ΤΟΜΟΣ 2 · Η ΣΥΝΕΧΕΙΑ ΤΟΥ «SQL ΜΕ POSTGRESQL»

PostgreSQL

σε production

Από το PREPARE μέχρι το failover.

26 κεφάλαια, 142 ασκήσεις

ΚΑΘΕ ΕΝΤΟΛΗ ΕΚΤΕΛΕΣΤΗΚΕ ΣΕ ΠΡΑΓΜΑΤΙΚΟΥΣ SERVERS POSTGRESQL 18.6

ΠΡΙΝ ΑΠΟ ΤΟ ΠΡΩΤΟ ΚΕΦΑΛΑΙΟ

Η βάση δουλεύει. Τώρα πρέπει να μην πέσει.

Ο πρώτος τόμος δίδαξε τη γλώσσα και τη μηχανή της. Queries, indexes, plans, MVCC και locks. Αυτός ο τόμος ξεκινά από εκεί και απαντά σε ό,τι ρωτά κανείς μόλις μια βάση αρχίσει να εξυπηρετεί πραγματικούς χρήστες. Πώς ρυθμίζεται ο server, τι κάνει όταν πέφτει το ρεύμα, πώς κρατάς backups που επαναφέρονται, πώς στήνεις standbys και κάνεις failover, πώς αλλάζεις σχήμα χωρίς να σταματήσεις την εφαρμογή.

Το πρώτο μέρος κλείνει κενά της SQL. Domains και enums, prepared statements, dynamic SQL, LISTEN και NOTIFY, δείγματα από μεγάλους πίνακες. Το δεύτερο ανοίγει τον server. Το τρίτο και το τέταρτο είναι backups, point in time recovery, replication και failover. Το πέμπτο είναι PgBouncer, migrations και bloat. Το τελευταίο είναι τρεις extensions, foreign data wrappers, pgvector και PostGIS.

Θεωρεί ότι ξέρεις όσα έχει ο πρώτος τόμος και ότι δουλεύεις άνετα σε terminal. Όπου χρειάζεται κάτι από τον πρώτο τόμο, το κείμενο λέει σε ποιο κεφάλαιο βρίσκεται.

Πώς δουλεύεται

Τα πέντε πρώτα κεφάλαια χρησιμοποιούν τη βάση του πρώτου τόμου. Από το κεφάλαιο 6 και μετά στήνεις δικούς σου servers σε ένα εργαστήριο στο `~/pglab`, με θύρες από 5501 και πάνω, χωρίς να αγγίξεις την PostgreSQL που ίσως έχεις ήδη. Κάθε εντολή του βιβλίου, SQL ή shell, εκτελέστηκε σε πραγματικούς servers PostgreSQL 18.6, και η έξοδος κάτω από αυτήν είναι η πραγματική. Χρόνοι, process IDs, LSNs και ώρες θα είναι διαφορετικά στη δική σου εκτέλεση, τα υπόλοιπα όχι.

Κάθε κεφάλαιο κλείνει με ασκήσεις. Κάτω από κάθε εκφώνηση υπάρχει το αναμενόμενο αποτέλεσμα και οι λύσεις με εξηγήσεις είναι σε χωριστό τόμο.

Ένα backup που δεν έχεις επαναφέρει δεν είναι backup. Ένας standby που δεν έχεις κάνει promote δεν είναι failover. Σε αυτόν τον τόμο τα κάνουμε όλα.

Ο ΧΑΡΤΗΣ ΤΟΥ ΒΙΒΛΙΟΥ

Περιεχόμενα

ΜΕΡΟΣ Ι

Ό,τι έλειπε από την SQL

Ο πρώτος τόμος έφτασε μέχρι τα query plans, τα locks και την Row Level Security. Πριν περάσουμε στον server, κλείνουμε πέντε κενά της γλώσσας που συναντάς σε κάθε σοβαρή εφαρμογή. Δικοί σου τύποι δεδομένων, prepared statements, dynamic SQL, ειδοποιήσεις από τη βάση και δείγματα από μεγάλους πίνακες.

ΚΕΦΑΛΑΙΟ 1

Domains, enums και composite types

Οι ενσωματωμένοι τύποι λένε τι μορφή έχει μια τιμή. Δεν λένε τι σημαίνει. Ένα email είναι `text`, αλλά δεν είναι κάθε `text` email. Η PostgreSQL σε αφήνει να ορίσεις δικούς σου τύπους πάνω στους ενσωματωμένους. Τα domains προσθέτουν κανόνες, τα enums περιορίζουν τις τιμές σε μια λίστα και οι composite types ομαδοποιούν πολλά πεδία σε ένα.

Η βάση του τόμου

Όλα τα κεφάλαια του πρώτου μέρους χρησιμοποιούν τη βάση `sqlbook` του πρώτου τόμου, με τους ίδιους πελάτες, προϊόντα και παραγγελίες. Αν δεν την έχεις, τη φτιάχνεις με το script του φακέλου `data`.

TERMINAL

```
bash data/setup.sh
psql -d sqlbook
```

Όπως και στον πρώτο τόμο, κάθε αποτέλεσμα κάτω από ένα query είναι το πραγματικό αποτέλεσμα στην PostgreSQL 18.

Ο ίδιος κανόνας σε πολλά σημεία

Ο πίνακας `customers` έχει email και τηλέφωνο. Το ίδιο και ο `suppliers`. Αν θέλεις να ελέγχεις ότι ένα email έχει τη σωστή μορφή, θα έγραφες το ίδιο `CHECK` σε κάθε πίνακα. Αν αύριο αλλάξει ο κανόνας, θα έπρεπε να βρεις όλα τα σημεία.

Ένα **domain** είναι ένας τύπος που βασίζεται σε άλλον και προσθέτει constraints. Ορίζεις τον κανόνα μία φορά και τον χρησιμοποιείς όπου θα χρησιμοποιούσες τον βασικό τύπο.

SQL

```
CREATE DOMAIN email AS text
  CHECK (VALUE ~ '^[^@\s]+@[^@\s]+\.[a-z]+$');

CREATE DOMAIN positive_money AS numeric(10,2)
  CHECK (VALUE > 0);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE DOMAIN

CREATE DOMAIN
```

Μέσα στο `CHECK` η λέξη `VALUE` είναι η τιμή που ελέγχεται. Τώρα ένας νέος πίνακας χρησιμοποιεί τους τύπους σαν να ήταν ενσωματωμένοι.

SQL

```
CREATE TABLE newsletter_signups (
  id         integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  address    email NOT NULL UNIQUE,
  voucher    positive_money,
  created_at timestamptz NOT NULL DEFAULT now()
);

INSERT INTO newsletter_signups (address, voucher)
VALUES ('anna@example.gr', 5), ('petros@example.gr', NULL);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 2
```

SQL

```
INSERT INTO newsletter_signups (address) VALUES ('anna at example.gr');
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  value for domain email violates check constraint "email_check"
```

Το μήνυμα λάθους αναφέρει το domain και όχι τον πίνακα. Αυτό βοηθά όταν ο ίδιος τύπος υπάρχει σε δέκα πίνακες. Βλέπεις αμέσως ποιος κανόνας παραβιάστηκε. Το `NULL` στο `voucher` πέρασε, γιατί ένα `CHECK` αποτυγχάνει μόνο όταν η συνθήκη βγει ψευδής. Το `NULL > 0` δεν είναι ψευδές.

Ένα domain λειτουργεί και σε casts, σε παραμέτρους functions και σε μεταβλητές PL/pgSQL. Ο έλεγχος γίνεται όπου μια τιμή μετατρέπεται στο domain.

SQL

```
SELECT '-3'::positive_money;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  value for domain positive_money violates check constraint "positive_money_check"
```

ΠΑΓΙΔΑ

Ένα domain μπορεί να έχει και `NOT NULL`, αλλά καλύτερα να το αποφύγεις. Ένα `LEFT JOIN` ή ένα subquery που δεν βρίσκει γραμμή παράγει `NULL` σε στήλη του domain χωρίς να γίνει έλεγχος, οπότε η εγγύηση δεν ισχύει παντού. Βάλε το `NOT NULL` στη στήλη του πίνακα, όπως στο `address` παραπάνω.

Αλλαγή κανόνα σε domain που χρησιμοποιείται

Οι κανόνες αλλάζουν. Ας πούμε ότι θέλεις να επιτρέπεις μόνο πεζά γράμματα στα emails. Ένα νέο constraint ελέγχει όλες τις υπάρχουσες τιμές σε κάθε πίνακα που χρησιμοποιεί το domain.

SQL

```
INSERT INTO newsletter_signups (address) VALUES ('Giorgos@example.gr');
```

ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 1
```

SQL

```
ALTER DOMAIN email ADD CONSTRAINT email_lowercase CHECK (VALUE = lower(VALUE));
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  column "address" of table "newsletter_signups" contains values that violate the new constraint
```

Η αλλαγή απέτυχε, γιατί μια υπάρχουσα τιμή δεν τηρεί τον νέο κανόνα. Με `NOT VALID` το constraint ισχύει από εδώ και πέρα για νέες τιμές και οι παλιές μένουν όπως είναι. Όταν τις διορθώσεις, το `VALIDATE CONSTRAINT` τις ελέγχει όλες και το constraint γίνεται πλήρες.

SQL

```
ALTER DOMAIN email ADD CONSTRAINT email_lowercase
  CHECK (VALUE = lower(VALUE)) NOT VALID;

UPDATE newsletter_signups SET address = lower(address);

ALTER DOMAIN email VALIDATE CONSTRAINT email_lowercase;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER DOMAIN

UPDATE 3

ALTER DOMAIN
```

Το ίδιο μοτίβο, πρώτα `NOT VALID` και μετά `VALIDATE`, θα το ξαναδούμε στο κεφάλαιο 22 για constraints σε μεγάλους πίνακες, όπου κάνει τη διαφορά ανάμεσα σε λίγα δευτερόλεπτα lock και σε πολλά λεπτά.

Enums

Η στήλη `orders.status` επιτρέπει έξι τιμές με ένα `CHECK`. Ένα **enum** είναι τύπος με σταθερή λίστα τιμών, που έχουν και σειρά. Η σειρά είναι αυτή με την οποία τις δήλωσες.

SQL

```
CREATE TYPE ticket_priority AS ENUM ('low', 'normal', 'high', 'urgent');

CREATE TABLE tickets (
  id       integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  subject  text NOT NULL,
  priority ticket_priority NOT NULL DEFAULT 'normal'
);

INSERT INTO tickets (subject, priority) VALUES
  ('Δεν φτάνει το email επιβεβαίωσης', 'high'),
  ('Αλλαγή διεύθυνσης', 'low'),
  ('Διπλή χρέωση κάρτας', 'urgent'),
  ('Ερώτηση για εγγύηση', DEFAULT);

SELECT id, subject, priority FROM tickets ORDER BY priority DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TYPE

CREATE TABLE

INSERT 0 4

 id |             subject              | priority
----+----------------------------------+----------
  3 | Διπλή χρέωση κάρτας              | urgent
  1 | Δεν φτάνει το email επιβεβαίωσης | high
  4 | Ερώτηση για εγγύηση              | normal
  2 | Αλλαγή διεύθυνσης                | low
(4 rows)
```

Το `ORDER BY priority` ακολουθεί τη σειρά του enum και όχι την αλφαβητική. Με `text` το `urgent` θα ερχόταν τελευταίο και το `high` πρώτο. Οι συγκρίσεις δουλεύουν επίσης με τη σειρά της δήλωσης.

SQL

```
SELECT subject FROM tickets WHERE priority >= 'high' ORDER BY id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
             subject
----------------------------------
 Δεν φτάνει το email επιβεβαίωσης
 Διπλή χρέωση κάρτας
(2 rows)
```

SQL

```
INSERT INTO tickets (subject, priority) VALUES ('Κάτι', 'critical');
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  invalid input value for enum ticket_priority: "critical"
LINE 1: ... INTO tickets (subject, priority) VALUES ('Κάτι', 'critical'...
                                                             ^
```

Νέα τιμή προστίθεται με `ALTER TYPE`, σε όποια θέση θέλεις. Τη βάζουμε ανάμεσα στο `normal` και το `high`.

SQL

```
ALTER TYPE ticket_priority ADD VALUE 'elevated' AFTER 'normal';

SELECT enum_range(NULL::ticket_priority);
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TYPE

            enum_range
-----------------------------------
 {low,normal,elevated,high,urgent}
(1 row)
```

Ένα enum έχει δύο περιορισμούς που πρέπει να ξέρεις πριν το διαλέξεις. Δεν μπορείς να αφαιρέσεις τιμή και δεν μπορείς να αλλάξεις τη σειρά τους. Αν μια τιμή δεν χρειάζεται πια, μένει στον τύπο για πάντα, εκτός αν φτιάξεις νέο τύπο και μετατρέψεις τη στήλη.

Επιλογή	Καλό για	Κόστος
enum	μικρές λίστες που σπάνια αλλάζουν και έχουν σειρά	δεν αφαιρείς τιμές
`text` με `CHECK`	λίστες που αλλάζουν πότε πότε	αλλαγή του `CHECK` ελέγχει όλο τον πίνακα
πίνακας αναφοράς με foreign key	λίστες που αλλάζουν συχνά ή έχουν επιπλέον πεδία	ένα join παραπάνω

Ο πίνακας αναφοράς είναι το πιο ευέλικτο. Μπορείς να προσθέσεις περιγραφή, μετάφραση ή σειρά ταξινόμησης χωρίς να αλλάξεις σχήμα. Η εφαρμογή μπορεί επίσης να διαβάζει τη λίστα από τη βάση.

Composite types

Ένας **composite type** είναι μια ομάδα πεδίων με όνομα, σαν μια γραμμή χωρίς πίνακα. Μπορεί να γίνει τύπος στήλης ή τύπος που επιστρέφει μια function.

SQL

```
CREATE TYPE address AS (
  street      text,
  city        text,
  postal_code text
);

CREATE TABLE warehouses (
  id       integer PRIMARY KEY,
  name     text NOT NULL,
  location address NOT NULL
);

INSERT INTO warehouses VALUES
  (1, 'Κεντρική αποθήκη', ROW('Πειραιώς 120', 'Αθήνα', '11854')),
  (2, 'Αποθήκη Βορρά', ROW('26ης Οκτωβρίου 90', 'Θεσσαλονίκη', '54627'));

SELECT name, (location).city, (location).postal_code
FROM warehouses
ORDER BY id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TYPE

CREATE TABLE

INSERT 0 2

       name       |    city     | postal_code
------------------+-------------+-------------
 Κεντρική αποθήκη | Αθήνα       | 11854
 Αποθήκη Βορρά    | Θεσσαλονίκη | 54627
(2 rows)
```

Οι παρενθέσεις στο `(location).city` χρειάζονται. Χωρίς αυτές η PostgreSQL θα διάβαζε το `location` ως όνομα πίνακα και το `city` ως στήλη του. Ένα πεδίο αλλάζει με την ίδια σύνταξη στο `SET`, χωρίς τις παρενθέσεις.

SQL

```
UPDATE warehouses SET location.postal_code = '11855' WHERE id = 1;

SELECT location FROM warehouses WHERE id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
UPDATE 1

           location
-------------------------------
 ("Πειραιώς 120",Αθήνα,11855)
(1 row)
```

Στις στήλες πινάκων οι composite types χρησιμοποιούνται σπάνια. Ένα ξεχωριστό πεδίο ανά στήλη είναι πιο εύκολο να μπει σε index, σε constraint και σε foreign key. Η πραγματική τους χρησιμότητα είναι αλλού.

Κάθε πίνακας είναι και τύπος

Για κάθε πίνακα η PostgreSQL ορίζει αυτόματα έναν composite type με το ίδιο όνομα και τις ίδιες στήλες. Γι' αυτό μπορείς να γράψεις το όνομα του alias μόνο του στο `SELECT` και να πάρεις ολόκληρη τη γραμμή ως μία τιμή.

SQL

```
SELECT c FROM customers AS c WHERE c.id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                                                c
-----------------------------------------------------------------------------------------
 (1,Μαρία,"Παπαδοπούλου",maria.p@example.gr,6944123456,1,1988-04-12,t,"2024-11-03 10:15:00+02")
(1 row)
```

Αυτή η μία τιμή περνά σε functions. Το `to_jsonb` μετατρέπει ολόκληρη τη γραμμή σε JSON, με τα ονόματα των στηλών ως κλειδιά. Είναι ο πιο σύντομος τρόπος να φτιάξεις ένα audit log ή μια απάντηση API.

SQL

```
SELECT jsonb_pretty(to_jsonb(c) - 'created_at') AS customer
FROM customers AS c
WHERE c.id = 2;
```

ΑΠΟΤΕΛΕΣΜΑ

```
             customer
-------------------------------------
 {                                  +
     "id": 2,                       +
     "email": "gpapad@example.gr",+
     "phone": "6977001122",         +
     "city_id": 2,                  +
     "last_name": "Παπαδόπουλος", +
     "birth_date": "1979-09-30",  +
     "first_name": "Γιώργος",     +
     "newsletter": false            +
 }
(1 row)
```

Το `jsonb_pretty` απλώς στοιχίζει το JSON σε γραμμές για να διαβάζεται.

Και μια function μπορεί να δέχεται ολόκληρη γραμμή ως όρισμα. Ο τύπος του ορίσματος είναι το όνομα του πίνακα.

```sql
CREATE FUNCTION display_name(c customers)
RETURNS text
LANGUAGE sql
IMMUTABLE
RETURN c.first_name || ' ' || left(c.last_name, 1) || '.';

SELECT c.id, display_name(c) FROM customers AS c WHERE c.id <= 3 ORDER BY c.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 id | display_name
----+--------------
  1 | Μαρία Π.
  2 | Γιώργος Π.
  3 | Ελένη Ν.
(3 rows)
```

Μια τέτοια function μπορεί να κληθεί και σαν να ήταν στήλη, με τη σύνταξη `c.display_name`. Το χρησιμοποιούν ORMs και εργαλεία GraphQL για «υπολογισμένα πεδία». Εμείς προτιμάμε τη ρητή κλήση, που φαίνεται αμέσως ότι είναι function.

Functions που επιστρέφουν πολλά πεδία

Μια function επιστρέφει μία τιμή. Αν χρειάζεσαι δύο, ένας composite type τις ενώνει σε μία.

```sql
CREATE TYPE price_range AS (low numeric, high numeric);

CREATE FUNCTION category_price_range(p_category integer)
RETURNS price_range
LANGUAGE sql
STABLE
RETURN (SELECT ROW(min(price), max(price))::price_range
        FROM products WHERE category_id = p_category);

SELECT category_price_range(9);

SELECT (category_price_range(9)).*;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TYPE

CREATE FUNCTION

 category_price_range
----------------------
 (24.90,329.00)
(1 row)

  low  |  high
-------+--------
 24.90 | 329.00
(1 row)
```

Το `.*` απλώνει τα πεδία σε στήλες. Προσοχή σε ένα σημείο. Όταν γράφεις `(f(x)).*` στο `SELECT`, η PostgreSQL μπορεί να καλέσει τη function μία φορά για κάθε πεδίο. Για ακριβές functions, κάλεσέ τη στο `FROM`, όπου εκτελείται μία φορά.

SQL

```
SELECT r.low, r.high
FROM category_price_range(9) AS r;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  low  |  high
-------+--------
 24.90 | 329.00
(1 row)
```

Πού βρίσκεις τους τύπους σου

Το psql έχει δύο εντολές για τους δικούς σου τύπους. Το `\dT` δείχνει όλους τους τύπους και το `\dD` τα domains με τα constraints τους. Ο πίνακας του `\dD` είναι φαρδύς, οπότε τον βλέπουμε με `\x`, που τυπώνει κάθε γραμμή ως λίστα πεδίων.

PSQL

```
\dT
\x on
\dD email
\x off
```

ΑΠΟΤΕΛΕΣΜΑ

```
           List of data types
 Schema |      Name       | Description
--------+-----------------+-------------
 public | address         | ∅
 public | email           | ∅
 public | positive_money  | ∅
 public | price_range     | ∅
 public | ticket_priority | ∅
```

```
(5 rows)

Expanded display is on.
List of domains
-[ RECORD 1 ]-----------------------------------------------------------------------------
Schema    | public
Name      | email
Type      | text
Collation | ∅
Nullable  | ∅
Default   | ∅
Check     | CHECK (VALUE ~ '^[^@\s]+@[^@\s]+\.[a-z]+$'::text) CHECK (VALUE = lower(VALUE))

Expanded display is off.
```

ΚΡΑΤΑ ΑΥΤΟ

Ένα domain είναι βασικός τύπος με constraints, που ορίζεις μία φορά και χρησιμοποιείς παντού. Νέο constraint σε domain που χρησιμοποιείται προσθέτεις με `NOT VALID` και ελέγχεις με `VALIDATE`. Ένα enum έχει σταθερή λίστα τιμών με σειρά, στην οποία προσθέτεις αλλά δεν αφαιρείς. Ένας composite type ενώνει πεδία σε μία τιμή. Κάθε πίνακας είναι και composite type, οπότε μια γραμμή περνά ολόκληρη σε functions και στο `to_jsonb`.

Ασκήσεις

ΑΣΚΗΣΗ 1.1 Κινητό τηλέφωνο

Φτιάξε domain `mobile_gr` πάνω σε `text` που δέχεται μόνο δέκα ψηφία που ξεκινούν από `69`. Δοκίμασε τη μετατροπή του `'6944123456'` και του `'2105550101'` στο domain σε δύο χωριστές εντολές.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE DOMAIN

     ok
------------
 6944123456
(1 row)

ERROR:  value for domain mobile_gr violates check constraint "mobile_gr_check"
```

ΑΣΚΗΣΗ 1.2 Κανόνας που δεν τηρούν όλοι

Φτιάξε domain `phone_digits` πάνω σε `text` με `CHECK (VALUE ~ '^[0-9]+$')`. Βρες πρώτα ποιοι προμηθευτές έχουν τηλέφωνο που δεν τηρεί τον κανόνα και μετά δοκίμασε να αλλάξεις τον τύπο της στήλης `suppliers.phone` σε `phone_digits`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE DOMAIN

 id |     name      |    phone
----+---------------+--------------
  5 | Nordic Office | +46855500404
(1 row)

ERROR:  value for domain phone_digits violates check constraint "phone_digits_check"
```

ΑΣΚΗΣΗ 1.3 Σειρά ταξινόμησης από enum

Φτιάξε enum `shipping_speed` με τιμές `economy`, `standard` και `express`, με αυτή τη σειρά. Σε ένα `VALUES` με τις τρεις τιμές ως text, ανακατεμένες, ταξινόμησέ τες από την πιο γρήγορη στην πιο αργή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TYPE

  speed
----------
 express
 standard
 economy
(3 rows)
```

ΑΣΚΗΣΗ 1.4 Νέα προτεραιότητα στην αρχή

Πρόσθεσε στο enum `ticket_priority` του κεφαλαίου την τιμή `trivial` πριν από το `low` και δείξε τη νέα λίστα τιμών με τη σειρά τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TYPE

 priority
----------
 trivial
 low
 normal
 elevated
 high
 urgent
(6 rows)
```

ΑΣΚΗΣΗ 1.5 Ολόκληρη η παραγγελία σε JSON

Επέστρεψε την παραγγελία 5 ως ένα JSON αντικείμενο που περιέχει όλες τις στήλες της γραμμής του `orders`, με τη χρήση του row type του πίνακα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                    order_json
-------------------------------------------------
 {                                               +
     "id": 5,                                    +
     "status": "delivered",                      +
     "created_at": "2025-02-06T22:45:00+02:00",+
     "shipped_at": "2025-02-11T00:45:00+02:00",+
     "coupon_code": null,                        +
     "customer_id": 5,                           +
     "shipping_city_id": 4                       +
 }
(1 row)
```

ΑΣΚΗΣΗ 1.6 Σύνοψη παραγγελίας με composite type

Φτιάξε composite type `order_summary` με πεδία `items integer` και `total numeric`. Φτιάξε function `summarize_order(p_order integer)` που επιστρέφει αυτόν τον τύπο με το πλήθος των γραμμών και το σύνολο `quantity * unit_price` της παραγγελίας. Κάλεσέ τη στο `FROM` για την παραγγελία 5.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TYPE

CREATE FUNCTION

 items | total
-------+-------
     1 | 34.90
(1 row)
```

ΚΕΦΑΛΑΙΟ 2

Prepared statements και generic plans

Κάθε query περνά από parsing, planning και εκτέλεση. Ένα prepared statement κάνει το parsing μία φορά και κρατά το plan για να το ξαναχρησιμοποιήσει. Αυτό κερδίζει χρόνο και χωρίζει τον κώδικα SQL από τις τιμές, αλλά φέρνει ένα νέο ερώτημα. Είναι καλό ένα plan για όλες τις τιμές μιας παραμέτρου;

PREPARE και EXECUTE

Ως τώρα κάθε query έφτανε στον server ως ένα κείμενο με τις τιμές μέσα του. Ο server το διάβαζε, έφτιαχνε plan, το εκτελούσε και τα ξεχνούσε όλα. Με `PREPARE` δίνεις όνομα σε ένα query με **παραμέτρους** `$1`, `$2` και ούτω καθεξής. Με `EXECUTE` το τρέχεις δίνοντας τιμές.

SQL

```
PREPARE customer_orders(integer) AS
  SELECT id, status, created_at::date AS day
  FROM orders
  WHERE customer_id = $1
  ORDER BY created_at;

EXECUTE customer_orders(5);
```

ΑΠΟΤΕΛΕΣΜΑ

```
PREPARE

 id  |  status   |    day
-----+-----------+------------
   5 | delivered | 2025-02-06
   6 | delivered | 2025-02-07
  51 | delivered | 2025-10-27
  83 | delivered | 2026-02-05
  87 | cancelled | 2026-02-19
  90 | delivered | 2026-02-23
 137 | delivered | 2026-06-06
(7 rows)
```

Ο τύπος κάθε παραμέτρου δηλώνεται στην παρένθεση μετά το όνομα. Αν τον παραλείψεις, η PostgreSQL τον συμπεραίνει από το πού χρησιμοποιείται η παράμετρος.

Το σημαντικό είναι ότι η τιμή `5` δεν γίνεται ποτέ μέρος του κειμένου SQL. Ταξιδεύει χωριστά και ο server τη χειρίζεται ως τιμή, όχι ως κώδικα. Γι' αυτό οι παράμετροι είναι η ουσιαστική άμυνα απέναντι στο SQL injection. Μια τιμή όπως `5; DROP TABLE orders` δεν μπορεί να γίνει εντολή, απλώς αποτυγχάνει ως μη έγκυρος ακέραιος.

SQL

```
EXECUTE customer_orders('5; DROP TABLE orders');
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  invalid input syntax for type integer: "5; DROP TABLE orders"
LINE 1: EXECUTE customer_orders('5; DROP TABLE orders')
                                ^
```

Στην πράξη δεν γράφεις `PREPARE` με το χέρι. Οι drivers των γλωσσών το κάνουν για σένα μέσω του πρωτοκόλλου της PostgreSQL, που έχει χωριστά μηνύματα για parse, bind και execute. Όταν γράφεις `WHERE id = $1` στο node-postgres ή `WHERE id = ?` στο JDBC, στον server φτάνει ένα prepared statement. Το `PREPARE` της SQL είναι ο τρόπος να δεις τη μηχανή από το psql.

Ένα prepared statement ζει στη σύνδεση

Τα prepared statements ανήκουν στη σύνδεση που τα έφτιαξε. Μια άλλη σύνδεση δεν τα βλέπει και όταν κλείσει η σύνδεση χάνονται.

SQL · ΣΥΝΕΔΡΙΑ Β

```
EXECUTE customer_orders(5);
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Β

```
ERROR:  prepared statement "customer_orders" does not exist
```

Η σύνδεση βλέπει τα δικά της στο `pg_prepared_statements`.

SQL

```
SELECT name, parameter_types, generic_plans, custom_plans
FROM pg_prepared_statements;
```

ΑΠΟΤΕΛΕΣΜΑ

```
      name       | parameter_types | generic_plans | custom_plans
-----------------+-----------------+---------------+--------------
 customer_orders | {integer}       |             0 |            1
(1 row)
```

Αυτή η λεπτομέρεια θα μας απασχολήσει στο κεφάλαιο 21. Ένας connection pooler μοιράζει λίγες συνδέσεις του server σε πολλούς clients. Αν ο client ετοιμάσει ένα statement σε μία σύνδεση και το εκτελέσει σε άλλη, αποτυγχάνει ακριβώς όπως η συνεδρία Β.

