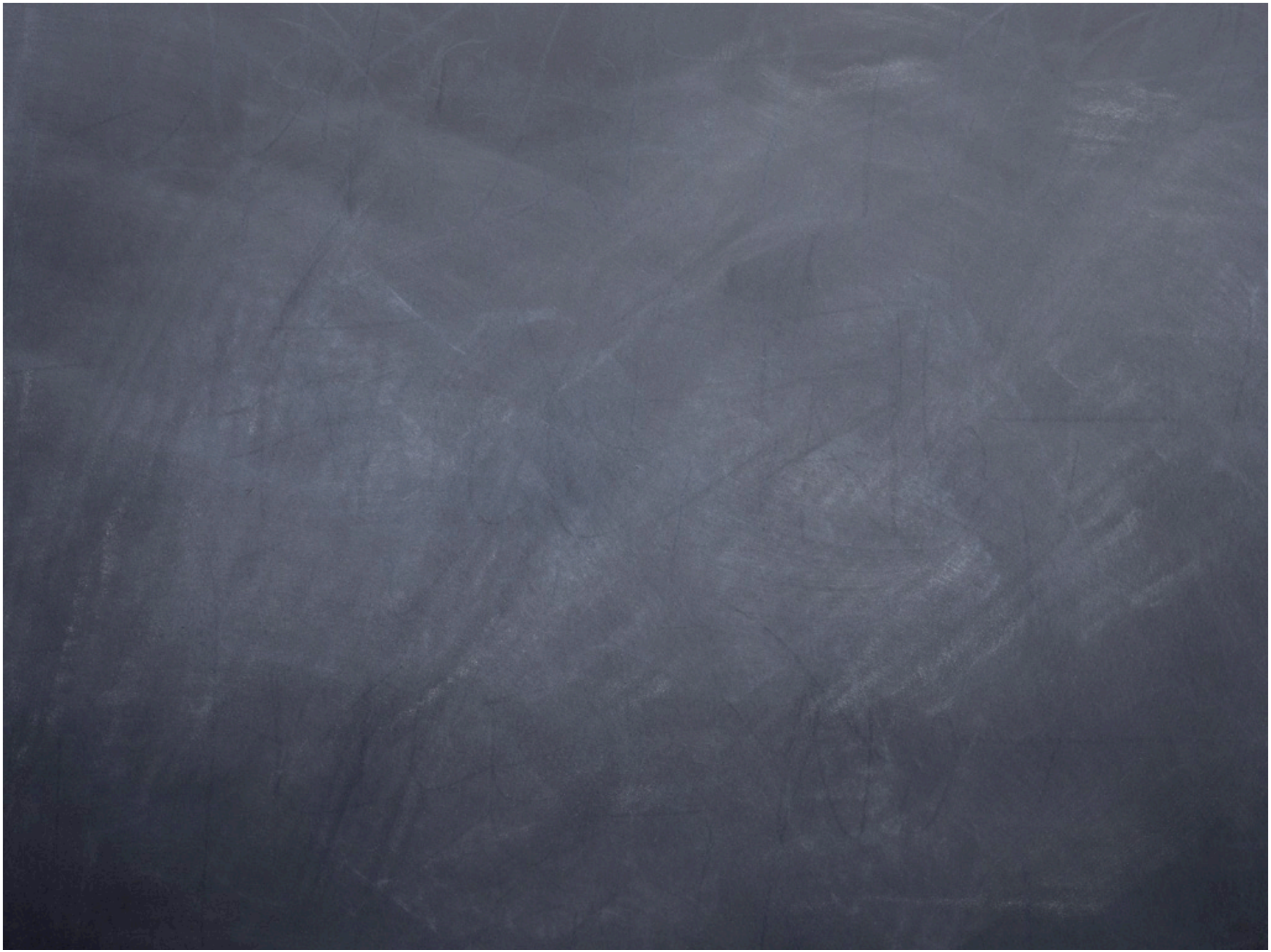
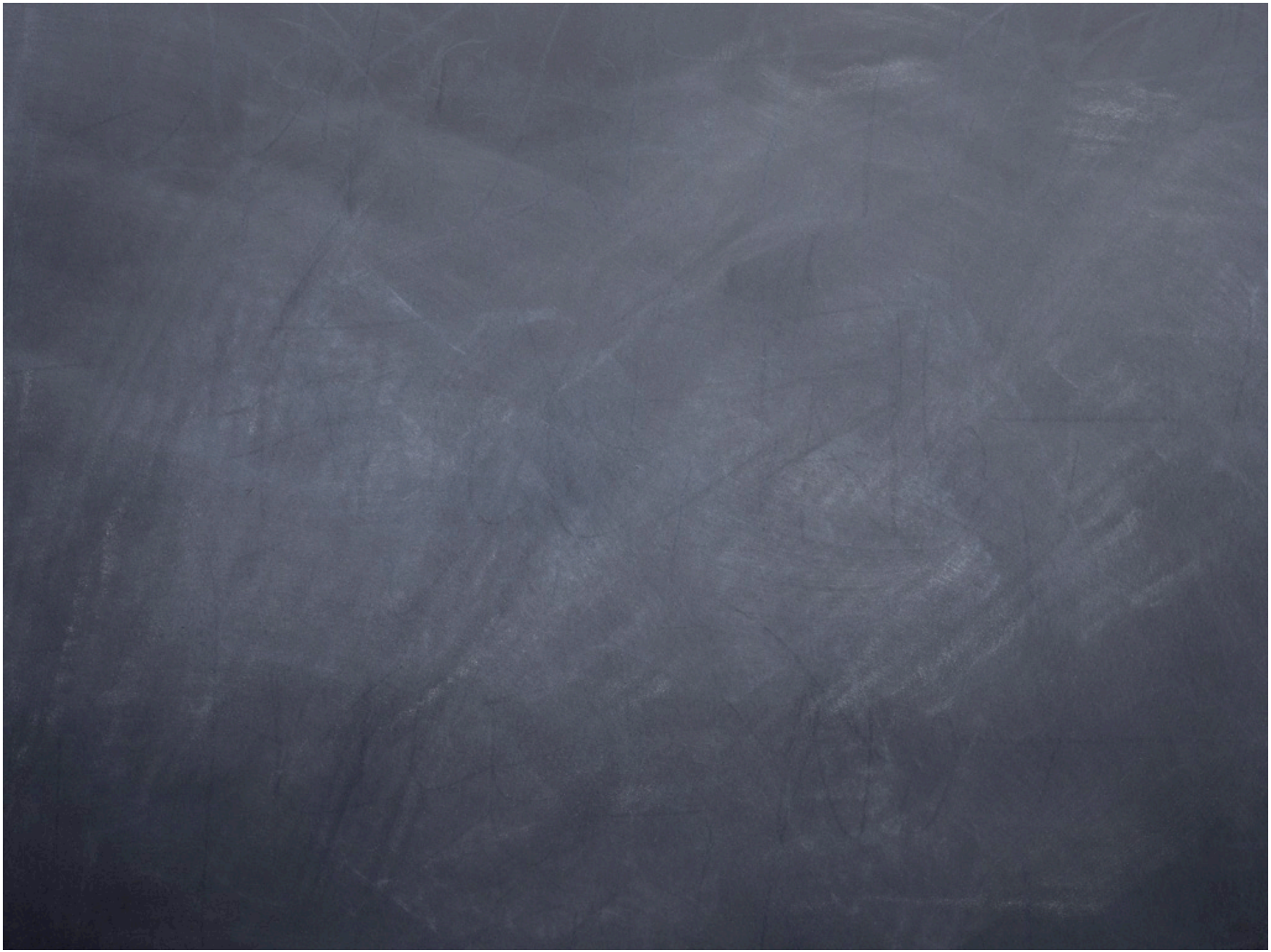
Learning PL/pgSQL

