Writeable CTEs

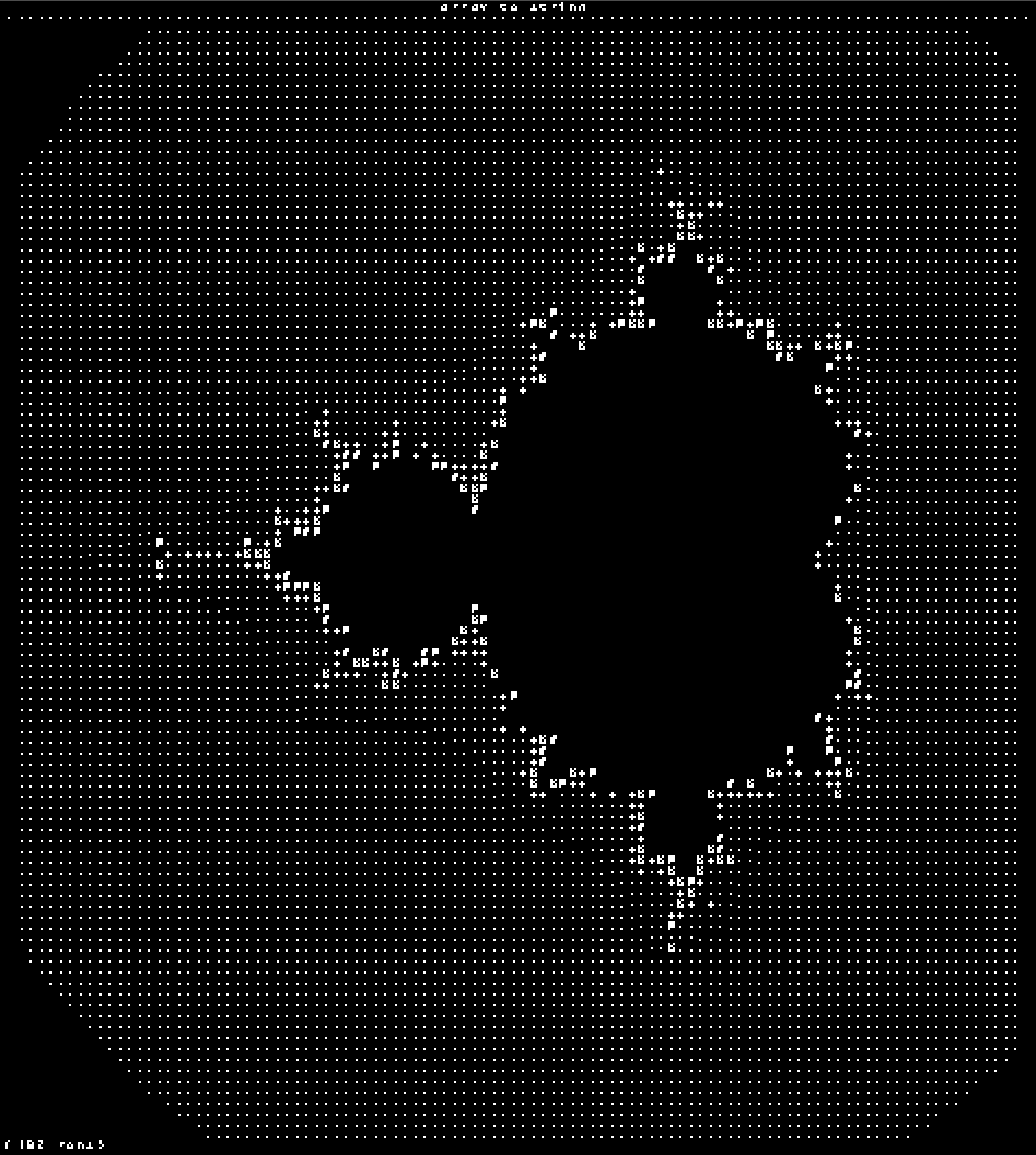
(the next big thing)

David Fetter david.fetter@pgexperts.com

PGX
POSTGRESQL
EXPERTS, INC.

