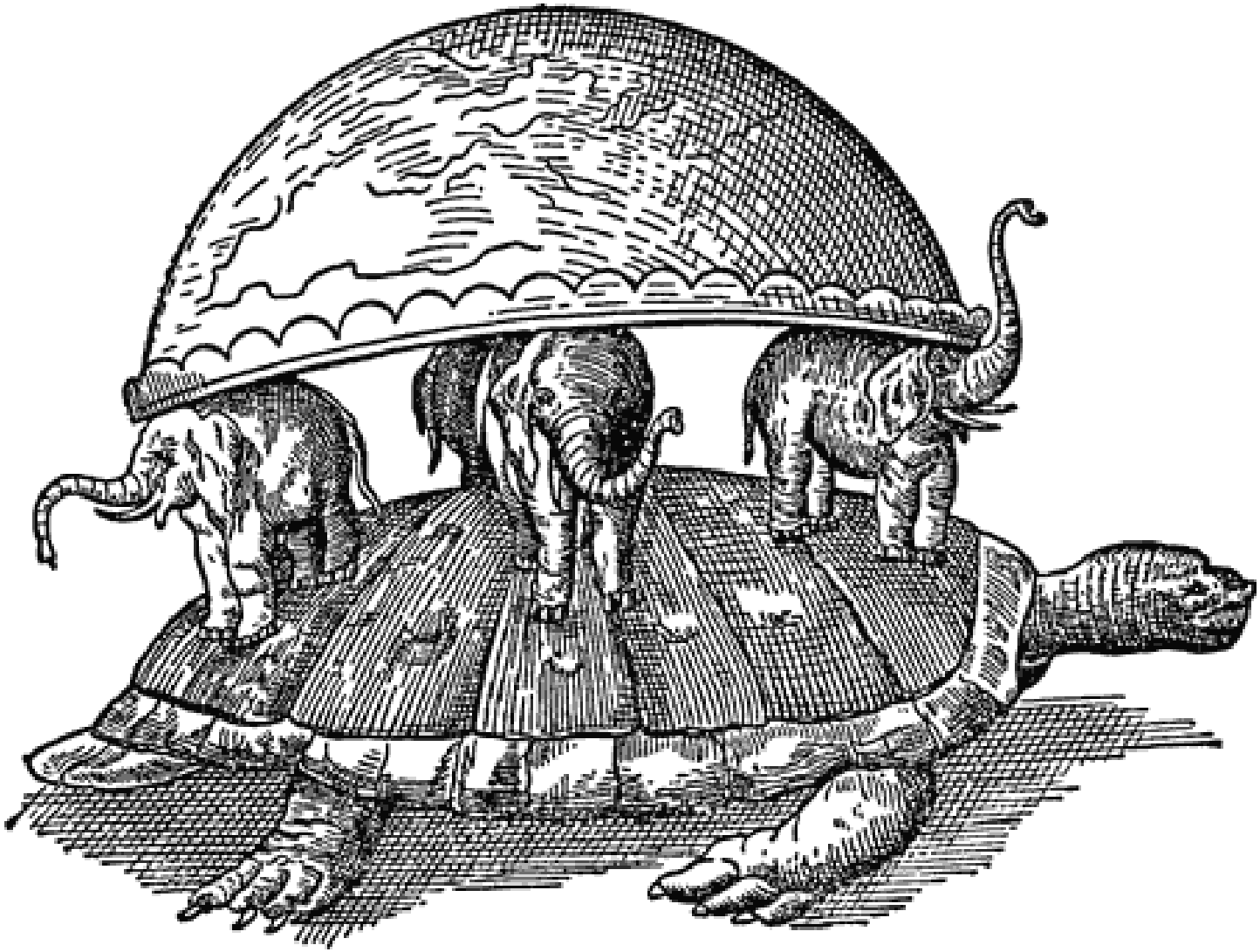
Generalized Inverted iNdex

Indexing several pointers per value, inversed cardinality

- Built for Text Search and Arrays
- Balanced
- API: `compare`, `consistent`
- API: `extractValue`, `extractQuery`
- operators: `@>` `<@` `&&` `=`



Extensions and data types

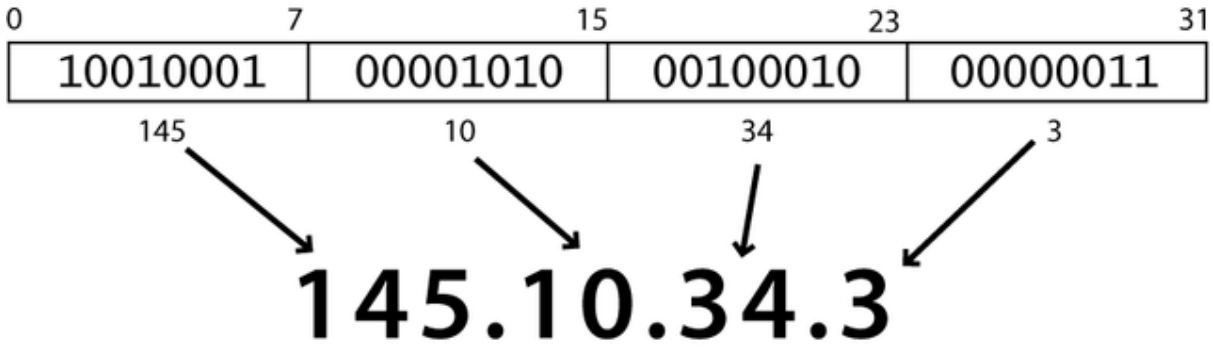


Some extensions example

46 Contribs, Community extensions, Private ones...

- hll
- cube
- ltree
- citext
- hstore
- earthdistance
- pgq
- pg_trgm
- wildspeed
- plproxy
- PostGIS
- ip4r
- intarray
- prefix
- pgfincore
- pgcrypto
- pg_stattuple
- pg_buffercache
- pg_stat_statements
- pgfincore