David Wheeler
Kineticode

Portland PostgreSQL Users Group
2006-07-19



But First...



SQL on Rails!

http://www.sqlonrails.org/

Learning PL/pgSQL

David Wheeler
Kineticode

Portland PostgreSQL Users Group
2006-07-19

Application Tiers

Application Tiers

- Application developers used to tiers

Application Tiers

- Application developers used to tiers
  - UI layer (Browser)

Application Tiers

- Application developers used to tiers
  - UI layer (Browser)
  - Application layer (Perl, Python, Ruby, PHP)

Application Tiers

- Application developers used to tiers
  - UI layer (Browser)
  - Application layer (Perl, Python, Ruby, PHP)
  - Database (PostgreSQL, MySQL, Oracle)

What Have We Learned?

What Have We Learned?

- Application layer

What Have We Learned?

- Application layer
  - P* languages

What Have We Learned?

- Application layer
  - P* languages
  - SQL

What Have We Learned?

- Application layer
  - P* languages
  - SQL
  - Templating

What Have We Learned?

What Have We Learned?

- UI layer

What Have We Learned?

- UI layer
  - X?HTML

What Have We Learned?

- UI layer
  - X?HTML
  - CSS

What Have We Learned?

- UI layer
  - X?HTML
  - CSS
  - JavaScript

What Have We Learned?

- UI layer
  - X?HTML
  - CSS
  - JavaScript
  - AJAX

What Have We Learned?

What Have We Learned?

- Database layer

What Have We Learned?

- Database layer
  - CREATE TABLE

What Have We Learned?

- Database layer
  - CREATE TABLE
  - CREATE INDEX

What Have We Learned?

- Database layer
  - CREATE TABLE
  - CREATE INDEX
  - Foreign key constraints

What Have We Learned?

- Database layer
  - CREATE TABLE
  - CREATE INDEX
  - Foreign key constraints
  - Erm...

What Have We Learned?

- Database layer
  - CREATE TABLE
  - CREATE INDEX
  - Foreign key constraints
  - Erm...
  - Is that it?

What about the Database?

What about the Database?

- What we haven't learned

What about the Database?

- What we haven't learned
  - Views

What about the Database?

- What we haven't learned
  - Views
  - Rules

What about the Database?

- What we haven't learned
  - Views
  - Rules
  - Triggers

What about the Database?

