PostgreSQL SQL Tricks I

SQL is a language where one task can be solved multiple ways with different efficiency. You can see effective use of PostgreSQL's possibilities on this page.

Google AdSense

[Newer tricks](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II)

**Obsah**

 [[skrýt](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I)]

* [1 Slow queries in stored procedures](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Slow_queries_in_stored_procedures)
* [2 Slow DISTINCT](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Slow_DISTINCT)
* [3 Taking quarter from interval](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Taking_quarter_from_interval)
* [4 Select random row](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Select_random_row)
* [5 Use exception carefully](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Use_exception_carefully)
* [6 Fast compare variables NEW and OLD in trigger's body](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Fast_compare_variables_NEW_and_OLD_in_trigger.27s_body)
* [7 Fast first n rows removing](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Fast_first_n_rows_removing)
* [8 Getting list of comma separated items](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Getting_list_of_comma_separated_items)
* [9 Using functions pgstattuple and pgstatindex](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Using_functions_pgstattuple_and_pgstatindex)
* [10 Diacritic removing](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Diacritic_removing)
* [11 Predicate IN optimalization](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Predicate_IN_optimalization)
* [12 BAR code validation](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#BAR_code_validation)
* [13 Using hash functions for ensuring uniqueness of texts](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Using_hash_functions_for_ensuring_uniqueness_of_texts)
* [14 Getting OID of function](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Getting_OID_of_function)
* [15 General array sort](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#General_array_sort)
* [16 Array transformation in PL/Perl](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Array_transformation_in_PL.2FPerl)
* [17 UPC code validation](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#UPC_code_validation)
* [18 Show only every n row](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Show_only_every_n_row)
* [19 Array to table](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Array_to_table)
* [20 LIKE optimalization](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#LIKE_optimalization)
* [21 Deferred constraints](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Deferred_constraints)
* [22 Removing of duplicate rows](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Removing_of_duplicate_rows)
* [23 Select first n rows from group](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Select_first_n_rows_from_group)
* [24 Cast to varchar](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Cast_to_varchar)
* [25 Create Aggregates](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I#Create_Aggregates)

Slow queries in stored procedures

Sometimes some users reports slow queries in stored procedure. Direct execution of same query is fast. Reason is simple. Stored procedures uses prepared statements and optimalization of prepared statements runs without knowledge of real parameters. This behave ensure safety against sql injection, but carry some problems, planner is blind. This problem isn't too often and solution is simply - we have to use dynamic queries. But we cannot to forget carefully check against sql injection. This kind of problems we can find with analyzing of prepared statements.

t2=# PREPARE pp(integer) AS SELECT count(\*) FROM foo WHERE a BETWEEN $1 + 100 AND $1 + 200;

t2=# EXPLAIN ANALYZE execute pp(100);

 QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------

 Aggregate (cost=438.60..438.61 rows=1 width=0) (actual time=16.149..16.150 rows=1 loops=1)

 -> Bitmap Heap Scan on foo (cost=13.39..437.35 rows=500 width=0) (actual time=1.391..5.458 rows=1033 loops=1)

 Recheck Cond: ((a >= ($1 + 100)) AND (a <= ($1 + 200)))

 -> Bitmap Index Scan on fx (cost=0.00..13.26 rows=500 width=0) (actual time=1.131..1.131 rows=1033 loops=1)

 Index Cond: ((a >= ($1 + 100)) AND (a <= ($1 + 200)))

 Total runtime: 16.340 ms

(6 rows)

t2=# EXPLAIN ANALYZE SELECT count(\*) FROM foo WHERE a BETWEEN 100 + 100 AND 100 + 200;

 QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------

 Aggregate (cost=432.21..432.22 rows=1 width=0) (actual time=15.930..15.932 rows=1 loops=1)

 -> Bitmap Heap Scan on foo (cost=22.21..429.78 rows=971 width=0) (actual time=1.179..13.338 rows=1033 loops=1)

 Recheck Cond: ((a >= 200) AND (a <= 300))

 -> Bitmap Index Scan on fx (cost=0.00..21.97 rows=971 width=0) (actual time=0.917..0.917 rows=1033 loops=1)

 Index Cond: ((a >= 200) AND (a <= 300))

 Total runtime: 16.058 ms

(6 rows)

Execution plans are nearly identical, so we can use prepared statements.

PostgreSQL 9.2 has redesigned implementation of prepared statements - described performance issues should be history.

Slow DISTINCT

Current implementation of DISTINCT needs sort, that can be slow. We should to use GROUP BY clause - this clause can to use hash aggregate - and it's significantly faster than DISTINCT.

postgres=# select count(\*) from (select distinct i from g) a;

 count

-------

 10001

(1 row)

Time: 1563,109 ms

postgres=# select count(\*) from (select i from g group by i) a;

 count

-------

 10001

(1 row)

Time: 594,481 ms

This topis is related to PostgreSQL 8.2 and older. PostgreSQL 8.3 and higher can to use a hash aggregation for distinct.

Taking quarter from interval

Interval arithmetic is little bit alchemy. Internally type interval carries days, months and years. Input value isn't normalized, it stays in entered units (i.e. 200 days are stored as 0 years, 0 months and 200 days). Some functions can't to work well with similar entered values (etc. EXTRACT QUARTER returns first quarter). We have to do normalization first with justify\_interval function (so interval 200 days will be stored as 0 years, 6 months and 20 days). From this value we take correct information about quarter:

postgres=# SELECT interval '200 days', EXTRACT(QUARTER FROM interval '300 days');

 interval | date\_part

----------+-----------

 200 days | 1

(1 row)

postgres=# SELECT justify\_interval(interval '200 days'),

 EXTRACT(QUARTER FROM justify\_interval('200 days'));

 justify\_interval | date\_part

------------------+-----------

 6 mons 20 days | 3

(1 row)

Author: Tom Lane from pgsql-bugs

Select random row

Hubert Lubaczewski showed in his [[blog](http://www.depesz.com/index.php/2007/09/16/my-thoughts-on-getting-random-row/)] problem with select random row from table. When I did one database audit I found typical but slow solution (from FAQ) ORDER BY random() LIMIT 1. This method never uses index and every similar query do: complete scan of table and sort of table. Next possible method WHERE id >= (random()\*C+1)::int LIMIT 1 is much faster, but when series has some gaps then prefer first rows after gaps. I used with success method based on using set of random numbers. Disadvantage of this method is risk of returning zero rows (then we have to call statement again):

SELECT id, ...

 FROM data

 WHERE id = ANY(ARRAY(

 SELECT (random()\*max\_id)::int

 FROM generate\_series(1,20)))

 LIMIT 1;

-- max\_id is constant equal max(id) from table + 20% reserve

-- id is primary key of table data

Use exception carefully

Trapping of exceptions has some overhead. So don't use it too often. In majority of cases this overhead is insignificant, but that can change when you are using cycles. Next two procedures add new item into table of unique records. If item exists, it returns false, else true.

CREATE OR REPLACE FUNCTION addnew1(integer)

RETURNS boolean AS $$

BEGIN

 -- test(a) .. primary key

 INSERT INTO test(a) VALUES($1);

 RETURN true;

EXCEPTION WHEN OTHERS

 RETURN false

END; $$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION addnew2(integer)

RETURNS boolean AS $$

BEGIN

 IF NOT EXISTS(SELECT a FROM test WHERE a = $1) THEN

 INSERT INTO test(a) VALUES($1);

 RETURN true;

 END IF;

 RETURN false;

END; $$ LANGUAGE plpgsql;

Procedure addnew1 is about 2 times slower than addnew2, but addnew2 is more robust. Every protected block in plpgsql creates new subtransaction. And in this case time for creating new subtrans + INSERT > SELECT + INSERT.

Fast compare variables NEW and OLD in trigger's body

We can run trigger effectively if trigger's body is executed only when any columns are changed. This test can be simple and fast with new (from 8.2) row operator *IS DISTINCT FROM*.

IF ROW(NEW.\*) IS DISTINCT FROM ROW(OLD.\*) THEN

 -- continue only if some is changed

 IF NEW.a1 IS DISTINCT FROM OLD.a1

 OR NEW.a2 IS DISTINCT FROM OLD.a2 THEN

 -- continue only if important columns are changed

 ...

 END IF;

END IF;

RETURN NEW;

or effectively:

IF ROW(NEW.a1, NEW.a2) IS DISTINCT FROM ROW(OLD.a1, OLD.a2) THEN

 -- important columns are changed

 ...

END IF;

RETURN NEW;

From 8.4 you can compare record values simply:

IF NEW IS DISTINCT FROM OLD THEN

 -- continue only if some is changed

 ...

END IF;

RETURN NEW;

Fast first n rows removing

PostgreSQL doesn't support LIMIT for UPDATE or DELETE statements. Sometimes we would to use it (for queue emulation, ...). Classic solution with IN and subquery isn't too effective:

postgres=# explain analyze delete from del where ctid in (select ctid from del limit 10 offset 0);

 QUERY PLAN

----------------------------------------------------------------------------------------------------------------------

 Merge IN Join (cost=804.79..854.94 rows=10 width=6) (actual time=63.392..63.464 rows=10 loops=1)

 Merge Cond: (public.del.ctid = "IN\_subquery".ctid)

 -> Sort (cost=804.39..829.39 rows=10000 width=6) (actual time=63.205..63.217 rows=11 loops=1)

 Sort Key: public.del.ctid

 Sort Method: quicksort Memory: 646kB

 -> Seq Scan on del (cost=0.00..140.00 rows=10000 width=6) (actual time=0.045..27.366 rows=9970 loops=1)

 -> Sort (cost=0.41..0.43 rows=10 width=6) (actual time=0.172..0.187 rows=10 loops=1)

 Sort Key: "IN\_subquery".ctid

 Sort Method: quicksort Memory: 17kB

 -> Limit (cost=0.00..0.14 rows=10 width=6) (actual time=0.021..0.127 rows=10 loops=1)

 -> Seq Scan on del (cost=0.00..140.00 rows=10000 width=6) (actual time=0.016..0.030 rows=10 loops=1)

 Total runtime: 63.661 ms

(12 rows)

Tom Lane's proposal is based on using of array:

postgres=# explain analyze delete from del where ctid = any (array(select ctid from del limit 10));

 QUERY PLAN

------------------------------------------------------------------------------------------------------------------

 Tid Scan on del (cost=0.14..40.37 rows=10 width=6) (actual time=0.200..0.237 rows=10 loops=1)

 TID Cond: (ctid = ANY ($0))

 InitPlan

 -> Limit (cost=0.00..0.14 rows=10 width=6) (actual time=0.046..0.139 rows=10 loops=1)

 -> Seq Scan on del (cost=0.00..140.00 rows=10000 width=6) (actual time=0.040..0.065 rows=10 loops=1)

 Total runtime: 0.376 ms

The core of this trick is protection from JOIN, what is not (in this case) effective as seq. scan with filter. This trick is usable in other cases. Generally we should to say, so for small N (N < 100) is predicate = ANY better than predicate IN (for PostgreSQL 8.2 and 8.3). But real efficiency depends on indexes.

Getting list of comma separated items

D. Dante Lorenso sent note about getting list from content of table:

SELECT a.id, a.name,

 array\_to\_string(ARRAY(

 SELECT b.name

 FROM b

 WHERE b.id = a.id

 ORDER BY b.name ASC

 ), ',') AS b\_names

 FROM a

 ORDER BY a.id ASC;

[table a]

 id | name

----+------

1 | one

2 | two

3 | three

4 | four

[table b]

 id | name

----+------

1 | pizza

1 | hot dog

2 | gorilla

2 | monkey

3 | apple

4 | cheese

4 | milk

4 | eggs

--result

 id | name | b\_names

----+-------+---------

1 | one | pizza,hot dog

2 | two | gorilla,monkey

3 | three | apple

4 | four | cheese,milk,eggs

Using functions pgstattuple and pgstatindex

You can find nice contrib package pgstattuple in PostgreSQL 8.1 and higher. This contrib module contains some diagnostics functions for detection of fraction of dead tuples and for detection of index's fragmentation. Using of these functions is more practical with next two table generating functions:

CREATE OR REPLACE FUNCTION print\_table\_dead\_tp(OUT table\_name varchar, OUT tuple\_count int8, OUT table\_len varchar, OUT free\_space varchar,

 OUT dead\_tuple\_percent numeric(5,2), OUT dead\_tuple\_count integer)

RETURNS SETOF RECORD AS $$

DECLARE r record; s record;

BEGIN

 FOR r IN SELECT c.oid, n.nspname || '.' || c.relname as "tablename"

 FROM pg\_catalog.pg\_class c

 LEFT JOIN

 pg\_catalog.pg\_namespace n

 ON n.oid = c.relnamespace

 WHERE c.relkind = 'r'

 AND n.nspname NOT IN ('pg\_catalog', 'pg\_toast')

 AND pg\_catalog.pg\_table\_is\_visible(c.oid)

 ORDER BY 2

 LOOP

 s := pgstattuple(r.oid);

 table\_name := r.tablename;

 tuple\_count := s.tuple\_count; dead\_tuple\_percent := s.dead\_tuple\_percent;

 dead\_tuple\_count := s.dead\_tuple\_count;

 table\_len := pg\_size\_pretty(s.table\_len);

 free\_space := pg\_size\_pretty(s.free\_space);

 RETURN NEXT;

 END LOOP;

 RETURN;

END; $$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION print\_indexes\_info(OUT index\_name varchar, OUT indexsize varchar,

 OUT leaf\_pages integer, OUT deleted\_pages integer,

 OUT leaf\_fragmentation double precision, OUT avg\_leaf\_density double precision)

RETURNS SETOF RECORD AS $$

DECLARE r record; o pgstatindex\_type;

BEGIN

 FOR r IN SELECT schemaname || '.' || indexname AS indexname FROM pg\_indexes LOOP

 BEGIN

 o := pgstatindex(r.indexname);

 index\_name := r.indexname;

 indexsize := pg\_size\_pretty(o.index\_size); deleted\_pages := o.deleted\_pages;

 leaf\_pages := o.leaf\_pages; avg\_leaf\_density := o.avg\_leaf\_density;

 leaf\_fragmentation := o.leaf\_fragmentation;

 RETURN NEXT;

 EXCEPTION WHEN OTHERS THEN

 indexsize := NULL; deleted\_pages := NULL; leaf\_fragmentation := NULL;

 index\_name := r.indexname;

 RETURN NEXT;

 END;

 END LOOP;

 RETURN;

END; $$ LANGUAGE plpgsql;

Diacritic removing

We can simply remove diacritic with calling *to\_ascii* function. There is small problem. This function doesn't support UTF8 encoding, so we have to use *convert* function.

postgres=# select to\_ascii(convert('Příliš žlutý kůň' using utf8\_to\_iso\_8859\_2),'latin2');

 to\_ascii

------------------

 Prilis zluty kun

(1 row)

Note for 7.4, name of conversion is "utf\_8\_to\_iso\_8859\_2". For 8.3 and 9.0 we have to use small workaround:

CREATE FUNCTION to\_ascii(bytea, name)

RETURNS text AS 'to\_ascii\_encname' LANGUAGE internal;

SELECT to\_ascii(convert\_to('Příliš žlutý kůň', 'latin2'),'latin2');

Predicate IN optimalization

I found note [pg\_performance](http://archives.postgresql.org/pgsql-performance/) about optimalization possibility of IN predicate when list of values is longer than eighty numbers. For longer list is better create constant subqueries with using multi values:

SELECT \*

 FROM tab

 WHERE x IN (1,2,3,..n); -- n > 70

-- faster case

SELECT \*

 FROM tab

 WHERE x IN (VALUES(10),(20));

Using VALUES is faster for bigger number of items, so don't use it for small set of values.