IP Ranges, ip4r



IP Ranges, ip4r

```
table geolite.blocks limit 10;
        iprange          | locid
-------------------------+-------
 1.0.0.0/24              |    17
 1.0.1.0-1.0.3.255       |    49
 1.0.4.0/23              | 14409
 1.0.6.0/23              |    17
 1.0.8.0/21              |    49
 1.0.16.0/20             | 14614
 1.0.32.0/19             | 47667
 1.0.64.0/18             |   111
 1.0.128.0-1.0.147.255   |   209
 1.0.148.0/24            | 22537
(10 rows)
```

IP Ranges, ip4r, Geolocation

PostgreSQL allows using SQL and JOINs to match IP4R with geolocation.

```
select *
  from geolite.blocks
  join geolite.location
       using(locid)
 where iprange
           >>=
       '74.125.195.147';
```

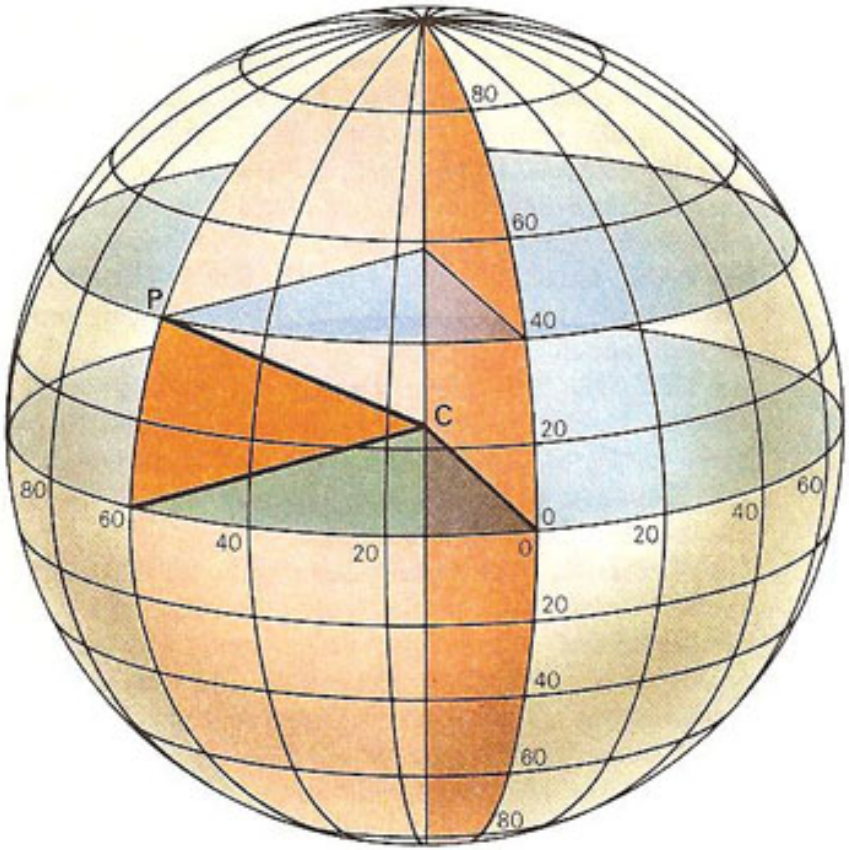
IP Ranges, ip4r, Geolocation

PostgreSQL allows using SQL and JOINs to match IP4R with geolocation.

```
select *
  from geolite.blocks
  join geolite.location
       using(locid)
 where iprange
           >>=
       '74.125.195.147';
```

```
-[ RECORD 1 ]----------------------
locid      | 2703
iprange    | 74.125.189.24-74.125.
country    | US
region     | CA
city       | Mountain View
postalcode | 94043
location   | (-122.0574,37.4192)
metrocode  | 807
areacode   | 650

Time: 1.335 ms
```

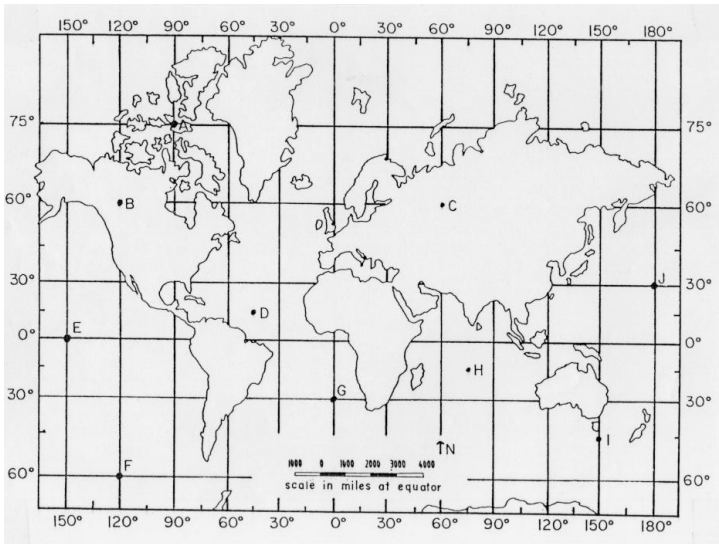
Earth Distance



How Far is The Nearest Pub

The `point` datatype is in-core

```
#   CREATE TABLE pubnames
    (
      id    bigint,
      pos   POINT,
      name text
    );
```



How Far is The Nearest Pub

```
  select name, pos
    from pubnames
order by pos <-> point(-6.25,53.346)
    limit 3;
```

```
          Pub Name             |            pos
-------------------------------+---------------------------
 Ned's                         | (-6.2519967,53.3458267)
 Sub Lounge                    | (-6.2542332,53.3469085)
 O'Neill's of Pearse Street    | (-6.2524389,53.3448589)
(3 rows)

Time: 18.679 ms
```