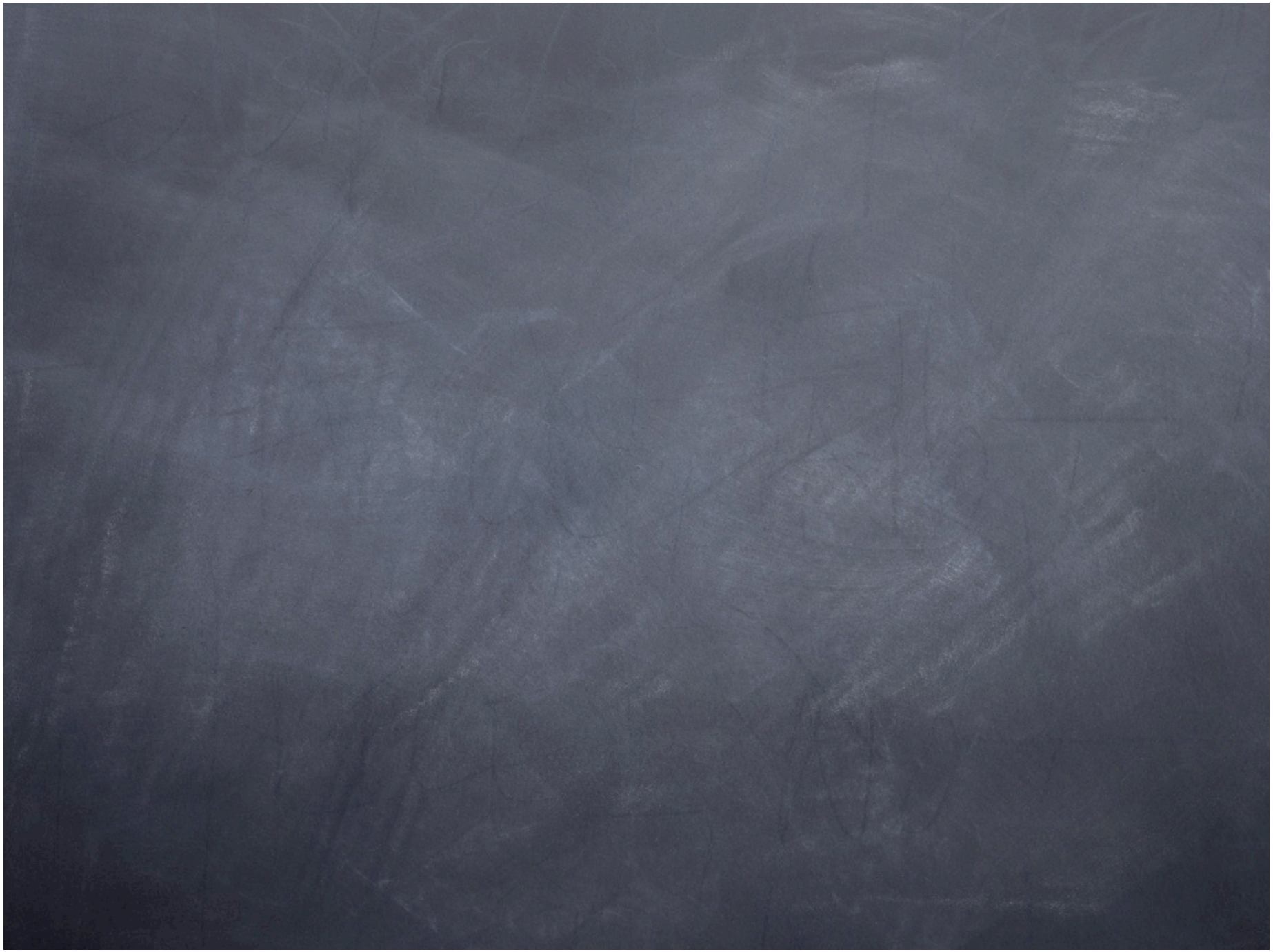
- What we haven't learned
  - Views
  - Rules
  - Triggers
  - Domains

What about the Database?

- What we haven't learned
  - Views
  - Rules
  - Triggers
  - Domains
  - Aggregates

What about the Database?

- What we haven't learned
  - Views
  - Rules
  - Triggers
  - Domains
  - Aggregates
  - Functions/Stored Procedures



Isn't it Time We Changed That?

PostgreSQL Functions

PostgreSQL Functions

- SQL
- PL/Perl
- PL/Python
- PL/TCL
- PL/Ruby
- PL/Java
- PL/PHP
- PL/pgSQL

What is PL/pgSQL?

What is PL/pgSQL?

- Procedural programming language

What is PL/pgSQL?

- Procedural programming language
- More powerful than SQL

What is PL/pgSQL?

- Procedural programming language
- More powerful than SQL
  - Variables

What is PL/pgSQL?

- Procedural programming language
- More powerful than SQL
  - Variables
  - Conditionals

What is PL/pgSQL?

- Procedural programming language
- More powerful than SQL
  - Variables
  - Conditionals
  - Looping constructs

What is PL/pgSQL?

- Procedural programming language
- More powerful than SQL
  - Variables
  - Conditionals
  - Looping constructs
  - Exceptions

What is PL/pgSQL?

- Procedural programming language
- More powerful than SQL
  - Variables
  - Conditionals
  - Looping constructs
  - Exceptions
- Similar to Oracle's PL/SQL

Adding PL/pgSQL

Adding PL/pgSQL

```
% createlang -U postgres plpgsql template1
```

CREATE FUNCTION

CREATE FUNCTION

```
CREATE OR REPLACE FUNCTION name (
```

CREATE FUNCTION

```
CREATE OR REPLACE FUNCTION name (
   type,
   type
```

CREATE FUNCTION

```
CREATE OR REPLACE FUNCTION name (
    type,
    type
) RETURNS type
```

CREATE FUNCTION

```
CREATE OR REPLACE FUNCTION name (
    type,
    type
) RETURNS type AS $$
```

CREATE FUNCTION

```
CREATE OR REPLACE FUNCTION name (
  type,
  type
) RETURNS type AS $$
  // Function Body
```

CREATE FUNCTION

```
CREATE OR REPLACE FUNCTION name (
    type,
    type
) RETURNS type AS $$
    // Function Body
$$ LANGUAGE langname attributes;
```

A Note on Examples

A Note on Examples

- Early examples calculate Fibonacci numbers

A Note on Examples

- Early examples calculate Fibonacci numbers
- Fibonacci numbers are a sequence

A Note on Examples

- Early examples calculate Fibonacci numbers
- Fibonacci numbers are a sequence
- 0, 1, 1, 2, 3, 5, 8, 13, 21...