BAR code validation

SQL Guru Joe Celko described [DBAzinu](http://www.dbazine.com/ofinterest/oi-articles/celko36) interesting technique for BAR code validation with multi values.

CREATE FUNCTION barcode\_sum(text)

RETURNS bigint AS

$$

 SELECT ABS(SUM(CAST(SUBSTRING($1 FROM Weights.seq FOR 1) AS INTEGER) \* Weights.wgt))

 FROM (VALUES (CAST(1 AS INTEGER), CAST(-1 AS INTEGER)),

 (2, +1), (3, -1), (4, +1), (5, -1),

 (6, +1), (7, -1), (8, +1), (9, -1), (10, +1),

 (11,-1), (12, +1)) AS weights(seq, wgt)

$$ LANGUAGE sql;

Our function can be used like:

CREATE TABLE products(

 barcode char(13) NOT NULL

 CONSTRAINT valid\_checkdigit

 check (MOD(barcode\_sum(barcode),10) = CAST(substring(barcode from 13 for 1) AS integer))

 CONSTRAINT all\_numeric\_checkdigit CHECK(barcode NOT SIMILAR TO '%[^0-9]%')

);

INSERT INTO products VALUES('2837232811227');

This trick is based on behave of char type. Shorted text than 13 chars is filled with spaces, so test NOT SIMILAR ensure correct number of digits.

Using hash functions for ensuring uniqueness of texts

Classic unique index isn't well solution for longer texts. It's natural using some of hash functions. Output of these functions is 32 char long text. This is hexadecimal number, so we can cut it by casting to bytea type. Attention: hash doesn't ensure 100% uniqueness.

root=# CREATE TABLE test(a text);

CREATE TABLE

root=# CREATE UNIQUE INDEX uidx ON test((decode(md5(a),'hex')));

CREATE INDEX

root=# CREATE UNIQUE INDEX uidx ON test((decode(md5(lower(replace(a,' ',''))),'hex')));

CREATE INDEX

root=# INSERT INTO test VALUES('příliš žluťoučký kůň');

INSERT 0 1

root=# INSERT INTO test VALUES('příliš žluťoučký kůň ');

ERROR: duplicate key violates unique constraint "uidx"

root=# INSERT INTO test VALUES('Příliš žluťoučký kůň');

ERROR: duplicate key violates unique constraint "uidx"

-- less risk, using two hash functions

-- CREATE UNIQUE INDEX uidx ON TEST((decode(md5(a),'hex')),(hashtext(a)));

Getting OID of function

Every database object in PostgreSQL has Oid - unique identifier. We can find it in database system's tables or we can get it with casting:

CREATE OR REPLACE FUNCTION a(integer, varchar) RETURNS void AS

$$

 BEGIN

 END;

$$ LANGUAGE plpgsql;

root=# SELECT 'a'::regproc::int;

ERROR: more than one function named "a"

root=# SELECT 'a(integer, varchar)'::regprocedure::int;

 int4

-------

 57507

(1 row)

root=# SELECT 57507::regprocedure;

 regprocedure

------------------------------

 a(integer,character varying)

(1 row)

Table's Oid we could to get with cast to regclass type:

SELECT 'oo'::regclass::int;

General array sort

Very nice exhibition of PostgreSQL's features is general procedure for sorting by David Fetter.

CREATE OR REPLACE FUNCTION array\_sort (ANYARRAY)

RETURNS ANYARRAY LANGUAGE SQL

AS $$

SELECT ARRAY(

 SELECT $1[s.i] AS "foo"

 FROM

 generate\_series(array\_lower($1,1), array\_upper($1,1)) AS s(i)

 ORDER BY foo

);

$$;

little bit faster shorter modernised variant:

CREATE OR REPLACE FUNCTION array\_sort (ANYARRAY)

RETURNS ANYARRAY LANGUAGE SQL

AS $$

SELECT ARRAY(SELECT unnest($1) ORDER BY 1)

$$

or

-- by Craig Ringer

CREATE OR REPLACE FUNCTION array\_sort (ANYARRAY)

RETURNS ANYARRAY LANGUAGE SQL

AS $$

SELECT array\_agg(x ORDER BY x) FROM unnest($1) x;

$$

If you need really fast sort of int arrays, then you have to look on intarray contrib module.

Array transformation in PL/Perl

PL/Perl doesn't support OUT arrays. We can use undocumented procedure \_plperl\_to\_pg\_array:

CREATE OR REPLACE FUNCTION foo(OUT a integer[], OUT b integer[]) AS $$

 return { a=> \_plperl\_to\_pg\_array([1,2,3]),

 b=> \_plperl\_to\_pg\_array([4,5,6]) };

UPC code validation

This is sample of validation of code with control number in Perl (PL/perl). I used this function for compare with PL/pgSQL and C languages. Perl is about 20% faster, but C is 10 times faster than Perl.

CREATE OR REPLACE FUNCTION ean\_perl (

 TEXT

) RETURNS boolean AS $\_$

 my $ean = length $\_[0] == 12 ? "0$\_[0]" : $\_[0];

 # Make sure we really have an EAN.

 return 'false' unless $ean =~ /^\d{13}$/;

 my @nums = split '', $ean;

 return 10 - (

 # Sum even numerals.

 ( ( $nums[1] + $nums[3] + $nums[5] + $nums[7] + $nums[9]

 + $nums[11]

 ) \* 3 # Multiply total by 3.

 # Add odd numerals except for checksum (12).

 ) + $nums[0] + $nums[2] + $nums[4] + $nums[6] + $nums[8] + $nums[10]

 # Compare to the checksum.

 ) % 10 == $nums[12] ? 'true' : 'false';

$\_$ LANGUAGE plperl immutable;

I found interesting solution by Joe Celko in [article](http://www.dbazine.com/ofinterest/oi-articles/celko36). After rewriting to PostgreSQL:

SELECT (

 SELECT mod(

 sum(

 CAST(substring('2837232811227' from s.seq for 1) AS integer) \*

 CASE mod(s.seq,2) WHEN 0 THEN 1 ELSE -1 END),

 10) FROM generate\_series(1,12) as s(seq)) =

 CAST(substring('2837232811227' FROM 13 FOR 1) AS integer);

Show only every n row

When we would to skip some rows from table, we should to use temporary sequences and modulo function:

 CREATE TEMP SEQUENCE number;

 SELECT \* FROM ( SELECT \*, nextval('number') AS number FROM datatable ) foo

 WHERE foo.number % 5 = 0;

With 8.4 this can be done without the use of a temporary sequence

 SELECT \* FROM ( SELECT \*, row\_number() over () AS number FROM datatable ) foo

 WHERE foo.number % 5 = 0;

Array to table

We can simply unpack array with generate\_series function:

 SELECT (ARRAY[1,3,4,6,7])[idx.n] FROM generate\_series(1,7) idx(n);

CREATE OR REPLACE FUNCTION unpack(anyarray)

RETURNS SETOF anyelement AS $$

SELECT $1[i]

 FROM generate\_series(array\_lower($1,1),

 array\_upper($1,1)) g(i);

$$ LANGUAGE sql STRICT IMMUTABLE;

postgres=# select unpack(array['a','b','c']);

 unpack

--------

 a

 b

 c

(3 rows)

With 8.4, we can use the unnest()-function:

test=\*# select unnest(array['a','b','c']);

 unnest

--------

 a

 b

 c

(3 rows)

Additionally, we can extend the unnest() function to return one row with a default value if the array is empty or null:

create or replace function unnest(anyarray, anyelement)

returns setof anyelement as $$

select unnest(case when array\_upper($1, 1) > 0 then $1 else array[$2] end);

$$ language sql immutable;

gp=> create temp table foo (x text[], y int) distributed by (y);

CREATE TABLE

gp=> insert into foo values

(array['a','b','c'], 1),

(null, 2),

('{}'::text[],3);

INSERT 0 3

gp=> select \*,array\_upper(x,1) as size from foo;

 x | y | size

---------+---+------

 {} | 3 |

 | 2 |

 {a,b,c} | 1 | 3

(3 rows)

gp=> select unnest(x),y from foo;

 unnest | y

--------+---

 a | 1

 b | 1

 c | 1

(3 rows)

gp=> select unnest(x,'empty'),y from foo;

 unnest | y

--------+---

 empty | 3

 empty | 2

 a | 1

 b | 1

 c | 1

(5 rows)

LIKE optimalization

PostgreSQL use index for LIKE only when:

* pattern doesn't start with % and \_ symbols,
* database cluster is initialized with C locale.

Last constraint we can pass by special index:

CREATE INDEX like\_index ON people(surname varchar\_pattern\_ops);

When we work wit pattern like %some (for internet domain searching) we should to use trick:

* create functional index on mirrored values.
* search with mirrored pattern

PostgreSQL doesn't allow reverse (mirror) function. We can use rvrs function from [Orafunc](http://postgres.cz/wiki/Oracle_functionality_%28en%29#PLVstr_string_and_PLVchr_char_functions) package or we can use PL/Perl.

CREATE OR REPLACE FUNCTION reverse(varchar) RETURNS varchar AS $$

 $reversed = reverse $\_[0];

 return $reversed;

$$ LANGUAGE plperlu IMMUTABLE;

CREATE INDEX rev\_email ON users( (reverse(email) ) varchar\_pattern\_ops );

SELECT \* FROM \_users WHERE reverse(email) LIKE reverse ('%.cz');

Without plperl:

CREATE OR REPLACE FUNCTION reverse(TEXT) RETURNS TEXT AS $$

 SELECT

 array\_to\_string(

 ARRAY

 ( SELECT substring($1, s.i,1) FROM generate\_series(length($1), 1, -1) AS s(i) ),

 '');

$$ LANGUAGE SQL IMMUTABLE;

Note: PostgreSQL 9.1 has buildin function for reverse string.

Deferred constraints

PostgreSQL run all constrains immediately. SQL know deferred constraints, that allows delay tests to time closely to commit. PostgreSQL hasn't full support. You can deferred only referential integrity constraints. But PostgreSQL supports undocumented deferred triggers, so we should use it. Syntax is similar to trigger definition:

 CREATE CONSTRAINT TRIGGER sc

 AFTER INSERT

 ON fieldtrip\_students

 INITIALLY DEFERRED

 FOR EACH ROW EXECUTE PROCEDURE field\_trip();

Removing of duplicate rows

Any table created without parameter *WITHOUT OIDS* has row unique identifier OID. We can use it for removing duplicate rows:

 DELETE FROM people WHERE

 NOT oid IN (SELECT MIN(oid) FROM people GROUP BY name, surname);

But new tables are created with this parameter usually. So we can't use OID. Every PostgreSQL row has another unique identifier - ctid. For ctid type we have not defined any aggregate functions, we have to use correlated subquery. I found this trick in Greg Mullane presentation.

 DELETE FROM lidi WHERE

 NOT ctid = (SELECT ctid FROM lidi l WHERE

 prijmeni=l.prijmeni AND jmeno=l.jmeno LIMIT 1);

Select first n rows from group

 SELECT \* FROM people WHERE id IN (

 SELECT id FROM people s

 WHERE people.category = s.category

 ORDER BY age LIMIT 2)

 ORDER BY category, age;

Other solution without correlated subquery is [SELF JOIN](http://www.xaprb.com/blog/2007/03/14/how-to-find-the-max-row-per-group-in-sql-without-subqueries/). I have to look on problem from different perspective. I have to ask on persons for which only one person in group is older.

SELECT s1.\*

 FROM people s1

 LEFT JOIN

 people s2

 ON s1.category = s2.category AND s1.age < s2.age

 GROUP BY s1.id, s1.category

 HAVING COUNT(s2.id) <= 1

 ORDER BY s1.category, COUNT(s2.id);

Cast to varchar

Every data type in PostreSQL has input and output functions. We can use these functions for support casting to varchar (if we miss it). Don't use this trick on PostgreSQL 8.3 and higher

 testdb011=# SELECT '(1,1)'::point::varchar;

 ERROR: cannot cast type point to character varying

but

 testdb011=# SELECT textin(point\_out('(1,1)'::point))::varchar;

 textin

 --------

 (1,1)

 (1 row)

Create Aggregates

I needed to be able to do a fast

SELECT count(\*) FROM sometable WHERE field='x'

instead of a simple

SELECT count (\*) FROM sometable

Both of the above queries initiate sequential scans on a table (which takes too much time). Here is what I created:

CREATE TABLE aggregate AS

 (SELECT result,count(result) AS count FROM original\_table

 GROUP BY result ORDER BY count DESC);

Further, my data contains many one-offs which I don't want to put into my aggregate table (I assume that if a value isn't in my aggregate table that it's too small to be useful)... here is what I do to leave those values out:

CREATE TABLE aggregate AS

 (SELECT \* FROM

 (SELECT result,count(result) AS count FROM original\_table

 GROUP BY result ORDER BY count DESC)

 AS countquery WHERE count > 3);

This method is fast enough. Because I have an index on the "result" field, It runs in under a minute against 3 million records. It takes approximately the same amount of time to generate counts for all the result values as it does to count the total number of records.