How Far is The Nearest Pub

```
CREATE INDEX on pubnames USING GIST(pos);
```

```
  select name,
         pos
    from pubnames
order by pos <-> point(-0.12,51)
   limit 3;
```

```
  name   |          pos
---------+------------------
 Ned's   | (-6.25,53.34
 Sub Lo  | (-6.25,53.34
 O'Neil  | (-6.25,53.34
(3 rows)

Time: 0.849 ms
```

How Far is The Nearest Pub, in Miles please.

```
create extension cube;
create extension earthdistance;
```

```
  select name,
   pos <@> point(-6.25,53.34) miles
    from pubnames
order by pos <-> point(-6.25,53.34)
   limit 3;
```

```
  name   | miles
---------+-------
 Ned's   |  0.06
 Sub Lo  |  0.07
 O'Neil  |  0.12
(3 rows)

Time: 1.335 ms
```

Some pubs far away from here...

```
select c.name as city,
pos <@> point(-6.25,53.34) as miles
 from pubnames p,
   lateral (select name
              from cities c
          order by c.pos <-> p.pos
             limit 1) c
order by pos <-> point(-6.25,53.34)
         desc
  limit 5;
```

```
    city    |  miles
------------+--------
 Canterbury | 399.44
 Canterbury | 378.91
 Canterbury | 392.08
 Canterbury | 397.30
 Canterbury | 379.68
(5 rows)

Time: 636.445 ms
```

Geolocation: ip4r meets earthdistance

Some pubs nearby... some place...

```
with geoloc as
 (
  select location as l
    from location
    join blocks using(locid)
   where iprange
         >>=
         '212.58.251.195'
 )
  select name,
         pos <@> l miles
    from pubnames, geoloc
order by pos <-> l
   limit 10;
```

```
          name          | miles
------------------------+-------
 Blue Anchor            | 0.299
 Dukes Head             | 0.360
 Blue Ball              | 0.337
 Bell (aka The Rat)     | 0.481
 on the Green           | 0.602
 Fox & Hounds           | 0.549
 Chequers               | 0.712
 Sportsman              | 1.377
 Kingswood Arms         | 1.205
 Tattenham Corner       | 2.007
(10 rows)

Time: 3.275 ms
```

Trigrams

Trigrams and similarity

similar but not quite *like* the same

```
create extension pg_trgm;

select show_trgm('tomy') as tomy,
       show_trgm('Tomy') as "Tomy",
       show_trgm('tom torn') as "tom torn",
       similarity('tomy', 'tom'),
       similarity('dim', 'tom');
```

```
-[ RECORD 1 ]-------------------------------
tomy       | {"  t"," to","my ",omy,tom}
Tomy       | {"  t"," to","my ",omy,tom}
tom torn   | {"  t"," to","om ",orn,"rn ",tom,tor}
similarity | 0.5
similarity | 0
```

Trigrams and typos

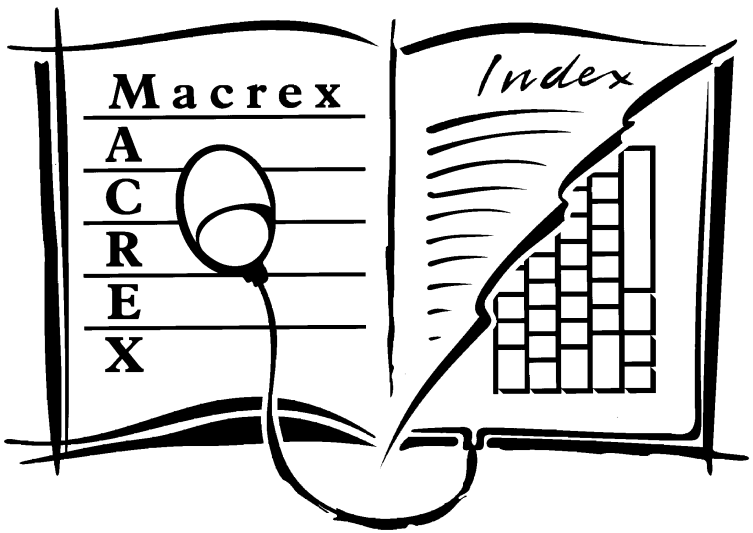
Use your data to help your users out

```
select actor
  from products
 where actor ~* 'tomy';
  actor
----------


(0 rows)

Time: <unregistered>
```

```
select actor
  from products
 where actor % 'tomy';
  actor
----------
 TOM TORN
 TOM DAY
(2 rows)

Time: 26.972 ms
```

Trigrams search indexing

```
create index on products using gist(actor gist_trgm_ops);
```




```
select actor
  from products
 where actor % 'tomy';
  actor
----------
 TOM TORN
 TOM DAY
(2 rows)

Time: 2.695 ms
```

Trigrams and autocompletion

Use your data to help your users out

```
explain (costs off)
 select * from products where actor ~* 'tomy';
                    QUERY PLAN
-----------------------------------------------
 Index Scan using products_actor_idx on products
   Index Cond: ((actor)::text ~* 'tomy'::text)
(2 rows)
```

Trigrams and autocompletion

Use your data to help your users out

```
  select actor
    from products
   where actor % 'fran'
order by actor <-> 'fran'
   limit 10;
```

```
     actor
--------------
 FRANK HAWKE
 FRANK BERRY
 FRANK POSEY
 FRANK HAWKE
 FRANCES DEE
 FRANK LEIGH
 FRANCES DAY
 FRANK FOSTER
 FRANK HORNE
 FRANK TOMEI
(10 rows)

Time: 2.960 ms
```