A Note on Examples

- Early examples calculate Fibonacci numbers
- Fibonacci numbers are a sequence
- 0, 1, 1, 2, 3, 5, 8, 13, 21...
- http://en.wikipedia.org/wiki/Fibonacci_number

A Note on Examples

- Early examples calculate Fibonacci numbers
- Fibonacci numbers are a sequence
- 0, 1, 1, 2, 3, 5, 8, 13, 21...
- http://en.wikipedia.org/wiki/Fibonacci_number
- Used here entirely for pedagogical purposes

A Note on Examples

- Early examples calculate Fibonacci numbers
- Fibonacci numbers are a sequence
- 0, 1, 1, 2, 3, 5, 8, 13, 21...
- http://en.wikipedia.org/wiki/Fibonacci_number
- Used here entirely for pedagogical purposes
- Shows off different PL/pgSQL features

A Note on Examples

- Early examples calculate Fibonacci numbers
- Fibonacci numbers are a sequence
- 0, 1, 1, 2, 3, 5, 8, 13, 21...
- http://en.wikipedia.org/wiki/Fibonacci_number
- Used here entirely for pedagogical purposes
- Shows off different PL/pgSQL features
- Without further ado...

Recursive Fibonacci Function

Recursive Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib (
  fib_for integer
) RETURNS integer AS $$
BEGIN
  IF fib_for < 2 THEN
    RETURN fib_for;
  END IF;
  RETURN fib(fib_for - 2) + fib(fib_for - 1);
END;
$$ LANGUAGE plpgsql strict;
```

Recursive Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib (
  fib_for integer
) RETURNS integer AS $$
BEGIN
  IF fib_for < 2 THEN
    RETURN fib_for;
  END IF;
  RETURN fib(fib_for - 2) + fib(fib_for - 1);
END;
$$ LANGUAGE plpgsql strict;
```

Recursive Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib (
  fib_for integer
) RETURNS integer AS $$
BEGIN
  IF fib_for < 2 THEN
    RETURN fib_for;
  END IF;
  RETURN fib(fib_for - 2) + fib(fib_for - 1);
END;
$$ LANGUAGE plpgsql strict;
```

Recursive Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib (
  fib_for integer
) RETURNS integer AS $$
BEGIN
  IF fib_for < 2 THEN
    RETURN fib_for;
  END IF;
  RETURN fib(fib_for - 2) + fib(fib_for - 1);
END;
$$ LANGUAGE plpgsql strict;
```

Recursive Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib (
  fib_for integer
) RETURNS integer AS $$
BEGIN
  IF fib_for < 2 THEN
    RETURN fib_for;
  END IF;
  RETURN fib(fib_for - 2) + fib(fib_for - 1);
END;
$$ LANGUAGE plpgsql strict;
```

Recursive Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib (
  fib_for integer
) RETURNS integer AS $$
BEGIN
  IF fib_for < 2 THEN
    RETURN fib_for;
  END IF;
  RETURN fib(fib_for - 2) + fib(fib_for - 1);
END;
$$ LANGUAGE plpgsql strict;
```

Recursive Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib (
  fib_for integer
) RETURNS integer AS $$
BEGIN
  IF fib_for < 2 THEN
    RETURN fib_for;
  END IF;
  RETURN fib(fib_for - 2) + fib(fib_for - 1);
END;
$$ LANGUAGE plpgsql strict;
```

Try It

Accessing the Database

Accessing the Database

- Use SQL to access relations

Accessing the Database

- Use SQL to access relations
- Example: Memoize the Fibonacci function

Accessing the Database

- Use SQL to access relations
- Example: Memoize the Fibonacci function
- Create a table to store values

Accessing the Database

- Use SQL to access relations
- Example: Memoize the Fibonacci function
- Create a table to store values

```
CREATE TABLE fib_cache (
  num integer PRIMARY KEY,
  fib integer NOT NULL
);
```

Memoized Fibonacci

Memoized Fibonacci

```
CREATE OR REPLACE FUNCTION fib_cached(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer;
BEGIN
  if fib_for < 2 THEN
    RETURN fib_for;
  END IF;

  SELECT INTO ret fib
  FROM   fib_cache
  WHERE  num = fib_for;

  IF ret IS NULL THEN
    ret := fib_cached(fib_for - 2) + fib_cached(fib_for - 1);
    INSERT INTO fib_cache (num, fib)
    VALUES (fib_for, ret);
  END IF;
  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Memoized Fibonacci

```
CREATE OR REPLACE FUNCTION fib_cached(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer;
BEGIN
  if fib_for < 2 THEN
    RETURN fib_for;
  END IF;

  SELECT INTO ret fib
  FROM   fib_cache
  WHERE  num = fib_for;

  IF ret IS NULL THEN
    ret := fib_cached(fib_for - 2) + fib_cached(fib_for - 1);
    INSERT INTO fib_cache (num, fib)
    VALUES (fib_for, ret);
  END IF;
  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Memoized Fibonacci

```
CREATE OR REPLACE FUNCTION fib_cached(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer;
BEGIN
  if fib_for < 2 THEN
    RETURN fib_for;
  END IF;

  SELECT INTO ret fib
  FROM   fib_cache
  WHERE  num = fib_for;