Now I can do a simple (very fast) query to get a count:

SELECT \* FROM aggregate where result='xxxxx'

PostgreSQL SQL Tricks II

SQL is a language where one task can be solved multiple ways with different efficiency. You can see effective use of PostgreSQL's possibilities on this page.

Google AdSense

[Older tricks](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_I) [Newer tricks](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III)

**Obsah**

 [[skrýt](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II)]

* [1 subscript shift to zero](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#subscript_shift_to_zero)
* [2 Record evolution](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Record_evolution)
* [3 Conversion between hex and dec numbers](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Conversion_between_hex_and_dec_numbers)
* [4 Fast interval (of time or ip addresses) searching with spatial indexes](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Fast_interval_.28of_time_or_ip_addresses.29_searching_with_spatial_indexes)
* [5 Short form for insert ROW variables to table](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Short_form_for_insert_ROW_variables_to_table)
* [6 ALTER TABLE ALTER COLUMN USING](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#ALTER_TABLE_ALTER_COLUMN_USING)
* [7 Quote\_ident for schema.name](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Quote_ident_for_schema.name)
* [8 Fast searching of longer prefix](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Fast_searching_of_longer_prefix)
* [9 Get count of substrings in string](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Get_count_of_substrings_in_string)
* [10 Timestamp recalculation for different timezone and its indexing](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Timestamp_recalculation_for_different_timezone_and_its_indexing)
* [11 Using IMMUTABLE functions as hints for the optimizer](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Using_IMMUTABLE_functions_as_hints_for_the_optimizer)
* [12 Autonomous transaction in PostgreSQL](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Autonomous_transaction_in_PostgreSQL)
* [13 Optimization of repetitive assignment to varchar variable with arrays](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Optimization_of_repetitive_assignment_to_varchar_variable_with_arrays)
* [14 Use IS DISTINCT FROM without COALESCE](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Use_IS_DISTINCT_FROM_without_COALESCE)
* [15 Port to Oracle, problems with to\_date function](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Port_to_Oracle.2C_problems_with_to_date_function)
* [16 Identification and correction of wrong chars in PostgreSQL dump](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Identification_and_correction_of_wrong_chars_in_PostgreSQL_dump)
* [17 Be carefull when using the NOT IN operator with NULL](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Be_carefull_when_using_the_NOT_IN_operator_with_NULL)
* [18 Fast specification of first rows of some table field](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Fast_specification_of_first_rows_of_some_table_field)
* [19 An employee with highest compensation](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#An_employee_with_highest_compensation)
* [20 Any other session variables](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Any_other_session_variables)
* [21 xpath function indexing](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#xpath_function_indexing)
* [22 psql together with less](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#psql_together_with_less)
* [23 Intersection of arrays](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Intersection_of_arrays)
* [24 VOID function in SQL language](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#VOID_function_in_SQL_language)
* [25 Fast select of next row](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#Fast_select_of_next_row)

subscript shift to zero

Implementation of these features is simple in C and it's little bit slow in PL/pgSQL (based on iterated shift). However, we can use the SQL parser capability to recognize an array range:

CREATE OR REPLACE FUNCTION shift\_idx(anyarray)

RETURNS anyarray AS $$

DECLARE

 mi int := array\_upper($1,1);

 offset int := array\_lower($1,1);

BEGIN

 RETURN '[0:' || mi - offset || '] = ' || $1::varchar;

END

$$ LANGUAGE plpgsql IMMUTABLE STRICT;

CREATE FUNCTION

postgres=# SELECT shift\_idx(array[1,2,3]);

 shift\_idx

---------------

 [0:2]={1,2,3}

(1 row)

Author: Pavel Stěhule

Record evolution

When we have functions that return a record but cannot use derived tables, we must use following syntax:

CREATE FUNCTION foo(OUT a int, OUT b int, IN c int)

RETURNS record AS $$

BEGIN

 a := c + 1; b := c + 2;

 RETURN;

END;

$$ LANGUAGE plpgsql;

CREATE TABLE t(c int);

INSERT INTO t VALUES(10),(20);

postgres=# SELECT c, foo(c) FROM t;

 c | foo

----+---------

 10 | (11,12)

 20 | (21,22)

(2 rows)

postgres=# SELECT c, (foo(c)).\* FROM t;

 c | a | b

----+----+----

 10 | 11 | 12

 20 | 21 | 22

(2 rows)

Conversion between hex and dec numbers

source: <http://www.varlena.com/GeneralBits/104.php>.

=# select to\_hex(11);

 to\_hex

 --------

 b

 (1 row)

postgres=# create or replace function to\_dec(text)

 returns integer as $$

 declare r int;

 begin

 execute E'select x\''||$1|| E'\'::integer' into r;

 return r;

 end

 $$ language plpgsql;

CREATE FUNCTION

postgres=# select to\_dec('ff');

 to\_dec

--------

 255

(1 row)

Faster variant (Michael Glaesemann):

CREATE FUNCTION hex2dec(in\_hex TEXT)

RETURNS INT

IMMUTABLE STRICT LANGUAGE sql AS $body$

 SELECT CAST(CAST(('x' || CAST($1 AS text)) AS bit(8)) AS INT);

$body$;

Variant that is able to handle arbitrary length input (output as varchar, adapt if needed):

CREATE OR REPLACE FUNCTION hex2dec (varchar)

RETURNS varchar as

$$

DECLARE

 \_x numeric;

 \_count int;

 \_digit int;

BEGIN

 \_x := 0;

 for \_count in 1..length($1) loop

 EXECUTE E'SELECT x\''||substring($1 from \_count for 1)|| E'\'::integer' INTO \_digit;

 \_x := \_x \* 16 + \_digit ;

 end loop;

 return \_x::varchar;

end

;

$$ language plpgsql immutable

;

Fast interval (of time or ip addresses) searching with spatial indexes

Searching intervals is usually slow, because the optimizer don't use an index. The reason lies in the dependency between the start and end columns. One solution is based on spatial indexes: it allows working with two dependent values as if they were a single value:

postgres=# EXPLAIN ANALYZE SELECT \* FROM testip WHERE 19999999 BETWEEN startip AND endip;

 QUERY PLAN

----------------------------------------------------------------

 Seq Scan on testip (cost=0.00..19902.00 rows=200814 width=12) (actual time=3.457..434.218 rows=1 loops=1)

 Filter: ((19999999 >= startip) AND (19999999 <= endip))

 Total runtime: 434.299 ms

(3 rows)

Time: 435,865 ms

postgres=# CREATE INDEX ggg ON testip USING gist ((box(point(startip,startip),point(endip,endip))) box\_ops);

CREATE INDEX

Time: 75530,079 ms

postgres=# EXPLAIN ANALYZE

 SELECT \*

 FROM testip

 WHERE box(point(startip,startip),point(endip,endip)) @> box(point (19999999,19999999), point(19999999,19999999));

 QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

 Bitmap Heap Scan on testip (cost=60.50..2550.14 rows=1000 width=12) (actual time=0.169..0.172 rows=1 loops=1)

 Recheck Cond: (box(point((startip)::double precision, (startip)::double precision), point((endip)::double precision, (endip)::double precision)) @> '(19999999,19999999),(19999999,19999999)'::box)

 -> Bitmap Index Scan on ggg (cost=0.00..60.25 rows=1000 width=0) (actual time=0.152..0.152 rows=1 loops=1)

 Index Cond: (box(point((startip)::double precision, (startip)::double precision), point((endip)::double precision, (endip)::double precision)) @> '(19999999,19999999),(19999999,19999999)'::box)

 Total runtime: 0.285 ms

(5 rows)

Time: 2,805 ms

Short form for insert ROW variables to table

postgres=# CREATE TABLE foo(a integer, b integer);

CREATE TABLE

postgres=# CREATE OR REPLACE FUNCTION fx()

 RETURNS void as $$

 DECLARE r foo;

 BEGIN

 SELECT INTO r \* FROM foo;

 INSERT INTO foo VALUES(r.\*);

 RETURN;

 END;

 $$ LANGUAGE plpgsql;

CREATE FUNCTION

postgres=# SELECT fx();

 fx

----

(1 row)

ALTER TABLE ALTER COLUMN USING

I hadn't known about the USING clause in ALTER TABLE ALTER COLUMN. I thought the old type had to be castable to the new type. But it's not true. Czech boolean literals cannot be cast directly to PostgreSQL's bool type, but, with the USING clause, I can do an ALTER COLUMN TYPE from varchar to bool:

postgres=# CREATE TABLE foo(a varchar);

CREATE TABLE

postgres=# INSERT INTO foo VALUES ('ano'); -- czech yes

INSERT 0 1

postgres=# ALTER TABLE foo ALTER COLUMN a TYPE boolean ;

ERROR: column "a" cannot be cast to type "pg\_catalog.bool"

postgres=# ALTER TABLE foo

 ALTER COLUMN a TYPE boolean

 USING CASE a

 WHEN 'ano' THEN true

 ELSE false END;

ALTER TABLE

postgres=# SELECT \* FROM foo;

 a

---

 t

(1 row)

Quote\_ident for schema.name

Using single quotes is one protection against SQL injection. The function *quote\_ident* checks its argument and, when the argument contains problematic chars, it returns the argument surrounded by single quotes. It's quite easy and useful. Problem is with schema.name pair, because the dot is a watched char:

postgres=# select quote\_ident('public.foo');

 quote\_ident

--------------

 "public.foo"

(1 row)

postgres=# select \* from "public.foo";

ERROR: relation "public.foo" does not exist

postgres=# select \* from public."foo";

 a

---

(0 rows)

postgres=# select \* from "public"."foo";

 a

---

(0 rows)

We can get around this limitation with custom functions:

CREATE OR REPLACE FUNCTION quote\_array(text[])

RETURNS text AS $$

SELECT array\_to\_string(array(SELECT quote\_ident($1[i])

 FROM generate\_series(1, array\_upper($1,1)) g(i)),

 '.')

$$ LANGUAGE SQL IMMUTABLE;

CREATE OR REPLACE FUNCTION quote\_schema\_ident(text)

RETURNS text AS $$

SELECT quote\_array(string\_to\_array($1,'.'))

$$ LANGUAGE SQL IMMUTABLE;

postgres=# select quote\_schema\_ident('public.foo tab');

 quote\_schema\_ident

--------------------

 public."foo tab"

(1 row)

Autor: [Pavel Stehule](http://postgres.cz/wiki/Pavel_Stehule).

Fast searching of longer prefix

Searching longest prefix is common task - mainly in communications. We search for the longest prefix for some number from from set of prefixes. I solved this task some years ago with stored procedure and modified method of halving of interval. Maybe little bit slower but simple and robust method was described by Hubert Lubaciewski ([Depesz](http://www.depesz.com/index.php/2008/03/04/searching-for-longest-prefix/#more-1176)).

--table contains about 5000 prefixes

postgres=# select \* from prefixesx limit 5;

 costcode\_name | costcode

--------------------+----------

 Afghanistan | 93

 Afghanistan Mobile | 9370

 Afghanistan Mobile | 9379

 Afghanistan Mobile | 9380

 Alaska (USA) | 1907

(5 rows)

Simple but slow solution with LIKE:

postgres=# SELECT \*

 FROM prefixesx

 WHERE '420724181000' LIKE costcode || '%'

 ORDER BY length(costcode) DESC

 LIMIT 1;

 costcode\_name | costcode

---------------------------------+----------

 Czech Republic Mobile - EuroTel | 42072

Fast solution is generating all possible prefixes from number and search this prefixes:

CREATE OR REPLACE FUNCTION prefixes(varchar)

RETURNS varchar[] AS $$

SELECT ARRAY(SELECT substring($1 FROM 1 FOR i)

 FROM generate\_series(1, length($1)) g(i))::varchar[];

$$ LANGUAGE sql IMMUTABLE;

and use query:

postgres=# SELECT \*

 FROM prefixesx

 WHERE costcode = ANY (prefixes('420724191000'))

 ORDER BY length(costcode) DESC

 LIMIT 1;

 costcode\_name | costcode

---------------------------------+----------

 Czech Republic Mobile - EuroTel | 42072

On 5000 rows is this query 4 times faster than LIKE.

Little bit slower is query with LIKE and LIKE firstchar || %

SELECT \*

 FROM prefixesx

 WHERE '420724181000' LIKE costcode || '%'

 AND costcode LIKE substring('420724191000' FROM 1 FOR 1) || '%'

 ORDER BY length(costcode) DESC

 LIMIT 1;

Get count of substrings in string

This function is based on popular trick - using replace and length function without cycle, that is typical for interprets where built-in functions are faster than iteration:

CREATE OR REPLACE FUNCTION CountInString(text,text)

RETURNS integer AS $$

 SELECT(Length($1) - Length(REPLACE($1, $2, ''))) / Length($2) ;

$$ LANGUAGE SQL IMMUTABLE;

Sent by Rodrigo E. De León Plicet. Another my (array based) solution:

CREATE OR REPLACE FUNCTION CountInString(text, text)

RETURNS integer AS $$

 SELECT Array\_upper(String\_to\_array($1,$2),1) - 1;

$$ LANGUAGE SQL IMMUTABLE;

Timestamp recalculation for different timezone and its indexing

The simplest specification of time at some timezone is using AT TIME ZONE operator. We can't indexing timestamp with timezone, because we can change current timezone, so we can create index only for timestamp at some specified timezone. Source: pgsql-general, Tom Lane

CREATE INDEX created\_tz\_paris ON data((created\_on AT TIME ZONE 'Europe/Paris'));

Using IMMUTABLE functions as hints for the optimizer

For highly specific data distributions, we can have a problem with prediction and generation of suboptimal execution plans. These problems we can solve by breaking a SQL query into more dependent queries and evaluating step by step. The following queries use real results (not estimations) precedent queries, and queries are executed optimally. Tom Lane's proposal is based on using IMMUTABLE functions that are evaluated before generating an execution plan:

CREATE OR REPLACE FUNCTION getTypesLieuFromTheme(codeTheme text)

RETURNS text[] AS $f$

SELECT ARRAY(SELECT codetylieu::text FROM rubtylieu WHERE codeth = $1);

$f$ LANGUAGE SQL IMMUTABLE;

in query

SELECT ...