Advanced Array Indexing with `intarray`

Last.fm allows users to tag tracks

```
  select t.tag,
         count(tt.tid) n
    from tid_tag tt
    join tags t
      on tt.tag = t.rowid
   where t.tag ~* 'setzer'
group by t.tag;
```

```
                tag                |  n
-----------------------------------+----
 the brian setzer orchestra        |  1
 setzer                            | 13
 rockabilly setzer style           |  4
 setzer is a true guitarhero       |  9
 brian setzer orchestra            |  3
 brian setzer is god               |  1
 brian setzer                      |  1
 brain setzer orchestra            |  2
(8 rows)

time: 644.826 ms
```

Last.fm allows users to tag tracks

```
create extension intarray;
```

```
  select tt.tid,
         array_agg(tags.rowid) tags
    from tags
         join tid_tag tt
           on tags.rowid = tt.tag
group by tt.tid
   limit 3;
```

```
 tid |    tags
-----+------------
   1 | {1,2}
   2 | {3,4}
   3 | {5,6,7,8}
(3 rows)

time: 942.074 ms
```

Prepare for intarray indexing

Denormalize the data set thanks to PostgreSQL Arrays

```
create table track_tags as
   select tt.tid, array_agg(tags.rowid) as tags
     from tags join tid_tag tt on tags.rowid = tt.tag
 group by tt.tid;

create index on track_tags using gin(tags gin__int_ops);
```

Search for several tags at once

Intersection of multiple criteria

```
select array_agg(rowid)
  from tags
 where tag = 'blues'
    or tag = 'rhythm and blues';
```

```
 array_agg
-----------
 {3,739}
(1 row)

time: 0.684 ms
```

The query_int data type

intarray has powerful indexing and searching facilities

```
select format('(%s)',
         array_to_string(
          array_agg(rowid), '&')
       )::query_int as query
  from tags
 where tag = 'blues'
    or tag = 'rhythm and blues';
```

```
  query
---------
 3 & 739
(1 row)

time: 0.747 ms
```

Putting it all together

```
with t(query) as (
 select format('(%s)',
         array_to_string(
          array_agg(rowid), '&')
        )::query_int as query
   from tags
  where tag = 'blues'
     or tag = 'rhythm and blues'
)
 select track.tid
   from track_tags tt
   join tids track
     on tt.tid = track.rowid, t
  where tt.tags @@ t.query
  limit 10;
```

```
                tid
--------------------
 TRCJLCC12903CBF4AE
 TRCIFOV128F92F6F4C
 TRCYUVJ128F425C8F1
 TRCNTFO128F92F6564
 TRCDRGT12903CE64BF
 TRCWAED128F42A837B
 TRCWFEM128F9320F94
 TRCQCQH128F932E707
 TRCUMTA12903CD67EE
 TRJJYUT12903CFB13B
(10 rows)

Time: 7.630 ms
```

HStore

HStore: NoSQL meets PostgreSQL

- Key-Value Store
- Denormlisation
- Indexing (GiST, GIN)
- Operators
- SQL

HStore: NoSQL meets PostgreSQL

```
create table preferences
(
  email        text      primary key,
  language     text,
  timezone     text,
  properies  hstore
);
```

HStore: NoSQL meets PostgreSQL

```
INSERT INTO preferences
VALUES
  ('dimitri@2ndQuadrant.fr', 'fr_FR', 'Europe/Paris',
   'skills => PostgreSQL,
    Extensions => "prefix, base64, pgextwlist, preprepare",
    Software => "pgloader"'),

  ('simon@2ndQuadrant.com', 'en_UK', 'Europe/London',
   'skills => "PostgreSQL, Replication",
    Software => "PostgreSQL, repmgr, pg_standby"');

INSERT 0 2
```

HStore: NoSQL meets PostgreSQL

```
~# select * from preferences;
-[ RECORD 1 ]----------------------------------------
email      | dimitri@2ndQuadrant.fr
language   | fr_FR
timezone   | Europe/Paris
properies  | "skills"=>"PostgreSQL",
           "Software"=>"pgloader",
           "Extensions"=>"prefix, base64, pgextwlist, prepre
-[ RECORD 2 ]----------------------------------------
email      | simon@2ndQuadrant.com
language   | en_UK
timezone   | Europe/London
properies  | "skills"=>"PostgreSQL, Replication",
           "Software"=>"PostgreSQL, repmgr, pg_standby"
```

HStore: NoSQL meets PostgreSQL

```
~# select email
     from preferences
    where properies ? 'Extensions';
          email
------------------------
 dimitri@2ndQuadrant.fr
(1 row)
```

HStore: NoSQL meets PostgreSQL

```
~# select email
     from preferences
    where (properies -> 'skills') ~ 'PostgreSQL';
          email
--------------------------
 dimitri@2ndQuadrant.fr
 simon@2ndQuadrant.com
(2 rows)
```

HStore and Parametrized Triggers

Compute *duration* in a before trigger

We need a table and some data

```
create table foo
(
  id       serial      primary key,
  d_start  timestamptz default now(),
  d_end    timestamptz,
  duration interval
);

insert into foo(d_start, d_end)
     select now() - 10 * random() * interval '1 min',
            now() + 10 * random() * interval '1 min'
       from generate_series(1, 10);
```

Populating an hstore from a record

Filling in a column when the name is a parameter

```
select (foo #= hstore('duration', '10 mins')).*
  from foo
  limit 1;

-[ RECORD 1 ]----------------------------
id       | 1
d_start  | 2013-10-24 18:57:36.725212+02
d_end    | 2013-10-24 19:04:56.171473+02
duration | 00:10:00
```

The hstore based *parametrized* trigger

```
create or replace function tg_duration()
 returns trigger
 language plpgsql
as '
declare
   hash hstore := hstore(NEW);
   duration interval;
begin
   duration :=  (hash -> TG_ARGV[1])::timestamptz
              - (hash -> TG_ARGV[0])::timestamptz;

   NEW := NEW #= hstore(TG_ARGV[2], duration::text);

   RETURN NEW;
end;
';
```