  IF ret IS NULL THEN
    ret := fib_cached(fib_for - 2) + fib_cached(fib_for - 1);
    INSERT INTO fib_cache (num, fib)
    VALUES (fib_for, ret);
  END IF;
  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Memoized Fibonacci

```
CREATE OR REPLACE FUNCTION fib_cached(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer;
BEGIN
  if fib_for < 2 THEN
    RETURN fib_for;
  END IF;

  SELECT INTO ret fib
  FROM   fib_cache
  WHERE  num = fib_for;

  IF ret IS NULL THEN
    ret := fib_cached(fib_for - 2) + fib_cached(fib_for - 1);
    INSERT INTO fib_cache (num, fib)
    VALUES (fib_for, ret);
  END IF;
  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Memoized Fibonacci

```
CREATE OR REPLACE FUNCTION fib_cached(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer;
BEGIN
  if fib_for < 2 THEN
    RETURN fib_for;
  END IF;

  SELECT INTO ret fib
  FROM   fib_cache
  WHERE  num = fib_for;

  IF ret IS NULL THEN
    ret := fib_cached(fib_for - 2) + fib_cached(fib_for - 1);
    INSERT INTO fib_cache (num, fib)
    VALUES (fib_for, ret);
  END IF;
  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Memoized Fibonacci

```
CREATE OR REPLACE FUNCTION fib_cached(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer;
BEGIN
  if fib_for < 2 THEN
    RETURN fib_for;
  END IF;

  SELECT INTO ret fib
  FROM   fib_cache
  WHERE  num = fib_for;

  IF ret IS NULL THEN
    ret := fib_cached(fib_for - 2) + fib_cached(fib_for - 1);
    INSERT INTO fib_cache (num, fib)
    VALUES (fib_for, ret);
  END IF;
  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Memoized Fibonacci

```
CREATE OR REPLACE FUNCTION fib_cached(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer;
BEGIN
  if fib_for < 2 THEN
    RETURN fib_for;
  END IF;

  SELECT INTO ret fib
  FROM   fib_cache
  WHERE  num = fib_for;

  IF ret IS NULL THEN
    ret := fib_cached(fib_for - 2) + fib_cached(fib_for - 1);
    INSERT INTO fib_cache (num, fib)
    VALUES (fib_for, ret);
  END IF;
  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Memoized Fibonacci

```
CREATE OR REPLACE FUNCTION fib_cached(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer;
BEGIN
  if fib_for < 2 THEN
    RETURN fib_for;
  END IF;

  SELECT INTO ret fib
  FROM   fib_cache
  WHERE  num = fib_for;

  IF ret IS NULL THEN
    ret := fib_cached(fib_for - 2) + fib_cached(fib_for - 1);
    INSERT INTO fib_cache (num, fib)
    VALUES (fib_for, ret);
  END IF;
  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Memoized Fibonacci

```
CREATE OR REPLACE FUNCTION fib_cached(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer;
BEGIN
  if fib_for < 2 THEN
    RETURN fib_for;
  END IF;

  SELECT INTO ret fib
  FROM   fib_cache
  WHERE  num = fib_for;

  IF ret IS NULL THEN
    ret := fib_cached(fib_for - 2) + fib_cached(fib_for - 1);
    INSERT INTO fib_cache (num, fib)
    VALUES (fib_for, ret);
  END IF;
  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Try It

Using SQL in PL/pgSQL

Using SQL in PL/pgSQL

- Any SQL can be used

Using SQL in PL/pgSQL

- Any SQL can be used
- Variables can be used in the SQL statements

Using SQL in PL/pgSQL

- Any SQL can be used
- Variables can be used in the SQL statements
- SQL statements are compiled into the function

Using SQL in PL/pgSQL

- Any SQL can be used
- Variables can be used in the SQL statements
- SQL statements are compiled into the function

```
INSERT  INTO fib_cache (num, fib)
VALUES  (fib_for, ret);
```

Using SQL in PL/pgSQL

- Any SQL can be used
- Variables can be used in the SQL statements
- SQL statements are compiled into the function

```
PREPARE some_insert(integer, integer) AS
INSERT  INTO fib_cache (num, fib)
VALUES  ($1, $2);
```

Using SQL in PL/pgSQL

- Any SQL can be used
- Variables can be used in the SQL statements
- SQL statements are compiled into the function

```
PREPARE some_insert(integer, integer) AS
INSERT  INTO fib_cache (num, fib)
VALUES  ($1, $2);

EXECUTE some_insert(fib_for, ret);
```

Looping Fibonacci Function

Looping Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib_fast(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer := 0;
  nxt integer := 1;
  tmp integer;
BEGIN
  FOR num IN 1..fib_for LOOP
    tmp := ret;
    ret := nxt;
    nxt := tmp + nxt;
  END LOOP;

  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Looping Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib_fast(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer := 0;
  nxt integer := 1;
  tmp integer;
BEGIN
  FOR num IN 1..fib_for LOOP
    tmp := ret;
    ret := nxt;
    nxt := tmp + nxt;
  END LOOP;

  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Looping Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib_fast(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer := 0;
  nxt integer := 1;
  tmp integer;
BEGIN
  FOR num IN 1..fib_for LOOP
    tmp := ret;
    ret := nxt;
    nxt := tmp + nxt;
  END LOOP;

  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Looping Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib_fast(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer := 0;
  nxt integer := 1;
  tmp integer;
BEGIN
  FOR num IN 1..fib_for LOOP
    tmp := ret;
    ret := nxt;
    nxt := tmp + nxt;
  END LOOP;