 WHERE ... AND gl.codetylieu = ANY(getTypesLieuFromTheme('RES'))

look execution plan doesn't contain function call and query to table rubtylieu too

-> Bitmap Heap Scan on genrelieu gl (cost=2.23..6.22 rows=88 width=4) (actual time=0.022..0.075 rows=88 loops=1)

 Recheck Cond: ((codetylieu)::text = ANY ('{RES}'::text[]))

 -> Bitmap Index Scan on ind\_genrelieu2 (cost=0.00..2.21 rows=88 width=0) (actual time=0.016..0.016 rows=88 loops=1)

 Index Cond: ((codetylieu)::text = ANY ('{RES}'::text[]))

Use this trick as last step. Test, please, by increasing the statistic number beforehand.

Autonomous transaction in PostgreSQL

Autonomous transaction is usable feature mainly for logging. This feature isn't supported in PostgreSQL yet. With untrusted languages we are able to emulate it. You can see solution by Jon Roberts.

CREATE OR REPLACE FUNCTION fn\_log\_error(p\_function varchar, p\_location int, p\_error varchar)

RETURNS void AS $$

DECLARE

 v\_sql varchar;

 v\_return varchar;

 v\_error varchar;

BEGIN

 PERFORM dblink\_connect('connection\_name', 'dbname=...');

 v\_sql := 'INSERT INTO error\_log (function\_name, location, error\_message, error\_time) '

 || 'VALUES (''' || p\_function\_name || ''', '

 || p\_location || ', ''' || p\_error || ''', clock\_timestamp())';

 SELECT INTO v\_return \*

 FROM dblink\_exec('connection\_name', v\_sql, false);

 --get the error message

 SELECT INTO v\_error \*

 FROM dblink\_error\_message('connection\_name');

 IF position('ERROR' in v\_error) > 0 OR position('WARNING' in v\_error) > 0 THEN

 RAISE EXCEPTION '%', v\_error;

 END IF;

 PERFORM dblink\_disconnect('connection\_name');

EXCEPTION

 WHEN others THEN

 PERFORM dblink\_disconnect('connection\_name');

 RAISE EXCEPTION '(%)', SQLERRM;

END;

$$ LANGUAGE plpgsql SECURITY DEFINER;

Optimization of repetitive assignment to varchar variable with arrays

plpgsql isn't a good language for intensive non SQL operations. It's best as glue for SQL statements or for prototyping. But sometimes plperl or plpython can be significantly faster. Plpgsql does not like cumulative iterations over varchar or array variables. When we can't use Perl, .. we could use SQL:

CREATE OR REPLACE FUNCTION SlowList(int) -- slow function, usable for N <= 100

RETURNS varchar AS $$

DECLARE s varchar = '';

BEGIN

 FOR i IN 1..$1 LOOP

 s := '<item>' || i || '</item>'; -- slow is s := s || ..

 END LOOP;

 RETURN s;

END; $$ LANGUAGE plpgsql IMMUTABLE;

CREATE OR REPLACE FUNCTON FastList(int) -- fast function

RETURNS varchar AS $$

BEGIN

 RETURN array\_to\_string(ARRAY(SELECT '<item>' || i || '</item>'

 FROM generate\_series(1, $1) g(i)),

 '');

END; $$ LANGUAGE plpgsql IMMUTABLE;

For N < 100 execution time is less then 4ms (so isn't any reason for dirty trick). For bigger N is execution time different (SlowList(10000) - 4000ms, FastList(10000) - 52ms). Author - Pavel Stěhule.

Use IS DISTINCT FROM without COALESCE

Sim Zacks reported slow queries after migration to 8.2.4. This version has problem with prediction of result of expression COALESCE(column, false) = false, and then execution plan is suboptimal. After correction PostgreSQL choose optimal execution plan.

SELECT ...

 FROM some\_tab

 LEFT JOIN

 some\_tab2

 ON sometab2.col IS DISTINCT FROM true; --> join false or NULL

So don't use COALESCE in WHERE clause

Next week somebody reported similar problem:

-- execution plan is suboptimal

((COALESCE((at\_type)::integer, 1) = 1) AND (COALESCE(at\_language, 0::numeric) = 0::numeric))

-- execution plan is optimal

(at\_type = 1 or at\_type is null) AND (at\_language = 0 or at\_language is null)

Port to Oracle, problems with to\_date function

Peter Eisentraut reported incompatibility to\_date function between PostreSQL and Oracle.

SQL> select to\_date('31-DEC-200700:00:00', 'dd-mon-yyyy hh24:mi:ss') from

dual;

TO\_DATE('

---------

31-DEC-07

On PostgreSQL:

select to\_date('31-DEC-200700:00:00', 'dd-mon-yyyy hh24:mi:ss');

 to\_date

--------------

 200700-12-31

Oracle ignores spaces in format string, but PostgreSQL requires exact format. This behavior can be changed in custom function that drops spaces (Author: Jon Roberts):

CREATE OR REPLACE FUNCTION fn\_to\_date(p\_date varchar, p\_format varchar)

RETURNS timestamp AS $$

 SELECT to\_timestamp(replace($1, ' ', ''), replace($2, ' ', ''));

$$ LANGUAGE SQL STRICT IMMUTABLE;

# select fn\_to\_date('31-DEC-200700:00:00', 'dd-mon-yyyy hh24:mi:ss');

 fn\_to\_date

---------------------

 2007-12-31 00:00:00

(1 row)

Identification and correction of wrong chars in PostgreSQL dump

8.2 is stricter than 8.1 for correct chars in dump. Reason is in SQL injection security fix (via incorrect UTF8 chars). This change of behavior can cause problems with migration. Fix is simple. We have to find wrong char and fix it.

$ iconv -f utf-8 -t utf-8 dump.sql > /dev/null

iconv: illegal input sequence at position 1000

$ head -c 1010 dump.sql | tail -c 20 | od -c

0000000 h . B u t i s n 222 t i t

0000020 h i s ?

0000024

Source: [http://lca2007.linux.org.au/att\_data/Miniconfs(2f)PostgreSQL/attachments/professional.pdf](http://lca2007.linux.org.au/att_data/Miniconfs%282f%29PostgreSQL/attachments/professional.pdf)

Be carefull when using the NOT IN operator with NULL

Some SQL developer dislike NULL. Other like it. I like it, too. NULL is necessary and saves a lot of work. NULL in a result expression clearly signals some problem in the input data. People who dislike NULL don't understand it and use the following argument, citing an "anomaly":

postgres=# SELECT \*

 FROM (VALUES(10),(20),(30)) v(a)

 WHERE a IN (10, 20, NULL); --> all is ok

 a

----

 10

 20

(2 rows)

postgres=# SELECT \*

 FROM (VALUES(10),(20),(30)) v(a)

 WHERE a NOT IN (10, 20, NULL);

 a

---

(0 rows) --> anomaly, expects 30

But it isn't an anomaly. The NOT IN operator checks each item of a list, because the left-hand value must be different than all of them. Since we can never have a comparison with NULL that results in anything but NULL, no value can fulfill this predicate.

Fast specification of first rows of some table field

My task is specific. I have address book with 100 000 items and I have to get all first chars of surnames. This task is one from a few tasks, where correlated subquery is faster than everything else. Because seq scan and agg function need scan all 100 000 records, but correlated subquery need only 32 access to functional index:

SELECT \*

 FROM (

 SELECT substring('abcdefghijklmnopqrstuvwxyzěščřžýáíé'

 FROM i For 1)

 FROM generate\_series(1,32) g(i)

 ) p(onechar)

 WHERE EXISTS(

 SELECT 1

 FROM address\_book

 WHERE substring(surname from 1 for 1) = p.onechar

 and active

 );

Version for support czech char CH (and some others: úůňľó):

 SELECT \*

 FROM

 (

 SELECT substring('abcdefghijklmnopqrstuvwxyzěščřžýáíéúůňľó' FROM i For 1) FROM generate\_series(1,40) g(i) UNION SELECT 'ch'

 ) p(onechar)

 WHERE

 EXISTS( SELECT 1 FROM a\_osoba WHERE lower(substring(prijmeni from 1 for char\_length(p.onechar))) = p.onechar );

An employee with highest compensation

It's typical school query: select the highest-paid employees in some group. This query can be solved with derivated table or with correlated subquery. We could use the nonstandard clause DISTINCT ON in PostgreSQL:

postgres=# SELECT \* FROM employees ;

 name | surname | department | payment

-----------+-----------+------------+-------

 Pavel | Stehule | 1 | 10000

 Zdenek | Stehule | 1 | 9000

 Vladimira | Stehulova | 2 | 9000

(3 rows)

postgres=# SELECT DISTINCT ON department \*

 FROM employees

 ORDER BY department, payment DESC;

 name | surname | department | payment

-----------+-----------+------------+-------

 Pavel | Stehule | 1 | 10000

 Vladimira | Stehulova | 2 | 9000

(2 rows)

Attention, DISTINCT ON isn't portable. A big disadvantage of this solution is that it shows only one person from every department, even when more employees has the same, highest, payment, yielding an incomplete result.

Any other session variables

PostgreSQL doesn't support server session variables. This mean, so we have to write stored procedures more often, because there are variables. Some substitution is module variables. These variables has to allowed in configuration. Sent by Andreas Kretschmer:

- define in your postgresql.conf:

 custom\_variable\_classes = 'myvar'

- use within psql:

test=# set myvar.benutzer = 'foo';

SET

test=\*# select \* from foo;

 id | name

----+------

 1 | foo

 2 | bar

(2 rows)

test=\*# select \* from foo where name=current\_setting('myvar.benutzer');

 id | name

----+------

 1 | foo

(1 row)

test=\*# set myvar.benutzer = 'none';

SET

test=\*# select \* from foo where name=current\_setting('myvar.benutzer');

 id | name

----+------

(0 rows)

xpath function indexing

8.3 has integrated xpath function. There is gap in XML support, because XML type isn't supported with GIST or GIN index. So xpath function returns array of xml values. But we can write custom casting to int array:

CREATE OR REPLACE FUNCTION xml\_list\_to\_int(xml[])

RETURNS int[] AS $$

SELECT ARRAY(SELECT to\_number(($1[i])::text,'999999.99')::int

 FROM generate\_series(1, array\_upper($1,1)) g(i))

$$ LANGUAGE SQL IMMUTABLE;

CREATE CAST (xml[] AS int[]) WITH FUNCTION xml\_list\_to\_int(xml[]);

-- array of integers are supported with GIST

CREATE INDEX fx ON foo USING

GIN((xpath('//id/text()',order\_in\_xml)::int[]));

psql together with less

psql has paging support, but it isn't comparable with the options offered by less. Here is example on how to setup psql to use less as the pager. Author: Merlin Moncure

#to profile

export PAGER=less

export LESS="-iMSx4 -FX"

#to .psqlrc

\timing

\pset pager always

Intersection of arrays

Very nice trick by [David Fetter](http://people.planetpostgresql.org/dfetter/). It's based well known trick ([universal sort of array](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II#General_array_sort)):

CREATE OR REPLACE FUNCTION array\_intersect(ANYARRAY, ANYARRAY)

RETURNS ANYARRAY

LANGUAGE SQL

AS $$

 SELECT ARRAY(

 SELECT $1[i] AS "the\_intersection"

 FROM generate\_series(

 array\_lower($1,1),

 array\_upper($1,1)

 ) AS i

 INTERSECT

 SELECT $2[j] AS "the\_intersection"

 FROM generate\_series(

 array\_lower($2,1),

 array\_upper($2,1)

 ) AS j

 );

$$;

An alternative and possibly faster way:

CREATE OR REPLACE FUNCTION array\_intersect(ANYARRAY, ANYARRAY)

RETURNS ANYARRAY

LANGUAGE SQL

AS $$

 SELECT ARRAY(SELECT UNNEST($1) INTERSECT SELECT UNNEST($2))

$$;

VOID function in SQL language

SQL functions can contain only SQL statements. The results of the last statements are results of the functions. The type of a result have to be compatible with the function's type. Because VOID is fictive type, no statement can return this type with one exception. We can cast NULL to VOID.

note: void returning functions are not allowed to be used through the binary protocol

CREATE OR REPLACE FUNCTION none()

RETURNS VOID AS $$ SELECT NULL::VOID; $$

LANGUAGE SQL;

Fast select of next row

For older application (ISAM appliaction works on series of rows) we have be able to search next row specified with some combination of values (a1, b1, c1). Traditional solution:

SELECT \*

 FROM data

 WHERE (a > a1)

 OR (a = a1 AND b > b1)

 OR (a = a1 AND b = b1 AND c > c1)

 ORDER BY a, b, c

 LIMIT 1;

Merlin Moncure noted simplified form that use row equation. This form is more readable and allows using multicolumn index:

SELECT \*

 FROM data

 WHERE (a, b, c) > (a1, b1, c1)

 ORDER BY a, b, c

 LIMIT 1;

ISAM access downgrade SQL serer performance. For effective SQL we have to operate over sets, but when we cannot rewrite application, we have not choice.

PostgreSQL SQL Tricks III

SQL is a language where one task can be solved multiple ways with different efficiency. You can see effective use of PostgreSQL's possibilities on this page.