Installing the trigger

To be able to modify what's inserted we need a `before` trigger

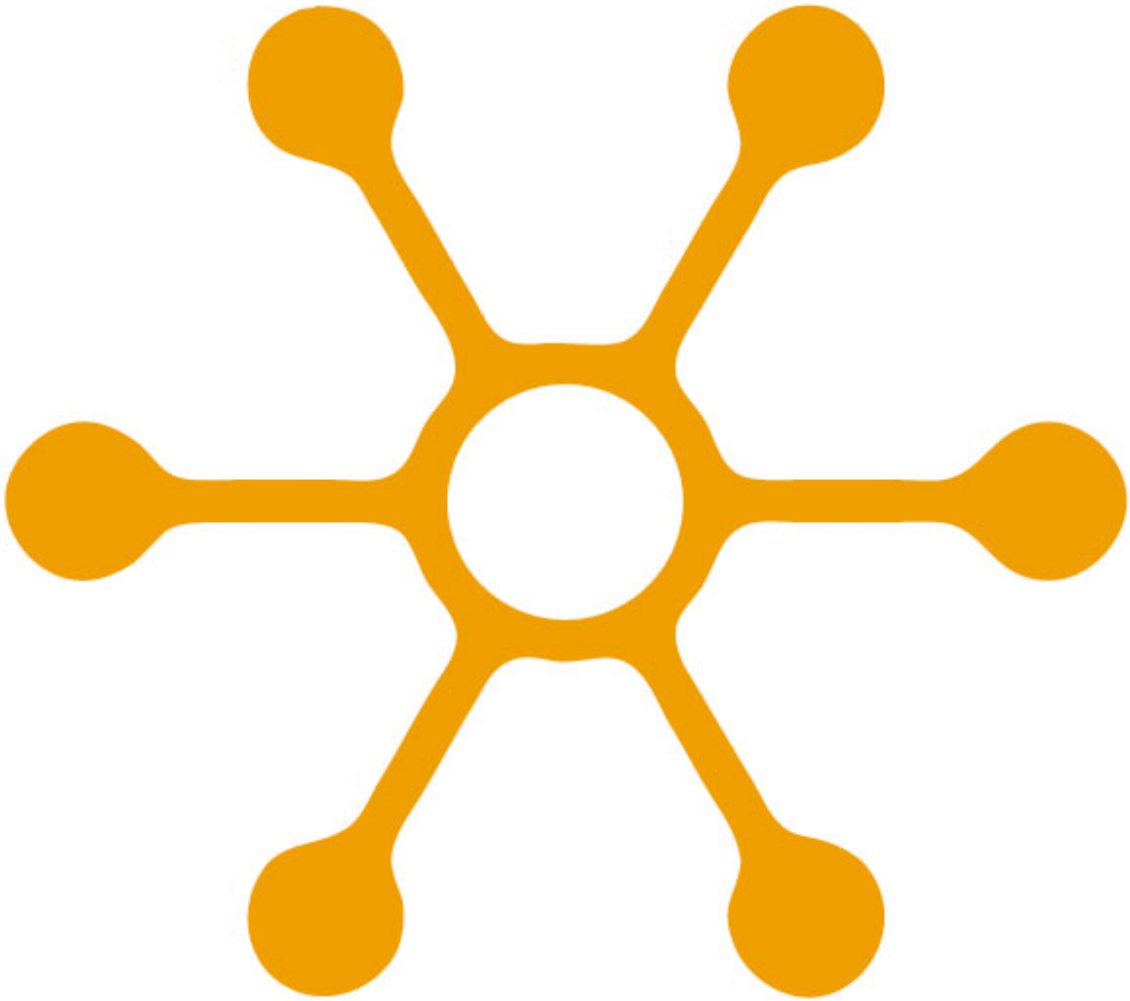
```
create trigger compute_duration
 before insert on foo
      for each row
execute procedure tg_duration('d_start',
                              'd_end',
                              'duration');
```

Using the trigger: watch the duration

```
truncate foo;
insert into foo(d_start, d_end)
     select now()
          - 10 * random()
               * interval '1 min',
            now()
          + 10 * random()
            * interval '1 min'
       from generate_series(1, 10);
```

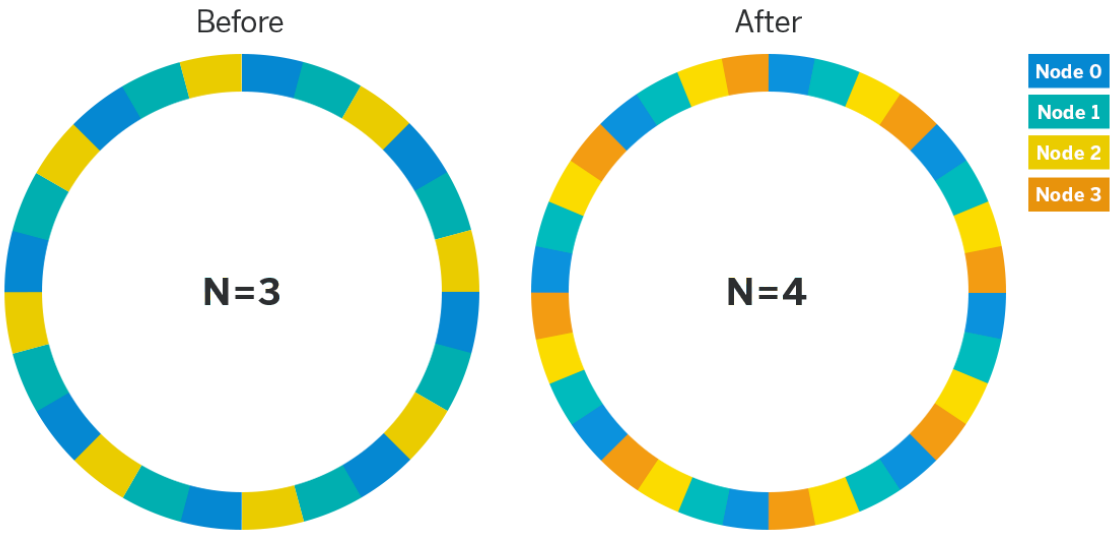
```
select duration
  from foo;
    duration
-----------------
 00:03:48.003135
 00:10:57.727407
 00:01:13.637183
 00:10:33.820578
 00:13:11.607287
 00:04:41.224213
 00:08:26.842229
 00:12:16.630843
 00:09:51.418547
 00:08:52.968195
(10 rows)
```