  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Looping Fibonacci Function

```
CREATE OR REPLACE FUNCTION fib_fast(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer := 0;
  nxt integer := 1;
  tmp integer;
BEGIN
  FOR num IN 1..fib_for LOOP
    tmp := ret;
    ret := nxt;
    nxt := tmp + nxt;
  END LOOP;

  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Try It

Set Returning Functions

Set Returning Functions

- Functions can return sets

Set Returning Functions

- Functions can return sets
- Similar to continuations

Set Returning Functions

- Functions can return sets
- Similar to continuations
- Think of a SELECT on a single-column table

Set Returning Functions

- Functions can return sets
- Similar to continuations
- Think of a SELECT on a single-column table
- Set returning function used like a table

Set Returning Functions

- Functions can return sets
- Similar to continuations
- Think of a SELECT on a single-column table
- Set returning function used like a table
- Easy to implement in PL/pgSQL

Fibonacci Set

Fibonacci Set

```
CREATE OR REPLACE FUNCTION fib_fast(
  fib_for int
) RETURNS integer AS $$
DECLARE
  ret integer := 0;
  nxt integer := 1;
  tmp integer;
BEGIN
  FOR num IN 1..fib_for LOOP

    tmp := ret;
    ret := nxt;
    nxt := tmp + nxt;
  END LOOP;

  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Fibonacci Set

```
CREATE OR REPLACE FUNCTION fibs_to(
  fib_for int
) RETURNS SETOF integer AS $$
DECLARE
  ret integer := 0;
  nxt integer := 1;
  tmp integer;
BEGIN
  FOR num IN 1..fib_for LOOP

    tmp := ret;
    ret := nxt;
    nxt := tmp + nxt;
  END LOOP;

  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Fibonacci Set

```
CREATE OR REPLACE FUNCTION fibs_to(
  fib_for int
) RETURNS SETOF integer AS $$
DECLARE
  ret integer := 0;
  nxt integer := 1;
  tmp integer;
BEGIN
  FOR num IN 1..fib_for LOOP
    RETURN NEXT ret;
    tmp := ret;
    ret := nxt;
    nxt := tmp + nxt;
  END LOOP;

  RETURN ret;
END;
$$ LANGUAGE plpgsql strict;
```

Fibonacci Set

```
CREATE OR REPLACE FUNCTION fibs_to(
  fib_for int
) RETURNS SETOF integer AS $$
DECLARE
  ret integer := 0;
  nxt integer := 1;
  tmp integer;
BEGIN
  FOR num IN 1..fib_for LOOP
    RETURN NEXT ret;
    tmp := ret;
    ret := nxt;
    nxt := tmp + nxt;
  END LOOP;

  RETURN NEXT ret;
END;
$$ LANGUAGE plpgsql strict;
```

Try It

Yea, but What Good are They?

Yea, but What Good are They?

- I'm glad you asked

Yea, but What Good are They?

- I'm glad you asked
- Let's get practical

Yea, but What Good are They?

- I'm glad you asked
- Let's get practical
- Let's manage an ordered many-to-many relationship

A Blogging App's Schema

A Blogging App's Schema

```
CREATE TABLE entry (
  id      SERIAL PRIMARY KEY,
  title   TEXT,
  content TEXT
);
```

A Blogging App's Schema

```
CREATE TABLE entry (
  id      SERIAL PRIMARY KEY,
  title   TEXT,
  content TEXT
);

CREATE TABLE tag (
  id SERIAL PRIMARY KEY,
  name text
);
```

A Blogging App's Schema

```
CREATE TABLE entry (
  id      SERIAL PRIMARY KEY,
  title   TEXT,
  content TEXT
);

CREATE TABLE tag (
  id SERIAL PRIMARY KEY,
  name text
);

CREATE TABLE entry_coll_tag (
  entry_id  integer REFERENCES entry(id)
                    ON UPDATE CASCADE
                    ON DELETE CASCADE,
  tag_id    integer REFERENCES tag(id)
                    ON UPDATE CASCADE
                    ON DELETE CASCADE,
  tag_order smallint,
  PRIMARY KEY (entry_id, tag_id)
);
```

A Blogging App's Schema

```
CREATE TABLE entry (
  id      SERIAL PRIMARY KEY,
  title   TEXT,
  content TEXT
);

CREATE TABLE tag (
  id SERIAL PRIMARY KEY,
  name text
);

CREATE TABLE entry_coll_tag (
  entry_id  integer REFERENCES entry(id)
                    ON UPDATE CASCADE
                    ON DELETE CASCADE,
  tag_id    integer REFERENCES tag(id)
                    ON UPDATE CASCADE
                    ON DELETE CASCADE,
  tag_order smallint,
  PRIMARY KEY (entry_id, tag_id)
);

CREATE UNIQUE INDEX idx_entry_coll_tag_order
ON entry_coll_tag (entry_id, tag_order);
```

SELECT Tags for an Entry

SELECT Tags for an Entry

```
SELECT tag.id, tag.name
FROM   tag, entry_coll_tag
WHERE  tag.id = entry_coll_tag.tag_id
       AND entry_coll_tag.entry_id = 1
ORDER BY entry_coll_tag.tag_order;
```

Try It

```
SELECT tag.id, tag.name
FROM   tag, entry_coll_tag
WHERE  tag.id = entry_coll_tag.tag_id
       AND entry_coll_tag.entry_id = 1
ORDER BY entry_coll_tag.tag_order;
```

Setting Tags

Setting Tags

```
# Use prepared statements.
insert = dbh.prepare('INSERT INTO entry (title, content) VALUES (?, ?)');
sel_id = dbh.prepare("SELECT CURRVAL('entry_id_seq')");

ins_coll = dbh.prepare('
    INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
    VALUES (?, ?, ?)
');