Custom και generic plans

Οι στήλες `generic_plans` και `custom_plans` μετρούν δύο διαφορετικούς τρόπους εκτέλεσης. Ένα **custom plan** φτιάχνεται σε κάθε `EXECUTE` για τις συγκεκριμένες τιμές, σαν να είχες γράψει τις τιμές μέσα στο

query. Ένα **generic plan** φτιάχνεται μία φορά χωρίς να ξέρει τις τιμές και ξαναχρησιμοποιείται. Ο πρώτος τρόπος δίνει καλύτερο plan, ο δεύτερος γλιτώνει το planning.

Η PostgreSQL αποφασίζει μόνη της. Οι πρώτες πέντε εκτελέσεις παίρνουν custom plan. Από την έκτη συγκρίνει το εκτιμώμενο κόστος του generic plan με το μέσο κόστος των custom plans που έφτιαξε. Αν το generic δεν είναι ακριβότερο, το κρατά και σταματά να κάνει planning.

Για να το δούμε στην πράξη χρειαζόμαστε δεδομένα με ανισοκατανομή. Στις δύο εκατομμύρια παραγγελίες του εργαστηρίου το `delivered` είναι η συντριπτική πλειονότητα και το `pending` είναι σπάνιο.

SQL

```
CREATE INDEX orders_status_idx ON lab.orders (status);
ANALYZE lab.orders;

SELECT status, count(*) FROM lab.orders GROUP BY status ORDER BY count(*);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

ANALYZE

  status   |  count
-----------+---------
 paid      |    1321
 shipped   |    1330
 pending   |    1349
 refunded  |   59898
 cancelled |  139461
 delivered | 1796641
(6 rows)
```

Ετοιμάζουμε ένα query ανά κατάσταση και το εκτελούμε πέντε φορές για το `delivered`, όπως θα έκανε μια εφαρμογή όπου η πρώτη οθόνη δείχνει τις παραδοτέες παραγγελίες.

SQL

```
PREPARE by_status(text) AS
  SELECT count(*), sum(total) FROM lab.orders WHERE status = $1;

EXECUTE by_status('delivered');
EXECUTE by_status('delivered');
EXECUTE by_status('delivered');
EXECUTE by_status('delivered');
EXECUTE by_status('delivered');
```

SQL

```
SELECT name, generic_plans, custom_plans
FROM pg_prepared_statements WHERE name = 'by_status';
```

ΑΠΟΤΕΛΕΣΜΑ

```
   name    | generic_plans | custom_plans
-----------+---------------+--------------
 by_status |             0 |            5
(1 row)
```

Πέντε custom plans. Στην έκτη εκτέλεση η PostgreSQL συγκρίνει. Το `EXPLAIN` δείχνει ποιο plan θα χρησιμοποιηθεί.

SQL

```
EXPLAIN (COSTS OFF) EXECUTE by_status('delivered');
```

ΑΠΟΤΕΛΕΣΜΑ

```
Finalize Aggregate
  ->  Gather
        Workers Planned: 2
        ->  Partial Aggregate
              ->  Parallel Bitmap Heap Scan on orders
                    Recheck Cond: (status = $1)
                    ->  Bitmap Index Scan on orders_status_idx
                          Index Cond: (status = $1)
```

Το `$1` στη συνθήκη σημαίνει ότι αυτό είναι το generic plan. Χωρίς τιμή, ο planner υπέθεσε ότι κάθε κατάσταση έχει περίπου το ένα έκτο των γραμμών. Το κόστος βγήκε κοντά στο κόστος των custom plans για το `delivered`. Άρα το κράτησε. Τώρα δες τι γίνεται όταν η εφαρμογή ζητήσει τις λίγες εκκρεμείς παραγγελίες.

SQL

```
EXPLAIN (ANALYZE, COSTS OFF, BUFFERS OFF) EXECUTE by_status('pending');
```

ΑΠΟΤΕΛΕΣΜΑ

```
Finalize Aggregate (actual time=2.270..2.980 rows=1.00 loops=1)
  ->  Gather (actual time=0.227..2.974 rows=3.00 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate (actual time=0.094..0.094 rows=1.00 loops=3)
              ->  Parallel Bitmap Heap Scan on orders (actual time=0.056..0.076 rows=449.67 loops=3)
                    Recheck Cond: (status = $1)
                    Heap Blocks: exact=34
                    ->  Bitmap Index Scan on orders_status_idx (actual time=0.053..0.053 rows=1349.00 loops=1)
                          Index Cond: (status = $1)
                          Index Searches: 1
Planning Time: 0.001 ms
Execution Time: 2.988 ms
```

Το `Planning Time` είναι σχεδόν μηδέν, γιατί δεν έγινε planning. Το plan όμως είναι το ίδιο plan με parallel workers και bitmap scan που φτιάχτηκε για ένα έκτο του πίνακα, ενώ τώρα χρειαζόμαστε 1.349 γραμμές. Σύγκρινε με το custom plan για την ίδια τιμή. Η ρύθμιση `plan_cache_mode` επιβάλλει τον τρόπο.

```sql
SET plan_cache_mode = force_custom_plan;
EXPLAIN (ANALYZE, COSTS OFF, BUFFERS OFF) EXECUTE by_status('pending');
RESET plan_cache_mode;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET

Aggregate (actual time=0.125..0.125 rows=1.00 loops=1)
  ->  Index Scan using orders_status_idx on orders (actual time=0.005..0.069 rows=1349.00 loops=1)
        Index Cond: (status = 'pending'::text)
        Index Searches: 1
Planning Time: 0.030 ms
Execution Time: 0.130 ms

RESET
```

Με την τιμή μπροστά του ο planner διάλεξε ένα απλό `Index Scan` χωρίς workers. Ο χρόνος εκτέλεσης είναι πολλαπλάσια μικρότερος και η διαφορά μεγαλώνει όσο μεγαλώνει ο πίνακας.

Η `plan_cache_mode` έχει τρεις τιμές. Η `auto` είναι η προεπιλογή που περιγράψαμε. Η `force_custom_plan` κάνει planning σε κάθε εκτέλεση και η `force_generic_plan` χρησιμοποιεί πάντα το generic. Μπορείς να την ορίσεις για μια σύνδεση, για έναν ρόλο ή για μία function.

ΠΑΓΙΔΑ

Αν ένα query είναι γρήγορο στο psql και αργό από την εφαρμογή, υποψιάσου generic plan. Στο psql γράφεις τιμές μέσα στο query και παίρνεις πάντα custom plan. Η εφαρμογή στέλνει παραμέτρους και μετά από πέντε εκτελέσεις μπορεί να πάρει generic. Δοκίμασε με `PREPARE` και έξι `EXECUTE` για να δεις ό,τι βλέπει η εφαρμογή.

Το generic plan χωρίς PREPARE

Από την έκδοση 16 το `EXPLAIN` δέχεται την επιλογή `GENERIC_PLAN`. Δείχνει το generic plan ενός query με παραμέτρους χωρίς να το ετοιμάσεις και χωρίς να το εκτελέσεις. Είναι βολικό όταν έχεις ένα query από τα logs της εφαρμογής, με τα `$1` και `$2` μέσα του.

```sql
EXPLAIN (GENERIC_PLAN, COSTS OFF)
SELECT id, total FROM lab.orders
WHERE customer_id = $1 AND created_at >= $2;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Gather
  Workers Planned: 2
  ->  Parallel Seq Scan on orders
        Filter: ((created_at >= $2) AND (customer_id = $1))
```

Δεν υπάρχει index στο `customer_id`, οπότε όποια τιμή κι αν έρθει το plan είναι ένα parallel sequential scan.

Όταν αλλάζει ο πίνακας

Ένα αποθηκευμένο plan εξαρτάται από τον πίνακα, τα indexes και τα statistics. Όταν κάποιο από αυτά αλλάξει, η PostgreSQL ακυρώνει το plan και το ξαναφτιάχνει στην επόμενη εκτέλεση. Αυτό γίνεται αυτόματα. Υπάρχει μία περίπτωση όμως που δεν μπορεί να καλυφθεί.

SQL

```
PREPARE city_by_id(integer) AS SELECT * FROM cities WHERE id = $1;
EXECUTE city_by_id(1);

ALTER TABLE cities ADD COLUMN code text;
EXECUTE city_by_id(1);
```

ΑΠΟΤΕΛΕΣΜΑ

```
PREPARE

 id | name  | region | population
----+-------+--------+------------
  1 | Αθήνα | Αττική |     643452
(1 row)

ALTER TABLE

ERROR:  cached plan must not change result type
```

Το `SELECT *` επέστρεφε τέσσερις στήλες την ώρα του `PREPARE`. Μετά το `ALTER TABLE` θα επέστρεφε πέντε. Ο client περιμένει τη μορφή που του δηλώθηκε, οπότε η PostgreSQL αρνείται. Σε μια εφαρμογή με pool συνδέσεων αυτό το σφάλμα εμφανίζεται για λίγα λεπτά μετά από κάθε migration που προσθέτει στήλη, ώσπου να ανανεωθούν όλες οι συνδέσεις. Η λύση είναι να αναφέρεις ρητά τις στήλες αντί για `*` ή να κάνεις `DEALLOCATE` και ξανά `PREPARE`.

SQL

```
DEALLOCATE city_by_id;
PREPARE city_by_id(integer) AS SELECT * FROM cities WHERE id = $1;
EXECUTE city_by_id(1);
```

ΑΠΟΤΕΛΕΣΜΑ

```
DEALLOCATE

PREPARE

 id | name  | region | population | code
----+-------+--------+------------+------
  1 | Αθήνα | Αττική |     643452 | ∅
(1 row)
```

Το `DEALLOCATE ALL` σβήνει όλα τα prepared statements της σύνδεσης.

ΚΡΑΤΑ ΑΥΤΟ

Ένα prepared statement χωρίζει το SQL από τις τιμές και κρατά plan για τις επόμενες εκτελέσεις. Ζει μόνο στη σύνδεση που το έφτιαξε. Οι πέντε πρώτες εκτελέσεις παίρνουν custom plan και μετά η PostgreSQL μπορεί να περάσει σε generic plan, που αγνοεί την τιμή. Σε δεδομένα με ανισοκατανομή αυτό δίνει αργά queries μόνο από την εφαρμογή. Το `plan_cache_mode` το ελέγχει και το `EXPLAIN (GENERIC_PLAN)` δείχνει το generic plan χωρίς εκτέλεση.

Ασκήσεις

ΑΣΚΗΣΗ 2.1 Προϊόντα κατηγορίας

Φτιάξε prepared statement `category_products` με μία παράμετρο `integer` που επιστρέφει όνομα και τιμή των ενεργών προϊόντων μιας κατηγορίας, με φθηνότερο πρώτο. Εκτέλεσέ το για την κατηγορία 9.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
PREPARE

         name          | price
-----------------------+--------
 Ασύρματο ποντίκι      |  24.90
 Webcam HD             |  59.00
 Μηχανικό πληκτρολόγιο |  89.00
 USB-C docking station | 139.00
 Οθόνη 27" 4K          | 329.00
(5 rows)
```

ΑΣΚΗΣΗ 2.2 Πάντα custom για τα σπάνια

Φτιάξε prepared statement `status_count(text)` με `SELECT count(*) FROM lab.orders WHERE status = $1` και εκτέλεσέ το έξι φορές για το `pending`. Δείξε πόσα custom και πόσα generic plans έγιναν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
PREPARE

 count
-------
  1349
(1 row)

 count
-------
  1349
(1 row)

 count
-------
  1349
(1 row)

 count
-------
  1349
(1 row)

 count
-------
  1349
(1 row)

 count
-------
  1349
(1 row)

 generic_plans | custom_plans
---------------+--------------
             0 |            6
(1 row)
```

ΑΣΚΗΣΗ 2.3 Generic plan με δύο παραμέτρους

Δείξε χωρίς εκτέλεση και χωρίς κόστη το generic plan του query που μετρά τις παραγγελίες του `lab.orders` με `total` ανάμεσα σε `$1` και `$2`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Finalize Aggregate
  ->  Gather
        Workers Planned: 2
        ->  Partial Aggregate
              ->  Parallel Seq Scan on orders
                    Filter: ((total >= $1) AND (total <= $2))
```

ΑΣΚΗΣΗ 2.4 Επιβολή generic plan

Με `plan_cache_mode = force_generic_plan` δείξε με `EXPLAIN (COSTS OFF)` το plan του `by_status` για την τιμή `refunded`, χωρίς να αφήσεις τη ρύθμιση αλλαγμένη.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SET

Finalize Aggregate
  ->  Gather
        Workers Planned: 2
        ->  Partial Aggregate
              ->  Parallel Bitmap Heap Scan on orders
                    Recheck Cond: (status = $1)
                    ->  Bitmap Index Scan on orders_status_idx
                          Index Cond: (status = $1)

RESET
```

ΑΣΚΗΣΗ 2.5 Στήλη που προστέθηκε

Φτιάξε prepared statement `supplier_row(integer)` με `SELECT * FROM suppliers WHERE id = $1`, εκτέλεσέ το για τον προμηθευτή 1, πρόσθεσε στήλη `rating smallint` στον `suppliers` και εκτέλεσέ το ξανά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
PREPARE

 id |         name          | city_id |              email               |   phone
----+-----------------------+---------+----------------------------------+------------
  1 | Βιβλιοδιανομή Αττικής |       1 | orders@vivliodianomi.example.gr | 2105550101
(1 row)

ALTER TABLE

ERROR:  cached plan must not change result type
```

ΑΣΚΗΣΗ 2.6 Καθαρή σύνδεση

Σβήσε όλα τα prepared statements της σύνδεσης και δείξε ότι το `pg_prepared_statements` είναι άδειο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
DEALLOCATE ALL

 prepared
----------
        0
(1 row)
```

ΚΕΦΑΛΑΙΟ 3

Dynamic SQL με EXECUTE και format

Οι παράμετροι του προηγούμενου κεφαλαίου αντικαθιστούν τιμές. Δεν μπορούν να αντικαταστήσουν ονόματα πινάκων ή στηλών. Όταν το ίδιο το query εξαρτάται από δεδομένα, όπως ένας πίνακας ανά μήνα ή μια αναφορά για κάθε πίνακα της βάσης, το query πρέπει να φτιαχτεί ως κείμενο και να εκτελεστεί. Αυτό λέγεται dynamic SQL και είναι ασφαλές μόνο αν ξέρεις πώς μπαίνουν τα ονόματα και οι τιμές μέσα στο κείμενο.

Γιατί δεν αρκούν οι παράμετροι

Ένα `$1` μπορεί να σταθεί μόνο εκεί όπου θα στεκόταν μια σταθερή τιμή. Το plan ενός query εξαρτάται από τους πίνακες και τις στήλες του, οπότε αυτά πρέπει να είναι γνωστά πριν έρθουν οι τιμές.

SQL

```
PREPARE count_any(text) AS SELECT count(*) FROM $1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  syntax error at or near "$1"
LINE 1: PREPARE count_any(text) AS SELECT count(*) FROM $1
                                                        ^
```

Για να μετρήσεις τις γραμμές ενός πίνακα που το όνομά του έρχεται ως δεδομένο, πρέπει να φτιάξεις το κείμενο `SELECT count(*) FROM orders` και να το εκτελέσεις. Αυτό γίνεται μέσα σε PL/pgSQL με την εντολή `EXECUTE`, που δεν έχει σχέση με το `EXECUTE` των prepared statements παρά μόνο στο όνομα.

Το format και οι τρεις μορφές του

Πριν εκτελέσουμε κάτι, χρειαζόμαστε ασφαλή τρόπο να φτιάχνουμε το κείμενο. Η function `format` δουλεύει όπως το `printf` άλλων γλωσσών, με τρεις μορφές που έχουν σημασία.

Μορφή	Τι κάνει	Για τι
`%s`	βάζει την τιμή όπως είναι	κείμενο που έχεις ήδη ελέγξει
`%I`	βάζει την τιμή ως identifier, με διπλά εισαγωγικά όπου χρειάζεται	ονόματα πινάκων, στηλών, schemas

Μορφή	Τι κάνει	Για τι
`%L`	βάζει την τιμή ως literal, με μονά εισαγωγικά και escaping	τιμές

SQL

```
SELECT format('%I', 'orders')           AS simple_name,
       format('%I', 'Order Items')      AS odd_name,
       format('%L', 'O''Brien')         AS literal,
       format('%L', NULL::text)         AS null_literal;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 simple_name |   odd_name    |  literal   | null_literal
-------------+---------------+------------+--------------
 orders      | "Order Items" | 'O''Brien' | NULL
(1 row)
```

Το `%I` άφησε το `orders` ως έχει και έβαλε εισαγωγικά στο όνομα με κενό και κεφαλαία. Το `%L` διπλασίασε το μονό εισαγωγικό μέσα στην τιμή, έτσι ώστε να μη μπορεί να κλείσει το string. Για `NULL` έγραψε `NULL` χωρίς εισαγωγικά, που είναι το σωστό μέσα σε SQL.

EXECUTE σε PL/pgSQL

Μια function που μετρά τις γραμμές οποιουδήποτε πίνακα φτιάχνει το query με `format` και το εκτελεί με `EXECUTE ... INTO`, που αποθηκεύει την πρώτη γραμμή του αποτελέσματος σε μεταβλητή.

SQL

```
CREATE FUNCTION exact_count(p_table text)
RETURNS bigint
LANGUAGE plpgsql
AS $$
DECLARE
  n bigint;
BEGIN
  EXECUTE format('SELECT count(*) FROM %I', p_table) INTO n;
  RETURN n;
END;
$$;

SELECT relname AS table_name, exact_count(relname) AS rows
FROM pg_stat_user_tables
WHERE schemaname = 'public'
ORDER BY rows DESC
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 table_name  | rows
-------------+------
 events      | 1395
 order_items |  310
 orders      |  162
 payments    |  155
 reviews     |   39
(5 rows)
```

Το `pg_stat_user_tables` έχει μία γραμμή για κάθε πίνακα. Η function καλείται για καθέναν και εκτελεί ένα διαφορετικό query κάθε φορά.

Το `%I` όμως δέχεται μόνο ένα όνομα. Για `lab.orders` θα έβαζε εισαγωγικά γύρω από όλο το `"lab.orders"` και θα έψαχνε πίνακα με τελεία στο όνομά του. Ο τύπος `regclass` λύνει το πρόβλημα καλύτερα. Είναι το όνομα ενός πίνακα που η PostgreSQL έχει ήδη επιβεβαιώσει ότι υπάρχει και όταν μετατρέπεται σε κείμενο βγαίνει σωστά γραμμένο, με schema και εισαγωγικά όπου χρειάζονται.

SQL

```
CREATE FUNCTION exact_count(p_table regclass)
RETURNS bigint
LANGUAGE plpgsql
AS $$
DECLARE
  n bigint;
BEGIN
  EXECUTE format('SELECT count(*) FROM %s', p_table) INTO n;
  RETURN n;
END;
$$;

SELECT exact_count('orders'::regclass) AS orders,
       exact_count('lab.customers'::regclass) AS lab_customers;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 orders | lab_customers
--------+---------------
    162 |        200000
(1 row)
```

SQL

```
SELECT exact_count('no_such_table'::regclass);
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  relation "no_such_table" does not exist
LINE 1: SELECT exact_count('no_such_table'::regclass)
                           ^
```

Τώρα έχουμε δύο functions με το ίδιο όνομα και διαφορετικό τύπο ορίσματος. Το `'orders'::regclass` διαλέγει τη δεύτερη. Ένας πίνακας που δεν υπάρχει απορρίπτεται στη μετατροπή σε `regclass`, πριν φτάσει στο `EXECUTE`. Εδώ το `%s` είναι ασφαλές, γιατί η τιμή είναι ήδη έγκυρο όνομα.

Τιμές με USING

Στο dynamic SQL οι τιμές δεν χρειάζεται να μπουν στο κείμενο. Το `EXECUTE` δέχεται `USING` με τιμές για τα `$1`, `$2` του κειμένου. Είναι οι παράμετροι του προηγούμενου κεφαλαίου, μέσα σε dynamic query.

SQL

```
CREATE FUNCTION count_where(p_table regclass, p_column text, p_value text)
RETURNS bigint
LANGUAGE plpgsql
AS $$
DECLARE
  n bigint;
BEGIN
  EXECUTE format('SELECT count(*) FROM %s WHERE %I = $1', p_table, p_column)
    INTO n
    USING p_value;
  RETURN n;
END;
$$;

SELECT count_where('orders', 'status', 'delivered') AS delivered,
       count_where('products', 'sku', 'BK-PRG-001') AS one_sku;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 delivered | one_sku
-----------+---------
       125 |       1
(1 row)
```

Ο κανόνας είναι απλός. Ονόματα μπαίνουν με `%I` ή `regclass`. Τιμές περνούν με `USING`. Το `%L` μένει για τις σπάνιες περιπτώσεις όπου η τιμή πρέπει να γραφτεί μέσα στο κείμενο, όπως σε μια εντολή DDL που δεν δέχεται παραμέτρους.

Πώς γίνεται το SQL injection

Για να καταλάβεις γιατί επιμένουμε, δες την ίδια function γραμμένη με συνένωση strings. Ο προγραμματιστής σκέφτηκε ότι η τιμή είναι πάντα απλό κείμενο.

```sql
CREATE FUNCTION count_status_unsafe(p_status text)
RETURNS bigint
LANGUAGE plpgsql
AS $$
DECLARE
  n bigint;
BEGIN
  EXECUTE 'SELECT count(*) FROM orders WHERE status = ''' || p_status || '''' INTO n;
  RETURN n;
END;
$$;

SELECT count_status_unsafe('pending') AS normal,
       count_status_unsafe($$x' OR 'a' = 'a$$) AS injected;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 normal | injected
--------+----------
      3 |      162
(1 row)
```

Η δεύτερη κλήση επέστρεψε όλες τις παραγγελίες. Το κείμενο που εκτελέστηκε ήταν `WHERE status = 'x' OR 'a' = 'a'`. Η τιμή έκλεισε το string και πρόσθεσε δική της συνθήκη. Σε μια function που κάνει `DELETE` ή τρέχει με δικαιώματα ιδιοκτήτη, η ίδια τεχνική διαβάζει ή σβήνει δεδομένα άλλων.

Το ίδιο κείμενο με `USING` είναι απλώς μια περίεργη κατάσταση που δεν υπάρχει.

```sql
SELECT count_where('orders', 'status', $$x' OR 'a' = 'a$$) AS safe;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 safe
------
    0
(1 row)
```

ΠΑΓΙΔΑ

Μια function με `SECURITY DEFINER` τρέχει με τα δικαιώματα του ιδιοκτήτη της. Αν περιέχει dynamic SQL, κάθε λάθος στο quoting γίνεται κλιμάκωση δικαιωμάτων. Σε τέτοιες functions ορίζεις και `SET search_path = pg_catalog, public`, ώστε ένας χρήστης να μη μπορεί να βάλει δικό του πίνακα ή δική του function με το ίδιο όνομα σε schema που προηγείται.

Αποτελέσματα με πολλές γραμμές

Το `EXECUTE ... INTO` κρατά μία γραμμή. Για function που επιστρέφει πίνακα χρησιμοποιείς `RETURN QUERY EXECUTE`. Εδώ μια μικρή αναφορά που δέχεται το όνομα της στήλης για την ομαδοποίηση.

```sql
CREATE FUNCTION orders_by(p_column text)
RETURNS TABLE (grp text, orders bigint)
LANGUAGE plpgsql
AS $$
BEGIN
  IF p_column NOT IN ('status', 'coupon_code', 'shipping_city_id') THEN
    RAISE EXCEPTION 'δεν επιτρέπεται ομαδοποίηση κατά %', p_column;
  END IF;
  RETURN QUERY EXECUTE format(
    'SELECT %1$I::text, count(*) FROM orders GROUP BY %1$I ORDER BY count(*) DESC',
    p_column);
END;
$$;

SELECT * FROM orders_by('status');
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

    grp    | orders
-----------+--------
 delivered |    125
 cancelled |     14
 shipped   |     12
 paid      |      5
 pending   |      3
 refunded  |      3
(6 rows)
```

Το `%1$I` σημαίνει «το πρώτο όρισμα ως identifier». Με τον αριθμό μπορείς να χρησιμοποιήσεις το ίδιο όρισμα πολλές φορές. Ο έλεγχος στην αρχή περιορίζει τις στήλες σε μια λίστα που ξέρεις. Το `%I` εμποδίζει το injection, αλλά μόνο η λίστα εμποδίζει κάποιον να ομαδοποιήσει κατά `email` και να δει κάθε email της βάσης.

```sql
SELECT * FROM orders_by('email');
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  δεν επιτρέπεται ομαδοποίηση κατά email
CONTEXT:  PL/pgSQL function orders_by(text) line 4 at RAISE
```

Εντολές μίας χρήσης με DO

Συχνά το dynamic SQL χρειάζεται μία φορά, σε ένα migration ή σε μια εργασία συντήρησης. Το `DO` εκτελεί ένα ανώνυμο block PL/pgSQL χωρίς να φτιάξεις function. Εδώ φτιάχνει έναν πίνακα αρχείου για κάθε έτος παραγγελιών.

SQL

```
DO $$
DECLARE
  y integer;
BEGIN
  FOR y IN
    SELECT DISTINCT extract(year FROM created_at)::integer FROM orders ORDER BY 1
  LOOP
    EXECUTE format(
      'CREATE TABLE %I AS SELECT * FROM orders WHERE created_at >= %L AND created_at < %L',
      'orders_' || y, make_date(y, 1, 1), make_date(y + 1, 1, 1));
    RAISE NOTICE 'orders_%', y;
  END LOOP;
END;
$$;

SELECT 'orders_2025' AS table_name, count(*) FROM orders_2025
UNION ALL
SELECT 'orders_2026', count(*) FROM orders_2026;
```

ΑΠΟΤΕΛΕΣΜΑ

```
NOTICE:  orders_2025
NOTICE:  orders_2026
DO

 table_name  | count
-------------+-------
 orders_2025 |    74
 orders_2026 |    88
(2 rows)
```

Εδώ το `%L` ήταν απαραίτητο, αφού το `CREATE TABLE AS` δεν δέχεται `USING`. Το `RAISE NOTICE` τυπώνει ένα μήνυμα για κάθε πίνακα, που βλέπεις στο psql.

Το του psql

Για εργασίες συντήρησης υπάρχει κάτι ακόμη πιο απλό, που δεν χρειάζεται PL/pgSQL. Γράφεις ένα query που επιστρέφει εντολές SQL ως κείμενο και το τελειώνεις με `\gexec` αντί για ερωτηματικό. Το psql εκτελεί κάθε τιμή του αποτελέσματος ως χωριστή εντολή.

PSQL

```
SELECT format('DROP TABLE %I', relname)
FROM pg_stat_user_tables
WHERE relname LIKE 'orders\_20%'
ORDER BY relname
\gexec
```

ΑΠΟΤΕΛΕΣΜΑ

```
DROP TABLE
DROP TABLE
```

Πριν το τρέξεις με `\gexec`, τρέχεις το ίδιο query με ερωτηματικό και διαβάζεις τις εντολές που θα εκτελεστούν. Είναι ο πιο ασφαλής τρόπος να κάνεις την ίδια αλλαγή σε πολλά αντικείμενα.

ΚΡΑΤΑ ΑΥΤΟ

Τα ονόματα πινάκων και στηλών δεν γίνονται παράμετροι, οπότε το query φτιάχνεται ως κείμενο και εκτελείται με `EXECUTE` μέσα σε PL/pgSQL. Ονόματα μπαίνουν με `format('%I')` ή ως `regclass`, τιμές περνούν με `USING` και το `%L` μένει για DDL. Η συνένωση strings με τιμές χρήστη είναι SQL injection. Όπου η επιλογή στήλης έρχεται από τον χρήστη, περιόρισέ τη σε λίστα. Το `DO` τρέχει ανώνυμο block και το `\gexec` εκτελεί εντολές που παράγει ένα query.