PL/Proxy



PL/Proxy

PL/Proxy is all about *Sharding*



We're going to use it for *Remote Procedure Call*

Classic Auditing

```
create table example
 (
   id serial,
   f1 text,
   f2 text
 );
```

```
create table audit
 (
  change_date timestamptz
              default now(),
  before hstore,
  after  hstore
 );
```

Classic trigger based Auditing

```
~# begin;
~*# update example set f1 = 'b' where id = 1;
~*# rollback;

~# select * from audit;
 change_date | before | after
-------------+--------+-------
(0 rows)
```

Seting up PL/Proxy

```
create extension plproxy;

       create server local
foreign data wrapper plproxy
             options (p0 'dbname=dim');

create user mapping
                for public
              server local
             options (user 'dim');
```

PL/Proxy: Basic Testing

```
create function test_proxy
           (i int)
   returns int
  language plproxy
as '
  cluster ''local'';
  select i;
';
```

```
select test_proxy(1);
 test_proxy
------------
          1
(1 row)

Time: 0.866 ms
```

Implementing Autonomous Transactions for Auditing

Trigger Functions 1/3: the trigger

```
create function audit_trigger()
  returns trigger
  language plpgsql
as '
begin
  perform audit_proxy(old, new);
  return new;
end;
';
```

Trigger Functions 2/3: the proxy

```
create function audit_proxy
 (
  old example,
  new example
 )
  returns void
  language plproxy
as '
  cluster ''local'';
  target audit;
';
```

Trigger Functions 3/3: the implementation

```
create function audit
 (
  old example,
  new example
 )
  returns void
  language SQL
as '
  INSERT INTO audit(before, after)
      SELECT hstore(old), hstore(new);
';
```

Trigger Definition

```
drop trigger if exists audit on example;

create trigger audit
  after update on example
      for each row
      -- careful, defaults to FOR EACH STATEMENT!
execute procedure audit_trigger();
```

Autonomous Auditing Transaction

```
~# begin;
BEGIN

~*# update example set f1 = 'b' where id = 1;
UPDATE 1

~*# rollback;
ROLLBACK
```

Autonomous Auditing Tranasction

We did `ROLLBACK;` the transaction

```
~# select change_date,
          before,
          after,
          after - before as diff
     from audit;
-[ RECORD 1 ]-------------------------------
change_date | 2013-10-14 14:29:09.685105+02
before      | "f1"=>"a", "f2"=>"a", "id"=>"1"
after       | "f1"=>"b", "f2"=>"a", "id"=>"1"
diff        | "f1"=>"b"
```

HyperLogLog

State of The Art Cardinality Estimation Algorithm

Creating the unique visitors tracking table

```
CREATE EXTENSION hll;

-- Create the destination table
CREATE TABLE daily_uniques (
    DATE            DATE UNIQUE,
    users           hll
);

-- Our first aggregate update
UPDATE daily_uniques
   SET users = hll_add(users,
                       hll_hash_text('123.123.123.123'))
 WHERE date = current_date;
```

Production ready updates

```
--
-- First upload a new batch, e.g. using
--    CREATE TEMP TABLE new_batch as VALUES(), (), ...;
--
WITH hll(agg) AS (
 SELECT hll_add_agg(hll_hash_text(value))
   FROM new_batch
)
 UPDATE daily_uniques
    SET users = CASE WHEN hll.agg IS NULL THEN users
                     ELSE hll_union(users, hll.agg)
                END
   FROM hll
  WHERE date = current_date;
```

Daily Reporting

```
with stats as (
  select date,
         #users as daily,

         #hll_union_agg(users)
         over() as total

    from daily_uniques
)
  select date,
         daily,
         daily/total*100
    from stats
order by date;
```

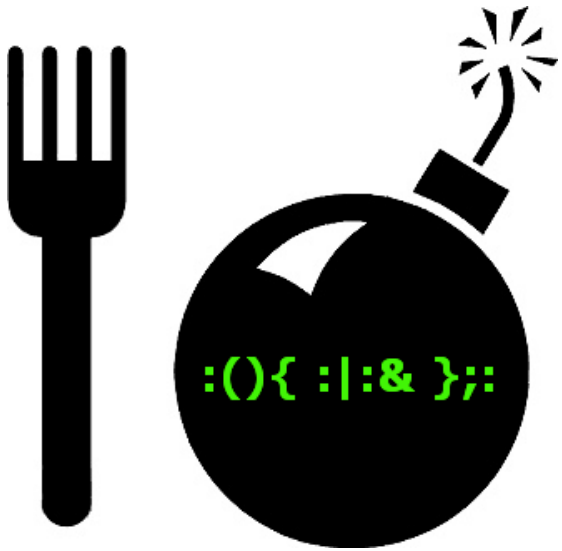
```
    date    | daily  | percent
------------+--------+---------
 2013-02-22 | 401677 |   25.19
 2013-02-23 | 660187 |   41.41
 2013-02-24 | 869980 |   54.56
 2013-02-25 | 154996 |    9.72
(4 rows)
```