Google AdSense

[Older tricks](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_II) [Fresh tricks](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks)

**Obsah**

 [[skrýt](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III)]

* [1 to\_string, to\_array functions](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#to_string.2C_to_array_functions)
* [2 Cast between "unix timestamp" and timestamp](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Cast_between_.22unix_timestamp.22_and_timestamp)
* [3 Domain for time zone](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Domain_for_time_zone)
* [4 Non-interactive entering a password for psql](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Non-interactive_entering_a_password_for_psql)
* [5 Faster table's list](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Faster_table.27s_list)
* [6 Array of arrays](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Array_of_arrays)
* [7 MySQL function *group\_concat* in PostgreSQL](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#MySQL_function_group_concat_in_PostgreSQL)
* [8 MySQL function *field* in PostgreSQL](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#MySQL_function_field_in_PostgreSQL)
* [9 The first from groups (with windows functions)](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#The_first_from_groups_.28with_windows_functions.29)
* [10 Bigint to ip](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Bigint_to_ip)
* [11 Marking files as orig (for difforig command) based on patch](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Marking_files_as_orig_.28for_difforig_command.29_based_on_patch)
* [12 Safe string to number conversion](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Safe_string_to_number_conversion)
* [13 Dropping milliseconds from timestamp](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Dropping_milliseconds_from_timestamp)
* [14 Attention on IS NULL and IS NOT NULL operators for composite types](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Attention_on_IS_NULL_and_IS_NOT_NULL_operators_for_composite_types)
* [15 Faster execution of PL/pgSQL function with explicit casting](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Faster_execution_of_PL.2FpgSQL_function_with_explicit_casting)
* [16 Protection from parallel run of some PL function](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Protection_from_parallel_run_of_some_PL_function)
* [17 Using scientific format](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Using_scientific_format)
* [18 Terminal's configuration](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Terminal.27s_configuration)
* [19 Moving tables from one schema to second schema](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Moving_tables_from_one_schema_to_second_schema)
* [20 Encoding number to any numeral system with CTE](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Encoding_number_to_any_numeral_system_with_CTE)
* [21 Get default values from table definition](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Get_default_values_from_table_definition)
* [22 Function to\_bin](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Function_to_bin)
* [23 SQL code for hierarchical break-down of domains](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#SQL_code_for_hierarchical_break-down_of_domains)
* [24 Solution for migration to 8.3 (problem with tsearch2)](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Solution_for_migration_to_8.3_.28problem_with_tsearch2.29)
* [25 Expect SQL functions to be really fast](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III#Expect_SQL_functions_to_be_really_fast)

to\_string, to\_array functions

Author: Pavel Stěhule. Functions *string\_to\_array* and *array\_to\_string* doesn't handle NULL value well. So I designed new functions with better NULL handling.

CREATE OR REPLACE FUNCTION to\_string(anyarray, sep text, nullstr text DEFAULT '')

RETURNS text AS $$

SELECT array\_to\_string(ARRAY(SELECT coalesce(v::text, $3)

 FROM unnest($1) g(v)),

 $2)

$$ LANGUAGE sql;

postgres=# select to\_string(array[1,2,3,4,null,5],',');

 to\_string

------------

 1,2,3,4,,5

(1 row)

postgres=# select to\_string(array[1,2,3,4,null,5],',','<NULL>');

 to\_string

------------------

 1,2,3,4,<NULL>,5

(1 row)

CREATE OR REPLACE FUNCTION to\_array(text, sep text, nullstr text DEFAULT '')

RETURNS text[] AS $$

 SELECT ARRAY(SELECT CASE

 WHEN v = $3 THEN NULL::text

 ELSE v END

 FROM unnest(string\_to\_array($1,$2)) g(v))

$$ LANGUAGE sql;

postgres=# select to\_array('1,2,3,4,,5',',');

 to\_array

------------------

 {1,2,3,4,NULL,5}

(1 row)

postgres=# select to\_array('1,2,3,4,<NULL>,5',',','<NULL>');

 to\_array

------------------

 {1,2,3,4,NULL,5}

(1 row)

postgres=# select to\_array('1,2,3,,5',',')::int[];

 to\_array

----------------

 {1,2,3,NULL,5}

(1 row)

This issue is fixed in modern versions, that allow third parameter for NULL substitution.

postgres=# SELECT array\_to\_string(ARRAY[10,10,NULL,10], ',','');

 array\_to\_string

─────────────────

 10,10,,10

(1 row)

Cast between "unix timestamp" and timestamp

Author:??

CREATE OR REPLACE FUNCTION convert\_timestamp\_to\_xtime(dt timestamp)

RETURNS integer AS $$

 SELECT EXTRACT(EPOCH FROM $1)::integer

$$ LANGUAGE sql;

CREATE OR REPLACE FUNCTION convert\_xtime\_to\_timestamp(xdt integer)

RETURNS timestamp AS $$

 SELECT to\_timestamp($1)::timestamp

$$ LANGUAGE sql;

Note: You can use casting to abstime type, but: "The types abstime and reltime are lower precision types which are used internally. You are discouraged from using these types in applications; these internal types might disappear in a future release."

Domain for time zone

David E. Wheleer sent a nice trick. PostgreSQL missing a data type for time zone and doesn't contain any function for time zone verification. But we can use operator AT:

CREATE OR REPLACE FUNCTION is\_timezone( tz TEXT ) RETURNS BOOLEAN as $$

BEGIN

 PERFORM now() AT TIME ZONE tz;

 RETURN TRUE;

EXCEPTION WHEN invalid\_parameter\_value THEN

 RETURN FALSE;

END;

$$ language plpgsql STABLE;

CREATE DOMAIN timezone AS CITEXT

CHECK ( is\_timezone( value ) );

Non-interactive entering a password for psql

*psql* missing any switch for password. People often ask, how can use psql inside scripts, when is necessary enter any password. One possibility is use a system variable *PGPASSWORD*.

postgres=# CREATE USER tom;

CREATE ROLE

postgres=# ALTER USER tom PASSWORD 'tiger';

ALTER ROLE

...

postgres=> \q

[pavel@nemesis ~]$ PGPASSWORD=tiger psql postgres -U tom

Welcome to psql 8.3.8, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms

 \h for help with SQL commands

 \? for help with psql commands

 \g or terminate with semicolon to execute query

 \q to quit

postgres=> ...

The sense of this method is hiding a password in process list. *Warning: On Linux (at least) this is still visible through /proc/<pid>/environ*

Another possibility is a .pgpass file in your home directory.

Faster table's list

If your database has a high number of tables - thousands and more, you will like next trick:

alter function pg\_table\_is\_visible(oid) cost 10;

by Tom Lane for 8.3 and 8.4. For 8.5 and newer it is default.

Array of arrays

PostgreSQL supports n-dimensional arrays, but doesn't support array of arrays, so we cannot use cumulative aggregate functions over arrays. Here is workaround:

postgres=# create type intarr as (f int[]);

CREATE TYPE

postgres=# select \* from t;

 a

─────────────

 {10,20,30}

 {1,2,3,4,5}

(2 rows)

postgres=# select array\_agg(a) from t;

ERROR: could not find array type for data type integer[]

postgres=# select array\_agg(distinct row(a)::intarr) from t;

 array\_agg

──────────────────────────────────────────

 {"(\"{1,2,3,4,5}\")","(\"{10,20,30}\")"}

(1 row)

postgres=# select (unnest(array\_agg(distinct row(a)::intarr))::intarr).f from t;

 f

─────────────

 {1,2,3,4,5}

 {10,20,30}

(2 rows)

Sent by Sam Mason.

MySQL function *group\_concat* in PostgreSQL

In MySQL we can find very useful aggregate function *group\_concat*. For this function we can define separator and ordering. This isn't possible in PostgreSQL. But we can use workaround via array functions:

postgres=# SELECT \* FROM x;

 cat | town

-----+---------

 1 | Benešov

 1 | Tábor

 1 | Písek

 2 | Praha

 2 | Hradec

 3 | Cheb

 3 | Aš

(7 rows)

postgres=# SELECT cat, array\_to\_string(ARRAY(SELECT unnest(array\_agg(town))

 ORDER BY 1),',')

 FROM x

 GROUP BY cat;

 kat | array\_to\_string

-----+---------------------

 1 | Benešov,Písek,Tábor

 3 | Aš,Cheb

 2 | Hradec,Praha

(3 rows)

Since PostgreSQL 9.0 this is natively available as *string\_agg()*

MySQL function *field* in PostgreSQL

Source: <http://stackoverflow.com/questions/1309624/simulating-mysqls-order-by-field-in-postgresql>

When we can explicitly to specify some order, we should to use function *field* in MySQL.

select \* from pet order by field(species, 'cat', 'dog', 'bird') desc;

+----------+--------+---------+------+------------+------------+

| name | owner | species | sex | birthday | death |

+----------+--------+---------+------+------------+------------+

| Fluffy | Harold | cat | f | 1993-02-04 | NULL |

| Claws | Gwen | cat | m | 1994-03-17 | NULL |

| Buffy | Harold | dog | f | 1989-05-13 | NULL |

| Fang | Benny | dog | m | 1990-08-27 | NULL |

| Bowser | Diane | dog | m | 1989-08-31 | 1995-07-29 |

| Chirpy | Gwen | bird | f | 1998-09-11 | NULL |

| Whistler | Gwen | bird | NULL | 1997-12-09 | NULL |

+----------+--------+---------+------+------------+------------+

This functionality is missing in PostgreSQL - we could to use CASE statement or we could to write own function. It's possible - we have variadic functions:

select \*

 from pet

 order by species, case species when 'cat' then 1

 when 'dog' then 2

 when 'bird' then 3

 else 0 end desc;

Source code field function:

CREATE OR REPLACE FUNCTION field(varchar, VARIADIC text[])

RETURNS int AS $$

 SELECT i

 FROM generate\_subscripts($2,1) g(i)

 WHERE $1 = $2[i]

 UNION ALL

 SELECT 0

 LIMIT 1 1

$$ LANGUAGE sql;

Complement of function field is function elt:

CREATE OR REPLACE FUNCTION elt(int, VARIADIC text[])

RETURNS text AS $$

 SELECT $2[$1];

$$ LANGUAGE sql;

The first from groups (with windows functions)

This exercise should be solved variously. One way is based on windows functions:

postgres=# SELECT \* FROM staff;

┌───────────┬───────────┬────────────┬──────────┐

│ name │ surname │ department │ salary │

├───────────┴───────────┴────────────┴──────────┤

│ Pavel │ Stehule │ 1 │ 10000.00 │

│ Zdenek │ Stehule │ 1 │ 9000.00 │

│ Vladimira │ Stehulova │ 2 │ 9000.00 │

└───────────────────────────────────────────────┘

(3 rows)

postgres=# SELECT \*

 FROM (SELECT \*, max(salary) OVER (PARTITION BY department)

 FROM staff) x

 WHERE max = salary;

┌───────────┬───────────┬────────────┬──────────┬──────────┐

│ name │ surname │ department │ salary │ max │

├───────────┴───────────┴────────────┴──────────┴──────────┤

│ Pavel │ Stehule │ 1 │ 10000.00 │ 10000.00 │

│ Vladimira │ Stehulova │ 2 │ 9000.00 │ 9000.00 │

└──────────────────────────────────────────────────────────┘

(2 rows)

Bigint to ip

By Jasen Betts:

create function bigint\_to\_inet(bigint) returns inet as $$

select (($1>>24&255)||'.'||($1>>16&255)||'.'||($1>>8&255)||'.'||($1>>0&255))::inet

$$ language sql;

Marking files as orig (for difforig command) based on patch

PostgreSQL hackers like commands cporig and difforig. cporig creates copy file with .orig extension. difforig search files with extension .orig and does cumulative diff i.e. patch. Sometimes we need mark modified files from patch. This process can be simplified with unix commands:

cat mnnotation.diff | egrep -o -e "^\\*\\*\\*.\*(sql|out|h|c|sgml|y|lex|Makefile)" | replace "\*\*\* " "" | xargs cporig

Safe string to number conversion

Sometimes we do not want to raise an exception when the converted value is not a number. We can prevent the exception by testing if the input value is a number or not:

CREATE OR REPLACE FUNCTION read\_int(varchar)

RETURNS int AS $$

 SELECT CASE WHEN $1 ~ e'^\\d+$' THEN $1::int END;

$$ LANGUAGE SQL IMMUTABLE STRICT;

Dropping milliseconds from timestamp

Usually we don't need timestamp value in maximum precision. For mostly people only seconds are significant. Timestamp type allows to define precision - and we could to use this feature:

postgres=# select current\_timestamp;

 now

------------------------------

 2009-05-23 20:42:21.57899+02

(1 row)

Time: 196,784 ms

postgres=# select current\_timestamp::timestamp(2);

 now

------------------------

 2009-05-23 20:42:27.74

(1 row)

Time: 51,861 ms

postgres=# select current\_timestamp::timestamp(0);

 now

---------------------

 2009-05-23 20:42:31

(1 row)

Time: 0,729 ms

Attention on IS NULL and IS NOT NULL operators for composite types

One may think that !(x IS NULL) = x IS NOT NULL is true in all cases. But there is an exception - composite types. When one field of a composite value is NULL and another field is NOT NULL, then result of both operators is false. IS NULL is true, only when all fields are NULL. IS NOT NULL is true, only when all fields are NOT NULL. For any case in between, then both operators return false.