Ασκήσεις

ΑΣΚΗΣΗ 3.1 Quoting για κάθε περίπτωση

Με ένα `format` φτιάξε την εντολή που αλλάζει σε `Nikos` το όνομα των γραμμών του πίνακα `Πελάτες 2026` με email `nikos'o@example.gr`, στη στήλη `first name`. Μην την εκτελέσεις, επέστρεψε μόνο το κείμενο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                                          sql
-------------------------------------------------------------------------------------------
 UPDATE "Πελάτες 2026" SET "first name" = 'Nikos' WHERE email = 'nikos''o@example.gr'
(1 row)
```

ΑΣΚΗΣΗ 3.2 Μέγιστη τιμή οποιασδήποτε στήλης

Φτιάξε function `max_of(p_table regclass, p_column text)` που επιστρέφει ως `text` τη μέγιστη τιμή της στήλης. Κάλεσέ τη για το `price` του `products` και για το `created_at` του `lab.orders`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 max_price |          last_order
-----------+-------------------------------
 2199.00   | 2026-07-01 02:44:55.05438+03
(1 row)
```

ΑΣΚΗΣΗ 3.3 Πελάτες ανά στήλη με ασφάλεια

Φτιάξε function `customers_matching(p_column text, p_value text)` που επιστρέφει `id` και `email` των πελατών όπου η στήλη ισούται με την τιμή. Επίτρεψε μόνο τις στήλες `last_name` και `phone`. Κάλεσέ τη για `last_name = 'Παπαδόπουλος'`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 id |       email
----+-------------------
  2 | gpapad@example.gr
(1 row)
```

ΑΣΚΗΣΗ 3.4 Πίνακες ανά τρίμηνο

Με ένα `DO` block φτιάξε τέσσερις άδειους πίνακες `payments_2025_q1` έως `payments_2025_q4` με τις ίδιες στήλες με τον `payments`. Μετά μέτρησε πόσοι πίνακες υπάρχουν με όνομα που αρχίζει από `payments_2025`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
DO

 tables
--------
      4
(1 row)
```

ΑΣΚΗΣΗ 3.5 Ένωση πολλών πινάκων

Φτιάξε με `string_agg` και `format` ένα query κειμένου που ενώνει με `UNION ALL` ένα `SELECT` ανά πίνακα του schema `lab`, όπου κάθε `SELECT` επιστρέφει το όνομα του πίνακα και το `count(*)` του. Επέστρεψε μόνο το κείμενο, ταξινομημένο κατά όνομα πίνακα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                              sql
-----------------------------------------------------------------
 SELECT 'customers' AS table_name, count(*) FROM lab.customers+
 UNION ALL                                                    +
 SELECT 'events' AS table_name, count(*) FROM lab.events      +
 UNION ALL                                                    +
 SELECT 'orders' AS table_name, count(*) FROM lab.orders      +
 UNION ALL                                                    +
 SELECT 'products' AS table_name, count(*) FROM lab.products
(1 row)
```

ΑΣΚΗΣΗ 3.6 Injection στην πράξη

Χρησιμοποίησε την `count_status_unsafe` του κεφαλαίου για να μετρήσεις τις παραγγελίες που δεν έχουν κατάσταση `delivered`, περνώντας μόνο ένα κατάλληλο string ως όρισμα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 not_delivered
---------------
            37
(1 row)
```

ΚΕΦΑΛΑΙΟ 4

LISTEN και NOTIFY

Μια εφαρμογή που περιμένει νέες παραγγελίες μπορεί να ρωτά τη βάση κάθε δευτερόλεπτο. Οι περισσότερες ερωτήσεις θα απαντηθούν «τίποτα νέο». Η PostgreSQL έχει έναν απλό μηχανισμό ειδοποιήσεων. Μια σύνδεση ακούει σε ένα κανάλι, μια άλλη στέλνει μήνυμα και η πρώτη το λαμβάνει αμέσως, χωρίς επιπλέον σύστημα μηνυμάτων.

Κανάλια και μηνύματα

Ένα **κανάλι** είναι απλώς ένα όνομα. Δεν το δημιουργείς, υπάρχει μόλις κάποιος ακούσει σε αυτό. Η συνεδρία Β ακούει στο κανάλι `new_order` με `LISTEN`.

SQL · ΣΥΝΕΔΡΙΑ Β

```
LISTEN new_order;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Β

```
LISTEN
```

Η συνεδρία Α στέλνει μήνυμα με `NOTIFY`. Το μήνυμα έχει κανάλι και προαιρετικά ένα κείμενο, το **payload**.

SQL · ΣΥΝΕΔΡΙΑ Α

```
NOTIFY new_order, 'ping 1';
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Α

```
NOTIFY
```

Ο server παραδίδει την ειδοποίηση σε κάθε σύνδεση που ακούει στο κανάλι. Το psql όμως τη δείχνει μόνο όταν εκτελέσει την επόμενη εντολή, γιατί μόνο τότε διαβάζει ό,τι έχει έρθει από τον server. Ας πούμε ότι η Β είναι ένας worker που, μόλις ειδοποιηθεί, ψάχνει τις παραγγελίες σε αναμονή.

SQL · ΣΥΝΕΔΡΙΑ Β

```
SELECT count(*) AS pending FROM orders WHERE status = 'pending';
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ B

```
 pending
---------
       3
(1 row)
Asynchronous notification "new_order" with payload "ping 1" received from server process with PID 75732.
```

Κάτω από το αποτέλεσμα εμφανίζεται η ειδοποίηση με το κανάλι, το payload και το process ID της σύνδεσης που την έστειλε. Μια εφαρμογή δεν χρειάζεται να εκτελέσει εντολή. Ο driver της ακούει στο socket και καλεί τον κώδικά σου μόλις έρθει ειδοποίηση.

Ειδοποιήσεις και transactions

Ένα `NOTIFY` μέσα σε transaction στέλνεται μόνο όταν γίνει commit. Αν το transaction αναιρεθεί, η ειδοποίηση δεν στέλνεται ποτέ. Αυτό είναι το πιο χρήσιμο χαρακτηριστικό του μηχανισμού. Ο worker δεν θα ψάξει ποτέ μια παραγγελία που δεν αποθηκεύτηκε.

SQL · ΣΥΝΕΔΡΙΑ A

```
BEGIN;
NOTIFY new_order, 'ping 2';
ROLLBACK;

BEGIN;
NOTIFY new_order, 'ping 3';
NOTIFY new_order, 'ping 3';
NOTIFY new_order, 'ping 4';
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ A

```
BEGIN

NOTIFY

ROLLBACK

BEGIN

NOTIFY

NOTIFY

NOTIFY

COMMIT
```

SQL · ΣΥΝΕΔΡΙΑ B

```
SELECT 'worker ξύπνησε' AS status;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ B

```
     status
----------------
 worker ξύπνησε
(1 row)
Asynchronous notification "new_order" with payload "ping 3" received from server process with PID 75732.
Asynchronous notification "new_order" with payload "ping 4" received from server process with PID 75732.
```

Η ειδοποίηση `ping 2` δεν ήρθε ποτέ. Από το δεύτερο transaction ήρθαν δύο και όχι τρεις. Ίδιες ειδοποιήσεις, με ίδιο κανάλι και ίδιο payload, μέσα στο ίδιο transaction ενώνονται σε μία. Ο worker δεν ενδιαφέρεται πόσες φορές του είπαν να ξυπνήσει.

pg_notify

Η εντολή `NOTIFY` θέλει το κανάλι και το payload γραμμένα μέσα στο κείμενο. Η function `pg_notify` δέχεται και τα δύο ως τιμές, οπότε χρησιμοποιείται σε queries και σε triggers. Ο πιο συνηθισμένος συνδυασμός είναι ένας trigger που ειδοποιεί για κάθε νέα γραμμή.

SQL · ΣΥΝΕΔΡΙΑ A

```
CREATE FUNCTION notify_new_order()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
  PERFORM pg_notify('new_order', NEW.id::text);
  RETURN NULL;
END;
$$;

CREATE TRIGGER orders_notify
AFTER INSERT ON orders
FOR EACH ROW EXECUTE FUNCTION notify_new_order();

INSERT INTO orders (customer_id, status, created_at)
VALUES (4, 'pending', '2026-07-01 09:00'), (9, 'pending', '2026-07-01 09:05');
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ A

```
CREATE FUNCTION

CREATE TRIGGER

INSERT 0 2
```

SQL · ΣΥΝΕΔΡΙΑ B

```
SELECT id, customer_id FROM orders WHERE status = 'pending' ORDER BY id DESC LIMIT 2;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Β

```
 id  | customer_id
-----+-------------
 164 |           9
 163 |           4
(2 rows)
Asynchronous notification "new_order" with payload "163" received from server process with PID 75732.
Asynchronous notification "new_order" with payload "164" received from server process with PID 75732.
```

Ο worker πήρε ένα μήνυμα για κάθε νέα παραγγελία, με το id της. Το payload είναι το id και όχι ολόκληρη η γραμμή και αυτό δεν είναι τυχαίο.

ΠΑΓΙΔΑ

Το payload έχει όριο 8.000 bytes και οι ειδοποιήσεις δεν αποθηκεύονται πουθενά. Αν ο worker έχει αποσυνδεθεί τη στιγμή του commit, δεν θα μάθει ποτέ για το μήνυμα. Γι' αυτό το payload είναι ένα id ή τίποτα και η αλήθεια βρίσκεται σε πίνακα. Η ειδοποίηση απλώς λέει «κοίτα τον πίνακα τώρα».

Ουρά με ειδοποιήσεις

Στο κεφάλαιο 38 του πρώτου τόμου φτιάξαμε ουρά εργασιών με `FOR UPDATE SKIP LOCKED`. Οι workers ρωτούσαν τον πίνακα ξανά και ξανά. Με `LISTEN` ο worker ρωτά μόνο όταν υπάρχει λόγος και επειδή οι ειδοποιήσεις μπορεί να χαθούν, ρωτά και μία φορά στην εκκίνηση και ανά τακτά διαστήματα. Το σχέδιο έχει τρία μέρη.

1. Η εφαρμογή γράφει την εργασία σε πίνακα και κάνει `NOTIFY` στο ίδιο transaction.
2. Ο worker κάνει `LISTEN` και μόλις ξεκινήσει αδειάζει ό,τι βρίσκεται ήδη στον πίνακα.
3. Σε κάθε ειδοποίηση, ή κάθε λίγα λεπτά αν δεν έρθει καμία, παίρνει εργασίες με `SKIP LOCKED` μέχρι να αδειάσει ο πίνακας.

Αν χαθεί μια ειδοποίηση, η εργασία περιμένει μέχρι τον επόμενο γύρο και δεν χάνεται. Αν έρθουν δέκα ειδοποιήσεις μαζί, ο worker αδειάζει την ουρά με την πρώτη και οι υπόλοιπες βρίσκουν άδειο πίνακα.

Στον κώδικα ενός worker σε Python με psycopg2 το σχέδιο γίνεται ένας βρόχος που περιμένει στο socket της σύνδεσης.

PYTHON

```
import select
import psycopg2

conn = psycopg2.connect("dbname=sqlbook")
conn.autocommit = True
conn.cursor().execute("LISTEN new_order")
drain_queue(conn)

while True:
    ready, _, _ = select.select([conn], [], [], 300)
    conn.poll()
    conn.notifies.clear()
    drain_queue(conn)
```

Το `select` περιμένει μέχρι να έρθει κάτι στη σύνδεση ή να περάσουν 300 δευτερόλεπτα. Σε κάθε περίπτωση ο worker αδειάζει την ουρά. Η function `drain_queue` είναι το `SKIP LOCKED` του πρώτου τόμου σε βρόχο.

Όρια του μηχανισμού

Οι ειδοποιήσεις που δεν έχουν παραδοθεί περιμένουν σε μια κοινή ουρά του server, μεγέθους 8 GB. Μια σύνδεση που ακούει αλλά κρατά ανοιχτό ένα transaction για ώρα δεν μπορεί να λάβει τίποτα και τα μηνύματα συσσωρεύονται. Η function `pg_notification_queue_usage` επιστρέφει πόσο γεμάτη είναι η ουρά.

SQL

```
SELECT pg_notification_queue_usage() AS queue_used;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 queue_used
------------
          0
(1 row)
```

Αν η ουρά γεμίσει, κάθε `NOTIFY` αποτυγχάνει και μαζί του το transaction που το έστειλε. Είναι σπάνιο, αλλά αξίζει να παρακολουθείς αυτή την τιμή.

Υπάρχουν και δύο περιορισμοί που θα συναντήσεις αργότερα. Το `LISTEN` δεν δουλεύει μέσω PgBouncer σε transaction mode, γιατί η σύνδεση του server αλλάζει μετά από κάθε transaction και μαζί χάνεται η εγγραφή στο κανάλι. Το κεφάλαιο 21 εξηγεί γιατί. Επίσης οι ειδοποιήσεις δεν περνούν σε standby servers, οπότε ένας worker ακούει πάντα στον primary.

Όταν ο worker δεν θέλει πια μηνύματα, σταματά με `UNLISTEN`. Το `UNLISTEN *` σταματά όλα τα κανάλια της σύνδεσης. Το `pg_listening_channels` δείχνει σε ποια κανάλια ακούει μια σύνδεση.

SQL · ΣΥΝΕΔΡΙΑ Β

```
SELECT pg_listening_channels();
UNLISTEN *;
SELECT count(*) AS channels FROM pg_listening_channels();
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Β

```
 pg_listening_channels
-----------------------
 new_order
(1 row)

UNLISTEN

 channels
----------
        0
(1 row)
```

ΚΡΑΤΑ ΑΥΤΟ

Το `LISTEN` εγγράφει μια σύνδεση σε κανάλι και το `NOTIFY` ή το `pg_notify` στέλνουν μήνυμα σε όποιον ακούει. Τα μηνύματα φεύγουν μόνο με commit και οι ίδιες ειδοποιήσεις του ίδιου transaction ενώνονται. Δεν αποθηκεύονται, οπότε η αλήθεια μένει σε πίνακα και η ειδοποίηση λέει μόνο πότε να κοιτάξεις. Το payload κρατά ένα id. Το `LISTEN` χρειάζεται σταθερή σύνδεση στον primary.

Ασκήσεις

ΑΣΚΗΣΗ 4.1 Ειδοποίηση χωρίς payload

Στη συνεδρία Α άκου στο κανάλι `cache_reset` και στείλε σε αυτό μια ειδοποίηση χωρίς payload.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
LISTEN

NOTIFY
Asynchronous notification "cache_reset" with payload "" received from server process with PID 75732.

UNLISTEN
```

ΑΣΚΗΣΗ 4.2 Δυναμικό κανάλι

Άκου στο κανάλι `customer_7` και στείλε σε αυτό με `pg_notify` το email του πελάτη 7, φτιάχνοντας το όνομα του καναλιού μέσα στο query από το id.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
LISTEN

 pg_notify
-----------

(1 row)
Asynchronous notification "customer_7" with payload "dimitra.k@example.gr" received from
server process with PID 75732.

UNLISTEN
```

ΑΣΚΗΣΗ 4.3 Ειδοποίηση για αλλαγή κατάστασης

Φτιάξε trigger στον `orders` που στέλνει στο κανάλι `order_status` payload της μορφής `id:νέα κατάσταση`, μόνο όταν αλλάζει η στήλη `status`. Άκου στο κανάλι και άλλαξε την κατάσταση της παραγγελίας 162 σε `paid`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

CREATE TRIGGER

LISTEN

UPDATE 1
Asynchronous notification "order_status" with payload "162:paid" received from server process with PID 75732.

UNLISTEN
```

ΑΣΚΗΣΗ 4.4 Rollback χωρίς ειδοποίηση

Άκου στο κανάλι `order_status`, άλλαξε μέσα σε transaction την κατάσταση της παραγγελίας 161 σε `cancelled` και κάνε `ROLLBACK`. Δείξε μετά την κατάσταση της παραγγελίας.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
LISTEN

BEGIN

UPDATE 1

ROLLBACK

 status
--------
 paid
(1 row)

UNLISTEN
```

ΑΣΚΗΣΗ 4.5 Σε ποια κανάλια ακούω

Άκου στα κανάλια `a_jobs` και `b_jobs`, δείξε τα κανάλια της σύνδεσης ταξινομημένα και σταμάτα να ακούς σε όλα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
LISTEN

LISTEN

 channel
---------
 a_jobs
 b_jobs
(2 rows)

UNLISTEN
```

ΚΕΦΑΛΑΙΟ 5

TABLESAMPLE και δείγματα

Για πολλές ερωτήσεις δεν χρειάζεσαι όλες τις γραμμές. Ο μέσος όρος δύο εκατομμυρίων παραγγελιών βγαίνει σχεδόν ίδιος από είκοσι χιλιάδες. Το `TABLESAMPLE` διαβάζει ένα τυχαίο κομμάτι του πίνακα. Υπάρχουν δύο τρόποι να διαλέξεις αυτό το κομμάτι, με πολύ διαφορετικό κόστος και πολύ διαφορετική ακρίβεια.

Ένα τοις εκατό των γραμμών

Το `TABLESAMPLE` μπαίνει αμέσως μετά το όνομα του πίνακα στο `FROM`. Δέχεται μια μέθοδο και ένα ποσοστό.

SQL

```
SELECT count(*) AS sampled, round(avg(total), 2) AS avg_total
FROM lab.orders TABLESAMPLE BERNOULLI (1);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 sampled | avg_total
---------+-----------
   20222 |    378.47
(1 row)
```

Περίπου 20.000 γραμμές, το ένα τοις εκατό του πίνακα. Αν ξανατρέξεις το query θα πάρεις λίγο διαφορετικούς αριθμούς, γιατί κάθε εκτέλεση διαλέγει άλλο δείγμα. Με `REPEATABLE` και έναν αριθμό, το seed, το δείγμα γίνεται ίδιο κάθε φορά, όσο δεν αλλάζει ο πίνακας. Όλα τα υπόλοιπα queries του κεφαλαίου το χρησιμοποιούν, ώστε να βλέπεις τα ίδια αποτελέσματα με το βιβλίο.

SQL

```
SELECT count(*) * 100 AS estimated_rows, round(avg(total), 2) AS avg_total
FROM lab.orders TABLESAMPLE BERNOULLI (1) REPEATABLE (42);

SELECT count(*) AS exact_rows, round(avg(total), 2) AS avg_total
FROM lab.orders;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 estimated_rows | avg_total
----------------+-----------
        1986600 |    383.39
(1 row)

 exact_rows | avg_total
------------+-----------
    2000000 |    380.08
(1 row)
```

Πολλαπλασιάζοντας το πλήθος του δείγματος επί 100 παίρνεις εκτίμηση για όλο τον πίνακα. Η εκτίμηση και ο μέσος όρος απέχουν λίγο από τις ακριβείς τιμές. Για μια αναφορά ή μια απόφαση αυτή η ακρίβεια φτάνει.

BERNOULLI και SYSTEM

Το `BERNOULLI` εξετάζει κάθε γραμμή και την κρατά με πιθανότητα ένα τοις εκατό. Για να το κάνει αυτό πρέπει να διαβάσει ολόκληρο τον πίνακα. Το `SYSTEM` αποφασίζει ανά σελίδα. Διαλέγει περίπου το ένα τοις εκατό των σελίδων και κρατά όλες τις γραμμές τους. Το `EXPLAIN` δείχνει τη διαφορά στο κόστος.

SQL

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF)
SELECT count(*) FROM lab.orders TABLESAMPLE SYSTEM (1) REPEATABLE (42);

EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF)
SELECT count(*) FROM lab.orders TABLESAMPLE BERNOULLI (1) REPEATABLE (42);
```

ΑΠΟΤΕΛΕΣΜΑ

```
Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=140 read=12 written=9
  ->  Sample Scan on orders (actual rows=17963.00 loops=1)
        Sampling: system ('1'::real) REPEATABLE ('42'::double precision)
        Buffers: shared hit=140 read=12 written=9
Planning Time: 0.030 ms
Execution Time: 1.087 ms

Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=15462 read=1462 written=43
  ->  Sample Scan on orders (actual rows=19866.00 loops=1)
        Sampling: bernoulli ('1'::real) REPEATABLE ('42'::double precision)
        Buffers: shared hit=15462 read=1462 written=43
Planning Time: 0.016 ms
Execution Time: 16.966 ms
```

Ο κόμβος `Sample Scan` είναι ίδιος. Τα buffers όμως διαφέρουν πάρα πολύ. Το `SYSTEM` άγγιξε περίπου 150 σελίδες και το `BERNOULLI` όλες τις σχεδόν 17.000 του πίνακα. Σε έναν πίνακα εκατοντάδων GB αυτό είναι η διαφορά ανάμεσα σε δευτερόλεπτα και σε ώρες.

Το `SYSTEM` όμως έχει ένα κρυφό κόστος. Οι γραμμές μιας σελίδας δεν είναι ανεξάρτητες μεταξύ τους. Στο `lab.orders`, όπως σε κάθε πίνακα όπου οι γραμμές μπαίνουν με τη σειρά που δημιουργούνται, μια σελίδα κρατά παραγγελίες της ίδιας ώρας. Οι παραγγελίες σε εκκρεμότητα είναι οι 1.349 πιο

πρόσφατες, άρα μαζεμένες σε λίγες δεκάδες σελίδες στο τέλος του πίνακα. Δες τι εκτιμά κάθε μέθοδος για αυτές με οκτώ διαφορετικά seeds.

SQL

```
SELECT seed,
       (SELECT count(*) * 100 FROM lab.orders TABLESAMPLE SYSTEM (1) REPEATABLE (seed)
        WHERE status = 'pending') AS system_estimate,
       (SELECT count(*) * 100 FROM lab.orders TABLESAMPLE BERNOULLI (1) REPEATABLE (seed)
        WHERE status = 'pending') AS bernoulli_estimate
FROM generate_series(1, 8) AS seed;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 seed | system_estimate | bernoulli_estimate
------+-----------------+--------------------
    1 |            3100 |               1700
    2 |               0 |               1400
    3 |               0 |               1100
    4 |            4600 |               1700
    5 |            4600 |               1200
    6 |            4400 |               1000
    7 |               0 |               1600
    8 |            4400 |                900
(8 rows)
```

Η σωστή τιμή είναι 1.349. Το `BERNOULLI` πέφτει πάντα κάπου κοντά. Το `SYSTEM` είτε δεν βρίσκει καμία εκκρεμή παραγγελία είτε βρίσκει τριπλάσιες, ανάλογα με το αν έπεσε σε μία από τις λίγες σελίδες του τέλους. Για τιμές απλωμένες σε όλο τον πίνακα, όπως ο μέσος όρος του `total`, το `SYSTEM` αρκεί. Για τιμές που είναι μαζεμένες σε μια περιοχή, χρειάζεσαι `BERNOULLI`.

Το `WHERE` εφαρμόζεται μετά τη δειγματοληψία. Το `TABLESAMPLE` διαλέγει πρώτα το ένα τοις εκατό του πίνακα και μετά κρατά όσες από αυτές τις γραμμές περνούν τη συνθήκη. Δεν παίρνεις το ένα τοις εκατό των εκκρεμών παραγγελιών.

Σταθερό πλήθος γραμμών

Συχνά δεν θέλεις ποσοστό αλλά ένα συγκεκριμένο πλήθος γραμμών, για παράδειγμα δέκα τυχαία προϊόντα για μια σελίδα προτάσεων. Ο συνηθισμένος τρόπος είναι `ORDER BY random() LIMIT 10`. Είναι σωστός αλλά διαβάζει όλο τον πίνακα και ταξινομεί κάθε γραμμή με τυχαίο κλειδί.

SQL

```
EXPLAIN (COSTS OFF)
SELECT id, name FROM lab.products ORDER BY random() LIMIT 10;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Limit
  ->  Sort
        Sort Key: (random())
        ->  Seq Scan on products
```

Η επέκταση `tsm_system_rows`, που έρχεται με την PostgreSQL, προσθέτει τη μέθοδο `SYSTEM_ROWS`. Διαβάζει σελίδες μέχρι να μαζέψει όσες γραμμές ζήτησες.

```sql
CREATE EXTENSION tsm_system_rows;

SELECT id, name FROM lab.products TABLESAMPLE SYSTEM_ROWS (5);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE EXTENSION

  id   |            name
-------+----------------------------
 32109 | Βραστήρας Pyxis E24-709
 32110 | Καλώδιο USB-C Atlas 82D-710
 32111 | Τοστιέρα Pyxis C70-711
 32112 | Βραστήρας Atlas 508-712
 32113 | Μικρόφωνο Orion 7B5-713
(5 rows)
```

Η `SYSTEM_ROWS` δεν δέχεται `REPEATABLE`, οπότε στη δική σου εκτέλεση θα δεις άλλα προϊόντα. Θα δεις όμως το ίδιο μοτίβο. Τα πέντε προϊόντα έχουν συνεχόμενα ids, γιατί βγήκαν όλα από την ίδια σελίδα, με το ίδιο μειονέκτημα που είδαμε στο `SYSTEM`. Για μια σελίδα προτάσεων συνήθως δεν πειράζει, για στατιστική δουλειά πειράζει. Υπάρχει και η αδελφή επέκταση `tsm_system_time` με τη μέθοδο `SYSTEM_TIME`, που διαβάζει σελίδες για όσα χιλιοστά του δευτερολέπτου ορίσεις.

Δείγμα ως νέος πίνακας

Όταν θέλεις να δοκιμάσεις ένα query ή ένα migration σε ρεαλιστικά δεδομένα, ένα αντίγραφο του ενός τοις εκατό είναι συχνά αρκετό και φτιάχνεται σε λίγα δευτερόλεπτα.

```sql
CREATE TABLE lab.orders_sample AS
SELECT * FROM lab.orders TABLESAMPLE BERNOULLI (1) REPEATABLE (2026);

SELECT count(*) AS rows,
       pg_size_pretty(pg_total_relation_size('lab.orders_sample')) AS size
FROM lab.orders_sample;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 19978

 rows  |  size
-------+---------
 19978 | 1744 kB
(1 row)
```

Ένα τυχαίο δείγμα παραγγελιών όμως δεν έχει τα items τους ούτε όλους τους πελάτες τους. Για ένα συνεκτικό υποσύνολο παίρνεις δείγμα από τον «γονικό» πίνακα και τα υπόλοιπα με joins. Για παράδειγμα, δείγμα πελατών και μετά όλες οι παραγγελίες τους.

ΣΗΜΕΙΩΣΗ

Για ένα γρήγορο πλήθος γραμμών δεν χρειάζεσαι καν δείγμα. Το `reltuples` του `pg_class` κρατά την εκτίμηση του τελευταίου `VACUUM` ή `ANALYZE`. Είναι αυτό που χρησιμοποιεί ο planner και διαβάζεται αμέσως, όσο μεγάλος κι αν είναι ο πίνακας.

SQL

```
SELECT reltuples::bigint AS estimated_rows
FROM pg_class
WHERE oid = 'lab.orders'::regclass;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 estimated_rows
----------------
        2000000
(1 row)
```

ΚΡΑΤΑ ΑΥΤΟ

Το `TABLESAMPLE` διαβάζει ένα τυχαίο ποσοστό ενός πίνακα και το `REPEATABLE` το κάνει επαναλήψιμο. Το `BERNOULLI` διαλέγει γραμμές, είναι ακριβές αλλά διαβάζει όλο τον πίνακα. Το `SYSTEM` διαλέγει σελίδες, είναι φθηνό αλλά μεροληπτεί όταν οι τιμές είναι μαζεμένες σε μια περιοχή του πίνακα. Το `WHERE` εφαρμόζεται μετά το δείγμα. Για σταθερό πλήθος υπάρχει το `SYSTEM_ROWS` και για γρήγορο πλήθος γραμμών το `reltuples`.