Current CTEs

```
WITH [RECURSIVE] t1 [(column type,…)] AS
(
    [SELECT | VALUES]
[UNION [ALL]
    [SELECT]
),
t2 AS…tn AS…
SELECT…
```

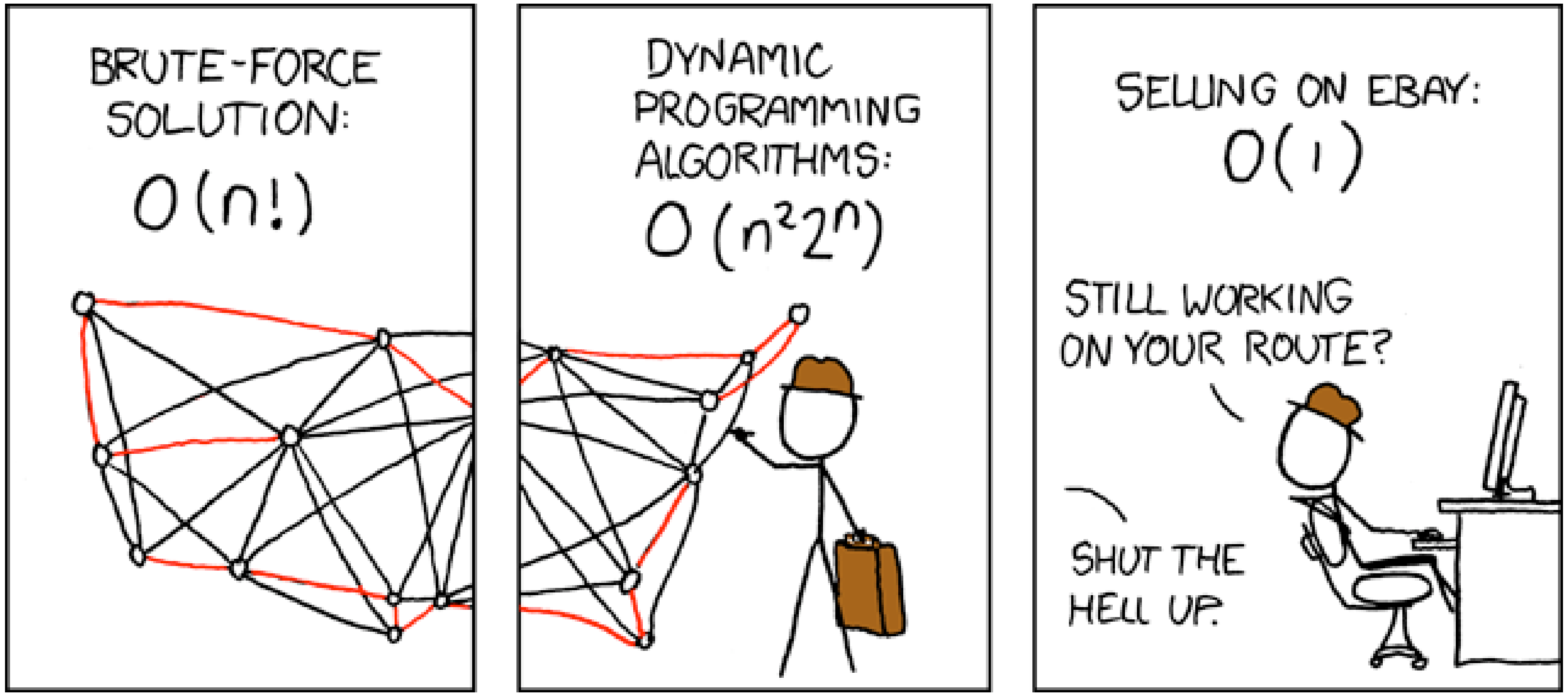


```
array to string

(102 rows)
```

Travelling Salesman Problem

Given a number of cities and the costs of travelling from any city to any other city, what is the least-cost round-trip route that visits each city exactly once and then returns to the starting city?