CREATE OR REPLACE FUNCTION test\_isnull()

RETURNS TABLE (a int, b int, isnull bool, isnotnull bool) AS $$

DECLARE r foo;

BEGIN

 isnull := r IS NULL; isnotnull := r IS NOT NULL; RETURN NEXT;

 a := NULL; b := 10; r := ROW(a, b);

 isnull := r IS NULL; isnotnull := r IS NOT NULL; RETURN NEXT;

 a := 10; b := 10; r := ROW(a, b);

 isnull := r IS NULL; isnotnull := r IS NOT NULL; RETURN NEXT;

 RETURN;

END;

$$ LANGUAGE plpgsql;

SELECT \* FROM test\_isnull();

 a | b | isnull | isnotnull

----+----+--------+-----------

 | | t | f

 | 10 | f | f

 10 | 10 | f | t

(3 rows)

Faster execution of PL/pgSQL function with explicit casting

PL/pgSQL execute right part of assignment statement independently on left part. Result of evaluation is checked and converted to destination type every time. This conversion should be little bit slow. So we could improve assignment statement via explicit casting to ensure type equality of left and right parts assignment statement. Use this technique carefully! And only when is it necessary.

create or replace function test1()

returns int as $$

declare s int := 0;

begin

 for i in 1..100000 loop s := 4e3; end loop; -- numeric constant

 return s;

end;

$$ language plpgsql immutable;

create or replace function test2()

returns int as $$

declare s int := 0;

begin

 for i in 1..100000 loop s := 4e3::int; end loop; -- int

 return s;

end;

$$ language plpgsql immutable;

postgres=# select test1();

 test1

-------

 4000

(1 row)

Time: 176,623 ms

postgres=# select test2();

 test2

-------

 4000

(1 row)

Time: 47,673 ms

Same problem is implicit conversion in return statement:

create or replace function test1() returns int as $$begin return 4e1; end; $$ language plpgsql;

create or replace function test2() returns int as $$begin return 4e1::int; end; $$ language plpgsql;

postgres=# select count(test1()) from generate\_series(1,100000);

 count

--------nebo v konverzi návratové hodnoty

 100000

(1 row)

Time: 682,005 ms

postgres=# select count(test2()) from generate\_series(1,100000);

 count

--------

 100000

(1 row)

Time: 528,099 ms

Protection from parallel run of some PL function

Author: Pavel Stěhule

Theoretically we should to detect an run of any function - but these information are not accessible from user interface (without C coding). We should to look to list of processed queries or we should to use advisory locks. Test based on select from pg\_stat\_activity view is useful only for function directly called by user. Functions called from other functions are invisible.

create or replace function long\_run()

returns void as $$

begin

 if exists(select procpid

 from pg\_stat\_activity

 where procpid <> pg\_backend\_pid()

 and current\_query like '%long\_run(%')

 then

 raise notice 'procedure is running';

 return;

 end if;

 perform pg\_sleep(10);

end$$

language plpgsql volatile;

CREATE FUNCTION

Next possibility is using advisory locks <http://www.postgresql.org/docs/8.3/interactive/functions-admin.html#FUNCTIONS-ADVISORY-LOCKS>.

create or replace function long\_run()

returns void as $$

declare foid oid:= 'long\_run'::regproc::oid;

begin

 if pg\_try\_advisory\_lock(foid::bigint) then

 perform pg\_sleep(10);

 perform pg\_advisory\_unlock(foid::bigint);

 else

 raise notice 'procedure is running';

 end if;

 return;

end;

$$ language plpgsql;

Using scientific format

PostgreSQL doesn't support EEEE format in to\_char function still (there is a patch for 8.5). We should to use external plperl function:

create or replace function floating\_format(double precision, int)

 returns varchar as $$

 sprintf("%.$\_[1]e", $\_[0])

$$ language plperl;

create or replace function floating\_format(double precision)

 returns varchar as $$

 sprintf("%e", $\_[0])

$$ language plperl;

postgres=# select floating\_format(10.223);

 floating\_format

-----------------

 1.022300e+01

(1 row)

postgres=# select floating\_format(10.223,2);

 floating\_format

-----------------

 1.02e+01

(1 row)

postgres=# select floating\_format(10.223,4);

 floating\_format

-----------------

 1.0223e+01

(1 row)

Terminal's configuration

I using two interfaces for access to PostgreSQL - emacs and psql. These application should be configured for more comfortable work:

* emacs (.emacs)

(setq-default truncate-lines t)

(ansi-color-for-comint-mode-on)

(setq lazy-lock-defer-on-scrolling t)

(setq inhibit-startup-echo-area-message t)

(setq inhibit-startup-message t)

(show-paren-mode t)

(setq show-paren-style 'mixed)

(fset 'yes-or-no-p 'y-or-n-p)

(global-set-key "\M-g" 'goto-line)

(setq sql-database "postgres")  ;; name of most often used database

;; don't use pager, empty title

(setq sql-postgres-options '("-P" "pager=off" "-P" "title= "))

;; interesting idea is setting of start of every new command on new line

;; (setq sql-postgres-options '("-P" "pager=off" "-P" "title= " "-v" "PROMPT1=%/.%n%#\n" "-v" "PROMPT2=" "-v" "PROMPT3="))

* psql

export PAGER="less -RSX"

psql ... -P pager=always

Moving tables from one schema to second schema

We have to use plpgsql function:

create or replace function mvtable(src\_schema varchar, dest\_schema varchar, mask varchar)

returns void as $$

declare r record;

begin

 for r in

 select \*

 from information\_schema.tables

 where table\_schema = src\_schema and table\_name like mask

 loop

 execute 'alter table ' || quote\_ident(r.table\_schema)

 || '.' || quote\_ident(r.table\_name)

 || ' set schema ' || quote\_ident(dest\_schema);

 end loop;

 return;

end;

$$ language plpgsql;

Encoding number to any numeral system with CTE

Autor: Pavel Stehule

postgres=# create function to\_base(num int, base int = 10)

 returns varchar as $$

 with recursive z as (

 select $1 as a, null::int as b, 0 as i

 union all

 select a/$2, a % $2, i+1

 from z

 where a > 0

 )

 select array\_to\_string(array(select substring('0123456789abcdef' from b + 1 for 1)

 from z

 where i > 0

 order by i desc),'');

 $$ language sql immutable;

postgres=# select to\_base(255);

 to\_base

---------

 255

(1 row)

postgres=# select to\_base(255,16);

 to\_base

---------

 ff

(1 row)

postgres=# select to\_base(255,8);

 to\_base

---------

 377

(1 row)

postgres=# select to\_base(255,2);

 to\_base

----------

 11111111

(1 row)

Get default values from table definition

One typical query is getting the default values for a record. Since these definitions are stored in the system catalogs (tables), they are accessible.

CREATE OR REPLACE FUNCTION eval(varchar)

RETURNS varchar AS $$

DECLARE result varchar;

BEGIN

 EXECUTE 'SELECT ' || $1 INTO result;

 RETURN result;

END;$$ LANGUAGE plpgsql STRICT;

CREATE OR REPLACE FUNCTION defaults(text,

 OUT attname name, OUT type varchar, OUT default\_val varchar)

RETURNS SETOF RECORD AS $$

SELECT a.attname,

 pg\_catalog.format\_type(a.atttypid, a.atttypmod),

 (SELECT eval(pg\_catalog.pg\_get\_expr(d.adbin, d.adrelid))

 FROM pg\_catalog.pg\_attrdef d

 WHERE d.adrelid = a.attrelid AND d.adnum = a.attnum AND a.atthasdef)

 FROM pg\_catalog.pg\_attribute a

 WHERE a.attrelid = $1::regclass::oid AND a.attnum > 0 AND NOT a.attisdropped

 ORDER BY a.attnum

$$ LANGUAGE SQL STRICT;

Second solution, but not always usable is using RETURNING clause.

Function to\_bin

PostgreSQL doesn't provide to\_bin function. Implementation (based on casting to bit type) is simple and fast:

create or replace function to\_bin(int)

returns varchar as $$

declare r varchar;

begin

 execute 'select ' || $1 || '::bit(' || case when $1 = 0 then 1 else trunc(log(2, $1)) + 1 end || ')' into r;

 return r;

end;

$$ language plpgsql immutable strict;

create or replace function bin(varchar)

returns varchar as $$

declare r varchar;

begin

 execute 'select b''' || $1 || '''::int' into r;

 return r;

end;

$$ language plpgsql immutable strict;

postgres=# select to\_bin(5), bin('101');

 to\_bin | bin

--------+-----

 101 | 5

(1 row)

SQL code for hierarchical break-down of domains

based on Depesc's ideas from <http://www.depesz.com/index.php/2008/12/01/getting-list-of-most-common-domains/#comment-27066>

postgres=# create or replace function domain\_list(varchar[])

 returns setof varchar as $$

 select array\_to\_string($1,’.')

 union all

 select domain\_list($1[2:array\_upper($1,1)])

 where array\_upper($1,1) > 1

 $$ language sql immutable strict;

CREATE FUNCTION

postgres=# select domain\_list(array['a','b','c','d']);

domain\_list

————-

a.b.c.d

b.c.d

c.d

d

(4 rows)

postgres=# create or replace function domain\_list(varchar)

 returns setof varchar as $$

 select domain\_list(string\_to\_array($1, ‘.’))

 $$ language sql immutable strict;

CREATE FUNCTION

postgres=# select domain\_list(’a.b.c.d’);

domain\_list

————-

a.b.c.d

b.c.d

c.d

d

Solution for migration to 8.3 (problem with tsearch2)

PostgreSQL documentation has a section with some advice that should to help with migration of older application (based on TSearch2) to PostgreSQL 8.3 and higher (with integrated fulltext). This works well when you migrate databases individually, but it fails when you use pg\_dump. As documented, you should do some pre steps - import TSearch compatible module into template1. But complete dump generates databases from template0 (you can't to change it). So you cannot load this dump, and you have to edit dump (advice - dump structure and data to two different files).

so, when you find bug:

ERROR: type "tsvector" already exists

You have to add import of tsearch2.sql to every creation database part in dump.

DROP DATABASE brigady;

CREATE DATABASE brigady WITH TEMPLATE = template0 ENCODING = 'UTF8';

--

-- PostgreSQL database dump complete

--

\connect brigady

-- new line

\i /usr/local/pgsql/share/contrib/tsearch2.sql

SET search\_path = public, pg\_catalog;

...

Expect SQL functions to be really fast

Much faster than PL/pgSQL - especially when they are wrappers around integrated features (in this example - integrated qsort). PL/pgSQL is usually fast to interpret - I know only about two slow operations - array update and string update. It's based on the internal architecture - every change generate a new object - it isn't significant for small objects (less than 1000), but for large objects this operations needs significant time. However, we can go a different way, using integrated functionality (this way is usually the fastest in any embedded language).

PostgreSQL use internal qsort -so, I wrote qsort in PL/pgSQL too (results will be comparable):

CREATE OR REPLACE FUNCTION quicksort(l integer, r integer, a int[])

RETURNS int[] AS $$

DECLARE akt int[] = a;

 i integer := l; j integer := r; x integer = akt[(l+r) / 2];

 w integer;

BEGIN

 LOOP

 WHILE akt[i] < x LOOP i := i + 1; END LOOP;

 WHILE x < akt[j] loop j := j - 1; END LOOP;

 IF i <= j THEN

 w := akt[i];

 akt[i] := akt[j]; akt[j] := w;

 i := i + 1; j := j - 1;

 END IF;

 EXIT WHEN i > j;

 END LOOP;

 IF l < j THEN akt := quicksort(l,j,akt); END IF;

 IF i < r then akt := quicksort(i,r,akt); END IF;

 RETURN akt;

END;

$$ LANGUAGE plpgsql STRICT IMMUTABLE;

postgres=# SELECT array\_upper(quicksort(1,10000,array\_agg(a)),1) FROM test;

 array\_upper

-------------

 10000

(1 row)

Time: 5918,531 ms

Then I wrote same function (note. some 8.4 features are used) in SQL:

CREATE OR REPLACE FUNCTION sort(anyarray)

RETURNS anyarray AS $$

 SELECT array(SELECT \* FROM unnest($1) ORDER BY 1);

$$ language sql;

postgres=# SELECT array\_upper(sort(array\_agg(a)),1) FROM test;

 array\_upper

-------------

 10000

(1 row)

Time: 35,980 ms

*Nota bene* - The SQL function is about 150 times faster than the PL/pgSQL function. This isn't surprising. Integrated sorting is well optimized and doesn't do a lot of large memory operations. For information: I tested bublesort too, and I stopped its execution after five minutes.

PostgreSQL SQL Tricks

SQL is a language where one task can be solved multiple ways with different efficiency. You can see effective use of PostgreSQL's possibilities on this page.

Google AdSense

[Older tricks](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks_III)

**Obsah**

 [[skrýt](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks)]

* [1 Setting DETAIL and HINT fields in PLPythonu exception](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Setting_DETAIL_and_HINT_fields_in_PLPythonu_exception)
* [2 Pretty xml formating](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Pretty_xml_formating)
* [3 Debugging communication between PostgreSQL and client](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Debugging_communication_between_PostgreSQL_and_client)
* [4 Fast replacing a index of PRIMARY KEY constraint](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Fast_replacing_a_index_of_PRIMARY_KEY_constraint)
* [5 Passing parameters from command line to DO statement](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Passing_parameters_from_command_line_to_DO_statement)
* [6 How to unnest an array with element indexes](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#How_to_unnest_an_array_with_element_indexes)
* [7 Allow only one NULL in column](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Allow_only_one_NULL_in_column)
* [8 Bytea to BLOB conversion](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Bytea_to_BLOB_conversion)
* [9 Only one column can hold NULL](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Only_one_column_can_hold_NULL)
* [10 Overwriting a application name in psql (for psql based script identification)](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Overwriting_a_application_name_in_psql_.28for_psql_based_script_identification.29)
* [11 Getting N unique most similar values with usage KNNsearch and pg\_tgrm module](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Getting_N_unique_most_similar_values_with_usage_KNNsearch_and_pg_tgrm_module)
* [12 Using custom casting when you migrate from Oracle to PostgreSQL](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Using_custom_casting_when_you_migrate_from_Oracle_to_PostgreSQL)
* [13 JSON creating](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#JSON_creating)
* [14 A array's fields trimming](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#A_array.27s_fields_trimming)
* [15 Simply query result processing in bash](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Simply_query_result_processing_in_bash)
* [16 Glue comment to query](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Glue_comment_to_query)
* [17 Logout all connections](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Logout_all_connections)
* [18 Taking first unlocked row from table](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Taking_first_unlocked_row_from_table)
* [19 Searching gaps in time series with window analytic functions](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Searching_gaps_in_time_series_with_window_analytic_functions)
* [20 Delete duplicate rows with window analytic functions](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Delete_duplicate_rows_with_window_analytic_functions)
* [21 LIKE to list of patterns](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#LIKE_to_list_of_patterns)
* [22 Enforcing unique name for cursor in PL/pgSQL](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Enforcing_unique_name_for_cursor_in_PL.2FpgSQL)
* [23 Conversion BLOB to Bytea](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Conversion_BLOB_to_Bytea)
* [24 Function for decoding of url code](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Function_for_decoding_of_url_code)
* [25 Emacs's configuration for PLpgSQL syntax highlighting when function is edited from console](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Emacs.27s_configuration_for_PLpgSQL_syntax_highlighting_when_function_is_edited_from_console)
* [26 Cast bytea to text](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Cast_bytea_to_text)
* [27 Simply age calculation from birth date](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Simply_age_calculation_from_birth_date)
* [28 Is current server master or slave?](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Is_current_server_master_or_slave.3F)
* [29 Dynamic modification some record's field](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Dynamic_modification_some_record.27s_field)
* [30 Iteration over RECORD variable inside trigger](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Iteration_over_RECORD_variable_inside_trigger)
* [31 Enforcing some chars in to\_char function](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Enforcing_some_chars_in_to_char_function)
* [32 Barrier against to unwanted conversion char(n) to text](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Barrier_against_to_unwanted_conversion_char.28n.29_to_text)
* [33 Get functions's DDL command](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#Get_functions.27s_DDL_command)
* [34 from Russian alphabet (Cyrillic) to ASCII](http://postgres.cz/wiki/PostgreSQL_SQL_Tricks#from_Russian_alphabet_.28Cyrillic.29_to_ASCII)