Ασκήσεις

ΑΣΚΗΣΗ 5.1 Εκτίμηση για τα checkout

Από δείγμα `BERNOULLI` του ενός τοις εκατό του `lab.events` με seed 7, εκτίμησε πόσα events έχουν `kind = 'checkout'`. Βάλε στην ίδια γραμμή και το ακριβές πλήθος.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 estimated | exact
-----------+--------
    139500 | 142762
(1 row)
```

ΑΣΚΗΣΗ 5.2 Μέση αξία ανά κατάσταση

Από δείγμα `SYSTEM` του δύο τοις εκατό του `lab.orders` με seed 3, υπολόγισε τη μέση αξία παραγγελίας για τις καταστάσεις `delivered`, `cancelled` και `refunded`, με δύο δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  status   | avg_total | sampled
-----------+-----------+---------
 cancelled |    382.14 |    2709
 delivered |    379.69 |   35695
 refunded  |    378.79 |    1170
(3 rows)
```

ΑΣΚΗΣΗ 5.3 Πόσες σελίδες διαβάζει

Με `EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF)` δείξε πόσες σελίδες διαβάζει ένα δείγμα `SYSTEM` του μισού τοις εκατό από το `lab.events` με seed 1.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=72
  ->  Sample Scan on events (actual rows=5268.00 loops=1)
        Sampling: system ('0.5'::real) REPEATABLE ('1'::double precision)
        Buffers: shared hit=72
Planning Time: 0.026 ms
Execution Time: 0.335 ms
```

ΑΣΚΗΣΗ 5.4 Δείγμα πελατών με τις παραγγελίες τους

Φτιάξε πίνακα `lab.customers_sample` με δείγμα `BERNOULLI` του μισού τοις εκατό των πελατών του `lab.customers` με seed 11. Μέτρησε μετά πόσους πελάτες έχει και πόσες παραγγελίες του `lab.orders` ανήκουν σε αυτούς.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 1019

 customers | orders
-----------+--------
      1019 |  10533
(1 row)
```

ΑΣΚΗΣΗ 5.5 Τρία τυχαία προϊόντα με φθηνό τρόπο

Δείξε με `EXPLAIN (COSTS OFF)` το plan ενός query που επιστρέφει τρία προϊόντα του `lab.products` με `SYSTEM_ROWS`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Sample Scan on products
  Sampling: system_rows ('3'::bigint)
```

ΑΣΚΗΣΗ 5.6 Εκτίμηση από τον planner

Δείξε το `reltuples` του `lab.events` και του `lab.customers` από το `pg_class`, ως ακέραιους, με το όνομα του πίνακα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  table_name   | estimated_rows
---------------+----------------
 lab.customers |         200000
 lab.events    |        1000000
(2 rows)
```

ΜΕΡΟΣ ΙΙ

Ο server

Ως τώρα ο server ήταν ένα κουτί που απαντούσε σε queries. Σε αυτό το μέρος τον στήνουμε από το μηδέν, τον ρυθμίζουμε και τον παρακολουθούμε. Διεργασίες, μνήμη, WAL, checkpoints, autovacuum και logs. Όλα όσα χρειάζεται να ξέρεις πριν κρατήσεις backups ή στήσεις replication.

ΚΕΦΑΛΑΙΟ 6

Το εργαστήριο και το data directory

Από εδώ και πέρα δεν αρκεί μία βάση. Θα στήνουμε δικούς μας servers, θα τους σταματάμε απότομα, θα τους αντιγράφουμε και θα τους συνδέουμε μεταξύ τους. Σε αυτό το κεφάλαιο φτιάχνουμε το εργαστήριο, στήνουμε τον πρώτο server με το χέρι και βλέπουμε από τι αποτελείται.

Το εργαστήριο

Όλοι οι servers του βιβλίου ζουν σε έναν φάκελο, το `~/pglab`. Ακούν σε θύρες από 5501 και πάνω. Δεν αγγίζουν την PostgreSQL που ίσως τρέχει ήδη στο μηχάνημά σου στη θύρα 5432. Χρειάζεσαι την PostgreSQL 18 με τα εργαλεία της στο `PATH`.

TERMINAL

```
postgres --version
```

ΕΞΟΔΟΣ

```
postgres (PostgreSQL) 18.6 (Homebrew)
```

Σε κάθε terminal που ανοίγεις για το εργαστήριο ορίζεις μερικές μεταβλητές περιβάλλοντος. Το `LAB` είναι ο φάκελος. Τα `PGHOST`, `PGPORT`, `PGUSER` και `PGDATABASE` είναι οι προεπιλογές που διαβάζουν όλα τα εργαλεία της PostgreSQL, από το `psql` μέχρι το `pg_dump`, όταν δεν τους δίνεις άλλες τιμές. Ο φάκελος `data` του βιβλίου μπαίνει στο `PATH` για το script `pglab`, που θα δούμε στο τέλος του κεφαλαίου.

TERMINAL

```
export LAB=~/pglab
export PGHOST=localhost PGPORT=5501 PGUSER=postgres PGDATABASE=sqlbook
export PATH="$HOME/pg-production/data:$PATH"
mkdir -p "$LAB" && cd "$LAB"
```

Όλες οι εντολές του βιβλίου τρέχουν μέσα στο `$LAB`. Στα αποτελέσματα ο φάκελος εμφανίζεται ως `~/pglab`.

initdb

Ένας PostgreSQL server διαχειρίζεται ένα **cluster**, δηλαδή έναν φάκελο με όλες τις βάσεις, τους ρόλους και τις ρυθμίσεις του. Η λέξη δεν σημαίνει εδώ πολλά μηχανήματα. Ένα cluster είναι ένας φάκελος και ένας server που τον διαχειρίζεται. Ο φάκελος λέγεται **data directory** και τον φτιάχνει το `initdb`.

TERMINAL

```
initdb -D "$LAB/first" -U postgres -A trust -E UTF8 \
  --locale-provider=icu --icu-locale=el-GR --locale=en_US.UTF-8
```

ΕΞΟΔΟΣ

```
The files belonging to this database system will be owned by user "you".
This user must also own the server process.

Using language tag "el-GR" for ICU locale "el-GR".
The database cluster will be initialized with this locale configuration:
  locale provider:   icu
  default collation: el-GR
  LC_COLLATE:  en_US.UTF-8
  LC_CTYPE:    en_US.UTF-8
  LC_MESSAGES: en_US.UTF-8
  LC_MONETARY: en_US.UTF-8
  LC_NUMERIC:  en_US.UTF-8
  LC_TIME:     en_US.UTF-8
The default text search configuration will be set to "english".

Data page checksums are enabled.

creating directory ~/pglab/first ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default "max_connections" ... 100
selecting default "shared_buffers" ... 128MB
selecting default time zone ... Europe/Athens
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

Success. You can now start the database server using:

    pg_ctl -D ~/pglab/first -l logfile start
```

Οι επιλογές λένε τα εξής. Το `-D` είναι ο φάκελος. Το `-U postgres` ονομάζει τον superuser `postgres`, αλλιώς θα έπαιρνε το όνομα του χρήστη του λειτουργικού. Το `-A trust` επιτρέπει σύνδεση χωρίς κωδικό, κάτι που κάνουμε μόνο σε εργαστήριο. Τα υπόλοιπα ορίζουν UTF8 και ελληνικό collation, όπως στη βάση του πρώτου τόμου.

Η έξοδος αναφέρει ότι τα **data checksums** είναι ενεργά. Από την έκδοση 18 είναι η προεπιλογή. Κάθε σελίδα των 8 KB έχει ένα checksum και ο server ελέγχει κάθε σελίδα που διαβάζει από τον δίσκο, ώστε μια αλλοίωση να φανεί ως σφάλμα και όχι ως λάθος δεδομένα. Το κόστος είναι μικρό και θα το χρειαστούμε στο κεφάλαιο 19.

Τι έχει ο φάκελος

TERMINAL

```
ls "$LAB/first"
```

```
ΕΞΟΔΟΣ

base
global
pg_commit_ts
pg_dynshmem
pg_hba.conf
pg_ident.conf
pg_logical
pg_multixact
pg_notify
pg_replslot
pg_serial
pg_snapshots
pg_stat
pg_stat_tmp
pg_subtrans
pg_tblspc
pg_twophase
PG_VERSION
pg_wal
pg_xact
postgresql.auto.conf
postgresql.conf
```

Τα περισσότερα ονόματα θα τα συναντήσουμε ξανά. Τα σημαντικά είναι λίγα.

Φάκελος ή αρχείο	Τι κρατά
`base`	ένας υποφάκελος για κάθε βάση, με ένα αρχείο για κάθε πίνακα και index
`global`	ό,τι ανήκει σε όλο το cluster, όπως οι ρόλοι και η λίστα των βάσεων
`pg_wal`	το write ahead log, το ημερολόγιο κάθε αλλαγής (κεφάλαιο 9)
`pg_xact`	η κατάσταση κάθε transaction, commit ή abort
`postgresql.conf`	οι ρυθμίσεις του server
`postgresql.auto.conf`	οι ρυθμίσεις που γράφει το `ALTER SYSTEM` (κεφάλαιο 7)
`pg_hba.conf`	ποιος συνδέεται από πού και πώς αποδεικνύει ποιος είναι
`PG_VERSION`	η major έκδοση που έφτιαξε τον φάκελο

Ο φάκελος ανήκει στον server. Δεν αλλάζεις ούτε σβήνεις αρχεία μέσα του με το χέρι, εκτός από τα αρχεία ρυθμίσεων. Ειδικά το `pg_wal` δεν είναι log με την έννοια των αρχείων καταγραφής. Αν το αδειάσεις για να κερδίσεις χώρο, χάνεις δεδομένα.

Ο πρώτος server

Πριν ξεκινήσει ο server, του λέμε σε ποια θύρα θα ακούει. Προσθέτουμε τρεις γραμμές στο τέλος του `postgresql.conf`. Όταν μια ρύθμιση εμφανίζεται δύο φορές, ισχύει η τελευταία.

TERMINAL

```
cat >> "$LAB/first/postgresql.conf" <<'EOF'
port = 5501
listen_addresses = 'localhost'
unix_socket_directories = ''
EOF
```

Το `listen_addresses` ορίζει σε ποιες διευθύνσεις δικτύου ακούει ο server. Το άδειο `unix_socket_directories` κλείνει τις συνδέσεις μέσω unix socket, ώστε όλα τα εργαλεία να συνδέονται μέσω `localhost`, όπως θα συνδεόταν μια εφαρμογή. Ο server ξεκινά με το `pg_ctl`, που τον τρέχει στο παρασκήνιο και γράφει το log του στο αρχείο του `-l`.

TERMINAL

```
pg_ctl -D "$LAB/first" -l "$LAB/first.log" start
pg_ctl -D "$LAB/first" status
```

ΕΞΟΔΟΣ

```
waiting for server to start.... done
server started
pg_ctl: server is running (PID: 75790)
/opt/homebrew/Cellar/postgresql@18/18.6/bin/postgres "-D" "~/pglab/first"
```

Το log γράφει τι έκανε ο server στην εκκίνηση.

TERMINAL

```
cat "$LAB/first.log"
```

ΕΞΟΔΟΣ

```
2026-09-28 09:46:38.249 EEST [75790] LOG:  starting PostgreSQL 18.6 (Homebrew) on aarch64-apple-
darwin25.6.0, compiled by Apple clang version 21.0.0 (clang-2100.1.1.101), 64-bit
2026-09-28 09:46:38.252 EEST [75790] LOG:  listening on IPv6 address "::1", port 5501
2026-09-28 09:46:38.252 EEST [75790] LOG:  listening on IPv4 address "127.0.0.1", port 5501
2026-09-28 09:46:38.257 EEST [75796] LOG:  database system was shut down at 2026-09-28 09:46:38 EEST
2026-09-28 09:46:38.260 EEST [75790] LOG:  database system is ready to accept connections
```

Μία διεργασία ανά σύνδεση

Η PostgreSQL δεν είναι ένα πρόγραμμα με πολλά threads. Είναι πολλές διεργασίες του λειτουργικού. Η πρώτη, ο **postmaster**, ξεκινά τις υπόλοιπες και περιμένει συνδέσεις. Το PID του είναι στην πρώτη γραμμή του `postmaster.pid`. Με το `pgrep -P` βρίσκουμε τα παιδιά του.

TERMINAL

```
ps -o pid,command -p "$(pgrep -d, -P "$(head -1 "$LAB/first/postmaster.pid")")"
```

ΕΞΟΔΟΣ

```
  PID COMMAND
75791 postgres: io worker 0
75792 postgres: io worker 1
75793 postgres: io worker 2
75794 postgres: checkpointer
75795 postgres: background writer
75797 postgres: walwriter
75798 postgres: autovacuum launcher
75799 postgres: logical replication launcher
```

Κάθε βοηθητική διεργασία έχει μία δουλειά. Το `checkpointer` και ο `background writer` γράφουν σελίδες από τη μνήμη στον δίσκο και ο `walwriter` γράφει το WAL. Θα τους δούμε στα κεφάλαια 8 και 9. Ο `autovacuum launcher` ξεκινά τα VACUUM του κεφαλαίου 10. Οι `io worker` είναι νέοι στην έκδοση 18 και κάνουν ασύγχρονες αναγνώσεις από τον δίσκο για λογαριασμό των queries.

Κάθε σύνδεση παίρνει δική της διεργασία, τον **backend**. Από μέσα η PostgreSQL δείχνει όλες τις διεργασίες στο `pg_stat_activity`. Η στήλη `backend_type` λέει τι είναι η καθεμία.

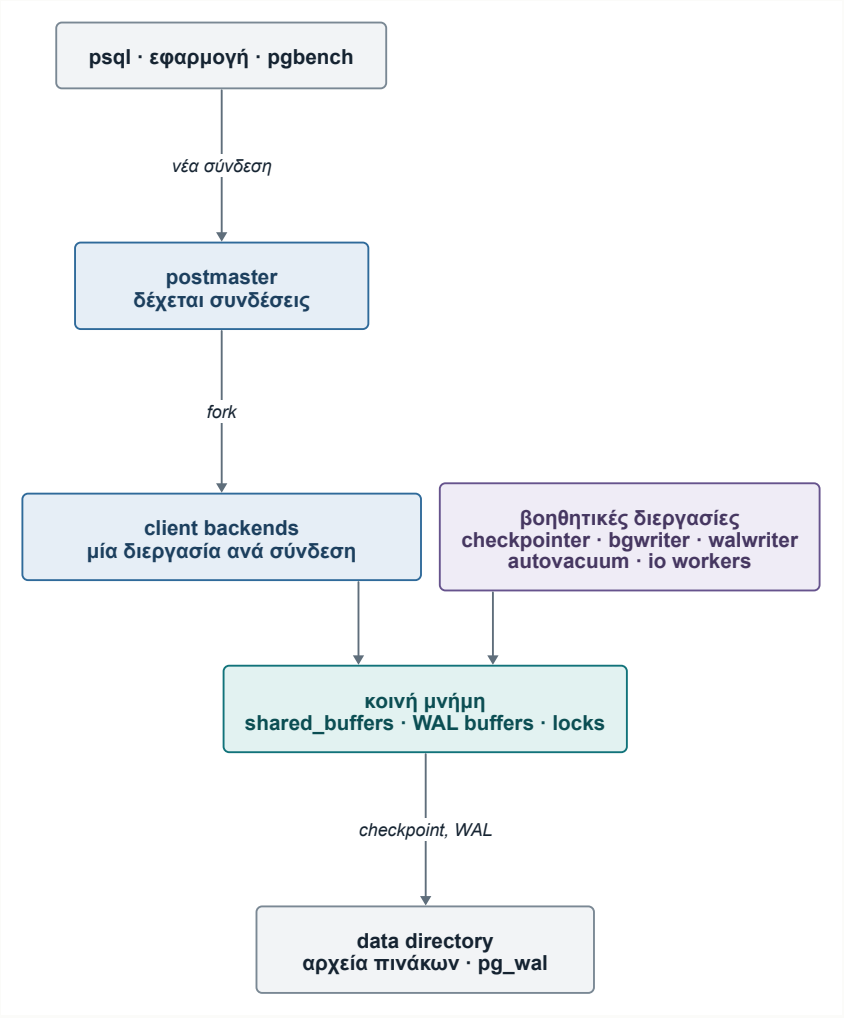
SQL

```
SELECT backend_type, count(*)
FROM pg_stat_activity
GROUP BY backend_type
ORDER BY backend_type;
```

ΑΠΟΤΕΛΕΣΜΑ

```
         backend_type          | count
-------------------------------+-------
 autovacuum launcher           |     1
 background writer             |     1
 checkpointer                  |     1
 client backend                |     1
 io worker                     |     3
 logical replication launcher  |     1
 walwriter                     |     1
(7 rows)
```

Ο μόνος `client backend` είναι η δική μας σύνδεση. Μία διεργασία ανά σύνδεση σημαίνει ότι κάθε σύνδεση κοστίζει μερικά MB μνήμης και χρόνο για να ξεκινήσει. Χίλιες συνδέσεις είναι χίλιες διεργασίες. Αυτό είναι ο λόγος ύπαρξης των connection poolers του κεφαλαίου 21.



Σχήμα 6.1 · Ο postmaster δέχεται συνδέσεις και ξεκινά έναν backend για καθεμία. Backends και βοηθητικές διεργασίες μοιράζονται την κοινή μνήμη.

pg_hba.conf

Κάθε νέα σύνδεση περνά από το `pg_hba.conf`, host based authentication. Κάθε γραμμή του έχει τύπο σύνδεσης, βάση, χρήστη, διεύθυνση και μέθοδο. Ο server διαβάζει τις γραμμές από πάνω προς τα κάτω και εφαρμόζει την πρώτη που ταιριάζει. Αν δεν ταιριάξει καμία, η σύνδεση απορρίπτεται. Οι γραμμές του `initdb` χωρίς τα σχόλια είναι λίγες.

TERMINAL

```
grep -v '^#' "$LAB/first/pg_hba.conf" | grep -v '^[[:space:]]*$'
```

ΕΞΟΔΟΣ

```
local   all             all                                     trust
host    all             all             127.0.0.1/32            trust
host    all             all             ::1/128                 trust
local   replication     all                                     trust
host    replication     all             127.0.0.1/32            trust
host    replication     all             ::1/128                 trust
```

Το `local` αφορά unix sockets και το `host` συνδέσεις TCP. Η λέξη `replication` στη θέση της βάσης αφορά τις συνδέσεις replication των κεφαλαίων 13 και 16. Όλες οι γραμμές έχουν `trust`. Ας φτιάξουμε έναν ρόλο για μια εφαρμογή που θα πρέπει να δίνει κωδικό.

TERMINAL

```
psql -d postgres -c "CREATE ROLE app LOGIN PASSWORD 'app-secret'"
```

ΕΞΟΔΟΣ

```
CREATE ROLE
```

Γράφουμε ξανά ολόκληρο το αρχείο, με δύο γραμμές για τον `app` στην αρχή. Η μέθοδος `scram-sha-256` ζητά κωδικό και τον ελέγχει χωρίς να ταξιδέψει ποτέ στο δίκτυο.

TERMINAL

```
cat > "$LAB/first/pg_hba.conf" <<'EOF'
# TYPE  DATABASE     USER  ADDRESS       METHOD
host    all          app   127.0.0.1/32  scram-sha-256
host    all          app   ::1/128       scram-sha-256
local   all          all                 trust
host    all          all   127.0.0.1/32  trust
host    all          all   ::1/128       trust
host    replication  all   127.0.0.1/32  trust
host    replication  all   ::1/128       trust
EOF
pg_ctl -D "$LAB/first" reload
```

ΕΞΟΔΟΣ

```
server signaled
```

Το `reload` λέει στον server να ξαναδιαβάσει τα αρχεία ρυθμίσεων χωρίς να σταματήσει. Οι αλλαγές στο `pg_hba.conf` ισχύουν από την επόμενη σύνδεση. Η επιλογή `-w` του `psql` σημαίνει να μη ζητήσει κωδικό από το πληκτρολόγιο.

TERMINAL

```
psql -w -U app -d postgres -c "SELECT current_user"
```

ΕΞΟΔΟΣ

```
psql: error: connection to server at "localhost" (::1), port 5501 failed: fe_sendauth: no password supplied
```

TERMINAL

```
PGPASSWORD=app-secret psql -U app -d postgres -c "SELECT current_user"
```

ΕΞΟΔΟΣ

```
 current_user
--------------
 app
(1 row)
```

Η PostgreSQL δείχνει πώς διάβασε το αρχείο στο view `pg_hba_file_rules`. Αν μια γραμμή έχει συντακτικό λάθος, εμφανίζεται εδώ με τη στήλη `error` και ο server αγνοεί ολόκληρο το νέο αρχείο στο reload.

SQL

```
SELECT line_number, type, database, user_name, address, auth_method
FROM pg_hba_file_rules
ORDER BY line_number;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 line_number | type  |   database    | user_name |  address  |  auth_method
-------------+-------+---------------+-----------+-----------+---------------
           2 | host  | {all}         | {app}     | 127.0.0.1 | scram-sha-256
           3 | host  | {all}         | {app}     | ::1       | scram-sha-256
           4 | local | {all}         | {all}     | ø         | trust
           5 | host  | {all}         | {all}     | 127.0.0.1 | trust
           6 | host  | {all}         | {all}     | ::1       | trust
           7 | host  | {replication} | {all}     | 127.0.0.1 | trust
           8 | host  | {replication} | {all}     | ::1       | trust
(7 rows)
```

ΠΑΓΙΔΑ

Σε server που ακούει σε δίκτυο δεν υπάρχει γραμμή `trust` για `host`. Η μέθοδος είναι `scram-sha-256` και η διεύθυνση όσο πιο στενή γίνεται. Η γραμμή `host all all 0.0.0.0/0 trust` σε server με δημόσια IP σημαίνει ότι όποιος βρει τη θύρα είναι superuser.

Σταμάτημα και ανάκαμψη

Το `pg_ctl stop` έχει τρεις τρόπους. Ο `smart` περιμένει να αποσυνδεθούν όλοι. Ο `fast`, η προεπιλογή, ακυρώνει τα transactions που τρέχουν, αποσυνδέει όλους και γράφει ό,τι έχει στη μνήμη στον δίσκο πριν σταματήσει. Ο `immediate` σταματά αμέσως, χωρίς να γράψει τίποτα, σαν να έπεσε το ρεύμα.

Γράφουμε μερικά δεδομένα και σταματάμε τον server με `immediate`.

TERMINAL

```
psql -d postgres -c "CREATE TABLE crash_test AS SELECT generate_series(1, 10000) AS n"
pg_ctl -D "$LAB/first" -m immediate stop
pg_ctl -D "$LAB/first" -l "$LAB/first.log" start >/dev/null
tail -n 7 "$LAB/first.log"
```

ΕΞΟΔΟΣ

```
SELECT 10000
waiting for server to shut down.... done
server stopped
2026-09-28 09:46:38.768 EEST [75840] LOG:  database system was not properly shut down; automatic recovery
in progress
2026-09-28 09:46:38.769 EEST [75840] LOG:  redo starts at 0/178D118
2026-09-28 09:46:38.772 EEST [75840] LOG:  invalid record length at 0/184D0C8: expected at least 24, got 0
2026-09-28 09:46:38.772 EEST [75840] LOG:  redo done at 0/184CF30 system usage: CPU: user: 0.00 s, system:
0.00 s, elapsed: 0.00 s
2026-09-28 09:46:38.773 EEST [75838] LOG:  checkpoint starting: end-of-recovery immediate wait
2026-09-28 09:46:38.776 EEST [75838] LOG:  checkpoint complete: wrote 73 buffers (0.4%), wrote 3 SLRU
buffers; 0 WAL file(s) added, 0 removed, 0 recycled; write=0.002 s, sync=0.001 s, total=0.003 s; sync
files=27, longest=0.001 s, average=0.001 s; distance=768 kB, estimate=768 kB; lsn=0/184D0C8, redo
lsn=0/184D0C8
2026-09-28 09:46:38.777 EEST [75834] LOG:  database system is ready to accept connections
```

Στην εκκίνηση ο server βρήκε ότι δεν σταμάτησε σωστά και έκανε **recovery**. Ξαναδιάβασε το WAL από το τελευταίο σημείο που ήξερε ότι όλα ήταν στον δίσκο και ξαναέκανε κάθε αλλαγή, το `redo`. Ο πίνακας είναι εκεί με όλες του τις γραμμές.

TERMINAL

```
psql -d postgres -c "SELECT count(*) FROM crash_test"
```

ΕΞΟΔΟΣ

```
 count
-------
 10000
(1 row)
```

Αυτός ο μηχανισμός, ένα ημερολόγιο που γράφεται πριν από τα δεδομένα και ξαναπαίζεται μετά από κάθε πτώση, είναι η βάση για όλα τα επόμενα κεφάλαια. Τα backups και η replication του μέρους III και IV είναι το ίδιο redo, σε άλλο μηχάνημα ή σε άλλη χρονική στιγμή.

Το script pglab

Κάθε κεφάλαιο από εδώ και πέρα χρειάζεται έναν ή περισσότερους servers με τις ίδιες ρυθμίσεις. Για να μη γράφουμε κάθε φορά τις ίδιες εντολές, ο φάκελος `data` του βιβλίου έχει το script `pglab`. Κάνει όσα κάναμε με το χέρι σε αυτό το κεφάλαιο, με δύο προσθήκες. Ορίζει ζώνη ώρας `Europe/Athens`, ώστε οι ώρες στα logs και στα αποτελέσματα να είναι ίδιες με του βιβλίου. Και όταν ξεκινά έναν standby του μέρους IV, περιμένει να συνδεθεί στον primary πριν επιστρέψει. Χωρίς ορίσματα τυπώνει τι κάνει.

TERMINAL

```
pglab
```

```
ΕΞΟΔΟΣ

Το εργαστήριο του βιβλίου. Κάθε server ζει σε έναν φάκελο μέσα στο $LAB.

  pglab init ONOMA θΥΡΑ   νέο cluster με τις ρυθμίσεις του βιβλίου
  pglab start ONOMA       ξεκινά τον server, log στο $LAB/ONOMA.log. Αν είναι
                          standby, περιμένει να αρχίσει να λαμβάνει WAL
  pglab stop ONOMA        σταματά τον server
  pglab restart ONOMA     σταματά και ξεκινά ξανά
  pglab data θΥΡΑ [--lab] φτιάχνει τη βάση sqlbook με τα δεδομένα του βιβλίου
  pglab ls                δείχνει τους servers και την κατάστασή τους
  pglab rm ONOMA          σταματά τον server και σβήνει τον φάκελό του
```

Σβήνουμε τον server του κεφαλαίου και φτιάχνουμε με το `pglab` τον `primary`, τον server που θα χρησιμοποιούμε από εδώ και πέρα. Το `pglab data` φορτώνει στη βάση `sqlbook` τα δεδομένα του πρώτου τόμου.

```
TERMINAL

pglab rm first
pglab init primary 5501
pglab start primary
pglab data 5501
pglab ls
```

```
ΕΞΟΔΟΣ

first: σβήστηκε
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
primary    θύρα 5501  σε λειτουργία
```

Τα επόμενα κεφάλαια ξεκινούν όλα με αυτές τις εντολές. Αν κάποιο πείραμα πάει στραβά, το `pglab rm` και οι ίδιες εντολές σε φέρνουν πίσω σε καθαρή κατάσταση σε λίγα δευτερόλεπτα.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Ένα cluster είναι ένα data directory και ένας server. Το `initdb` φτιάχνει τον φάκελο, το `pg_ctl` ξεκινά, σταματά και ξαναφορτώνει ρυθμίσεις. Κάθε σύνδεση είναι μια διεργασία και οι βοηθητικές διεργασίες γράφουν στον δίσκο, κάνουν autovacuum και διαβάζουν ασύγχρονα. Το `pg_hba.conf` αποφασίζει ποιος συνδέεται με την πρώτη γραμμή που ταιριάζει. Μετά από κάθε απότομο σταμάτημα ο server κάνει recovery ξαναπαίζοντας το WAL.