OBTW

```
With CTE and Windowing, SQL is Turing Complete.
```

What Didn't the Old Syntax Do?

WRITE!

```
WITH [RECURSIVE] t1 [(column type,…)] AS
(
     [SELECT | VALUES |
   (INSERT | UPDATE | DELETE) [RETURNING]]
[UNION [ALL]
     [SELECT | VALUES |
   (INSERT | UPDATE | DELETE) [RETURNING]]
)
(SELECT | INSERT | UPDATE | DELETE)     …
```

For 8.5:
Simple Partition Management

```
CREATE TABLE log (
    ts TIMESTAMPTZ NOT NULL,
    msg TEXT
);
```

For 8.5:
Simple Partition Management

```
CREATE TABLE log_200901 ()
INHERITS(log);

ALTER TABLE log_200901 ADD
CONSTRAINT right_month CHECK(
    ts >= '2009-01-01' AND
    ts <  '2009-02-01');
```

For 8.5:
Simple Partition Management

```
johto@postgres:54321=# WITH
t1 AS (DELETE FROM ONLY log WHERE ts < '2009-02-01' RETURNING *),
INSERT INTO log_200901 SELECT * FROM t1;
INSERT 0 83240
```

What you'll be able to do:

```
WITH t AS (
     DELETE FROM ONLY log WHERE ts >= '2009-01-01'
                       AND ts < '2009-02-01'
                       RETURNING *)
INSERT INTO log_200901
SELECT * FROM t;



                       QUERY PLAN
---------------------------------------------------------
 Insert (cost=27.40..27.52 rows=83240 width=40)
   ->  CTE Scan on t  (cost=27.40..27.52 rows=83240 width=40)
         CTE t
           ->  Delete  (cost=0.00..27.40 rows=83240 width=6)
                 ->  Seq Scan on log  (cost=0.00..27.40 rows=83240 width=6)
                       Filter: (..)
(6 rows)
```

What you can do now: Partition Management

```
johto@postgres:54321=# WITH
t1 AS (DELETE FROM ONLY log WHERE ts < '2009-02-01' RETURNING *),
t2 AS (INSERT INTO log_200901 SELECT * FROM t1)
SELECT min(ts), max(ts), count(*) FROM t1;
             min              |             max              | count
------------------------------|------------------------------|-------
 2009-01-01 00:00:01.6416-08 | 2009-01-30 23:58:38.6976-08 | 83240
(1 row)
```

Query Clustering: I/O Minimization

```
CREATE TABLE person (
    id SERIAL PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    CHECK (CASE WHEN first_name IS NULL THEN 0 ELSE 1 END +
           CASE WHEN last_name IS NULL THEN 0 ELSE 1 END >= 1)
    birthdate DATE NOT NULL,
    gender TEXT
);
```

Query Clustering: I/O Minimization

```
CREATE TABLE im (
    id SERIAL PRIMARY KEY,
    provider TEXT NOT NULL, /* should be fk */
    handle TEXT NOT NULL
);
```

Query Clustering: I/O Minimization

```
CREATE TABLE phone (
    id SERIAL PRIMARY KEY,
    country_code TEXT NOT NULL,
    phone_number TEXT NOT NULL,
    extension TEXT
);
```

Query Clustering: I/O Minimization

```
CREATE TABLE street (
    id SERIAL PRIMARY KEY,
    street1 TEXT NOT NULL,
    street2 TEXT,
    street3 TEXT,
    city TEXT NOT NULL,
    state TEXT,
    country TEXT NOT NULL,
    post_code TEXT
);
```

Query Clustering: I/O Minimization

```
CREATE TABLE person_im (
    person_id INTEGER NOT NULL REFERENCES person (id),
    im_id INTEGER NOT NULL REFERENCES im (id),
    UNIQUE (person_id, im_id)
);

CREATE TABLE person_phone (
    person_id INTEGER NOT NULL REFERENCES person (id),
    phone_id INTEGER NOT NULL REFERENCES phone (id),
    UNIQUE (person_id, phone_id)
);

CREATE TABLE person_street (
    person_id INTEGER NOT NULL REFERENCES person (id),
    street_id INTEGER NOT NULL REFERENCES street (id),
    UNIQUE (person_id, street_id)
);
```