Monthly Reporting

```
  select to_char(date, 'YYYY/MM'),
         #hll_union_agg(users)
    from daily_uniques
group by 1;
```

```
  month   | monthly
----------+---------
 2013/02 | 1960380
(1 row)
```

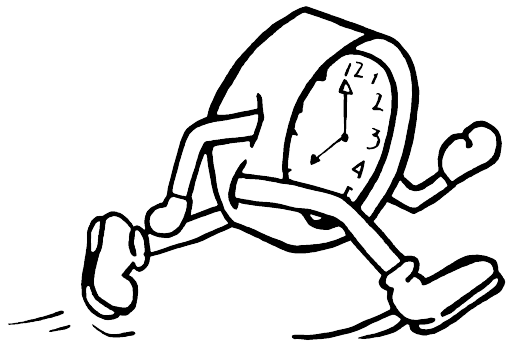
New in 9.3: Background Workers



New in 9.3: Background Workers

Start autonomous user processes within the database server

- Job Scheduler (autovacuum like maintainance)
- PGQ Ticker
- Replication Tasks
- Parallel Queries



Background Workers C API

```
void _PG_init(void)
{
  BackgroundWorker worker;

  worker.bgw_flags = BGWORKER_SHMEM_ACCESS |
       BGWORKER_BACKEND_DATABASE_CONNECTION;
  worker.bgw_start_time    = BgWorkerStart_RecoveryFinished;
  worker.bgw_main          = worker_spi_main;
  worker.bgw_sighup        = worker_spi_sighup;
  worker.bgw_sigterm       = worker_spi_sigterm;
  worker.bgw_name          = "count relations";
  worker.bgw_restart_time = BGW_NEVER_RESTART;
  worker.bgw_main_arg      = NULL;
  RegisterBackgroundWorker(&worker);

  BackgroundWorkerInitializeConnection("dbname", "username");
}
```

Background Workers C API

`bgw_start_time`

- `BgWorkerStart_PostmasterStart`
- `BgWorkerStart_ConsistentState`
- `BgWorkerStart_RecoveryFinished`

Background Workers and SPI

```
StartTransactionCommand();
SPI_connect();
PushActiveSnapshot(GetTransactionSnapshot());

/* build query, might be static string */

SPI_execute(query, true/false /* read only */, 0);

/* process query results */

SPI_finish();
PopActiveSnapshot();
CommitTransactionCommand();
```

New in 9.3: Background Workers

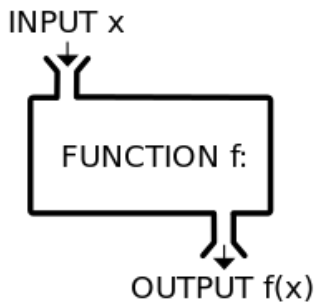
Can request *shared memory*, server must be restarted.

```
shared_preload_libraries = 'count_relations'
```

Extensions: Let's make a new one!

A new integer data type: base36

Internally store `bigint`, reuse *internals*



Only code the *input/output* functions, in `C`

A new integer data type: base36

```
create extension base36;
```

i	x	i	x	i	x
0	0	10000	7PS	100000000	1NJCHS
1	1	10001	7PT	100000001	1NJCHT
2	2	10002	7PU	100000002	1NJCHU
3	3	10003	7PV	100000003	1NJCHV
4	4	10004	7PW	100000004	1NJCHW
5	5	10005	7PX	100000005	1NJCHX
6	6	10006	7PY	100000006	1NJCHY
7	7	10007	7PZ	100000007	1NJCHZ
8	8	10008	7Q0	100000008	1NJCI0
9	9	10009	7Q1	100000009	1NJCI1
10	A	10010	7Q2	100000010	1NJCI2

Input Output Functions

```
CREATE OR REPLACE FUNCTION base36_in(cstring)
RETURNS base36
AS '$libdir/base36'
LANGUAGE C IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION base36_out(base36)
RETURNS cstring
AS '$libdir/base36'
LANGUAGE C IMMUTABLE STRICT;
```

Input Output Functions

```
CREATE OR REPLACE FUNCTION base36_recv(internal)
RETURNS base36
AS '$libdir/base36'
LANGUAGE C IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION base36_send(base36)
RETURNS bytea
AS '$libdir/base36'
LANGUAGE C IMMUTABLE STRICT;
```

Some C code

```
#include "postgres.h"

#ifndef PG_VERSION_NUM
#error "Unsupported too old PostgreSQL version"
#endif

#if  PG_VERSION_NUM / 100 != 903 \
  && PG_VERSION_NUM / 100 != 904
#error "Unknown or unsupported PostgreSQL version"
#endif

PG_MODULE_MAGIC;
```

Some C code

```c
static inline
base36 base36_from_str(const char *str)
{
  /* ... C code here ... */
}

static inline
char *base36_to_str(base36 c)
{
  /* ... C code here ... */
}
```

Interfacing with PostgreSQL

```
Datum base36_in(PG_FUNCTION_ARGS);
Datum base36_out(PG_FUNCTION_ARGS);
Datum base36_recv(PG_FUNCTION_ARGS);
Datum base36_send(PG_FUNCTION_ARGS);
Datum base36_cast_to_text(PG_FUNCTION_ARGS);
Datum base36_cast_from_text(PG_FUNCTION_ARGS);
Datum base36_cast_to_bigint(PG_FUNCTION_ARGS);
Datum base36_cast_from_bigint(PG_FUNCTION_ARGS);
```

Interfacing with PostgreSQL

```
PG_FUNCTION_INFO_V1(base36_in);
Datum
base36_in(PG_FUNCTION_ARGS)
{
    char *str = PG_GETARG_CSTRING(0);
    PG_RETURN_INT64(base36_from_str(str));
}

PG_FUNCTION_INFO_V1(base36_out);
Datum
base36_out(PG_FUNCTION_ARGS)
{
  base36 c = PG_GETARG_INT64(0);
  PG_RETURN_CSTRING(base36_to_str(c));
}
```