Ασκήσεις

ΑΣΚΗΣΗ 6.1 Ο φάκελος της βάσης

Βρες το OID της βάσης `sqlbook` και δείξε ότι υπάρχει υποφάκελος με αυτό το όνομα μέσα στο `base` του `primary`.

```
ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

OID: 16384
~/pglab/primary/base/16384
```

ΑΣΚΗΣΗ 6.2 Το αρχείο ενός πίνακα

Βρες με `pg_relation_filepath` σε ποιο αρχείο βρίσκεται ο πίνακας `orders` και δείξε το μέγεθός του σε bytes με `wc -c`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
base/16384/16501
   16384
```

ΑΣΚΗΣΗ 6.3 Δεύτερος server

Φτιάξε με το `pglab` δεύτερο server με όνομα `second` στη θύρα 5502, ξεκίνα τον και δείξε με `pglab ls` και τους δύο. Μετά σβήσε τον.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
second: νέο cluster στο $LAB/second, θύρα 5502
second: σε λειτουργία στη θύρα 5502
primary    θύρα 5501  σε λειτουργία
second     θύρα 5502  σε λειτουργία
second: σβήστηκε
```

ΑΣΚΗΣΗ 6.4 Οι κανόνες πρόσβασης

Δείξε από το `pg_hba_file_rules` του `primary` τη γραμμή, τον τύπο, τη βάση και τη μέθοδο κάθε κανόνα. Μέτρησε μετά πόσοι κανόνες είναι `trust`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 line_number | type  |   database    | auth_method
-------------+-------+---------------+-------------
         117 | local | {all}         | trust
         119 | host  | {all}         | trust
         121 | host  | {all}         | trust
         124 | local | {replication} | trust
         125 | host  | {replication} | trust
         126 | host  | {replication} | trust
(6 rows)

 trust_rules
-------------
           6
(1 row)
```

ΑΣΚΗΣΗ 6.5 Πόσες διεργασίες ανοίγει μια σύνδεση

Άνοιξε δύο συνδέσεις στο παρασκήνιο με `psql -c "SELECT pg_sleep(2)" &` και μέτρησε από τρίτη σύνδεση πόσοι `client backend` υπάρχουν. Περίμενε μετά να τελειώσουν με `wait`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
3
```

ΑΣΚΗΣΗ 6.6 Recovery μετά από πτώση

Γράψε στον `primary` έναν πίνακα `after_crash` με 1.000 γραμμές, σταμάτησέ τον με `immediate`, ξεκίνα τον και δείξε τις γραμμές του log για την αρχή και το τέλος του redo. Μέτρησε μετά τις γραμμές του πίνακα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 1000
waiting for server to shut down.... done
server stopped
primary: σε λειτουργία στη θύρα 5501
2026-09-28 09:46:43.751 EEST [76072] LOG:  redo starts at 0/178D118
2026-09-28 09:46:43.766 EEST [76072] LOG:  redo done at 0/1D040D8 system usage: CPU:
user: 0.00 s, system: 0.01 s, elapsed: 0.01 s
1000
```

ΚΕΦΑΛΑΙΟ 7

Ρυθμίσεις και από πού έρχονται

Η PostgreSQL έχει πάνω από τριακόσιες ρυθμίσεις. Δεν χρειάζεται να τις ξέρεις όλες. Χρειάζεται να ξέρεις πού ορίζεται μια ρύθμιση, ποια τιμή ισχύει τελικά και τι πρέπει να κάνεις για να αλλάξει. Κάποιες αλλάζουν για ένα query, κάποιες με ένα reload και κάποιες μόνο με restart του server.

Ο server του κεφαλαίου

Κάθε κεφάλαιο του εργαστηρίου ξεκινά με καθαρό server. Αν έχεις ήδη `primary` από το προηγούμενο κεφάλαιο, το `pglab rm primary` τον σβήνει.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
```

pg_settings

Κάθε ρύθμιση έχει μία γραμμή στο view `pg_settings`. Οι πιο χρήσιμες στήλες λένε την τιμή, τη μονάδα της, από πού ήρθε και πότε μπορεί να αλλάξει.

SQL

```
SELECT name, setting, unit, context, source
FROM pg_settings
WHERE name IN ('shared_buffers', 'work_mem', 'max_connections',
               'statement_timeout', 'log_min_duration_statement')
ORDER BY name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
             name             | setting | unit |  context   |       source
------------------------------+---------+------+------------+--------------------
 log_min_duration_statement   | -1      | ms   | superuser  | default
 max_connections              | 100     | ∅    | postmaster | configuration file
 shared_buffers               | 16384   | 8kB  | postmaster | configuration file
 statement_timeout            | 0       | ms   | user       | default
 work_mem                     | 4096    | kB   | user       | default
(5 rows)
```

Το `setting` είναι πάντα κείμενο στη μονάδα της στήλης `unit`. Το `shared_buffers` είναι 16384 σελίδες των 8 KB, δηλαδή 128 MB. Το `SHOW` δείχνει την ίδια τιμή σε πιο φιλική μορφή.

SQL

```
SHOW shared_buffers;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 shared_buffers
----------------
 128MB
(1 row)
```

Η στήλη `source` λέει ποιο επίπεδο έδωσε την τιμή. Το `default` σημαίνει ότι κανείς δεν την όρισε και ισχύει η τιμή που είναι γραμμένη στον κώδικα της PostgreSQL. Το `configuration file` σημαίνει ότι ήρθε από το `postgresql.conf`. Το `initdb` έγραψε εκεί το `shared_buffers` και το `max_connections` αφού έλεγξε τι αντέχει το μηχάνημα.

Το context

Η στήλη `context` είναι η πιο σημαντική. Λέει πότε μπορεί να αλλάξει μια ρύθμιση.

Context	Αλλάζει	Παραδείγματα
`postmaster`	μόνο με restart του server	`shared_buffers`, `max_connections`, `port`
`sighup`	με reload, για όλο τον server	`log_line_prefix`, `autovacuum_naptime`
`superuser`	με `SET` από superuser	`log_min_duration_statement`
`user`	με `SET` από οποιονδήποτε	`work_mem`, `statement_timeout`
`internal`	ποτέ, είναι σταθερές του build	`block_size`

Οι υπόλοιπες τιμές, `backend` και `superuser-backend`, αφορούν ρυθμίσεις που ορίζονται στη σύνδεση και μένουν ίδιες ως το τέλος της. Πόσες ρυθμίσεις έχει κάθε κατηγορία;

SQL

```
SELECT context, count(*) FROM pg_settings GROUP BY context ORDER BY count(*) DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
      context       | count
--------------------+-------
 user               |   151
 sighup             |   104
 postmaster         |    69
 superuser          |    49
 internal           |    20
 superuser-backend  |     4
 backend            |     2
(7 rows)
```

Οι περισσότερες αλλάζουν χωρίς restart. Οι 69 του `postmaster` όμως περιλαμβάνουν ό,τι αφορά μνήμη που δεσμεύεται στην εκκίνηση, θύρες, replication και shared libraries. Μια αλλαγή τους σε production σημαίνει σύντομη διακοπή, οπότε προγραμματίζεται.

Έξι επίπεδα

Μια ρύθμιση μπορεί να οριστεί σε πολλά σημεία. Όταν οριστεί σε περισσότερα από ένα, κερδίζει το πιο ειδικό.

1. Η προεπιλογή του κώδικα.
2. Το `postgresql.conf`.
3. Το `postgresql.auto.conf`, που γράφει το `ALTER SYSTEM`.
4. Η βάση, με `ALTER DATABASE ... SET`.
5. Ο ρόλος, με `ALTER ROLE ... SET`, ή ο ρόλος σε μια βάση, με `ALTER ROLE ... IN DATABASE ... SET`.
6. Η σύνδεση με `SET` και το transaction με `SET LOCAL`.

Τα επίπεδα 4 και 5 εφαρμόζονται τη στιγμή της σύνδεσης. Ας τα δούμε με το `statement_timeout`, που ακυρώνει όποιο query ξεπεράσει τον χρόνο του.

SQL

```
ALTER DATABASE sqlbook SET statement_timeout = '30s';
CREATE ROLE reporting LOGIN;
ALTER ROLE reporting SET statement_timeout = '5min';
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER DATABASE

CREATE ROLE

ALTER ROLE
```

Μια νέα σύνδεση ως `postgres` παίρνει την τιμή της βάσης και μια νέα σύνδεση ως `reporting` την τιμή του ρόλου.

TERMINAL

```
psql -c "SELECT current_user, current_setting('statement_timeout')"
psql -U reporting -c "SELECT current_user, current_setting('statement_timeout')"
```

ΕΞΟΔΟΣ

```
 current_user | current_setting
--------------+-----------------
 postgres     | 30s
(1 row)

 current_user | current_setting
--------------+-----------------
 reporting    | 5min
(1 row)
```

Ο ρόλος κέρδισε τη βάση. Έτσι ορίζεις ότι η εφαρμογή κόβει κάθε query στα 30 δευτερόλεπτα, ενώ ο ρόλος των αναφορών έχει περισσότερο περιθώριο. Το `\drds` του psql δείχνει όλες αυτές τις ρυθμίσεις μαζί.

PSQL

```
\drds
```

ΑΠΟΤΕΛΕΣΜΑ

```
              List of settings
   Role    | Database |        Settings
-----------+----------+------------------------
 reporting | ∅        | statement_timeout=5min
 ∅         | sqlbook  | statement_timeout=30s
(2 rows)
```

Το `SET` αλλάζει την τιμή για την υπόλοιπη σύνδεση και το `SET LOCAL` μόνο ως το τέλος του transaction. Η στήλη `source` του `pg_settings` δείχνει κάθε φορά από ποιο επίπεδο ήρθε η τιμή. Τα τρέχουμε σε νέα σύνδεση του psql, αφού η σύνδεση του πρώτου query του κεφαλαίου άνοιξε πριν από το `ALTER DATABASE` και κρατά ακόμη την προεπιλογή.

PSQL

```
SELECT setting, source FROM pg_settings WHERE name = 'statement_timeout';

BEGIN;
SET LOCAL statement_timeout = '2s';
SELECT setting, source FROM pg_settings WHERE name = 'statement_timeout';
COMMIT;

SELECT setting, source FROM pg_settings WHERE name = 'statement_timeout';
```

ΑΠΟΤΕΛΕΣΜΑ

```
 setting |  source
---------+----------
 30000   | database
(1 row)

BEGIN
SET
 setting | source
---------+---------
 2000    | session
(1 row)

COMMIT
 setting |  source
---------+----------
 30000   | database
(1 row)
```

Η νέα σύνδεση πήρε την τιμή της βάσης. Το `SET LOCAL` την άλλαξε για ένα transaction και χάθηκε με το `COMMIT`. Είναι ο σωστός τρόπος να δώσεις σε ένα βαρύ query περισσότερο χρόνο χωρίς να ξεχάσεις την αλλαγή στη σύνδεση, κάτι που μετράει όταν η σύνδεση επιστρέφει σε pool.

ALTER SYSTEM και reload

Για ρυθμίσεις όλου του server μπορείς να ανοίξεις το `postgresql.conf` σε editor. Το `ALTER SYSTEM` κάνει το ίδιο από SQL. Γράφει στο `postgresql.auto.conf`, που διαβάζεται μετά το `postgresql.conf` και άρα υπερισχύει. Ελέγχει επίσης το όνομα και την τιμή πριν γράψει.

SQL

```
ALTER SYSTEM SET log_min_duration_statement = '250ms';
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SYSTEM
```

SQL

```
ALTER SYSTEM SET log_min_duration_statment = '250ms';
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  unrecognized configuration parameter "log_min_duration_statment"
```

TERMINAL

```
cat "$LAB/primary/postgresql.auto.conf"
```

ΕΞΟΔΟΣ

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
log_min_duration_statement = '250ms'
```

Το αρχείο άλλαξε, αλλά ο server δεν το έχει διαβάσει ακόμη. Η ρύθμιση έχει context `superuser`, άρα φτάνει ένα reload. Το `pg_reload_conf()` κάνει από SQL ό,τι έκανε το `pg_ctl reload` του προηγούμενου κεφαλαίου.

SQL

```
SELECT pg_reload_conf();
```

ΑΠΟΤΕΛΕΣΜΑ

```
 pg_reload_conf
----------------
 t
(1 row)
```

Το reload είναι ασύγχρονο. Ο postmaster στέλνει σήμα σε κάθε διεργασία και κάθε διεργασία ξαναδιαβάζει τα αρχεία πριν εκτελέσει την επόμενη εντολή της. Ένα `SHOW` που ακολουθεί αμέσως στο ίδιο script μπορεί να δει ακόμη την παλιά τιμή. Μετά από μια στιγμή η νέα τιμή ισχύει.

SQL

```
SHOW log_min_duration_statement;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 log_min_duration_statement
----------------------------
 250ms
(1 row)
```

Το `ALTER SYSTEM RESET` σβήνει τη γραμμή από το αρχείο και η ρύθμιση επιστρέφει στην τιμή του `postgresql.conf` μετά το επόμενο reload.

Ρυθμίσεις που θέλουν restart

Ας διπλασιάσουμε το `shared_buffers`. Το `ALTER SYSTEM` δέχεται την αλλαγή και το reload δεν παραπονιέται, αλλά η τιμή δεν αλλάζει.

SQL

```
ALTER SYSTEM SET shared_buffers = '256MB';
SELECT pg_reload_conf();

SELECT name, setting, pending_restart
FROM pg_settings WHERE name = 'shared_buffers';
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SYSTEM

 pg_reload_conf
----------------
 t
(1 row)

      name      | setting | pending_restart
----------------+---------+-----------------
 shared_buffers | 16384   | f
(1 row)
```

Το `pending_restart` λέει ότι υπάρχει νέα τιμή στα αρχεία που περιμένει restart. Σε production αυτό το query είναι ένας καλός έλεγχος μετά από κάθε αλλαγή ρυθμίσεων. Κάνουμε restart και ξαναρωτάμε.

TERMINAL

```
pglab restart primary
```

ΕΞΟΔΟΣ

```
primary: σε λειτουργία στη θύρα 5501
```

SQL

```
SELECT name, setting, pending_restart
FROM pg_settings WHERE name = 'shared_buffers';
```

ΑΠΟΤΕΛΕΣΜΑ

```
      name      | setting | pending_restart
----------------+---------+-----------------
 shared_buffers | 32768   | f
(1 row)
```

Λάθη στο postgresql.conf

Όταν γράφεις με το χέρι στο `postgresql.conf`, κανείς δεν ελέγχει τι έγραψες μέχρι το επόμενο reload. Ας προσθέσουμε μια σωστή ρύθμιση και μία με ορθογραφικό λάθος.

TERMINAL

```
cat >> "$LAB/primary/postgresql.conf" <<'EOF'
work_mem = '16MB'
wrok_mem = '8MB'
EOF
pg_ctl -D "$LAB/primary" reload
sleep 1
tail -n 2 "$LAB/primary.log"
```

ΕΞΟΔΟΣ

```
server signaled
2026-09-28 09:46:48.257 EEST [76218] LOG:  unrecognized configuration parameter
"wrok_mem" in file "~/pglab/primary/postgresql.conf" line 898
2026-09-28 09:46:48.257 EEST [76218] LOG:  configuration file "~/pglab/primary/
postgresql.conf" contains errors; no changes were applied
```

Ο server βρήκε άγνωστη ρύθμιση και δεν εφάρμοσε καμία αλλαγή από το αρχείο, ούτε τη σωστή. Αυτή η συμπεριφορά σε προστατεύει από μισές αλλαγές, αλλά πρέπει να κοιτάξεις το log για να τη δεις. Το view `pg_file_settings` δείχνει τι βρίσκεται στα αρχεία, αν εφαρμόστηκε και ποιο είναι το λάθος.

SQL

```
SELECT sourceline, name, setting, applied, error
FROM pg_file_settings
WHERE name IN ('work_mem', 'wrok_mem');

SHOW work_mem;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 sourceline |   name   | setting | applied |                error
------------+----------+---------+---------+---------------------------------------
        897 | work_mem | 16MB    | f       | ∅
        898 | wrok_mem | 8MB     | f       | unrecognized configuration parameter
(2 rows)

 work_mem
----------
 4MB
(1 row)
```

Διορθώνουμε το λάθος. Σε Linux και macOS το `sed` έχει διαφορετική σύνταξη για επεξεργασία αρχείου στη θέση του, οπότε γράφουμε το αποτέλεσμα σε νέο αρχείο και το μεταφέρουμε πάνω στο παλιό.

TERMINAL

```
grep -v '^wrok_mem' "$LAB/primary/postgresql.conf" > "$LAB/pg.conf.new"
mv "$LAB/pg.conf.new" "$LAB/primary/postgresql.conf"
pg_ctl -D "$LAB/primary" reload
```

ΕΞΟΔΟΣ

```
server signaled
```

SQL

```
SHOW work_mem;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 work_mem
----------
 16MB
(1 row)
```

ΣΗΜΕΙΩΣΗ

Σε έναν πραγματικό server δεν επεξεργάζεσαι ένα αρχείο με εκατοντάδες γραμμές σχολίων. Βάζεις στο τέλος του `postgresql.conf` τη γραμμή `include_dir = 'conf.d'` και κρατάς τις δικές σου ρυθμίσεις σε μικρά αρχεία μέσα σε αυτόν τον φάκελο, ένα ανά θέμα. Είναι εύκολο να τα δεις με μια ματιά και να τα βάλεις σε version control.

ΚΡΑΤΑ ΑΥΤΟ

Το `pg_settings` δείχνει κάθε ρύθμιση, από πού ήρθε και το context της. Οι ρυθμίσεις `postmaster` θέλουν restart, οι `sighup` reload και οι `user` αλλάζουν με `SET`. Κερδίζει το πιο ειδικό επίπεδο, με σειρά αρχεία, βάση, ρόλος, σύνδεση και transaction. Το `ALTER SYSTEM` γράφει ελεγμένες τιμές στο `postgresql.auto.conf`. Μετά από κάθε αλλαγή ελέγχεις το `pending_restart` και το `pg_file_settings`, γιατί ένα λάθος στο αρχείο ακυρώνει όλη την αλλαγή.

Ασκήσεις

ΑΣΚΗΣΗ 7.1 Τι άλλαξε από τις προεπιλογές

Δείξε όλες τις ρυθμίσεις του `primary` που δεν έχουν `source` ίσο με `default`, `override` ή `client`, με το όνομα, την τιμή, τη μονάδα και την πηγή τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
            name             |      setting       | unit |       source
-----------------------------+--------------------+------+--------------------
 autovacuum_worker_slots     | 16                 | ø    | configuration file
 DateStyle                   | ISO, MDY           | ø    | configuration file
 default_text_search_config  | pg_catalog.english | ø    | configuration file
 dynamic_shared_memory_type  | posix              | ø    | configuration file
 lc_messages                 | en_US.UTF-8        | ø    | configuration file
 lc_monetary                 | en_US.UTF-8        | ø    | configuration file
 lc_numeric                  | en_US.UTF-8        | ø    | configuration file
 lc_time                     | en_US.UTF-8        | ø    | configuration file
 listen_addresses            | localhost          | ø    | configuration file
 log_min_duration_statement  | 250                | ms   | configuration file
 log_timezone                | Europe/Athens      | ø    | configuration file
 max_connections             | 100                | ø    | configuration file
 max_wal_size                | 1024               | MB   | configuration file
 min_wal_size                | 80                 | MB   | configuration file
 port                        | 5501               | ø    | configuration file
 shared_buffers              | 32768              | 8kB  | configuration file
 TimeZone                    | Europe/Athens      | ø    | configuration file
 unix_socket_directories     |                    | ø    | configuration file
 work_mem                    | 16384              | kB   | configuration file
 statement_timeout           | 30000              | ms   | database
(20 rows)
```

ΑΣΚΗΣΗ 7.2 Μέγεθος σε bytes

Δείξε το `shared_buffers`, το `work_mem` και το `maintenance_work_mem` σε μορφή που διαβάζεται, με `pg_size_pretty` πάνω στο `setting` και τη μονάδα του `pg_settings`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
         name          |  size
-----------------------+--------
 maintenance_work_mem | 64 MB
 shared_buffers       | 256 MB
 work_mem             | 16 MB
(3 rows)
```

ΑΣΚΗΣΗ 7.3 Ρύθμιση ρόλου σε μία βάση

Φτιάξε ρόλο `batch` με login και όρισε `work_mem = '64MB'` μόνο όταν συνδέεται στη βάση `sqlbook`. Δείξε από νέα σύνδεση ως `batch` την τιμή που παίρνει.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE ROLE
ALTER ROLE
64MB
```

ΑΣΚΗΣΗ 7.4 Ένα restart που περιμένει

Άλλαξε με `ALTER SYSTEM` το `max_connections` σε 150 και κάνε reload. Δείξε το `setting` και το `pending_restart` του. Κάνε restart τον `primary` και δείξε τα ξανά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SYSTEM
t
100|t
primary: σε λειτουργία στη θύρα 5501
150|f
```

ΑΣΚΗΣΗ 7.5 Χρόνος μόνο για ένα transaction

Μέσα σε ένα transaction όρισε `work_mem` 256 MB μόνο για αυτό το transaction και δείξε την τιμή. Μετά το `COMMIT` δείξε ξανά την τιμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

SET

 work_mem
----------
 256MB
(1 row)

COMMIT

 work_mem
----------
 16MB
(1 row)
```

ΑΣΚΗΣΗ 7.6 Ρυθμίσεις ανά context

Μέτρησε πόσες ρυθμίσεις του `pg_settings` που ξεκινούν από `autovacuum` έχει κάθε context.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  context   | count
------------+-------
 postmaster |     3
 sighup     |    13
(2 rows)
```

ΚΕΦΑΛΑΙΟ 8

Μνήμη

Η PostgreSQL χρησιμοποιεί μνήμη σε τρία επίπεδα. Μια κοινή cache σελίδων για όλες τις συνδέσεις και μνήμη εργασίας για κάθε ταξινόμηση και hash κάθε query. Το τρίτο επίπεδο είναι η cache αρχείων του λειτουργικού, που η PostgreSQL δεν ελέγχει αλλά υπολογίζει. Η σωστή ρύθμιση δεν είναι «όσο περισσότερη τόσο καλύτερα». Είναι ένας προϋπολογισμός που πρέπει να βγαίνει ακόμη και στη χειρότερη ώρα του server.

Ο server του κεφαλαίου

Χρειαζόμαστε τα μεγάλα δεδομένα του εργαστηρίου, οπότε το `pglab data` παίρνει και το `--lab`. Η φόρτωση διαρκεί περίπου δεκαπέντε δευτερόλεπτα.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501 --lab
pglab restart primary
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
primary: σε λειτουργία στη θύρα 5501
```

Το restart αδειάζει τη μνήμη του server, ώστε να ξεκινήσουμε από καθαρή cache.

Ο χάρτης της μνήμης

Μνήμη	Ρύθμιση	Ποιος τη χρησιμοποιεί
κοινή cache σελίδων	`shared_buffers`	όλες οι συνδέσεις μαζί, δεσμεύεται στην εκκίνηση
buffers του WAL	`wal_buffers`	όλες οι συνδέσεις, μικρή
μνήμη εργασίας	`work_mem`	κάθε κόμβος ταξινόμησης ή hash, σε κάθε query
μνήμη συντήρησης	`maintenance_work_mem`	`CREATE INDEX`, `VACUUM`, `ALTER TABLE ADD FOREIGN KEY`

Μνήμη	Ρύθμιση	Ποιος τη χρησιμοποιεί
cache του λειτουργικού	καμία, την υπολογίζει το `effective_cache_size`	ό,τι διαβάζεται από αρχεία

Κάθε σελίδα που διαβάζει ένα query περνά πρώτα από το `shared_buffers`. Αν δεν είναι εκεί, η PostgreSQL τη ζητά από το λειτουργικό. Το λειτουργικό έχει τη δική του cache και ίσως τη βρει εκεί χωρίς να πάει στον δίσκο. Γι' αυτό μια σελίδα `read` στο `EXPLAIN` του πρώτου τόμου δεν σημαίνει απαραίτητα δίσκο. Σημαίνει ότι η σελίδα δεν ήταν στη δική μας cache.

Τι υπάρχει στο shared_buffers

Η επέκταση `pg_buffercache` δείχνει κάθε buffer, δηλαδή κάθε θέση των 8 KB στο `shared_buffers`. Για καθένα λέει ποια σελίδα ποιου πίνακα κρατά.

SQL

```
CREATE EXTENSION pg_buffercache;

SELECT buffers_used, buffers_unused, buffers_dirty
FROM pg_buffercache_summary();
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE EXTENSION

 buffers_used | buffers_unused | buffers_dirty
--------------+----------------+---------------
          221 |          16163 |            54
(1 row)
```

Μετά το restart σχεδόν όλα τα 16.384 buffers είναι άδεια. Ας διαβάσουμε ολόκληρο το `lab.orders`, έναν πίνακα 132 MB, όσο δηλαδή το `shared_buffers`. Περιμένεις ότι μετά θα γεμίσει η cache με παραγγελίες.

SQL

```
SELECT count(*) FROM lab.orders;

SELECT c.oid::regclass AS relation, count(*) AS buffers
FROM pg_buffercache AS b
JOIN pg_class AS c ON b.relfilenode = pg_relation_filenode(c.oid)
WHERE b.reldatabase = (SELECT oid FROM pg_database WHERE datname = current_database())
GROUP BY c.oid
ORDER BY buffers DESC
LIMIT 4;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  count
---------
 2000000
(1 row)

   relation   | buffers
--------------+---------
 lab.orders   |     282
 pg_attribute |      61
 pg_proc      |      21
 pg_class     |      18
(4 rows)
```

Μόνο μερικές εκατοντάδες σελίδες του πίνακα έμειναν στη cache, από τις σχεδόν 17.000 που διαβάστηκαν. Δεν είναι λάθος. Όταν ένα sequential scan διαβάζει πίνακα μεγαλύτερο από το ένα τέταρτο του `shared_buffers`, η PostgreSQL του δίνει ένα μικρό **ring buffer** και ξαναχρησιμοποιεί τις ίδιες λίγες θέσεις ξανά και ξανά. Έτσι μια αναφορά που σαρώνει έναν τεράστιο πίνακα δεν διώχνει από τη μνήμη τις σελίδες που χρειάζονται όλα τα υπόλοιπα queries. Τα ίδια ισχύουν για το `VACUUM` και το `COPY`.

Αν θέλεις πραγματικά έναν πίνακα στη μνήμη, για παράδειγμα μετά από restart, το `pg_prewarm` τον φορτώνει ολόκληρο.

SQL

```
CREATE EXTENSION pg_prewarm;
SELECT pg_prewarm('lab.customers') AS pages_loaded;

SELECT c.oid::regclass AS relation, count(*) AS buffers
FROM pg_buffercache AS b
JOIN pg_class AS c ON b.relfilenode = pg_relation_filenode(c.oid)
WHERE b.reldatabase = (SELECT oid FROM pg_database WHERE datname = current_database())
GROUP BY c.oid
ORDER BY buffers DESC
LIMIT 4;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE EXTENSION

 pages_loaded
--------------
         3249
(1 row)

   relation    | buffers
---------------+---------
 lab.customers |    3249
 lab.orders    |     282
 pg_attribute  |      61
 pg_proc       |      22
(4 rows)
```

Ποια σελίδα φεύγει

Όταν το `shared_buffers` γεμίσει, κάθε νέα σελίδα πρέπει να πάρει τη θέση μιας παλιάς. Κάθε buffer έχει ένα **usage count** από 0 έως 5. Αυξάνεται κάθε φορά που κάποιος χρησιμοποιεί τη σελίδα. Όταν χρειάζεται θέση, ένας δείκτης γυρίζει κυκλικά τα buffers και μειώνει το usage count καθενός που συναντά. Το πρώτο που βρίσκει με μηδέν ελευθερώνεται. Ο αλγόριθμος λέγεται **clock sweep**. Οι σελίδες που χρησιμοποιούνται συχνά έχουν ψηλό usage count και επιβιώνουν πολλούς γύρους.

SQL

```
SELECT usagecount, count(*) AS buffers
FROM pg_buffercache
WHERE usagecount IS NOT NULL
GROUP BY usagecount
ORDER BY usagecount;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 usagecount | buffers
------------+---------
          1 |    3626
          2 |      18
          3 |      24
          4 |      23
          5 |     138
(5 rows)
```

Οι σελίδες του `pg_prewarm` διαβάστηκαν μία φορά και έχουν usage count 1. Όσες έχουν 5 είναι κυρίως σελίδες του καταλόγου, όπως το `pg_class` και το `pg_attribute`, που τις διαβάζει κάθε query.

Πόσο συχνά βρίσκει ο server τη σελίδα στη μνήμη

Το `pg_stat_database` μετρά για κάθε βάση τα `blks_hit` και τα `blks_read`. Τα πρώτα είναι σελίδες που βρέθηκαν στο `shared_buffers` και τα δεύτερα σελίδες που ζητήθηκαν από το λειτουργικό. Ο λόγος τους είναι το **cache hit ratio**.

SQL

```
SELECT blks_hit, blks_read,
       round(100.0 * blks_hit / nullif(blks_hit + blks_read, 0), 1) AS hit_pct
FROM pg_stat_database
WHERE datname = current_database();
```

ΑΠΟΤΕΛΕΣΜΑ

```
 blks_hit | blks_read | hit_pct
----------+-----------+---------
 19203349 |     74213 |    99.6
(1 row)
```

Πάνω από 99 τοις εκατό, αλλά μην ξεγελιέσαι. Τα εκατομμύρια hits ήρθαν κυρίως από τη φόρτωση του εργαστηρίου, όπου κάθε `INSERT` γράφει σε σελίδα που μόλις έφτιαξε στη μνήμη. Ένας μέσος όρος για όλη τη ζωή του server κρύβει τις κακές ώρες. Τον μετράς ως διαφορά ανάμεσα σε δύο στιγμές, για

παράδειγμα κάθε λεπτό από ένα σύστημα παρακολούθησης. Τον βλέπεις και ανά πίνακα στο `pg_statio_user_tables`, που δείχνει ποιος πίνακας δεν χωρά στη μνήμη.