Query Clustering: I/O Minimization

```
WITH t_person AS (
    INSERT INTO person (first_name, last_name)
    VALUES ('David', 'Fetter')
    RETURNING id
),
```

Query Clustering: I/O Minimization

```
t_im AS (
    INSERT INTO im (provider, handle)
    VALUES
        ('Yahoo!', 'dfetter'),
        ('AIM', 'dfetter666'),
        ('XMPP', 'david.fetter@gmail.com')
    RETURNING id
),
t_person_im AS (
    INSERT INTO person_im
    SELECT * FROM t_person CROSS JOIN t_im
),
```

Query Clustering: I/O Minimization

```
t_phone (phone_id) AS (
    INSERT INTO phone (country_code, phone_number)
    VALUES
        ('+1','415 235 3778'),
        ('+1','510 893 6100')
    RETURNING id
),
t_person_phone AS (
    INSERT INTO person_phone
    SELECT * FROM t_person CROSS JOIN t_phone
),
```

Query Clustering: I/O Minimization

```
t_street AS (
    INSERT INTO street (street1, city, state, country, post_code)
    VALUES
        ('2500B Magnolia Street', 'Oakland','California','USA','94607-2410'),
        ('2166 Hayes Street Suite 200', 'San Francisco','California','USA','94117')
),
t_person_street AS (
    INSERT INTO person_street
    SELECT * FROM t_person CROSS JOIN t_street
)
```

Query Clustering:
I/O Minimization

```
VALUES(true);
```

Query Clustering: Transaction Management

```
CREATE TABLE foo (
    id SERIAL PRIMARY KEY,
    bar_id INTEGER NOT NULL
);

CREATE TABLE bar (
    id SERIAL PRIMARY KEY,
    foo_id INTEGER NOT NULL REFERENCES foo(id)
                           ON DELETE CASCADE
                           INITIALLY DEFERRED
);

ALTER TABLE foo ADD FOREIGN KEY (bar_id) REFERENCES bar(id)
                                   ON DELETE CASCADE
                                   INITIALLY DEFERRED;
```

Query Clustering: Transaction Management

```
WITH t AS
(
   INSERT INTO foo(id, bar_id)
   VALUES(
     DEFAULT,
     nextval(pg_get_serial_sequence('bar', 'id'))
  )
   RETURNING id AS foo_id, bar_id
)
INSERT INTO bar(id,foo_id)
SELECT bar_id, foo_id FROM t RETURNING *;
```

How'd He Do That?!?

First try: David digs into the grammar and gets cut a few times.

How'd He Do That?!?

First try: Marko reworks the planner. It needs to know when it creates a ModifyTable node. These used to have another name.

How'd He Do That?!?

First try: Marko reworks the executor. It needs new nodes. Mmmm...nodes.

How'd He Do That?!?

Marko reworks the executor, Part II:
Copy & Paste. Now it's getting ugly...

How'd He Do That?!?

```
Jaime Casanova, Tom Lane, and Robert Haas look at
the reworked executor.

D'oh!
```

How'd He Do That?!?

FAIL!

Way too much code copying from top level to the new nodes.

How'd He Do That?!?

Planner changes for ModifyTable node (a few)

How'd He Do That?!?

Executor changes: ONE new node called ModifyTable

How'd He Do That?!?

Johto restructures the whole code base for the ModifyTable node. "The usual stuff," (he said casually) for new nodes.

How'd He Do That?!?

WIN!

Next Steps

INSERT, UPDATE and DELETE on the top level.
RECURSIVE
Optimization

How'd He Do That?!?

Questions?

Comments?

Thank You!

http://2009.pgday.eu/feedback

PGX
POSTGRESQL
EXPERTS, INC.