CREATE TYPE

```
CREATE TYPE base36 (
        INPUT           = base36_in,
        OUTPUT          = base36_out,
        RECEIVE         = base36_recv,
        SEND            = base36_send,
        LIKE            = bigint,
        CATEGORY        = 'N'
);
COMMENT ON TYPE base36
     IS 'bigint written in base36: [0-9A-Z]+';
```

A minimum amount of CAST

```
CREATE FUNCTION text(base36)
RETURNS text
AS '$libdir/base36',
   'base36_cast_to_text'
LANGUAGE C IMMUTABLE STRICT;

CREATE CAST (text as base36)
  WITH FUNCTION base36(text)
    AS IMPLICIT;
```

```
CREATE CAST (base36 as text)
  WITH FUNCTION text(base36);

CREATE CAST (bigint as base36)
    WITHOUT FUNCTION
         AS IMPLICIT;
CREATE CAST (base36 as bigint)
    WITHOUT FUNCTION
         AS IMPLICIT;
```

Reuse *internals*: comparison functions

```
CREATE OR REPLACE FUNCTION base36_eq(base36, base36)
RETURNS boolean LANGUAGE internal IMMUTABLE AS 'int8eq';

CREATE OR REPLACE FUNCTION base36_ne(base36, base36)
RETURNS boolean LANGUAGE internal IMMUTABLE AS 'int8ne';

CREATE OR REPLACE FUNCTION base36_lt(base36, base36)
RETURNS boolean LANGUAGE internal IMMUTABLE AS 'int8lt';

CREATE OR REPLACE FUNCTION base36_le(base36, base36)
RETURNS boolean LANGUAGE internal IMMUTABLE AS 'int8le';
```

Reuse *internals*: comparison functions

```
CREATE OR REPLACE FUNCTION base36_gt(base36, base36)
RETURNS boolean LANGUAGE internal IMMUTABLE AS 'int8gt';

CREATE OR REPLACE FUNCTION base36_ge(base36, base36)
RETURNS boolean LANGUAGE internal IMMUTABLE AS 'int8ge';

CREATE OR REPLACE FUNCTION base36_cmp(base36, base36)
RETURNS integer LANGUAGE internal IMMUTABLE AS 'btint8cmp';
```

Register operators

```
CREATE OPERATOR = (
          LEFTARG = base36,
         RIGHTARG = base36,
        PROCEDURE = base36_eq,
       COMMUTATOR = '=',
           NEGATOR = '<>',
         RESTRICT = eqsel,
             JOIN = eqjoinsel
);
COMMENT ON OPERATOR =(base36, base36) IS 'equals?';
```

Add in btree index support, stolen from bigint

```
CREATE OPERATOR CLASS btree_base36_ops
DEFAULT FOR TYPE base36 USING btree
AS
        OPERATOR        1       <  ,
        OPERATOR        2       <= ,
        OPERATOR        3       =  ,
        OPERATOR        4       >= ,
        OPERATOR        5       >  ,
        FUNCTION        1       base36_cmp(base36, base36);
```

Packaging an extension

Packaging an extension

We need a Makefile

```
EXTENSION    = base36
MODULES      = base36
DATA         = base36--1.0.sql base36.control

LDFLAGS=-lrt

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

Installing an extension

```
$ make install
/usr/local/bin/ccache /usr/bin/gcc -O2 -Wall -Wmissing-prototy
/usr/local/bin/ccache /usr/bin/gcc -O2 -Wall -Wmissing-prototy
/bin/sh /Users/dim/pgsql/ddl/lib/pgxs/src/makefiles/../../conf
/bin/sh /Users/dim/pgsql/ddl/lib/pgxs/src/makefiles/../../conf
/bin/sh /Users/dim/pgsql/ddl/lib/pgxs/src/makefiles/../../conf
/usr/bin/install -c -m 644 base36.control '/Users/dim/pgsql/do
/usr/bin/install -c -m 644 base36--1.0.sql base36.control '/Us
/usr/bin/install -c -m 755  base36.so '/Users/dim/pgsql/ddl/li
```

Enjoying our new extension

```
create extension base36;

create table demo(i bigint, x base36);

insert into demo(i, x)
     select n, n::bigint
       from generate_series(0, 10) t(n);

insert into demo(i, x)
     select n, n::bigint
       from generate_series(10000, 10010) t(n);

insert into demo(i, x)
     select n, n::bigint
       from generate_series(100000000, 100000010) t(n);

create index on demo(x);
```

A new integer data type: base36

```
create extension base36;
```

i	x	i	x	i	x
0	0	10000	7PS	100000000	1NJCHS
1	1	10001	7PT	100000001	1NJCHT
2	2	10002	7PU	100000002	1NJCHU
3	3	10003	7PV	100000003	1NJCHV
4	4	10004	7PW	100000004	1NJCHW
5	5	10005	7PX	100000005	1NJCHX
6	6	10006	7PY	100000006	1NJCHY
7	7	10007	7PZ	100000007	1NJCHZ
8	8	10008	7Q0	100000008	1NJCI0
9	9	10009	7Q1	100000009	1NJCI1
10	A	10010	7Q2	100000010	1NJCI2

PostgreSQL is YeSQL!

Recap

We saw a number of extensions, each with a practical use case

ip4r	IP Ranges and Geolocation
Earth	Longitude, Latitude, Computing distances on a map
Trigrams	Fixing typos, autocompletion
Intarray	Indexing Tag Searches
HStore	Schemaless development, Generic Auditing triggers
PL/Proxy	Sharding, RPC, Autonomous Transactions
HLL	Cardinalities, Unique Visitors
BGWorkers	PostgreSQL managed processes
base36	`bigints` with letters

Questions?

Now is the time to ask!