Πόσο shared_buffers

Το `shared_buffers` δεν χρειάζεται να είναι τεράστιο, γιατί από πίσω υπάρχει η cache του λειτουργικού. Μια σελίδα που διώχνεται από το `shared_buffers` πολύ πιθανόν είναι ακόμη στη μνήμη του λειτουργικού. Ο κοινός κανόνας είναι περίπου το ένα τέταρτο της μνήμης του μηχανήματος, για server που τρέχει μόνο PostgreSQL. Πολύ περισσότερο σημαίνει ότι οι ίδιες σελίδες κρατιούνται δύο φορές, μία στη μνήμη της PostgreSQL και μία του λειτουργικού.

Το `effective_cache_size` δεν δεσμεύει τίποτα. Λέει στον planner πόση μνήμη υπάρχει συνολικά για cache, το `shared_buffers` μαζί με την cache του λειτουργικού. Με μεγαλύτερη τιμή ο planner θεωρεί πιθανότερο ότι οι σελίδες ενός index θα βρεθούν στη μνήμη και προτιμά πιο εύκολα index scans. Συνήθως ορίζεται στο μισό έως τρία τέταρτα της μνήμης του μηχανήματος.

work_mem σε όλο τον server

Στο κεφάλαιο 32 του πρώτου τόμου είδαμε ότι μια ταξινόμηση που δεν χωρά στο `work_mem` γράφει προσωρινά αρχεία. Εδώ μας ενδιαφέρει πώς το βλέπεις σε όλο τον server, όχι σε ένα `EXPLAIN`. Η ρύθμιση `log_temp_files` γράφει στο log κάθε προσωρινό αρχείο μεγαλύτερο από το όριό της. Με 0 γράφει όλα.

SQL

```
ALTER SYSTEM SET log_temp_files = 0;
SELECT pg_reload_conf();
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SYSTEM

 pg_reload_conf
----------------
 t
(1 row)
```

SQL

```
EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, BUFFERS OFF)
SELECT customer_id, sum(total) AS spent
FROM lab.orders
GROUP BY customer_id
ORDER BY spent DESC
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Limit (actual rows=5.00 loops=1)
  ->  Sort (actual rows=5.00 loops=1)
        Sort Key: (sum(total)) DESC
        Sort Method: top-N heapsort  Memory: 25kB
        ->  Finalize GroupAggregate (actual rows=199642.00 loops=1)
```

```
              Group Key: customer_id
              ->  Gather Merge (actual rows=544744.00 loops=1)
                    Workers Planned: 2
                    Workers Launched: 2
                    ->  Sort (actual rows=181581.33 loops=3)
                          Sort Key: customer_id
                          Sort Method: external merge  Disk: 14208kB
                          Worker 0:  Sort Method: external merge  Disk: 13992kB
                          Worker 1:  Sort Method: external merge  Disk: 13976kB
                          ->  Partial HashAggregate (actual rows=181581.33 loops=3)
                                Group Key: customer_id
                                Batches: 21  Memory Usage: 8249kB  Disk Usage: 19232kB
                                Worker 0:  Batches: 21  Memory Usage: 8249kB  Disk Usage: 15920kB
                                Worker 1:  Batches: 21  Memory Usage: 8249kB  Disk Usage: 15904kB
                                ->  Parallel Seq Scan on orders (actual rows=666666.67 loops=3)
Planning Time: 0.445 ms
Execution Time: 558.086 ms
```

Η ομαδοποίηση 200.000 πελατών δεν χώρεσε στα 4 MB. Το `HashAggregate` έσπασε σε 21 batches και έγραψε στον δίσκο και η ταξινόμηση που ακολουθεί έγινε με `external merge`. Κάθε worker του parallel plan έχει δικό του `work_mem`, άρα τρεις διεργασίες έγραψαν αρχεία. Στο log φαίνεται κάθε αρχείο με το μέγεθός του.

TERMINAL

```
grep 'temporary file' "$LAB/primary.log" | tail -n 3
```

ΕΞΟΔΟΣ

```
2026-09-28 09:47:09.881 EEST [76554] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp76554.0", size 16285696
2026-09-28 09:47:09.883 EEST [76540] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp76540.1", size 14548992
2026-09-28 09:47:09.913 EEST [76540] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp76540.0", size 19693568
```

Και στο `pg_stat_database` υπάρχει το σύνολο για κάθε βάση από την τελευταία επαναφορά των στατιστικών.

SQL

```
SELECT temp_files, pg_size_pretty(temp_bytes) AS temp_size
FROM pg_stat_database
WHERE datname = current_database();
```

ΑΠΟΤΕΛΕΣΜΑ

```
 temp_files | temp_size
------------+-----------
          7 | 101 MB
(1 row)
```

Αν αυτός ο αριθμός μεγαλώνει γρήγορα, κάποια queries θέλουν περισσότερη μνήμη ή καλύτερο index. Με περισσότερο `work_mem` το ίδιο query δεν γράφει τίποτα.

SQL

```
SET work_mem = '64MB';

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, BUFFERS OFF)
SELECT customer_id, sum(total) AS spent
FROM lab.orders
GROUP BY customer_id
ORDER BY spent DESC
LIMIT 5;

RESET work_mem;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET

Limit (actual rows=5.00 loops=1)
  ->  Sort (actual rows=5.00 loops=1)
        Sort Key: (sum(total)) DESC
        Sort Method: top-N heapsort  Memory: 25kB
        ->  HashAggregate (actual rows=199642.00 loops=1)
              Group Key: customer_id
              Batches: 1  Memory Usage: 73745kB
              ->  Seq Scan on orders (actual rows=2000000.00 loops=1)
Planning Time: 0.026 ms
Execution Time: 645.016 ms

RESET
```

Το `HashAggregate` χρησιμοποίησε πάνω από 64 MB μνήμης σε ένα batch. Τα hash nodes έχουν όριο `work_mem` επί `hash_mem_multiplier`, που είναι 2 από προεπιλογή, γιατί ένα hash που σπάει σε batches κοστίζει περισσότερο από μια ταξινόμηση που γράφει στον δίσκο. Άρα αυτό το query θα μπορούσε να πάρει ως 128 MB. Ο planner διάλεξε επίσης plan χωρίς workers, αφού πλέον δεν είχε λόγο να μοιράσει τη δουλειά.

Ένας προϋπολογισμός

Ας κάνουμε τον λογαριασμό για έναν server 16 GB με `max_connections = 200`. Το `shared_buffers` παίρνει 4 GB. Κάθε σύνδεση χρειάζεται μερικά MB μόνο και μόνο για να υπάρχει, ας πούμε 10, άρα 2 GB στο χειρότερο. Μένουν περίπου 10 GB για την cache του λειτουργικού και για το `work_mem`.

Αν το `work_mem` είναι 64 MB και 200 συνδέσεις τρέχουν ταυτόχρονα ένα query με δύο hash nodes, η θεωρητική χειρότερη περίπτωση είναι 200 επί 2 επί 128 MB, δηλαδή 50 GB. Ο server θα έμενε χωρίς μνήμη και το λειτουργικό θα σκότωνε διεργασίες. Γι’ αυτό το `work_mem` του server μένει μικρό, 16 έως 32 MB σε έναν τέτοιο server. Μεγαλώνει με `ALTER ROLE ... SET` για τον ρόλο των αναφορών ή με `SET LOCAL` για ένα συγκεκριμένο βαρύ query. Αυτό είναι επίσης ο λόγος που οι λίγες συνδέσεις του pooler του κεφαλαίου 21 είναι ασφαλέστερες από τις πολλές.

Το `maintenance_work_mem` είναι λιγότερο επικίνδυνο. Το χρησιμοποιούν λίγες διεργασίες ταυτόχρονα, όσα `CREATE INDEX` τρέχεις και οι workers του autovacuum. Με 1 GB ένα μεγάλο `CREATE INDEX` ταξινομεί στη μνήμη και τελειώνει πολύ πιο γρήγορα.

ΠΑΓΙΔΑ

Σε Linux με πολλά GB `shared_buffers`, η μνήμη χωρίζεται σε σελίδες των 4 KB του λειτουργικού και ο πίνακας που τις αντιστοιχίζει γίνεται κι αυτός μεγάλος σε κάθε διεργασία. Τα **huge pages** του Linux, σελίδες των 2 MB, το λύνουν. Η PostgreSQL τα χρησιμοποιεί αν είναι διαθέσιμα, με τη ρύθμιση `huge_pages = try`, αλλά πρέπει να τα ενεργοποιήσεις στο λειτουργικό.

ΚΡΑΤΑ ΑΥΤΟ

Το `shared_buffers` είναι η κοινή cache σελίδων, περίπου το ένα τέταρτο της μνήμης. Τα μεγάλα sequential scans χρησιμοποιούν ring buffer και δεν τη γεμίζουν. Το `pg_buffercache` δείχνει τι περιέχει και το `pg_stat_database` το hit ratio. Το `effective_cache_size` είναι μόνο υπόδειξη για τον planner. Το `work_mem` ισχύει ανά κόμβο, ανά worker και ανά σύνδεση, τα hash nodes παίρνουν το διπλάσιο. Το `log_temp_files` δείχνει ποια queries δεν χωρούν. Κάνε τον λογαριασμό για τη χειρότερη ώρα, όχι για τη μέση.

Ασκήσεις

ΑΣΚΗΣΗ 8.1 Ποιοι πίνακες είναι στη μνήμη

Δείξε για τους πίνακες του schema `lab` πόσα MB από τις σελίδες τους βρίσκονται τώρα στο `shared_buffers`, ταξινομημένα από τον μεγαλύτερο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    relation     |   cached
-----------------+------------
 lab.customers   | 25 MB
 lab.orders      | 5264 kB
 lab.orders_pkey | 8192 bytes
(3 rows)
```

ΑΣΚΗΣΗ 8.2 Hit ratio ανά πίνακα

Από το `pg_statio_user_tables` δείξε για τους πίνακες του `lab` τα `heap_blks_hit`, `heap_blks_read` και το ποσοστό hit με ένα δεκαδικό.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  relname  | heap_blks_hit | heap_blks_read | hit_pct
-----------+---------------+----------------+---------
 customers |       4206716 |          12998 |    99.7
 events    |       1067725 |            594 |    99.9
 orders    |       4035119 |         100703 |    97.6
 products  |         53571 |           2235 |    96.0
(4 rows)
```

ΑΣΚΗΣΗ 8.3 Ταξινόμηση που χωρά

Με `EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, BUFFERS OFF)` δείξε το plan της ταξινόμησης των `lab.customers` κατά `last_name`, `first_name`, `email`, πρώτα με την προεπιλογή του `work_mem` και μετά με 32 MB μόνο για αυτό το transaction.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Sort (actual rows=200000.00 loops=1)
  Sort Key: last_name, first_name, email
  Sort Method: external merge  Disk: 12680kB
  ->  Seq Scan on customers (actual rows=200000.00 loops=1)
Planning Time: 0.098 ms
Execution Time: 333.913 ms

BEGIN

SET

Sort (actual rows=200000.00 loops=1)
  Sort Key: last_name, first_name, email
  Sort Method: quicksort  Memory: 20629kB
  ->  Seq Scan on customers (actual rows=200000.00 loops=1)
Planning Time: 0.023 ms
Execution Time: 354.272 ms

COMMIT
```

ΑΣΚΗΣΗ 8.4 Μνήμη για τον ρόλο των αναφορών

Φτιάξε ρόλο `reports` με login και όρισε `work_mem` 128 MB για αυτόν. Δείξε από νέα σύνδεση ως `reports` την τιμή του `work_mem` και την πηγή της.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE ROLE
ALTER ROLE
 setting | unit | source
---------+------+--------
 131072  | kB   | user
(1 row)
```

ΑΣΚΗΣΗ 8.5 Πόσα προσωρινά αρχεία

Μηδένισε τα στατιστικά της βάσης με `pg_stat_reset()`, τρέξε ταξινόμηση όλων των `lab.orders` κατά `total` με `work_mem` 4 MB και δείξε τα `temp_files` και το μέγεθός τους από το `pg_stat_database`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 pg_stat_reset
---------------

(1 row)

   id
--------
 188570
(1 row)

 pg_stat_force_next_flush
--------------------------

(1 row)

 temp_files | temp_size
------------+-----------
          2 | 60 MB
(1 row)
```

ΑΣΚΗΣΗ 8.6 Φόρτωση στη μνήμη

Φόρτωσε με `pg_prewarm` το primary key index του `lab.orders` και δείξε πόσες σελίδες του βρίσκονται στο `shared_buffers`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 pages_loaded
--------------
         5486
(1 row)

 buffers
---------
    5486
(1 row)
```

ΚΕΦΑΛΑΙΟ 9

WAL, checkpoints και durability

Ένα commit πρέπει να επιβιώσει από μια πτώση ρεύματος. Η PostgreSQL δεν το πετυχαίνει γράφοντας κάθε αλλαγμένη σελίδα στον δίσκο τη στιγμή του commit. Γράφει μια σύντομη περιγραφή της αλλαγής σε ένα ημερολόγιο, το write ahead log. Τις σελίδες τις γράφει αργότερα. Αυτό το κεφάλαιο εξηγεί τι γράφεται στο WAL, πόσο μεγάλο βγαίνει και πότε οι σελίδες φτάνουν τελικά στον δίσκο.

Ο server του κεφαλαίου

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
```

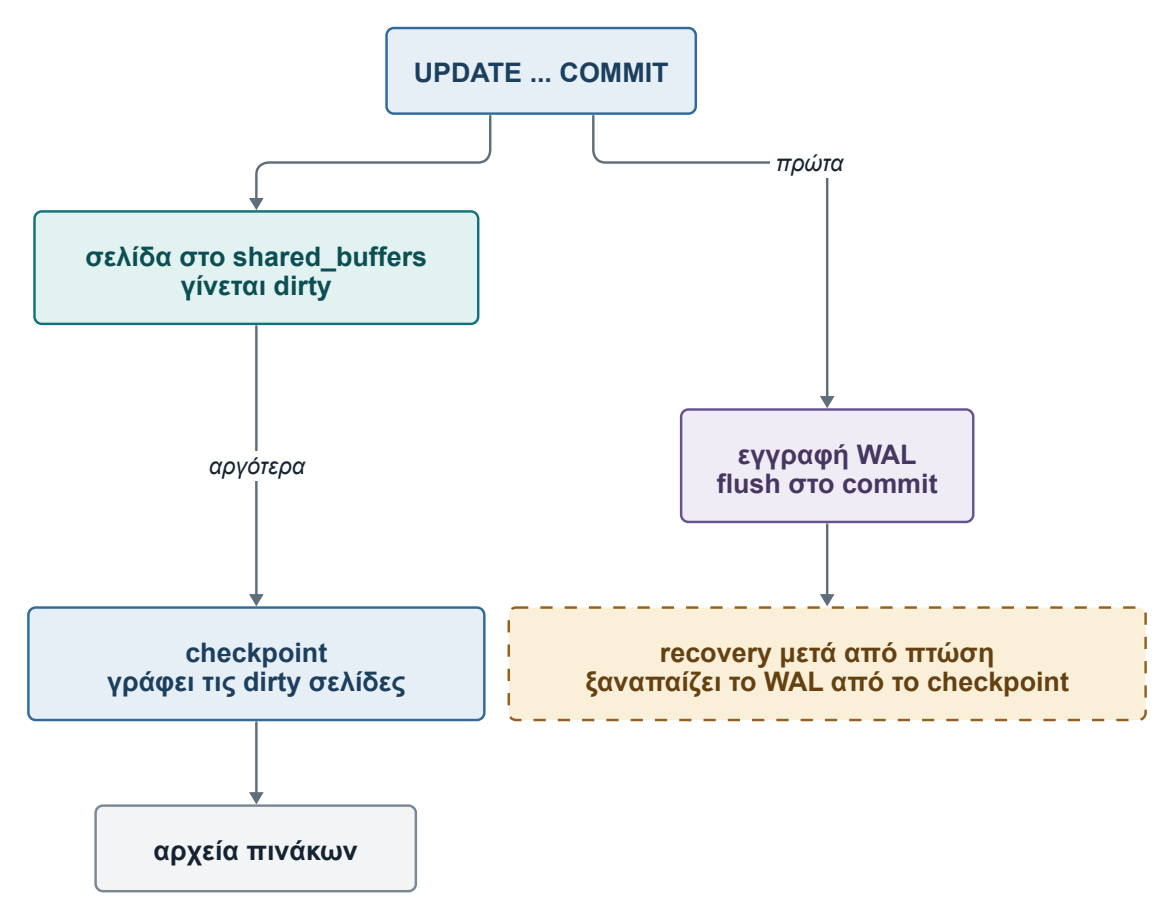
ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
```

Πρώτα το ημερολόγιο

Όταν ένα `UPDATE` αλλάζει μια γραμμή, η PostgreSQL αλλάζει τη σελίδα στο `shared_buffers` και προσθέτει μια εγγραφή στο **WAL** που περιγράφει την αλλαγή. Η σελίδα μένει στη μνήμη ως **dirty** και θα γραφτεί στον δίσκο κάποια στιγμή αργότερα. Στο commit η PostgreSQL περιμένει μόνο ένα πράγμα, να φτάσουν στον δίσκο οι εγγραφές του WAL μέχρι και την εγγραφή του commit.

Αυτό είναι γρηγορότερο για δύο λόγους. Το WAL γράφεται σειριακά, στο τέλος ενός αρχείου, ενώ οι σελίδες είναι σκόρπιες σε όλο τον δίσκο. Και μια εγγραφή WAL είναι συνήθως μερικές δεκάδες bytes, ενώ μια σελίδα είναι 8 KB. Αν ο server πέσει, το recovery του κεφαλαίου 6 ξαναπαίζει το WAL και φέρνει κάθε σελίδα στην κατάσταση που είχε μετά το τελευταίο commit.



Σχήμα 9.1 · Το commit περιμένει μόνο το WAL. Οι σελίδες φτάνουν στα αρχεία με το checkpoint και μετά από πτώση το recovery ξαναπαίζει το WAL από το τελευταίο checkpoint.

Κάθε θέση στο WAL έχει μια διεύθυνση, το **LSN**, log sequence number. Είναι ένας αριθμός που μεγαλώνει πάντα και γράφεται ως δύο δεκαεξαδικά χωρισμένα με κάθετο.

SQL

```
SELECT pg_current_wal_lsn() AS lsn,
       pg_walfile_name(pg_current_wal_lsn()) AS file;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    lsn    |           file
-----------+--------------------------
 0/1CF3730 | 000000010000000000000001
(1 row)
```

Το WAL είναι χωρισμένο σε αρχεία των 16 MB, τα **segments**, στον φάκελο `pg_wal`. Το όνομα κάθε αρχείου βγαίνει από το LSN.

TERMINAL

```
ls "$LAB/primary/pg_wal"
```

ΕΞΟΔΟΣ

```
000000010000000000000001
archive_status
summaries
```

Ο φάκελος `archive_status` χρησιμεύει στο archiving του κεφαλαίου 13 και ο `summaries` στα incremental backups του κεφαλαίου 15.

Τι γράφει ένα INSERT

Η επέκταση `pg_walinspect` διαβάζει το WAL από SQL. Κρατάμε το LSN πριν και μετά από ένα μικρό transaction και ζητάμε τις εγγραφές ανάμεσα. Το `\gset` του psql αποθηκεύει τις στήλες ενός αποτελέσματος σε μεταβλητές του psql, που χρησιμοποιούνται μετά ως `:'όνομα'`.

PSQL

```
CREATE EXTENSION pg_walinspect;
CREATE TABLE notes (id integer PRIMARY KEY, body text);
CHECKPOINT;

SELECT pg_current_wal_lsn() AS before \gset
INSERT INTO notes VALUES (1, 'πρώτη σημείωση');
SELECT pg_current_wal_lsn() AS after \gset

SELECT resource_manager, record_type, record_length, fpi_length, description
FROM pg_get_wal_records_info(:'before', :'after');
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE EXTENSION
CREATE TABLE
CHECKPOINT
INSERT 0 1
 resource_manager | record_type  | record_length | fpi_length |         description
------------------+--------------+---------------+------------+-----------------------------
 XLOG             | FPI_FOR_HINT |          3169 |       3120 | ∅
 XLOG             | FPI_FOR_HINT |          4973 |       4924 | ∅
 XLOG             | FPI_FOR_HINT |          3629 |       3580 | ∅
 XLOG             | FPI_FOR_HINT |          2357 |       2308 | ∅
 Heap             | INSERT+INIT  |            87 |          0 | off: 1, flags: 0x00
 Btree            | NEWROOT      |            90 |          0 | level: 0
 Btree            | INSERT_LEAF  |            64 |          0 | off: 1
 Transaction      | COMMIT       |            34 |          0 | 2026-09-28 09:47:15.35412+03
(8 rows)
```

Οι τέσσερις τελευταίες εγγραφές είναι το ίδιο το `INSERT`. Μία στο heap με τη γραμμή, δύο στο B-tree του primary key, που μόλις απέκτησε την πρώτη του σελίδα και την πρώτη του τιμή. Η τελευταία είναι το commit. Οι τέσσερις πρώτες είναι ολόκληρες σελίδες. Η στήλη `fpi_length` είναι το μέγεθος αυτών των **full page images** και θέλει εξήγηση.

Full page writes

Ο δίσκος δεν εγγυάται ότι μια σελίδα των 8 KB γράφεται ολόκληρη ή καθόλου. Αν το ρεύμα κοπεί στη μέση, μπορεί να μείνει μισή παλιά και μισή νέα, μια **torn page**. Το WAL περιγράφει αλλαγές, όχι ολόκληρες σελίδες. Μια αλλαγή δεν μπορεί να εφαρμοστεί πάνω σε σελίδα που είναι άχρηστη.

Γι' αυτό η πρώτη αλλαγή κάθε σελίδας μετά από κάθε checkpoint γράφει στο WAL ολόκληρη τη σελίδα. Όσες αλλαγές ακολουθούν στην ίδια σελίδα μέχρι το επόμενο checkpoint γράφουν μόνο την περιγραφή τους. Στο recovery η PostgreSQL ξεκινά από την εικόνα της σελίδας και εφαρμόζει τις αλλαγές πάνω της.

Οι εγγραφές `FPI_FOR_HINT` είναι μια ειδική περίπτωση. Όταν η PostgreSQL διαβάζει μια γραμμή και μαθαίνει ότι το transaction που την έγραψε έκανε commit, σημειώνει στη γραμμή ένα **hint bit**, για να μη χρειαστεί να το ψάξει ξανά. Με ενεργά checksums, ακόμη και αυτή η μικρή αλλαγή αλλάζει το checksum της σελίδας, οπότε η σελίδα γράφεται ολόκληρη στο WAL αν είναι η πρώτη της αλλαγή μετά από checkpoint. Εδώ ήταν σελίδες του καταλόγου που διάβασε το `INSERT`. Το `CHECKPOINT` λίγο πριν έκανε όλες τις σελίδες «πρώτες».

Η διαφορά φαίνεται καθαρά όταν αλλάζεις πολλές γραμμές δύο φορές στη σειρά.

PSQL

```
CREATE TABLE wal_demo AS
SELECT n AS id, repeat('x', 100) AS pad FROM generate_series(1, 100000) AS n;
CHECKPOINT;

SELECT pg_current_wal_lsn() AS a \gset
UPDATE wal_demo SET pad = repeat('y', 100) WHERE id <= 20000;
SELECT pg_current_wal_lsn() AS b \gset
UPDATE wal_demo SET pad = repeat('z', 100) WHERE id <= 20000;
SELECT pg_current_wal_lsn() AS c \gset

SELECT pg_size_pretty(pg_wal_lsn_diff(:'b', :'a')) AS first_update,
       pg_size_pretty(pg_wal_lsn_diff(:'c', :'b')) AS second_update;

SELECT 'first' AS pass, sum(count) AS records,
       pg_size_pretty(sum(record_size)) AS records_size,
       pg_size_pretty(sum(fpi_size)) AS page_images
FROM pg_get_wal_stats(:'a', :'b')
UNION ALL
SELECT 'second', sum(count), pg_size_pretty(sum(record_size)), pg_size_pretty(sum(fpi_size))
FROM pg_get_wal_stats(:'b', :'c');
```

ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 100000
CHECKPOINT
UPDATE 20000
UPDATE 20000
 first_update | second_update
--------------+---------------
 18 MB        | 4761 kB
(1 row)

  pass  | records | records_size | page_images
--------+---------+--------------+-------------
 first  |   41733 | 4634 kB      | 13 MB
 second |   40346 | 4608 kB      | 0 bytes
(2 rows)
```

Το ίδιο `UPDATE` έγραψε πολλαπλάσιο WAL την πρώτη φορά. Οι εγγραφές των αλλαγών έχουν σχεδόν το ίδιο μέγεθος και στα δύο. Η διαφορά είναι ακριβώς οι εικόνες των σελίδων. Όσο πιο συχνά γίνονται checkpoints, τόσο πιο συχνά κάθε σελίδα γίνεται ξανά «πρώτη» και τόσο περισσότερο WAL γράφει ο server για την ίδια δουλειά.

Οι εικόνες σελίδων συμπιέζονται καλά. Με `wal_compression` η PostgreSQL τις συμπιέζει πριν τις γράψει.

PSQL

```
ALTER SYSTEM SET wal_compression = lz4;
SELECT pg_reload_conf();
SELECT pg_sleep(1);
CHECKPOINT;

SELECT pg_current_wal_lsn() AS a \gset
UPDATE wal_demo SET pad = repeat('w', 100) WHERE id <= 20000;
SELECT pg_current_wal_lsn() AS b \gset

SELECT pg_size_pretty(pg_wal_lsn_diff(:'b', :'a')) AS compressed_first_update,
       pg_size_pretty(sum(fpi_size)) AS page_images
FROM pg_get_wal_stats(:'a', :'b');
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)

 pg_sleep
----------

(1 row)

CHECKPOINT
UPDATE 20000
 compressed_first_update | page_images
-------------------------+-------------
 5349 kB                 | 550 kB
(1 row)
```

Οι εικόνες σελίδων έπεσαν από αρκετά MB σε μερικές εκατοντάδες KB. Το κόστος είναι λίγη CPU. Σε servers με replication, όπου το WAL ταξιδεύει στο δίκτυο και αποθηκεύεται στα backups, το `wal_compression` αξίζει σχεδόν πάντα. Η τιμή `lz4` υπάρχει μόνο αν η PostgreSQL χτίστηκε με lz4, όπως σχεδόν όλα τα πακέτα. Αλλιώς υπάρχει το `pglz`, πιο αργό.

Checkpoints

Ένα **checkpoint** γράφει στον δίσκο όλες τις dirty σελίδες του `shared_buffers` και σημειώνει στο WAL ότι μέχρι αυτό το LSN όλα είναι στον δίσκο. Μετά από πτώση, το recovery ξεκινά από το τελευταίο checkpoint και όχι από την αρχή του χρόνου. Και τα segments του WAL πριν από αυτό δεν χρειάζονται πια, οπότε ανακυκλώνονται.

Ένα checkpoint ξεκινά για έναν από δύο λόγους. Πέρασε το `checkpoint_timeout`, 5 λεπτά από προεπιλογή, ή γράφτηκε τόσο WAL από το προηγούμενο ώστε να πλησιάζει το `max_wal_size`, 1 GB από

προεπιλογή. Ο πρώτος λόγος είναι ο φυσιολογικός. Ο δεύτερος σημαίνει ότι ο server γράφει περισσότερο απ' όσο προβλέψαμε.

Το view `pg_stat_checkpointer` μετρά τα δύο είδη. Το `num_timed` είναι checkpoints λόγω χρόνου και το `num_requested` όλα τα υπόλοιπα, μαζί με όσα ζήτησε κάποιος με την εντολή `CHECKPOINT`.

SQL

```
SELECT num_timed, num_requested, buffers_written
FROM pg_stat_checkpointer;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 num_timed | num_requested | buffers_written
-----------+---------------+-----------------
         0 |             3 |            5278
(1 row)
```

Ας δούμε τι γίνεται με πολύ μικρό `max_wal_size`. Το `pgbench`, ένα εργαλείο φόρτου που έρχεται με την PostgreSQL, φτιάχνει τους δικούς του πίνακες με το `-i`. Με `-s 5` ο `pgbench_accounts` έχει 500.000 γραμμές. Μετά αλλάζουμε όλες τις γραμμές τρεις φορές.

TERMINAL

```
psql -c "ALTER SYSTEM SET max_wal_size = '64MB'" -c "SELECT pg_reload_conf()"
pgbench -i -s 5 -q >/dev/null 2>&1
for i in 1 2 3; do
  psql -q -c "UPDATE pgbench_accounts SET abalance = abalance + 1"
done
grep 'checkpoint starting' "$LAB/primary.log" | tail -n 2
grep -E 'too frequently|HINT' "$LAB/primary.log" | tail -n 2
```

ΕΞΟΔΟΣ