# Do everything inside a transaction.
dbh.begin;

# Insert the new entry.
insert.execute(entry.title, entry.content);
sel_id.execute;
entry.id = sel_id.fetch;

# Associate the tags with the entry.
i = 0;
foreach tag in (tag_array) {
    ins_coll.execute(entry.id, tag.id, ++i);
}

# Make it so!
dbh.commit;
```

What's Wrong with That?

What's Wrong with That?

- It's Slow

What's Wrong with That?

- It's Slow
  - A separate query for each tag

What's Wrong with That?

- It's Slow
  - A separate query for each tag
  - What if there were 100 tags?

What's Wrong with That?

- It's Slow
  - A separate query for each tag
  - What if there were 100 tags?
- There's a race condition

What's Wrong with That?

- It's Slow
  - A separate query for each tag
  - What if there were 100 tags?
- There's a race condition
  - Tag order can be wrong

What's Wrong with That?

- It's Slow
  - A separate query for each tag
  - What if there were 100 tags?
- There's a race condition
  - Tag order can be wrong
  - When two processes update tags for the same entry at the same time

How can PL/pgSQL Help?

How can PL/pgSQL Help?

- A function can associate tags with entries

How can PL/pgSQL Help?

- A function can associate tags with entries
- It can carefully control for the race condition

How can PL/pgSQL Help?

- A function can associate tags with entries
- It can carefully control for the race condition
- It can be called once to associate all tags with a given entry

Let's Do It!

Let's Do It!

```
CREATE OR REPLACE FUNCTION entry_coll_tag_set (
    obj_id   integer,
    coll_ids integer[]
) RETURNS VOID AS $$
BEGIN
    PERFORM true FROM entry WHERE id = obj_id FOR UPDATE;

    UPDATE entry_coll_tag
    SET    tag_order = -tag_order
    WHERE  entry_id = obj_id

    FOR iloop IN 1..array_upper(coll_ids, 1) LOOP
        IF coll_ids[iloop] IS NULL THEN
            CONTINUE;
        END IF;

        UPDATE entry_coll_tag
        SET    tag_order = iloop
        WHERE  entry_id = obj_id
               AND tag_id = coll_ids[iloop];

        IF FOUND IS false THEN
            INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
            VALUES (obj_id, coll_ids[iloop], iloop);
        END IF;
    END LOOP;

    DELETE FROM entry_coll_tag
    WHERE  entry_id = obj_id AND tag_order < 0;
END;
$$ LANGUAGE plpgsql;
```

Let's Do It!

```
CREATE OR REPLACE FUNCTION entry_coll_tag_set (
    obj_id   integer,
    coll_ids integer[]
) RETURNS VOID AS $$
BEGIN
    PERFORM true FROM entry WHERE id = obj_id FOR UPDATE;

    UPDATE entry_coll_tag
    SET    tag_order = -tag_order
    WHERE  entry_id = obj_id

    FOR iloop IN 1..array_upper(coll_ids, 1) LOOP
        IF coll_ids[iloop] IS NULL THEN
            CONTINUE;
        END IF;

        UPDATE entry_coll_tag
        SET    tag_order = iloop
        WHERE  entry_id = obj_id
               AND tag_id = coll_ids[iloop];

        IF FOUND IS false THEN
            INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
            VALUES (obj_id, coll_ids[iloop], iloop);
        END IF;
    END LOOP;

    DELETE FROM entry_coll_tag
    WHERE  entry_id = obj_id AND tag_order < 0;
END;
$$ LANGUAGE plpgsql;
```

Let's Do It!

```
CREATE OR REPLACE FUNCTION entry_coll_tag_set (
    obj_id   integer,
    coll_ids integer[]
) RETURNS VOID AS $$
BEGIN
    PERFORM true FROM entry WHERE id = obj_id FOR UPDATE;

    UPDATE entry_coll_tag
    SET    tag_order = -tag_order
    WHERE  entry_id = obj_id

    FOR iloop IN 1..array_upper(coll_ids, 1) LOOP
        IF coll_ids[iloop] IS NULL THEN
            CONTINUE;
        END IF;

        UPDATE entry_coll_tag
        SET    tag_order = iloop
        WHERE  entry_id = obj_id
               AND tag_id = coll_ids[iloop];

        IF FOUND IS false THEN
            INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
            VALUES (obj_id, coll_ids[iloop], iloop);
        END IF;
    END LOOP;

    DELETE FROM entry_coll_tag
    WHERE  entry_id = obj_id AND tag_order < 0;
END;
$$ LANGUAGE plpgsql;
```

Let's Do It!

```
CREATE OR REPLACE FUNCTION entry_coll_tag_set (
    obj_id   integer,
    coll_ids integer[]
) RETURNS VOID AS $$
BEGIN
    PERFORM true FROM entry WHERE id = obj_id FOR UPDATE;

    UPDATE entry_coll_tag
    SET    tag_order = -tag_order
    WHERE  entry_id = obj_id

    FOR iloop IN 1..array_upper(coll_ids, 1) LOOP
        IF coll_ids[iloop] IS NULL THEN
            CONTINUE;
        END IF;

        UPDATE entry_coll_tag
        SET    tag_order = iloop
        WHERE  entry_id = obj_id
               AND tag_id = coll_ids[iloop];

        IF FOUND IS false THEN
            INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
            VALUES (obj_id, coll_ids[iloop], iloop);
        END IF;
    END LOOP;

    DELETE FROM entry_coll_tag
    WHERE  entry_id = obj_id AND tag_order < 0;
END;
$$ LANGUAGE plpgsql;
```

Let's Do It!