Setting DETAIL and HINT fields in PLPythonu exception

Author: Pavel Stěhule PLPythonu doesn't allow to set all possible fields in PostgreSQL exception. There are a workaround for setting DETAIL and HINT:

postgres=# do $$

x = plpy.SPIError('Nazdarek');

x.spidata = (100, "Some detail", "some hint", None, None);

raise x;

$$ language plpythonu;

ERROR: plpy.SPIError: Nazdarek

DETAIL: Some detail

HINT: some hint

CONTEXT: Traceback (most recent call last):

 PL/Python anonymous code block, line 4, in <module>

 raise x;

PL/Python anonymous code block

Pretty xml formating

Author: Pavel Stěhule (internet - xslt template)

Extension *xml2* supports xslt transformations. With function *xslt\_process* a pretty printing of any xml documenty pretty simple:

create or replace function xml\_pretty(xml)

returns xml as $$

 select xslt\_process($1,

'<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:strip-space elements="\*" />

<xsl:output method="xml" indent="yes" />

<xsl:template match="node() | @\*">

<xsl:copy>

<xsl:apply-templates select="node() | @\*" />

</xsl:copy>

</xsl:template>

</xsl:stylesheet>')::xml

$$ language sql immutable strict;

Example:

postgres=# select xml\_pretty('<root><a><b>some text</b></a></root>');

 xml\_pretty

----------------------

 <root> +

 <a> +

 <b>some text</b>+

 </a> +

 </root> +

(1 row)

Debugging communication between PostgreSQL and client

Author: Pavel Stěhule

There is great tool for this purpose - [pgShark](https://github.com/dalibo/pgshark).

unbuffer ./pgs-debug --host 172.30.34.72 -i lo --port 6432 | while read line; do echo `date +"%T.%3N"` $line; done | gzip > /mnt/ebs/pgsharklog.gz

Fast replacing a index of PRIMARY KEY constraint

Author: Petr Novak

We can use a following steps, when we have replace a index for PRIMARY KEY constraint quickly:

BEGIN;

CREATE UNIQUE INDEX CONCURRENTLY tab\_pkey\_idx2 ON tab(id);

ALTER TABLE tab

 DROP CONSTRAINT tab\_pkey CASCADE,

 ADD CONSTRAINT tab\_pkey PRIMARY KEY USING INDEX tab\_pkey\_idx2;

ALTER TABLE second\_tab

 ADD CONSTRAINT second\_tab\_fkey FOREIGN KEY (tab\_id) REFERENCES tab(id) NOT VALID;

COMMIT;

Next step should be a FOREIGN KEY constraint validation.

Passing parameters from command line to DO statement

Author: Pavel Stehule

PostgreSQL statement DO doesn't support parameters. But we can pass parameters there by psql and server side session parameters:

bash-4.1$ cat test.sh

echo "

set myvars.msgcount TO :'msgcount';

DO \$\$

BEGIN

 FOR i IN 1..current\_setting('myvars.msgcount')::int LOOP

 RAISE NOTICE 'Hello';

 END LOOP;

END \$\$" | psql postgres -v msgcount=$1

bash-4.1$ sh test.sh 3

SET

Time: 0.341 ms

NOTICE: Hello

NOTICE: Hello

NOTICE: Hello

DO

Time: 2.415 ms

How to unnest an array with element indexes

Sometimes we need to unnest a array with indexes. There are more ways how to do it:

Author: Pavel Stěhule (PostgreSQL 9.1)

CREATE OR REPLACE FUNCTION unnest\_rownum(anyarray)

RETURNS TABLE (id int, element anyelement) AS $$

BEGIN

 id := 1;

 FOREACH element IN array $1

 LOOP

 RETURN NEXT;

 id := id + 1;

 END LOOP;

 RETURN;

END

$$ LANGUAGE plpgsql;

postgres=# select \* from unnest\_rownum(ARRAY['A','B','C']);

 id | element

----+---------

 1 | A

 2 | B

 3 | C

(3 rows)

Author: Tom Lane (PostgreSQL 9.3)

SELECT i, arraycol[i]

 FROM tab,

 LATERAL generate\_subscripts(arraycol, 1) as i;

9.4 supports clause WITH ORDINALITY:

postgres=# SELECT \* FROM unnest(ARRAY['A','D','C']) WITH ORDINALITY;

 unnest | ordinality

--------+------------

 A | 1

 D | 2

 C | 3

(3 rows)

Allow only one NULL in column

Author: Pavel Stěhule

UNIQUE constraint ensures unique values in column(s). But NULL is not a comparable value and unique constraint ignores these values. Sometime we would to allow only one NULL in column. There is a simply solution based on partial index and functional (here only constant) index.

postgres=# CREATE TABLE omega(a int UNIQUE);

CREATE TABLE

postgres=# INSERT INTO omega VALUES(NULL);

INSERT 0 1

postgres=# INSERT INTO omega VALUES(NULL);

INSERT 0 1

postgres=# TRUNCATE omega;

TRUNCATE TABLE

postgres=# CREATE UNIQUE INDEX ON omega ((1)) WHERE a IS NULL;

CREATE INDEX

postgres=# INSERT INTO omega VALUES(NULL);

INSERT 0 1

postgres=# INSERT INTO omega VALUES(NULL);

ERROR: duplicate key value violates unique constraint "omega\_expr\_idx"

DETAIL: Key ((1))=(1) already exists.

postgres=# UPDATE omega SET a = 10;

UPDATE 1

postgres=# INSERT INTO omega VALUES(NULL);

INSERT 0 1

Time: 7.601 ms

Bytea to BLOB conversion

Author: unknown

Store bytea string to LO object.

CREATE OR REPLACE FUNCTION make\_lo(bytea)

RETURNS oid AS $$

DECLARE

 loid oid;

 fd integer;

 bytes integer;

BEGIN

 loid := lo\_creat(-1);

 fd := lo\_open(loid, 131072);

 bytes := lowrite(fd, $1);

 IF (bytes != LENGTH($1)) THEN

 RAISE EXCEPTION 'Not all data copied to blob';

 END IF;

 PERFORM lo\_close(fd);

 RETURN loid;

END;

$$ LANGUAGE plpgsql STRICT;

Only one column can hold NULL

Author: Daniel Vérité

How we can calculate how much columns hold a NULL value? Nice example of variadic function usage.

CREATE OR REPLACE FUNCTION public.null\_count(VARIADIC anyarray)

 RETURNS integer

 LANGUAGE sql

AS $function$

 SELECT sum(CASE WHEN v IS NULL THEN 1 ELSE 0 END)::int FROM unnest($1) g(v)

$function$

Usage:

CREATE TABLE xxx(

 a int,

 b int,

 c int,

 CHECK (null\_count(a,b,c) <= 1))

Overwriting a application name in psql (for psql based script identification)

Author: Erik Rijkers

Application\_name is one of connection params, that we can attach to log event and later we can to identify a SQL statement producent. There are more possibilities how to do it (one is a example of connection string in url format).

[pavel@localhost ~]$ PGAPPNAME=splunk psql -c "select application\_name from pg\_stat\_activity where pid = pg\_backend\_pid();" postgres

 application\_name

------------------

 splunk

(1 row)

[pavel@localhost ~]$ psql -c "select application\_name from pg\_stat\_activity where pid = pg\_backend\_pid();" "dbname=postgres application\_name=splunk"

 application\_name

------------------

 splunk

(1 row)

[pavel@localhost ~]$ psql -c "select application\_name from pg\_stat\_activity where pid = pg\_backend\_pid();" postgresql://localhost/postgres?application\_name=splunk

 application\_name

------------------

 splunk

(1 row)

Getting N unique most similar values with usage KNNsearch and pg\_tgrm module

Author: Tom Lane

SELECT DISTINCT ON (village\_name <-> 'Benešov') village\_name, village\_name<->'Benešov'

 FROM villages

 ORDER BY village\_name <-> 'Benešov'

 LIMIT 10;

 village\_name | ?column?

-----------------------+----------

 Benešov | 0

 Benešovice | 0.416667

 Dolní Benešov | 0.428571

 Benešov u Semil | 0.5

 Benešov nad Černou | 0.578947

 Benešov nad Ploučnicí | 0.636364

 Benecko | 0.666667

 Benetice | 0.692308

 Bečov | 0.727273

 Bezkov | 0.75

(10 rows)

postgres=# EXPLAIN SELECT DISTINCT ON (village\_name <-> 'Benešov') village\_name, village\_name<->'Benešov'

 FROM obce

 ORDER BY village\_name <-> 'Benešov'

 LIMIT 10;

 QUERY PLAN

-------------------------------------------------------------------------------------------------

 Limit (cost=0.00..2.76 rows=10 width=10)

 -> Unique (cost=0.00..1474.75 rows=5341 width=10)

 -> Index Scan using villages\_village\_name\_idx on obce (cost=0.00..1433.14 rows=16644 width=10)

 Order By: (village\_name <-> 'Benešov'::text)

(4 rows)

Using custom casting when you migrate from Oracle to PostgreSQL

Author: Pavel Stěhule

Oracle doesn't support boolean data type, and developers use a varchar(1) or number(1) instead. It is a issue in PostgreSQL because this width is too less for storing a strings "true" or "false". But we can redefine a default casting (there is used a ugly hack - direct update of system tables under superuser rights). This solution is fast hack - changing to PostgreSQL's boolean is much better.

CREATE OR REPLACE FUNCTION public.boolcast\_varchar(boolean)

 RETURNS character varying

 LANGUAGE sql

AS $$

 SELECT CASE WHEN $1 THEN 't' ELSE 'f' END

$$

UPDATE pg\_cast SET castfunc='boolcast\_varchar'::regproc,

 castcontext='i'

 WHERE castsource=16 and casttarget=1043;

CREATE TABLE test(a varchar(1));

INSERT INTO test VALUES(true);

JSON creating

Author: Merlin

select

 row\_to\_json(t1)

from (

 select

 'joe' as username,

 (select project from (values(1, 'prj1')) as project(project\_id,

project\_name)) as project

) t1;

A array's fields trimming

Author: a\_horse\_with\_no\_name

postgres=# select \* from g;

 a

--------------------------------

 {" ssss ssss","sss ssss "}

 {" ssss ssssaaaa "}

(2 rows)

postgres=# select row\_number() over(), unnest(a) e

 from g;

 row\_number | e

------------+-----------------

 1 | ssss ssss

 1 | sss ssss

 2 | ssss ssssaaaa

(3 rows)

postgres=# select array\_agg(trim(x.e))

 from (select row\_number() over() rn, unnest(a) e

 from g) x

 group by rn;

 array\_agg

---------------------------

 {"ssss ssss","sss ssss"}

 {"ssss ssssaaaa"}

(2 rows)

Attention: This trick depends on imlementation - and should not work in future releases. If you use a 9.3 and newer PostgreSQL, use a LATERAL join instead.

Simply query result processing in bash

Author: Pavel Stěhule

*psql* has not complete macro language, but we can use *bash* and we can use features, that provides bash:

pavel ~ $ psql postgres -A -t --field-separator=" " \

> -c "copy (select 1,'Ahoj Svete', i

> from generate\_series(1,3) g(i))

> to stdout delimiter ' '" | \

> while read var1 var2 var3;

> do

> echo "a=$var1,b=$var2,c=$var3 ";

> done

a=1,b=Ahoj Svete,c=1

a=1,b=Ahoj Svete,c=2

a=1,b=Ahoj Svete,c=3

Similar solution designed by Hubert Depesz Lubaczewski:

psql -qAtX -c "copy (select \* from t) to stdout" | while IFS=$'\t' read -r a b c; do echo -e "a=[$a] b=[$b] c=[$c]"; done

Glue comment to query

Author: Thomas Kellerer

When you need append notice to query, you can use nested comment:

SELECT /\* my comments, that I would to see in PostgreSQL log \*/

 a, b, c

 FROM mytab;

Logout all connections

Author: Pavel Stěhule

Execute query as user *postgres*:

postgres=# select \* from pg\_stat\_activity ;

 datid | datname | pid | usesysid | usename | application\_name | client\_addr | client\_hostname | client\_port | backend\_start |

-------+----------+-------+----------+----------+------------------+-------------+-----------------+-------------+-------------------------------+

 12894 | postgres | 21091 | 10 | postgres | psql | | | -1 | 2012-03-06 09:16:02.466983+01 |

 12894 | postgres | 21103 | 16384 | pavel | psql | | | -1 | 2012-03-06 09:17:02.827352+01 |

 12894 | postgres | 21104 | 16384 | pavel | psql | | | -1 | 2012-03-06 09:17:12.176979+01 |

(3 rows)

postgres=# select pg\_terminate\_backend(pid)

 from pg\_stat\_activity

 where pid <> pg\_backend\_pid() ;

 pg\_terminate\_backend

----------------------

 t

 t

(2 rows)

postgres=# select \* from pg\_stat\_activity ;

 datid | datname | pid | usesysid | usename | application\_name | client\_addr | client\_hostname | client\_port | backend\_start |

-------+----------+-------+----------+----------+------------------+-------------+-----------------+-------------+-------------------------------+

 12894 | postgres | 21091 | 10 | postgres | psql | | | -1 | 2012-03-06 09:16:02.466983+01 |

(1 row)

Taking first unlocked row from table

Sometimes we need first unlocked row (queue implementation). We can use following trick:

postgres=# select \* from queue;

 id | processed | cmd

----+-----------+-------

 1 | f | task1

 2 | f | task2

 3 | f | task2

 4 | f | task3

(4 rows)

Usage:

-- consument1

postgres=# begin;

BEGIN

postgres=# select \* from queue where not processed and pg\_try\_advisory\_xact\_lock(id) for update limit 1;

 id | processed | cmd

----+-----------+-------

 1 | f | task1

(1 row)

postgres=# update queue set processed = true where id = 1;

UPDATE 1

postgres=# commit;

COMMIT

-- consument2

postgres=# begin;

BEGIN

postgres=# select \* from queue where not processed and pg\_try\_advisory\_xact\_lock(id) for update limit 1;

 id | processed | cmd

----+-----------+-------

 2 | f | task2

(1 row)

postgres=# update queue set processed = true where id = 2;

UPDATE 1

postgres=# commit;

COMMIT

With advisory locks we can implement unblocked queue parallel processing.