```
ALTER SYSTEM
 pg_reload_conf
----------------
 t
(1 row)

2026-09-28 09:47:22.096 EEST [76680] LOG:  checkpoint starting: wal
2026-09-28 09:47:22.421 EEST [76680] LOG:  checkpoint starting: wal
2026-09-28 09:47:22.421 EEST [76680] LOG:  checkpoints are occurring too frequently (0 seconds apart)
2026-09-28 09:47:22.421 EEST [76680] HINT:  Consider increasing the configuration parameter "max_wal_size".
```

Τα checkpoints ξεκινούν με αιτία `wal` και η PostgreSQL προειδοποιεί ότι γίνονται πολύ συχνά, με υπόδειξη να μεγαλώσεις το `max_wal_size`. Όταν τα δεις σε ένα πραγματικό log, αυτό κάνεις.

SQL

```
SELECT num_timed, num_requested, buffers_written
FROM pg_stat_checkpointer;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 num_timed | num_requested | buffers_written
-----------+---------------+-----------------
         0 |            21 |           70692
(1 row)
```

Ένα checkpoint δεν γράφει όλες τις σελίδες μαζί. Τις απλώνει στο 90% του χρόνου ως το επόμενο προγραμματισμένο checkpoint, όπως ορίζει το `checkpoint_completion_target`, για να μη γονατίσει ο δίσκος. Ενδιάμεσα ο **background writer** γράφει μερικές dirty σελίδες που θα χρειαστούν σύντομα, ώστε ένα query που ψάχνει ελεύθερο buffer να μη χρειαστεί να γράψει το ίδιο.

Το log γράφει για κάθε checkpoint πόσες σελίδες έγραψε και πόσο κράτησε. Η ρύθμιση `log_checkpoints` είναι ενεργή από προεπιλογή.

TERMINAL

```
grep 'checkpoint complete' "$LAB/primary.log" | tail -n 1
```

ΕΞΟΔΟΣ

```
2026-09-28 09:47:22.421 EEST [76680] LOG:  checkpoint complete: wrote 2527 buffers
(15.4%), wrote 0 SLRU buffers; 0 WAL file(s) added, 1 removed, 1 recycled; write=0.313
s, sync=0.008 s, total=0.326 s; sync files=3, longest=0.007 s, average=0.003 s;
distance=30577 kB, estimate=38796 kB; lsn=0/2718FEB8, redo lsn=0/250DB1C8
```

ΣΗΜΕΙΩΣΗ

Ο κανόνας για το `max_wal_size` είναι απλός. Βάλε το αρκετά μεγάλο ώστε σχεδόν όλα τα checkpoints να είναι `timed`. Ο χώρος του `pg_wal` μπορεί να φτάσει ως αυτό το μέγεθος, οπότε ο δίσκος πρέπει να τον έχει. Μεγαλύτερο διάστημα ανάμεσα στα checkpoints σημαίνει λιγότερα full page images, αλλά μεγαλύτερο recovery μετά από πτώση, αφού υπάρχει περισσότερο WAL για ξαναπαίξιμο.

Πόσο ασφαλές είναι ένα commit

Τρεις ρυθμίσεις αλλάζουν την εγγύηση που περιγράψαμε. Οι δύο δεν πρέπει να τις αγγίξεις ποτέ σε production.

Ρύθμιση	Τι κάνει όταν είναι off	Τι χάνεις σε πτώση
`fsync`	ο server δεν ζητά ποτέ από το λειτουργικό να γράψει στον δίσκο	ίσως ολόκληρο το cluster, από αλλοιωμένα δεδομένα
`full_page_writes`	καμία εικόνα σελίδας στο WAL	σελίδες που δεν διορθώνονται μετά από torn write
`synchronous_commit`	το commit επιστρέφει πριν γραφτεί το WAL στον δίσκο	τα commits των τελευταίων κλασμάτων του δευτερολέπτου

Το `synchronous_commit = off` είναι διαφορετικό από τα άλλα δύο. Δεν αλλοιώνει τίποτα. Σε μια πτώση χάνονται μερικά από τα τελευταία commits, ως τρεις φορές το `wal_writer_delay`, δηλαδή λιγότερο από ένα δευτερόλεπτο. Η βάση μένει συνεπής, σαν να μην έγιναν ποτέ. Μπορείς να το ορίσεις για ένα

transaction με `SET LOCAL`. Ένα log επισκέψεων ή ένα event analytics μπορεί να το ανεχτεί, μια πληρωμή όχι.

SQL

```
BEGIN;
SET LOCAL synchronous_commit = off;
INSERT INTO events (customer_id, kind, occurred_at) VALUES (1, 'view', now());
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

SET

INSERT 0 1

COMMIT
```

ΠΑΓΙΔΑ

Σε macOS το `fsync` δεν αναγκάζει τον δίσκο να αδειάσει την εσωτερική του cache, οπότε σε laptop ένα commit είναι γρήγορο με όποια ρύθμιση κι αν δοκιμάσεις. Σε ένα Linux server με κανονικούς δίσκους το `synchronous_commit = off` πολλαπλασιάζει τα commits ανά δευτερόλεπτο. Μη βγάζεις συμπεράσματα για απόδοση commit από το laptop σου.

Πίνακες χωρίς WAL

Ένας **unlogged** πίνακας δεν γράφει καθόλου WAL. Οι εγγραφές του είναι πολύ πιο γρήγορες, αλλά μετά από πτώση ο server δεν μπορεί να ξέρει αν είναι συνεπής, οπότε τον αδειάζει. Ούτε περνά σε replicas. Είναι για δεδομένα που μπορείς να ξαναφτιάξεις, όπως έναν πίνακα που φορτώνεις από αρχείο πριν τον επεξεργαστείς.

PSQL

```
CREATE UNLOGGED TABLE staging AS SELECT n FROM generate_series(1, 100000) AS n;

SELECT pg_current_wal_lsn() AS a \gset
INSERT INTO staging SELECT n FROM generate_series(1, 100000) AS n;
SELECT pg_current_wal_lsn() AS b \gset
INSERT INTO wal_demo SELECT n, 'x' FROM generate_series(100001, 200000) AS n;
SELECT pg_current_wal_lsn() AS c \gset

SELECT pg_size_pretty(pg_wal_lsn_diff(:'b', :'a')) AS unlogged_insert,
       pg_size_pretty(pg_wal_lsn_diff(:'c', :'b')) AS logged_insert;

SELECT count(*) AS staging_rows FROM staging;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 100000
INSERT 0 100000
INSERT 0 100000
 unlogged_insert | logged_insert
-----------------+---------------
 0 bytes         | 6855 kB
(1 row)

 staging_rows
--------------
       200000
(1 row)
```

Το ίδιο πλήθος γραμμών δεν έγραψε καθόλου WAL στον unlogged πίνακα και αρκετά MB στον κανονικό. Τώρα προσομοιώνουμε πτώση.

TERMINAL

```
pg_ctl -D "$LAB/primary" -m immediate stop
pglab start primary
psql -c "SELECT count(*) AS staging_rows FROM staging"
```

ΕΞΟΔΟΣ

```
waiting for server to shut down.... done
server stopped
primary: σε λειτουργία στη θύρα 5501
 staging_rows
--------------
            0
(1 row)
```

Ο πίνακας υπάρχει ακόμη, άδειος. Ένα κανονικό σταμάτημα θα τον κρατούσε ακέραιο. Μόνο μια πτώση τον αδειάζει.

ΚΡΑΤΑ ΑΥΤΟ

Κάθε αλλαγή γράφεται πρώτα στο WAL και το commit περιμένει μόνο το WAL. Η θέση στο WAL είναι το LSN και το `pg_walinspect` δείχνει τις εγγραφές. Η πρώτη αλλαγή κάθε σελίδας μετά από checkpoint γράφει ολόκληρη τη σελίδα, οπότε τα συχνά checkpoints φουσκώνουν το WAL και το `wal_compression` το μικραίνει. Τα checkpoints πρέπει να είναι σχεδόν όλα `timed`, αλλιώς μεγαλώνεις το `max_wal_size`. Το `fsync` και το `full_page_writes` μένουν πάντα on. Το `synchronous_commit = off` χάνει τα τελευταία commits χωρίς να αλλοιώσει δεδομένα. Οι unlogged πίνακες αδειάζουν μετά από πτώση.

Ασκήσεις

ΑΣΚΗΣΗ 9.1 Το WAL ενός index

Μέτρησε πόσο WAL γράφει η δημιουργία ενός index στο `pgbench_accounts(abalance)`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
16 MB
```

ΑΣΚΗΣΗ 9.2 Οι εγγραφές ενός UPDATE

Με `pg_walinspect` δείξε τις εγγραφές WAL που γράφει ένα `UPDATE` του `body` της σημείωσης 1 στον πίνακα `notes`, με τον resource manager, τον τύπο και το μήκος τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 resource_manager | record_type  | record_length
------------------+--------------+---------------
 XLOG             | FPI_FOR_HINT |           129
 Heap             | HOT_UPDATE   |            87
 Transaction      | COMMIT       |            34
(3 rows)
```

ΑΣΚΗΣΗ 9.3 Τα αρχεία του pg_wal

Δείξε το όνομα του τρέχοντος segment και μέτρησε πόσα αρχεία segments υπάρχουν στο `pg_wal` του `primary`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
000000010000000000000028
4
```

ΑΣΚΗΣΗ 9.4 Checkpoint με το χέρι

Κάνε `CHECKPOINT` και δείξε από το `pg_stat_checkpointer` πώς άλλαξαν τα `num_requested` και `num_done`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 num_requested | num_done
---------------+----------
             1 |        1
(1 row)

CHECKPOINT

 pg_stat_clear_snapshot
------------------------

(1 row)

 num_requested | num_done
---------------+----------
             2 |        2
(1 row)
```

ΑΣΚΗΣΗ 9.5 Unlogged σε logged

Μετάτρεψε τον πίνακα `staging` σε κανονικό με `ALTER TABLE ... SET LOGGED`, αφού πρώτα βάλεις 100.000 γραμμές. Μέτρησε πόσο WAL έγραψε η μετατροπή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
6318 kB
p
```

ΑΣΚΗΣΗ 9.6 Commit χωρίς αναμονή

Μέσα σε ένα transaction όρισε `synchronous_commit` σε `off` μόνο για αυτό το transaction, δείξε την τιμή και πρόσθεσε ένα event. Δείξε την τιμή ξανά μετά το commit.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

SET

 synchronous_commit
--------------------
 off
(1 row)

INSERT 0 1

COMMIT

 synchronous_commit
--------------------
 on
(1 row)
```

ΚΕΦΑΛΑΙΟ 10

Autovacuum σε βάθος

Στον πρώτο τόμο είδαμε ότι κάθε `UPDATE` και `DELETE` αφήνει νεκρές εκδόσεις γραμμών και ότι το `VACUUM` τις καθαρίζει. Ο autovacuum το κάνει αυτόματα. Εδώ βλέπουμε πότε αποφασίζει να τρέξει, πώς το ρυθμίζεις για πίνακες που δεν ταιριάζουν στις προεπιλογές, τι τον εμποδίζει και γιατί ένα `VACUUM` που δεν τρέχει ποτέ μπορεί να σταματήσει ολόκληρη τη βάση.

Ο server του κεφαλαίου

Εκτός από τα δεδομένα του βιβλίου φορτώνουμε και τους πίνακες του `pgbench`. Ο autovacuum ελέγχει κάθε βάση κάθε λεπτό, όπως ορίζει το `autovacuum_naptime`. Για να μην περιμένουμε, στο εργαστήριο τον βάζουμε να ελέγχει κάθε δευτερόλεπτο. Ζητάμε επίσης να γράφει στο log κάθε εκτέλεσή του.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pgbench -i -s 5 -q >/dev/null 2>&1
psql -q -c "ALTER SYSTEM SET autovacuum_naptime = '1s'" \
        -c "ALTER SYSTEM SET log_autovacuum_min_duration = 0" \
        -c "SELECT pg_reload_conf()"
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
 pg_reload_conf
----------------
 t
(1 row)
```

Πότε τρέχει

Ο autovacuum τρέχει `VACUUM` σε έναν πίνακα όταν οι νεκρές γραμμές του ξεπεράσουν ένα όριο. Το όριο έχει ένα σταθερό μέρος και ένα αναλογικό.

ΑΠΟΤΕΛΕΣΜΑ

```
όριο = autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor × γραμμές
```

Με τις προεπιλογές, 50 και 0,2, ένας πίνακας 500.000 γραμμών χρειάζεται 100.050 νεκρές γραμμές. Για τα `INSERT` υπάρχει χωριστό όριο με τα `autovacuum_vacuum_insert_*`, ώστε και πίνακες που μόνο μεγαλώνουν να περνούν από `VACUUM` και να παγώνουν οι γραμμές τους, κάτι που θα δούμε παρακάτω. Για το `ANALYZE` το όριο έχει τα `autovacuum_analyze_*`, 50 και 0,1.

Το query που ακολουθεί υπολογίζει για κάθε πίνακα πόσες νεκρές γραμμές έχει και πόσες χρειάζονται για το επόμενο autovacuum. Το `current_setting` επιστρέφει την τιμή μιας ρύθμισης ως κείμενο.

SQL

```
SELECT relname, n_live_tup, n_dead_tup,
       current_setting('autovacuum_vacuum_threshold')::integer
         + current_setting('autovacuum_vacuum_scale_factor')::numeric * n_live_tup
         AS vacuum_at
FROM pg_stat_user_tables
WHERE relname LIKE 'pgbench%'
ORDER BY relname;
```

ΑΠΟΤΕΛΕΣΜΑ

```
     relname      | n_live_tup | n_dead_tup | vacuum_at
------------------+------------+------------+-----------
 pgbench_accounts |     500000 |          0 |  100050.0
 pgbench_branches |          5 |          0 |      51.0
 pgbench_history  |          0 |          0 |      50.0
 pgbench_tellers  |         50 |          0 |      60.0
(4 rows)
```

Αλλάζουμε 150.000 λογαριασμούς. Είναι περισσότεροι από το όριο, οπότε μέσα σε ένα δευτερόλεπτο ο autovacuum θα ξυπνήσει.

TERMINAL

```
psql -c "UPDATE pgbench_accounts SET abalance = 1 WHERE aid <= 150000"
sleep 5
```

ΕΞΟΔΟΣ

```
UPDATE 150000
```

SQL

```
SELECT relname, n_dead_tup, last_autovacuum IS NOT NULL AS vacuumed,
       autovacuum_count, autoanalyze_count
FROM pg_stat_user_tables
WHERE relname = 'pgbench_accounts';
```

ΑΠΟΤΕΛΕΣΜΑ

```
     relname      | n_dead_tup | vacuumed | autovacuum_count | autoanalyze_count
------------------+------------+----------+------------------+-------------------
 pgbench_accounts |          0 | t        |                1 |                 1
(1 row)
```

Οι νεκρές γραμμές καθαρίστηκαν. Τρέξαμε και `ANALYZE`, αφού άλλαξαν περισσότερες από το 10% των γραμμών. Το log γράφει τι ακριβώς έκανε.

TERMINAL

```
grep -A 12 'automatic vacuum of table "sqlbook.public.pgbench_accounts"' \
  "$LAB/primary.log" | head -n 13
```

ΕΞΟΔΟΣ

```
2026-09-28 09:47:27.576 EEST [76983] LOG:  automatic vacuum of table "sqlbook.public.pgbench_accounts":
index scans: 1
    pages: 0 removed, 10656 remain, 4920 scanned (46.17% of total), 0 eagerly scanned
    tuples: 150000 removed, 499988 remain, 0 are dead but not yet removable
    removable cutoff: 836, which was 0 XIDs old when operation ended
    new relfrozenxid: 831, which is 5 XIDs ahead of previous value
    frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
    visibility map: 4920 pages set all-visible, 2460 pages set all-frozen (0 were all-visible)
    index scan needed: 2460 pages from table (23.09% of total) had 150000 dead item identifiers removed
    index "pgbench_accounts_pkey": pages: 1786 in total, 0 newly deleted, 0 currently deleted, 0 reusable
    avg read rate: 32.501 MB/s, avg write rate: 0.034 MB/s
    buffer usage: 13168 hits, 958 reads, 1 dirtied
    WAL usage: 10663 records, 1 full page images, 1908390 bytes, 0 buffers full
    system usage: CPU: user: 0.05 s, system: 0.00 s, elapsed: 0.23 s
```

Οι γραμμές που αξίζει να διαβάζεις είναι λίγες. Το `pages` λέει πόσες σελίδες σάρωσε. Το `VACUUM` δεν διαβάζει ολόκληρο τον πίνακα, μόνο τις σελίδες που δεν είναι σημειωμένες στο **visibility map** ως all visible, δηλαδή σελίδες όπου όλες οι γραμμές είναι ορατές σε όλους. Το `tuples` λέει πόσες νεκρές γραμμές αφαίρεσε και πόσες ήταν νεκρές αλλά δεν μπορούσαν να αφαιρεθούν. Το `index scan` λέει ότι πέρασε και από τα indexes για να σβήσει τους δείκτες προς τις γραμμές που αφαίρεσε.

Πίνακες που δεν ταιριάζουν στις προεπιλογές

Το 20% είναι λογικό για έναν πίνακα χιλιάδων γραμμών. Για έναν πίνακα 500 εκατομμυρίων σημαίνει 100 εκατομμύρια νεκρές γραμμές πριν από το πρώτο `VACUUM`, ένας τεράστιος πίνακας με χιλιάδες φουσκωμένες σελίδες και ένα `VACUUM` που κρατά ώρες. Η PostgreSQL 18 έβαλε ένα ανώτατο όριο, το `autovacuum_vacuum_max_threshold`, 100 εκατομμύρια από προεπιλογή, αλλά για πολλούς πίνακες είναι ακόμη πολύ.

Οι ρυθμίσεις του autovacuum ορίζονται και ανά πίνακα, ως **storage parameters**.

SQL

```
ALTER TABLE pgbench_accounts SET (
  autovacuum_vacuum_scale_factor = 0.01,
  autovacuum_vacuum_threshold = 1000
);

SELECT relname, reloptions FROM pg_class WHERE relname = 'pgbench_accounts';
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

     relname      |                                      reloptions
------------------+---------------------------------------------------------------------------------------
 pgbench_accounts | {fillfactor=100,autovacuum_vacuum_scale_factor=0.01,autovacuum_vacuum_threshold=1000}
(1 row)
```

Τώρα το όριο είναι 1.000 συν 1% των γραμμών, δηλαδή 6.000. Ο autovacuum θα τρέχει πιο συχνά, αλλά κάθε φορά με λιγότερη δουλειά. Για πίνακες που δέχονται πολλά `UPDATE`, όπως ουρές εργασιών ή μετρητές, αυτή είναι σχεδόν πάντα η σωστή κατεύθυνση.

Το αντίθετο σπάνια χρειάζεται. Με `autovacuum_enabled = false` σταματάς τον autovacuum για έναν πίνακα, κάτι που δεν πρέπει να μείνει έτσι. Ακόμη και τότε ο autovacuum θα τρέξει για να αποτρέψει wraparound, όπως θα δούμε.

Πόσο γρήγορα δουλεύει

Ο autovacuum είναι σκόπιμα αργός, για να μην ανταγωνίζεται την εφαρμογή. Κάθε σελίδα που διαβάζει ή λερώνει κοστίζει μονάδες. Όταν μαζέψει όσες ορίζει το `vacuum_cost_limit`, 200 από προεπιλογή, κοιμάται για `autovacuum_vacuum_cost_delay`, 2 ms από προεπιλογή. Με τις προεπιλογές ένα worker διαβάζει από τον δίσκο το πολύ μερικές δεκάδες MB το δευτερόλεπτο. Σε έναν σύγχρονο server με SSD και πίνακες εκατοντάδων GB αυτό είναι λίγο.

Υπάρχουν επίσης το πολύ `autovacuum_max_workers` workers ταυτόχρονα, 3 από προεπιλογή, για όλες τις βάσεις του server. Το όριο κόστους μοιράζεται ανάμεσά τους. Αν έχεις πολλούς μεγάλους πίνακες και οι νεκρές γραμμές μεγαλώνουν ενώ ο autovacuum τρέχει συνέχεια, ανεβάζεις το `vacuum_cost_limit` σε 1.000 ή 2.000 και ίσως τα workers. Από την έκδοση 18 το `autovacuum_max_workers` αλλάζει με reload, μέχρι τον αριθμό του `autovacuum_worker_slots`.

Τι εμποδίζει το VACUUM

Μια νεκρή γραμμή αφαιρείται μόνο όταν κανένα transaction δεν μπορεί πια να τη δει. Ένα transaction που ξεκίνησε πριν πεθάνει η γραμμή μπορεί να τη δει, οπότε όσο είναι ανοιχτό η γραμμή μένει. Ας το δούμε. Η συνεδρία B ανοίγει ένα transaction σε `REPEATABLE READ` και διαβάζει κάτι, οπότε κρατά ένα snapshot.

SQL · ΣΥΝΕΔΡΙΑ B

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT count(*) FROM pgbench_branches;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ B

```
BEGIN

 count
-------
     5
(1 row)
```

Η συνεδρία A αλλάζει όλους τους tellers και τρέχει `VACUUM` με `VERBOSE`, που τυπώνει ό,τι γράφει ο autovacuum στο log.

SQL · ΣΥΝΕΔΡΙΑ A

```
UPDATE pgbench_tellers SET tbalance = tbalance + 1;
VACUUM (VERBOSE) pgbench_tellers;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ A

```
UPDATE 50

INFO:  vacuuming "sqlbook.public.pgbench_tellers"
INFO:  finished vacuuming "sqlbook.public.pgbench_tellers": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total), 0 eagerly scanned
tuples: 0 removed, 100 remain, 50 are dead but not yet removable
removable cutoff: 852, which was 1 XIDs old when operation ended
new relfrozenxid: 852, which is 27 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 0 pages set all-visible, 0 pages set all-frozen (0 were all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 23 hits, 0 reads, 0 dirtied
WAL usage: 1 records, 0 full page images, 299 bytes, 0 buffers full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Οι 50 νεκρές γραμμές είναι `dead but not yet removable`. Το `removable cutoff` είναι το παλαιότερο transaction που βλέπει ακόμη κάποιος και λέει πόσα transactions πίσω βρίσκεται.

Σε production αυτό είναι το πιο συχνό πρόβλημα με το `VACUUM`. Μια σύνδεση μένει `idle in transaction` για ώρες και σε όλη τη βάση, όχι μόνο στους πίνακες που διάβασε, καμία νεκρή γραμμή δεν αφαιρείται. Το `backend_xmin` του `pg_stat_activity` δείχνει ποια σύνδεση κρατά πίσω τον ορίζοντα. Η συνάρτηση `age` λέει πόσα transactions πίσω είναι.

SQL · ΣΥΝΕΔΡΙΑ A

```
SELECT pid, state, age(backend_xmin) AS xmin_age, left(query, 40) AS query
FROM pg_stat_activity
WHERE backend_xmin IS NOT NULL AND pid <> pg_backend_pid()
ORDER BY age(backend_xmin) DESC;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ A

```
  pid  |        state        | xmin_age |                  query
-------+---------------------+----------+-----------------------------------------
 77041 | idle in transaction |        1 | SELECT count(*) FROM pgbench_branches
(1 row)
```

Η συνεδρία B είναι `idle in transaction` και το τελευταίο της query ήταν το `SELECT` που πήρε το snapshot. Μόλις κλείσει, το επόμενο `VACUUM` αφαιρεί τις γραμμές.

SQL · ΣΥΝΕΔΡΙΑ B

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ B

```
COMMIT
```

SQL · ΣΥΝΕΔΡΙΑ A

```
VACUUM (VERBOSE) pgbench_tellers;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Α

```
INFO:  vacuuming "sqlbook.public.pgbench_tellers"
INFO:  finished vacuuming "sqlbook.public.pgbench_tellers": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total), 0 eagerly scanned
tuples: 50 removed, 50 remain, 0 are dead but not yet removable
removable cutoff: 853, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 1 pages set all-visible, 0 pages set all-frozen (0 were all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 23 hits, 0 reads, 0 dirtied
WAL usage: 3 records, 0 full page images, 612 bytes, 0 buffers full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Την ίδια επίδραση έχουν ένα replication slot που δεν καταναλώνεται και ένα standby με `hot_standby_feedback`, που θα δούμε στο κεφάλαιο 17, καθώς και τα prepared transactions. Η ρύθμιση `idle_in_transaction_session_timeout` αποσυνδέει όποια σύνδεση μείνει ανοιχτή σε transaction χωρίς να κάνει τίποτα περισσότερο από τον χρόνο της. Σε μια εφαρμογή ένα λεπτό είναι γενναιόδωρο όριο.

Freeze και wraparound

Κάθε transaction που αλλάζει δεδομένα παίρνει ένα **transaction ID**, το xid. Κάθε γραμμή θυμάται το xid που την έγραψε και η PostgreSQL αποφασίζει ποιες γραμμές βλέπει ένα transaction συγκρίνοντας xids. Το xid όμως είναι αριθμός των 32 bits. Μετά από περίπου 4 δισεκατομμύρια transactions ξαναρχίζει από την αρχή και η σύγκριση λειτουργεί κυκλικά. Κάθε xid βλέπει δύο δισεκατομμύρια transactions ως παρελθόν και δύο ως μέλλον.

Αν μια γραμμή γράφτηκε πριν από περισσότερα από δύο δισεκατομμύρια transactions, το xid της θα έμοιαζε ξαφνικά με μέλλον και η γραμμή θα εξαφανιζόταν. Γι' αυτό το `VACUUM` **παγώνει** τις παλιές γραμμές. Τις σημειώνει ως frozen, που σημαίνει «ορατή σε όλους για πάντα». Από εκεί και πέρα το xid τους δεν μετράει.

Κάθε πίνακας κρατά στο `relfrozenxid` το παλαιότερο xid που μπορεί να έχει ακόμη κάποια μη παγωμένη γραμμή του και κάθε βάση το ελάχιστο των πινάκων της στο `datfrozenxid`. Η ηλικία αυτών των τιμών είναι το πιο σημαντικό νούμερο παρακολούθησης μιας PostgreSQL.

SQL

```
SELECT datname, age(datfrozenxid) AS xid_age
FROM pg_database
ORDER BY xid_age DESC;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  datname  | xid_age
-----------+---------
 postgres  |     109
 sqlbook   |     109
 template1 |     109
 template0 |     109
(4 rows)
```

Όταν ένας πίνακας φτάσει τα 200 εκατομμύρια, το `autovacuum_freeze_max_age`, ο autovacuum τρέχει σε αυτόν ένα `VACUUM` «to prevent wraparound», ακόμη κι αν ο πίνακας δεν έχει ούτε μία νεκρή γραμμή και

ακόμη κι αν ο autovacuum είναι απενεργοποιημένος. Αυτό το `VACUUM` σαρώνει όλο τον πίνακα και δεν σταματά όταν κάποιος ζητήσει lock. Σε πολύ μεγάλο πίνακα μπορεί να κρατήσει ώρες.

> **ΠΑΓΙΔΑ**
> Αν για οποιονδήποτε λόγο το freeze δεν προλαβαίνει, για παράδειγμα επειδή ένα ξεχασμένο transaction ή replication slot κρατά πίσω τον ορίζοντα, η ηλικία συνεχίζει να μεγαλώνει. Λίγο πριν από τα δύο δισεκατομμύρια η PostgreSQL σταματά να δέχεται νέα transactions που γράφουν, για να προστατέψει τα δεδομένα. Η βάση γίνεται στην πράξη read only μέχρι να τρέξει `VACUUM`. Ένα alert όταν το `age(datfrozenxid)` ξεπερνά τα 500 εκατομμύρια σε γλιτώνει.

Ας δούμε πώς μεγαλώνει η ηλικία. Το `pgbench` τρέχει transactions που γράφουν, οπότε καθένα καταναλώνει ένα xid.

TERMINAL

```
psql -At -c "SELECT age(relfrozenxid) FROM pg_class WHERE relname = 'pgbench_branches'"
pgbench -n -t 2000 -c 2 2>&1 | grep 'number of transactions actually processed'
psql -At -c "SELECT age(relfrozenxid) FROM pg_class WHERE relname = 'pgbench_branches'"
```

ΕΞΟΔΟΣ