```
CREATE OR REPLACE FUNCTION entry_coll_tag_set (
    obj_id   integer,
    coll_ids integer[]
) RETURNS VOID AS $$
BEGIN
    PERFORM true FROM entry WHERE id = obj_id FOR UPDATE;

    UPDATE entry_coll_tag
    SET    tag_order = -tag_order
    WHERE  entry_id = obj_id

    FOR iloop IN 1..array_upper(coll_ids, 1) LOOP
        IF coll_ids[iloop] IS NULL THEN
            CONTINUE;
        END IF;

        UPDATE entry_coll_tag
        SET    tag_order = iloop
        WHERE  entry_id = obj_id
               AND tag_id = coll_ids[iloop];

        IF FOUND IS false THEN
            INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
            VALUES (obj_id, coll_ids[iloop], iloop);
        END IF;
    END LOOP;

    DELETE FROM entry_coll_tag
    WHERE  entry_id = obj_id AND tag_order < 0;
END;
$$ LANGUAGE plpgsql;
```

Let's Do It!

```
CREATE OR REPLACE FUNCTION entry_coll_tag_set (
    obj_id   integer,
    coll_ids integer[]
) RETURNS VOID AS $$
BEGIN
    PERFORM true FROM entry WHERE id = obj_id FOR UPDATE;

    UPDATE entry_coll_tag
    SET    tag_order = -tag_order
    WHERE  entry_id = obj_id

    FOR iloop IN 1..array_upper(coll_ids, 1) LOOP
        IF coll_ids[iloop] IS NULL THEN
            CONTINUE;
        END IF;

        UPDATE entry_coll_tag
        SET    tag_order = iloop
        WHERE  entry_id = obj_id
               AND tag_id = coll_ids[iloop];

        IF FOUND IS false THEN
            INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
            VALUES (obj_id, coll_ids[iloop], iloop);
        END IF;
    END LOOP;

    DELETE FROM entry_coll_tag
    WHERE  entry_id = obj_id AND tag_order < 0;
END;
$$ LANGUAGE plpgsql;
```

Let's Do It!

```
CREATE OR REPLACE FUNCTION entry_coll_tag_set (
    obj_id   integer,
    coll_ids integer[]
) RETURNS VOID AS $$
BEGIN
    PERFORM true FROM entry WHERE id = obj_id FOR UPDATE;

    UPDATE entry_coll_tag
    SET    tag_order = -tag_order
    WHERE  entry_id = obj_id

    FOR iloop IN 1..array_upper(coll_ids, 1) LOOP
        IF coll_ids[iloop] IS NULL THEN
            CONTINUE;
        END IF;

        UPDATE entry_coll_tag
        SET    tag_order = iloop
        WHERE  entry_id = obj_id
               AND tag_id = coll_ids[iloop];

        IF FOUND IS false THEN
            INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
            VALUES (obj_id, coll_ids[iloop], iloop);
        END IF;
    END LOOP;

    DELETE FROM entry_coll_tag
    WHERE  entry_id = obj_id AND tag_order < 0;
END;
$$ LANGUAGE plpgsql;
```

Let's Do It!

```
CREATE OR REPLACE FUNCTION entry_coll_tag_set (
    obj_id   integer,
    coll_ids integer[]
) RETURNS VOID AS $$
BEGIN
    PERFORM true FROM entry WHERE id = obj_id FOR UPDATE;

    UPDATE entry_coll_tag
    SET    tag_order = -tag_order
    WHERE  entry_id = obj_id

    FOR iloop IN 1..array_upper(coll_ids, 1) LOOP
        IF coll_ids[iloop] IS NULL THEN
            CONTINUE;
        END IF;

        UPDATE entry_coll_tag
        SET    tag_order = iloop
        WHERE  entry_id = obj_id
               AND tag_id = coll_ids[iloop];

        IF FOUND IS false THEN
            INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
            VALUES (obj_id, coll_ids[iloop], iloop);
        END IF;
    END LOOP;

    DELETE FROM entry_coll_tag
    WHERE  entry_id = obj_id AND tag_order < 0;
END;
$$ LANGUAGE plpgsql;
```

Let's Do It!

```
CREATE OR REPLACE FUNCTION entry_coll_tag_set (
    obj_id   integer,
    coll_ids integer[]
) RETURNS VOID AS $$
BEGIN
    PERFORM true FROM entry WHERE id = obj_id FOR UPDATE;

    UPDATE entry_coll_tag
    SET    tag_order = -tag_order
    WHERE  entry_id = obj_id

    FOR iloop IN 1..array_upper(coll_ids, 1) LOOP
        IF coll_ids[iloop] IS NULL THEN
            CONTINUE;
        END IF;

        UPDATE entry_coll_tag
        SET    tag_order = iloop
        WHERE  entry_id = obj_id
               AND tag_id = coll_ids[iloop];

        IF FOUND IS false THEN
            INSERT INTO entry_coll_tag (entry_id, tag_id, tag_order)
            VALUES (obj_id, coll_ids[iloop], iloop);
        END IF;
    END LOOP;

    DELETE FROM entry_coll_tag
    WHERE  entry_id = obj_id AND tag_order < 0;
END;
$$ LANGUAGE plpgsql;
```

Try It

Try It

```
SELECT entry_coll_tag_set(1, '{1,4,6,3}');
```

Learn More

- My O'Reilly Articles
  http://www.oreillynet.com/pub/au/1059
- The PostgreSQL Documentation
  http://www.postgresql.org/docs/current/static/plpgsql.html
- PostgreSQL: Introduction and Concepts
  http://www.postgresql.org/files/documentation/books/aw_pgsql/node165.html

Thank You!

Learning PL/pgSQL

David Wheeler
Kineticode

Portland PostgreSQL Users Group
2006-07-19