Searching gaps in time series with window analytic functions

Source: Magnus Hagander <http://blog.hagander.net/archives/203-Finding-gaps-in-partitioned-sequences.html>

Function *lag* can be used for any sorted series:

SELECT \* FROM (

 SELECT

 gropid,

 year,

 month,

 seq,

 seq-lag(seq,1) OVER (PARTITION BY groupid, year, month ORDER BY seq) AS gap FROM mytable

) AS t

WHERE NOT (t.gap=1)

ORDER BY groupid, year, month, seq

Delete duplicate rows with window analytic functions

We can use a window function for very effective removing of duplicate rows:

DELETE FROM tab

 WHERE id IN (SELECT id

 FROM (SELECT row\_number() OVER (PARTITION BY column\_with\_duplicate\_values), id

 FROM tab) x

 WHERE x.row\_number > 1);

Some PostgreSQL's optimized version (with ctid):

DELETE FROM tab

 WHERE ctid = ANY(ARRAY(SELECT ctid

 FROM (SELECT row\_number() OVER (PARTITION BY column\_with\_duplicate\_values), ctid

 FROM tab) x

 WHERE x.row\_number > 1));

LIKE to list of patterns

PostgreSQL cannot to use (I)LIKE together list of patterns (in conformance with ANSI SQL):

-- this is not valid

WHERE somecolumn LIKE ('%some%','%someelse%')

But we can use arrays or regular expressions:

-- this is valid

WHERE somecolumn ~~\* any(array['%some%', '%someelse']));

Author: Tom Lane

Enforcing unique name for cursor in PL/pgSQL

PL/pgSQL cursors uses PostgreSQL SQL cursors. When we use PL/pgSQL cursor, then PL/pgSQL runtime creates SQL cursor with same name. This behave is unwanted sometimes. But when we know, so PL/pgSQL cursor is +/- text variable with SQL name, we can assign NULL to this variable. Then PL/pgSQL runtime cannot to use a predefined name and it will generate a unique name:

DECLARE

 mycursor CURSOR FOR SELECT \* FROM int4\_tbl WHERE f1 > id;

 newid INTEGER;

 out TEXT;

BEGIN

 out := id::text || ' ';

 mycursor := null;

 OPEN mycursor;

 ..

Conversion BLOB to Bytea

Source: <http://snipplr.com/view/62576/convert-oid-to-bytea/>

CREATE OR REPLACE FUNCTION merge\_oid(val oid)

RETURNS bytea AS $$

DECLARE

 merged bytea;

 arr bytea;

BEGIN

 FOR arr IN SELECT DATA

 FROM pg\_largeobject

 WHERE loid = val

 ORDER BY pageno

 LOOP

 IF merged IS NULL THEN

 merged := arr;

 ELSE

 merged := merged || arr;

 END IF;

 END LOOP;

 RETURN merged;

END

$$ LANGUAGE plpgsql;

Function for decoding of url code

Author: Marti Raudsepp

CREATE OR REPLACE FUNCTION url\_decode(input text) RETURNS text

LANGUAGE plpgsql IMMUTABLE STRICT AS $$

DECLARE

 bin bytea = '';

 byte text;

BEGIN

 FOR byte IN (select (regexp\_matches(input, '(%..|.)', 'g'))[1]) LOOP

 IF length(byte) = 3 THEN

 bin = bin || decode(substring(byte, 2, 2), 'hex');

 ELSE

 bin = bin || byte::bytea;

 END IF;

 END LOOP;

 RETURN convert\_from(bin, 'utf8');

END

$$;

Usage:

ohs=# select url\_decode('Hell%C3%B6%20World%21');

 url\_decode

──────────────

 Hellö World!

(1 row)

Another version (by Marc Mamin)

CREATE OR REPLACE FUNCTION urldecode\_arr(url text)

RETURNS text AS $$

BEGIN

 RETURN

 (WITH str AS (SELECT CASE WHEN $1 ~ '^%[0-9a-fA-F][0-9a-fA-F]' THEN array[''] END

 || regexp\_split\_to\_array ($1, '(%[0-9a-fA-F][0-9a-fA-F])+', 'i') plain,

 ARRAY(SELECT (regexp\_matches ($1, '((?:%[0-9a-fA-F][0-9a-fA-F])+)', 'gi'))[1]) encoded)

 SELECT coalesce(string\_agg(plain[i] || coalesce( convert\_from(decode(replace(encoded[i], '%',''), 'hex'), 'utf8'), ''), ''), $1)

 FROM str,

 (SELECT generate\_series(1, array\_upper(encoded,1) + 2) i FROM str) blah);

END

$$ LANGUAGE plpgsql IMMUTABLE STRICT;

Emacs's configuration for PLpgSQL syntax highlighting when function is edited from console

Author: Peter Eisentraut

modify file .profile

PSQL\_EDITOR=emacs;

export PSQL\_EDITOR.

modify file .emacs

(add-to-list 'auto-mode-alist

 '("/psql.edit.[0-9]+\\'" . sql-mode))

Cast bytea to text

When we need cast content of bytea variable to adequate text value directly (content of bytea is char codes), then following function is solution:

CREATE OR REPLACE FUNCTION bytea\_to\_text(bytea)

RETURNS text AS $$

SELECT convert\_from($1, current\_setting('server\_encoding'))

$$ LANGUAGE sql;

xxx=# SELECT bytea\_to\_text('žluťoučký kůň se napil žluté vody'::bytea);

 bytea\_to\_text

───────────────────────────────────

 žluťoučký kůň se napil žluté vody

(1 row)

Simply age calculation from birth date

postgres=# SELECT CURRENT\_DATE;

 date

------------

 2011-09-20

(1 row)

postgres=# SELECT EXTRACT(YEAR FROM age('1972-08-20'::date));

 date\_part

-----------

 39

(1 row)

Is current server master or slave?

When we need to identify server's state, then we can use a function:

SELECT pg\_is\_in\_recovery();

Author: Simon Riggs

Dynamic modification some record's field

PL/pgSQL can modify record's field only when field specification is static. Some fast C libraries are generally available, but when we have no necessary rights, then we cannot use it. Then we can use function:

CREATE OR REPLACE FUNCTION x.setfield2(anyelement, text, text)

 RETURNS anyelement

 LANGUAGE plpgsql

AS $function$

DECLARE

 \_name text;

 \_values text[];

 \_value text;

 \_attnum int;

BEGIN

 FOR \_name, \_attnum

 IN SELECT a.attname, a.attnum

 FROM pg\_catalog.pg\_attribute a

 WHERE a.attrelid = (SELECT typrelid

 FROM pg\_type

 WHERE oid = pg\_typeof($1)::oid)

 LOOP

 IF \_name = $2 THEN

 \_value := $3;

 ELSE

 EXECUTE 'SELECT (($1).' || quote\_ident(\_name) || ')::text' INTO \_value USING $1;

 END IF;

 \_values[\_attnum] := COALESCE('"' || replace(replace(\_value, '"', '""'), '''', '''''') || '"', '');

 END LOOP;

 EXECUTE 'SELECT (' || pg\_typeof($1)::text || '''(' || array\_to\_string(\_values,',') || ')'').\*' INTO $1;

 RETURN $1;

END;

$function$;

postgres=# select setfield2(mypoint '(10,)', 'b', '33');

 setfield2

───────────

 (10,33)

(1 row)

Time: 9,480 ms

Author: Pavel Stěhule

Erwin Brandstetter designed better (more simply, faster) solution:

-- Erwin 1

CREATE OR REPLACE FUNCTION x.setfield3(anyelement, text, text)

RETURNS anyelement

AS $body$

DECLARE

 \_list text;

BEGIN

\_list := (

 SELECT string\_agg(x.fld, ',')

 FROM (

 SELECT CASE WHEN a.attname = $2

 THEN quote\_literal($3)

 ELSE quote\_ident(a.attname)

 END AS fld

 FROM pg\_catalog.pg\_attribute a

 WHERE a.attrelid = (SELECT typrelid

 FROM pg\_type

 WHERE oid = pg\_typeof($1)::oid)

 ORDER BY a.attnum

 ) x

);

EXECUTE '

SELECT ' || \_list || '

FROM (SELECT $1.\*) x'

USING $1

INTO $1;

RETURN $1;

END;

$body$ LANGUAGE plpgsql;

or modernised final version

CREATE FUNCTION f\_setfield(INOUT \_comp\_val anyelement, \_field text, \_val text)

 RETURNS anyelement AS

$func$

BEGIN

EXECUTE 'SELECT ' || array\_to\_string(ARRAY(

 SELECT CASE WHEN attname = \_field

 THEN '$2'

 ELSE '($1).' || quote\_ident(attname)

 END AS fld

 FROM pg\_catalog.pg\_attribute

 WHERE attrelid = pg\_typeof(\_comp\_val)::text::regclass

 AND attnum > 0

 AND attisdropped = FALSE

 ORDER BY attnum

 ), ',')

USING \_comp\_val, \_val

INTO \_comp\_val;

END

$func$ LANGUAGE plpgsql;

Iteration over RECORD variable inside trigger

Author: Tom Lane (for PostgreSQL 8.4 and higher)

CREATE OR REPLACE FUNCTION dynamic\_trigger()

RETURNS TRIGGER

LANGUAGE plpgsql

AS $$

DECLARE

 ri RECORD;

 t TEXT;

BEGIN

 RAISE NOTICE E'\n Operation: %\n Schema: %\n Table: %',

 TG\_OP,

 TG\_TABLE\_SCHEMA,

 TG\_TABLE\_NAME;

 FOR ri IN

 SELECT ordinal\_position, column\_name, data\_type

 FROM information\_schema.columns

 WHERE

 table\_schema = quote\_ident(TG\_TABLE\_SCHEMA)

 AND table\_name = quote\_ident(TG\_TABLE\_NAME)

 ORDER BY ordinal\_position

 LOOP

 EXECUTE 'SELECT ($1).' || ri.column\_name || '::text' INTO t USING NEW;

 RAISE NOTICE E'Column\n number: %\n name: %\n type: %\n value: %.',

 ri.ordinal\_position,

 ri.column\_name,

 ri.data\_type,

 t;

 END LOOP;

 RETURN NEW;

END; $$;

Note: This technique needs one dynamic SQL per one record's field - functions from PL Toolbox library or from Hstore contrib module are more effective.

Enforcing some chars in to\_char function

Sometimes we needs to show some chars, that are same like special chars (has some semantic) from mask. Text in quotes (in format mask) is not interpreted and it is showed without changes:

postgres=# select to\_char(current\_timestamp,'YYYY-MM-DDTHH24:MI:SS');

 to\_char

-----------------------

 2010-02-07THH24:38:10

(1 row)

postgres=# select to\_char(current\_timestamp,'YYYY-MM-DD"T"HH24:MI:SS');

 to\_char

---------------------

 2010-02-07T07:38:22

(1 row)

Author: Milen A. Radev

Barrier against to unwanted conversion char(n) to text

Almost all string functions have parameters of text type. PostgreSQL ensures automatic conversion from char(n) to varchar. But there are a few cases where this behave causes problems - mainly for older applications, where char(n) type was used (there is a clean advice - don't use this type). In some legacy application is comparison between char(n) type and literal constant with spaces and it doesn't work in PostgreSQL now:

SELECT ...

 WHERE substr(somecolumn,1,4) = 'ab ';

It doesn't work, because substr function has text parameter - and returns text and 'ab' is not equal to 'ab '. But we can overload substr function with char(n) datatype (it is possible because varchar and char uses same binary format) - and it doesn't do unwanted conversion from char(n) to text:

create or replace function substr(character, int, int) returns character as $$

select substr($1::cstring::text,$2,$3)

$$ language sql;

Tome Lane wrote more effective proposal:

create function substr(char,int,int) returns char

 strict immutable language internal as 'text\_substr' ;

It is little bit dirty trick, but 100% functional.

postgres=# create table f(a character(5));

CREATE TABLE

postgres=# insert into f values('a'),('ab'),('abc');

INSERT 0 3

postgres=# select \* from f;

 a

-------

 a

 ab

 abc

(3 rows)

postgres=# select \* from f where substr(a,1,3) = 'a ';

 a

-------

 a

(1 row)

postgres=# select \* from f where substr(a,1,3) = 'ab ';

 a

-------

 ab

(1 row)

Get functions's DDL command

PostgreSQL allows simple way to get full source code of any function:

postgres=# select oid from pg\_proc where proname = 'fu';

 oid

-------

 16389

(1 row)

postgres=# select pg\_get\_functiondef(16389);

 pg\_get\_functiondef

---------------------------------------------------------------------

 CREATE OR REPLACE FUNCTION public.fu(anyelement)

 RETURNS anyelement

 LANGUAGE plpgsql

 AS $function$

begin

 raise notice '%', $1;

 return $1;

end; $function$+

(1 row)

from Russian alphabet (Cyrillic) to ASCII

source: <http://leha.papulovskiy.ru/blog/>

CREATE OR REPLACE FUNCTION urltranslit(text) RETURNS text as $$

SELECT

regexp\_replace(

 replace(

 replace(

 replace(

 replace(

 replace(

 replace(

 replace(

 translate(

 lower($1),

 'абвгдеёзийклмнопрстуфхыэъь',

 'abvgdeezijklmnoprstufhye'

 ), 'ж', 'zh'

 ), 'ц', 'ts'

 ), 'ч', 'ch'

 ), 'ш', 'sh'

 ), 'щ', 'sch'

 ), 'ю', 'yu'

 ), 'я', 'ya'

 )

 ,

 '[^a-z]+',

 '-',

 'g'

)

$$ LANGUAGE SQL;