```
29
number of transactions actually processed: 4000/4000
4029
```

Κάθε transaction του pgbench και κάθε άλλο transaction σε οποιονδήποτε πίνακα της βάσης, μεγάλωσε την ηλικία όλων των πινάκων κατά ένα. Ένα `VACUUM (FREEZE)` παγώνει όλες τις γραμμές του πίνακα και μηδενίζει σχεδόν την ηλικία του.

TERMINAL

```
psql -q -c "VACUUM (FREEZE) pgbench_branches"
psql -At -c "SELECT age(relfrozenxid) FROM pg_class WHERE relname = 'pgbench_branches'"
```

ΕΞΟΔΟΣ

```
0
```

Πρόοδος ενός VACUUM

Όσο τρέχει ένα `VACUUM`, είτε δικό σου είτε του autovacuum, το `pg_stat_progress_vacuum` δείχνει σε ποια φάση βρίσκεται και πόσες σελίδες έχει σαρώσει. Ένα `VACUUM` σε μεγάλο πίνακα χωρίς πρόοδο στο `heap_blks_scanned` για πολλή ώρα είναι σημάδι ότι κάτι το κρατά, συνήθως ένα lock.

SQL

```
SELECT pid, relid::regclass AS relation, phase,
       heap_blks_scanned, heap_blks_total
FROM pg_stat_progress_vacuum;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 pid | relation | phase | heap_blks_scanned | heap_blks_total
-----+----------+-------+-------------------+-----------------
(0 rows)
```

Εδώ είναι άδειο, αφού κανένα `VACUUM` δεν τρέχει αυτή τη στιγμή. Το ίδιο query σε έναν πολυάσχολο server δείχνει τα workers του autovacuum.

ΚΡΑΤΑ ΑΥΤΟ

Ο autovacuum τρέχει σε έναν πίνακα όταν οι νεκρές γραμμές ξεπεράσουν `threshold + scale_factor × γραμμές`. Για μεγάλους πίνακες και πίνακες με πολλά `UPDATE` χαμηλώνεις το `scale_factor` ανά πίνακα. Το `vacuum_cost_limit` ορίζει πόσο γρήγορα δουλεύει. Καμία νεκρή γραμμή δεν αφαιρείται όσο ένα παλαιότερο transaction είναι ανοιχτό, οπότε παρακολουθείς το `backend_xmin` και ορίζεις `idle_in_transaction_session_timeout`. Το freeze προστατεύει από το wraparound των xids. Το `age(datfrozenxid)` πρέπει να μένει πολύ κάτω από τα δύο δισεκατομμύρια.

Ασκήσεις

ΑΣΚΗΣΗ 10.1 Ποιοι πίνακες πλησιάζουν το όριο

Για τους πίνακες του `public` με τουλάχιστον μία νεκρή γραμμή, δείξε το `n_dead_tup` και το ποσοστό του ορίου vacuum που έχουν φτάσει, με ένα δεκαδικό, χρησιμοποιώντας τις ρυθμίσεις του server.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     relname      | n_dead_tup | pct_of_threshold
------------------+------------+------------------
 pgbench_tellers  |         65 |            108.3
 pgbench_accounts |       3876 |              3.9
(2 rows)
```

ΑΣΚΗΣΗ 10.2 Ρύθμιση για την ουρά

Φτιάξε πίνακα `job_queue (id bigint PRIMARY KEY, payload text)` με `autovacuum_vacuum_scale_factor` 0 και `autovacuum_vacuum_threshold` 500 κατευθείαν στο `CREATE TABLE`. Δείξε το `reloptions` του.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

                            reloptions
------------------------------------------------------------------
 {autovacuum_vacuum_scale_factor=0,autovacuum_vacuum_threshold=500}
(1 row)
```

ΑΣΚΗΣΗ 10.3 Η παλαιότερη σύνδεση σε transaction

Άνοιξε στο παρασκήνιο ένα `psql` που ξεκινά transaction, διαβάζει και περιμένει τρία δευτερόλεπτα. Όσο περιμένει, δείξε από άλλη σύνδεση την κατάσταση και την ηλικία του `backend_xmin` κάθε σύνδεσης που έχει.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 state  | xmin_age |       query
--------+----------+--------------------
 active |        3 | SELECT pg_sleep(3)
(1 row)
```

ΑΣΚΗΣΗ 10.4 Ηλικία ανά πίνακα

Δείξε τους πέντε πίνακες της βάσης με τη μεγαλύτερη ηλικία `relfrozenxid`, μόνο πίνακες και όχι indexes ή views, μαζί με το μέγεθός τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      table_name       | xid_age |  size
-----------------------+---------+--------
 pg_statistic_ext_data |    4113 | 16 kB
 pg_authid             |    4113 | 72 kB
 pg_foreign_table      |    4113 | 16 kB
 pg_type               |    4113 | 248 kB
 pg_user_mapping       |    4113 | 24 kB
(5 rows)
```

ΑΣΚΗΣΗ 10.5 Freeze και visibility map

Τρέξε `VACUUM (FREEZE, VERBOSE)` στον `pgbench_tellers` και εντόπισε στην έξοδο πόσες σελίδες σημειώθηκαν all frozen.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
INFO:  aggressively vacuuming "sqlbook.public.pgbench_tellers"
INFO:  finished vacuuming "sqlbook.public.pgbench_tellers": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total), 0 eagerly scanned
tuples: 0 removed, 50 remain, 0 are dead but not yet removable
removable cutoff: 4857, which was 0 XIDs old when operation ended
new relfrozenxid: 4857, which is 191 XIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 50 tuples frozen
visibility map: 0 pages set all-visible, 1 pages set all-frozen (1 were all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 23 hits, 0 reads, 0 dirtied
WAL usage: 3 records, 0 full page images, 526 bytes, 0 buffers full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

ΑΣΚΗΣΗ 10.6 Autovacuum ανά πίνακα

Δείξε για τους πίνακες του `pgbench` πόσες φορές έτρεξε σε καθέναν autovacuum και autoanalyze και πόσες γραμμές έχουν αλλάξει από το τελευταίο analyze.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     relname      | autovacuum_count | autoanalyze_count | n_mod_since_analyze
------------------+------------------+-------------------+---------------------
 pgbench_accounts |                1 |                 1 |                4000
 pgbench_branches |                0 |                 1 |                   0
 pgbench_history  |                1 |                 1 |                   0
 pgbench_tellers  |                1 |                 1 |                  50
(4 rows)
```

ΚΕΦΑΛΑΙΟ 11

Παρακολούθηση

Όταν μια εφαρμογή αργεί, η πρώτη ερώτηση είναι ποια queries αργούν και τι περιμένουν. Η PostgreSQL δίνει τις απαντήσεις από δύο πηγές. Το log καταγράφει γεγονότα, όπως ένα αργό query ή μια αναμονή για lock. Τα views στατιστικών μετρούν συνεχώς τι κάνει ο server. Σε αυτό το κεφάλαιο στήνουμε και τα δύο όπως θα τα στήναμε σε production.

Ο server του κεφαλαίου

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pgbench -i -s 5 -q >/dev/null 2>&1
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
```

Ένα log που βοηθά

Το log της PostgreSQL από προεπιλογή γράφει λίγα. Σφάλματα, checkpoints, την εκκίνηση και το σταμάτημα. Τέσσερις ρυθμίσεις το κάνουν χρήσιμο εργαλείο.

Ρύθμιση	Τιμή	Γιατί
`log_line_prefix`	`'%m [%p] %q%u@%d '`	ώρα, PID, χρήστης και βάση σε κάθε γραμμή
`log_min_duration_statement`	`'250ms'`	κάθε query που κράτησε περισσότερο
`log_lock_waits`	`on`	κάθε αναμονή για lock μεγαλύτερη από το `deadlock_timeout`
`log_temp_files`	`0`	κάθε προσωρινό αρχείο, όπως στο κεφάλαιο 8

Στο prefix το `%m` είναι η ώρα, το `%p` το PID, το `%u` ο χρήστης και το `%d` η βάση. Το `%q` λέει ότι ό,τι ακολουθεί γράφεται μόνο για γραμμές που αφορούν συνδέσεις, όχι για τις βοηθητικές διεργασίες.

SQL

```
ALTER SYSTEM SET log_line_prefix = '%m [%p] %q%u@%d ';
ALTER SYSTEM SET log_min_duration_statement = '250ms';
ALTER SYSTEM SET log_lock_waits = on;
ALTER SYSTEM SET log_temp_files = 0;
SELECT pg_reload_conf();
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SYSTEM

ALTER SYSTEM

ALTER SYSTEM

ALTER SYSTEM

 pg_reload_conf
----------------
 t
(1 row)
```

Ένα query που κρατά μισό δευτερόλεπτο γράφεται στο log με τη διάρκειά του.

TERMINAL

```
psql -q -c "SELECT pg_sleep(0.5)" >/dev/null
grep 'duration:' "$LAB/primary.log" | tail -n 1
```

ΕΞΟΔΟΣ

```
2026-09-28 09:47:39.821 EEST [77250] postgres@sqlbook LOG:  duration: 501.239 ms
statement: SELECT pg_sleep(0.5)
```

Το όριο των 250 ms είναι σημείο εκκίνησης. Σε μια εφαρμογή που απαντά σε milliseconds, ένα query του μισού δευτερολέπτου είναι ήδη πρόβλημα. Αν όμως το όριο είναι πολύ χαμηλό, το log γεμίζει και το ίδιο το logging κοστίζει.

Τώρα μια αναμονή για lock. Η συνεδρία A κρατά ένα row lock και η B περιμένει.

SQL · ΣΥΝΕΔΡΙΑ A

```
BEGIN;
UPDATE products SET stock = stock - 1 WHERE id = 3;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ A

```
BEGIN

UPDATE 1
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
UPDATE products SET stock = stock - 2 WHERE id = 3;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Β

```
-- η εντολή περιμένει
```

SQL · ΣΥΝΕΔΡΙΑ Α

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Α

```
COMMIT
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
UPDATE 1
```

TERMINAL

```
grep -A 2 'still waiting' "$LAB/primary.log" | tail -n 3
```

ΕΞΟΔΟΣ

```
2026-09-28 09:47:40.837 EEST [77261] postgres@sqlbook LOG:  process 77261 still waiting
for ShareLock on transaction 831 after 1001.107 ms
2026-09-28 09:47:40.837 EEST [77261] postgres@sqlbook DETAIL:  Process holding the lock:
77248. Wait queue: 77261.
2026-09-28 09:47:40.837 EEST [77261] postgres@sqlbook CONTEXT:  while updating tuple
(0,3) in relation "products"
```

Μετά από ένα δευτερόλεπτο αναμονής, όσο είναι το `deadlock_timeout`, ο server έγραψε ποια διεργασία περιμένει, για ποιο lock, ποια την εμποδίζει και ποιο ήταν το query. Αυτό είναι ό,τι χρειάζεσαι για να βρεις το transaction που κρατούσε το lock, ακόμη κι αν έχει κλείσει εδώ και ώρες.

auto_explain

Το log λέει ότι ένα query ήταν αργό, όχι γιατί. Η επέκταση `auto_explain` γράφει στο log το plan κάθε query που ξεπερνά ένα όριο, σαν να είχες τρέξει `EXPLAIN ANALYZE` εκείνη τη στιγμή. Είναι πολύτιμο για queries που είναι αργά μόνο μερικές φορές, με συγκεκριμένες τιμές ή σε ώρες φόρτου.

Φορτώνεται για μία σύνδεση με `LOAD` ή για όλες με τη ρύθμιση `session_preload_libraries`. Εδώ το δοκιμάζουμε σε μία σύνδεση, με όριο μηδέν ώστε να γραφτεί σίγουρα το plan του query μας.

PSQL

```
LOAD 'auto_explain';
SET auto_explain.log_min_duration = 0;
SET auto_explain.log_analyze = on;
SET auto_explain.log_timing = off;

SELECT count(*) FROM pgbench_accounts WHERE abalance = 0 AND bid = 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
LOAD
SET
SET
SET
 count
--------
 100000
(1 row)
```

TERMINAL

```
grep -A 12 'plan:' "$LAB/primary.log" | tail -n 12
```

ΕΞΟΔΟΣ

```
2026-09-28 09:47:42.053 EEST [77295] postgres@sqlbook LOG:  duration: 16.893 ms  plan:
    Query Text: SELECT count(*) FROM pgbench_accounts WHERE abalance = 0 AND bid = 3;
    Finalize Aggregate  (cost=12427.04..12427.05 rows=1 width=8) (actual rows=1.00 loops=1)
      ->  Gather  (cost=12426.83..12427.04 rows=2 width=8) (actual rows=3.00 loops=1)
            Workers Planned: 2
            Workers Launched: 2
            ->  Partial Aggregate  (cost=11426.83..11426.84 rows=1 width=8) (actual rows=1.00 loops=3)
                  ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..11322.00 rows=41930 width=0)
(actual rows=33333.33 loops=3)
                        Filter: ((abalance = 0) AND (bid = 3))
                        Rows Removed by Filter: 133333
```

Το `log_analyze` μετρά τις πραγματικές γραμμές κάθε κόμβου, όπως το `EXPLAIN ANALYZE`. Κοστίζει λίγο σε κάθε query, ακόμη και σε όσα δεν γράφονται τελικά στο log. Το `log_timing = off` κρατά αυτό το κόστος μικρό. Σε production το `auto_explain` με όριο μερικών δευτερολέπτων είναι φθηνή ασφάλεια.

pg_stat_statements

Το log βλέπει μεμονωμένα αργά queries. Δεν βλέπει ένα query που κρατά 2 ms αλλά εκτελείται εκατό χιλιάδες φορές την ώρα και που μπορεί να είναι το μεγαλύτερο κόστος του server. Το `pg_stat_statements` κρατά σύνολα για κάθε διαφορετικό query. Το είδαμε στο κεφάλαιο 36 του πρώτου τόμου. Εδώ το στήνουμε από την αρχή, όπως σε νέο server. Μετά το χρησιμοποιούμε σε πραγματικό φόρτο.

Η επέκταση χρειάζεται κοινή μνήμη που δεσμεύεται στην εκκίνηση, οπότε μπαίνει στο `shared_preload_libraries` και θέλει restart.

TERMINAL

```
psql -q -c "ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements'"
pglab restart primary
psql -q -c "CREATE EXTENSION pg_stat_statements"
pgbench -n -c 4 -t 1000 2>&1 | grep 'actually processed'
```

ΕΞΟΔΟΣ

```
primary: σε λειτουργία στη θύρα 5501
number of transactions actually processed: 4000/4000
```

Το pgbench έτρεξε 4.000 transactions από τέσσερις συνδέσεις. Ποια queries κόστισαν τον περισσότερο χρόνο συνολικά;

SQL

```
SELECT left(query, 60) AS query, calls,
       round(total_exec_time::numeric, 1) AS total_ms,
       round(mean_exec_time::numeric, 3) AS mean_ms,
       rows
FROM pg_stat_statements
WHERE dbid = (SELECT oid FROM pg_database WHERE datname = current_database())
ORDER BY total_exec_time DESC
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
                            query                             | calls | total_ms | mean_ms | rows
--------------------------------------------------------------+-------+----------+---------+------
 UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE a |  4000 |     51.3 |   0.013 | 4000
 UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE b |  4000 |     50.9 |   0.013 | 4000
 UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE ti |  4000 |     26.9 |   0.007 | 4000
 CREATE EXTENSION pg_stat_statements                          |     1 |     14.6 |  14.589 |    0
 SELECT abalance FROM pgbench_accounts WHERE aid = $1         |  4000 |      7.7 |   0.002 | 4000
(5 rows)
```

Οι τιμές μέσα στα queries έγιναν `$1`, `$2`. Η επέκταση κανονικοποιεί κάθε query αντικαθιστώντας τις σταθερές με παραμέτρους, οπότε τα 4.000 `UPDATE` με διαφορετικούς λογαριασμούς είναι μία γραμμή με `calls = 4000`. Οι χρόνοι στη δική σου εκτέλεση θα διαφέρουν. Πρόσεξε όμως το `UPDATE` του `pgbench_branches`. Αλλάζει έναν πίνακα πέντε γραμμών και κοστίζει όσο το `UPDATE` ενός πίνακα 500.000 γραμμών. Οι συνδέσεις περιμένουν η μία την άλλη για τα row locks των λίγων branches και ο χρόνος αναμονής μετράει στον χρόνο εκτέλεσης.

Οι στήλες `shared_blks_hit` και `shared_blks_read` δείχνουν ποια queries διαβάζουν πολλές σελίδες και το `temp_blks_written` ποια γράφουν προσωρινά αρχεία. Όταν αλλάζεις κάτι και θέλεις καθαρή μέτρηση, το `pg_stat_statements_reset()` μηδενίζει τα πάντα.

Τι περιμένουν οι συνδέσεις

Το `pg_stat_activity` έχει δύο στήλες που λένε τι περιμένει κάθε σύνδεση αυτή τη στιγμή, το `wait_event_type` και το `wait_event`. Όταν είναι κενές, η σύνδεση τρέχει στη CPU. Μια μόνο ματιά δεν λέει πολλά. Πολλές ματιές σε σειρά δίνουν μια εικόνα του πού πηγαίνει ο χρόνος. Φτιάχνουμε έναν πίνακα για δείγματα και παίρνουμε 30 δείγματα, ένα κάθε 100 ms, όσο τρέχει το pgbench με οκτώ συνδέσεις.

SQL

```
CREATE TABLE wait_samples (wait_type text, wait_event text);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE
```

TERMINAL

```
pgbench -n -c 8 -T 5 >/dev/null 2>&1 &
sleep 1
for i in $(seq 1 30); do
  psql -q -c "INSERT INTO wait_samples
              SELECT coalesce(wait_event_type, 'CPU'), coalesce(wait_event, 'running')
              FROM pg_stat_activity WHERE application_name = 'pgbench'"
  sleep 0.1
done
wait
```

SQL

```
SELECT wait_type, wait_event, count(*) AS samples,
       round(100.0 * count(*) / sum(count(*)) OVER (), 1) AS pct
FROM wait_samples
GROUP BY wait_type, wait_event
ORDER BY samples DESC
LIMIT 6;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 wait_type |  wait_event   | samples | pct
-----------+---------------+---------+------
 Client    | ClientRead    |     127 | 52.9
 CPU       | running       |      75 | 31.3
 Lock      | transactionid |      23 |  9.6
 IO        | WalWrite      |      11 |  4.6
 LWLock    | WALWrite      |       3 |  1.3
 LWLock    | LockManager   |       1 |  0.4
(6 rows)
```

Η εικόνα αλλάζει από εκτέλεση σε εκτέλεση, αλλά τα μεγάλα κομμάτια μένουν. Ο τύπος `Lock` με `transactionid` είναι συνδέσεις που περιμένουν ένα άλλο transaction να τελειώσει, εδώ για τις γραμμές του `pgbench_branches`. Ο τύπος `IO` με `WalSync` ή `WalWrite` είναι commits που περιμένουν το WAL. Ο τύπος `Client` με `ClientRead` είναι συνδέσεις που περιμένουν την εφαρμογή να στείλει την επόμενη εντολή. Αυτή η τεχνική λέγεται **wait event sampling** και είναι η βάση πολλών εργαλείων παρακολούθησης.

I/O ανά διεργασία

Το `pg_stat_io` μετρά αναγνώσεις, εγγραφές, hits και evictions ανά είδος διεργασίας και ανά είδος αντικειμένου. Απαντά σε ερωτήσεις όπως «ποιος γράφει σελίδες στον δίσκο, το checkpointer ή τα backends;». Αν τα backends γράφουν πολλές σελίδες μόνα τους, ο background writer και τα checkpoints δεν προλαβαίνουν.

SQL

```
SELECT backend_type, context,
       reads, writes, hits, evictions
FROM pg_stat_io
WHERE object = 'relation' AND (reads > 0 OR writes > 0)
ORDER BY backend_type, context;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    backend_type     |  context  | reads | writes |  hits   | evictions
---------------------+-----------+-------+--------+---------+-----------
 autovacuum launcher | normal    |     1 |      0 |       1 |         0
 background worker   | bulkread  |   986 |      0 |    2655 |         0
 checkpointer        | normal    |     ø |   3092 |       ø |         ø
 client backend      | bulkread  |  1193 |      0 |    2029 |         0
 client backend      | bulkwrite |     0 |   6212 |    8066 |         0
 client backend      | normal    |  9937 |      0 | 1644919 |         0
 client backend      | vacuum    |   394 |      0 |    2525 |         0
 standalone backend  | normal    |   479 |   1050 |   97339 |         0
 standalone backend  | vacuum    |     8 |      0 |     946 |         0
(9 rows)
```

Το `context` ξεχωρίζει το `normal` από το `bulkread` και το `bulkwrite`, που είναι οι ring buffers του κεφαλαίου 8. Υπάρχει και το `vacuum`. Από την έκδοση 18 το view έχει και στήλες σε bytes και μετρά και το WAL, με `object = 'wal'`.

Τι να παρακολουθείς

Ένα σύστημα παρακολούθησης, όπως το Prometheus με τον `postgres_exporter`, διαβάζει αυτά τα views κάθε λίγα δευτερόλεπτα. Ό,τι κι αν χρησιμοποιήσεις, οι ενδείξεις που αξίζουν alert είναι λίγες.

Ένδειξη	Πηγή	Πρόβλημα όταν
ηλικία xid	`age(datfrozenxid)`	ξεπερνά τα 500 εκατομμύρια
παλαιότερο transaction	`xact_start` και `backend_xmin` στο `pg_stat_activity`	ανοιχτό για λεπτά
συνδέσεις	πλήθος στο `pg_stat_activity`	πλησιάζει το `max_connections`
deadlocks και rollbacks	`pg_stat_database`	αυξάνονται απότομα
checkpoints	`pg_stat_checkpointer`	πολλά `num_requested`
lag replication	`pg_stat_replication`, κεφάλαιο 17	μεγαλώνει
χώρος δίσκου	το λειτουργικό	κάτω από 20% ελεύθερο

ΚΡΑΤΑ ΑΥΤΟ

Ρύθμισε το log με χρήσιμο `log_line_prefix`, `log_min_duration_statement`, `log_lock_waits` και `log_temp_files`. Το `auto_explain` γράφει το plan των αργών queries. Το `pg_stat_statements` χρειάζεται `shared_preload_libraries` και restart και δείχνει ποια κανονικοποιημένα queries κοστίζουν περισσότερο συνολικά. Τα `wait_event` του `pg_stat_activity`, σε δείγματα, δείχνουν τι περιμένουν οι συνδέσεις και το `pg_stat_io` ποιος διαβάζει και γράφει.

Ασκήσεις

ΑΣΚΗΣΗ 11.1 Τα πιο αργά κατά μέσο όρο

Δείξε από το `pg_stat_statements` τα τρία queries της βάσης με τον μεγαλύτερο μέσο χρόνο εκτέλεσης, από όσα εκτελέστηκαν τουλάχιστον 100 φορές, με τις κλήσεις και τον μέσο χρόνο σε ms με τρία δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
                            query                              | calls | mean_ms
---------------------------------------------------------------+-------+---------
 UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE b | 75044 |   0.063
 UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE ti | 75044 |   0.017
 UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE a | 75044 |   0.009
(3 rows)
```

ΑΣΚΗΣΗ 11.2 Ένα query, πολλές τιμές

Μηδένισε το `pg_stat_statements`, τρέξε δύο φορές το query που μετρά τους πελάτες μιας πόλης, μία για την πόλη 1 και μία για την πόλη 2. Δείξε μετά πώς καταγράφηκαν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 reset
-------
 t
(1 row)

 count
-------
     7
(1 row)

 count
-------
     6
(1 row)

                        query                        | calls
-----------------------------------------------------+-------
 SELECT count(*) FROM customers WHERE city_id = $1 |     2
(1 row)
```

ΑΣΚΗΣΗ 11.3 Μεγάλα sequential scans

Δείξε από το `pg_stat_user_tables` τους τρεις πίνακες με τις περισσότερες γραμμές ανά sequential scan κατά μέσο όρο, μαζί με το πλήθος των sequential scans και το σύνολο των γραμμών που διάβασαν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     relname      | seq_scan | seq_tup_read | rows_per_scan
------------------+----------+--------------+---------------
 pgbench_accounts |        5 |      1000000 |        200000
 pgbench_tellers  |    75045 |      3752250 |            50
 products         |        4 |           60 |            15
(3 rows)
```

ΑΣΚΗΣΗ 11.4 Πόσα αργά queries

Μέτρησε πόσες γραμμές του log του `primary` είναι για query που ξεπέρασε το `log_min_duration_statement` και δείξε τη μεγαλύτερη διάρκεια.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
3
duration: 2109.653 ms
```

ΑΣΚΗΣΗ 11.5 Υγεία της βάσης

Δείξε από το `pg_stat_database` για τη βάση `sqlbook` τα commits, τα rollbacks, τα deadlocks και το ποσοστό των rollbacks στο σύνολο των transactions με δύο δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 xact_commit | xact_rollback | deadlocks | rollback_pct
-------------+---------------+-----------+--------------
       75278 |             0 |         0 |         0.00
(1 row)
```

ΑΣΚΗΣΗ 11.6 Ποιος γράφει στον δίσκο

Άθροισε από το `pg_stat_io` τις εγγραφές σελίδων σχέσεων ανά είδος διεργασίας και δείξε μόνο όσα έγραψαν κάτι.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    backend_type    | writes
--------------------+--------
 client backend     |   6212
 checkpointer       |   3092
 standalone backend |   1050
(3 rows)
```

ΜΕΡΟΣ ΙΙΙ

Backup και ανάκτηση

Ένα backup αξίζει όσο το restore που έχεις δοκιμάσει. Σε αυτό το μέρος βλέπουμε τα δύο είδη backup της PostgreSQL, το logical με το `pg_dump` και το physical με αντίγραφο του data directory και του WAL. Με το δεύτερο φτάνουμε στην επαναφορά σε οποιοδήποτε δευτερόλεπτο του παρελθόντος.

ΚΕΦΑΛΑΙΟ 12

pg_dump και pg_restore

Το `pg_dump` εξάγει μια βάση ως εντολές που την ξαναφτιάχνουν. Δουλεύει όσο η βάση είναι σε λειτουργία, παράγει ένα αρχείο που διαβάζεται από οποιαδήποτε νεότερη έκδοση και επιτρέπει να επαναφέρεις ένα μόνο κομμάτι. Είναι το εργαλείο για μικρές βάσεις, για αντίγραφα σε άλλο μηχάνημα και για upgrades. Έχει όμως και όρια που πρέπει να ξέρεις πριν στηρίξεις σε αυτό την ασφάλεια των δεδομένων σου.

Ο server του κεφαλαίου

Προσθέτουμε έναν ρόλο εφαρμογής που έχει κάποια δικαιώματα και έναν πίνακα, για να δούμε τι γίνεται με αυτά στο restore.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
psql -q -c "CREATE ROLE app LOGIN" \
        -c "GRANT SELECT ON customers TO app" \
        -c "ALTER TABLE reviews OWNER TO app" \
        -c "ANALYZE"
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
```

Ένα dump

TERMINAL

```
pg_dump -Fc -f sqlbook.dump sqlbook
ls -l sqlbook.dump
```

ΕΞΟΔΟΣ

```
-rw-r--r--@ 1 you  staff  53070 Sep 28 09:47 sqlbook.dump
```

Το `-Fc` σημαίνει custom format, ένα συμπιεσμένο αρχείο που διαβάζει μόνο το `pg_restore`. Η βάση του βιβλίου χωρά σε λίγες δεκάδες KB. Το `pg_dump` δεν αντιγράφει αρχεία. Συνδέεται όπως κάθε client, ανοίγει ένα transaction σε `REPEATABLE READ` και διαβάζει τα πάντα από ένα snapshot. Όσο κι αν κρατήσει, το dump δείχνει τη βάση όπως ήταν τη στιγμή που ξεκίνησε. Οι αλλαγές που γίνονται στο μεταξύ δεν το επηρεάζουν.

Το `pg_restore -l` τυπώνει τον πίνακα περιεχομένων του αρχείου.

TERMINAL

```
pg_restore -l sqlbook.dump | sed -n '2,12p'
pg_restore -l sqlbook.dump | grep -E 'TABLE (DATA )?public (cities|orders) '
```

ΕΞΟΔΟΣ

```
; Archive created at 2026-09-28 09:47:50 EEST
;     dbname: sqlbook
;     TOC Entries: 74
;     Compression: gzip
;     Dump Version: 1.16-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 18.6 (Homebrew)
;     Dumped by pg_dump version: 18.6 (Homebrew)
;
219; 1259 16385 TABLE public cities postgres
228; 1259 16501 TABLE public orders postgres
3958; 0 16385 TABLE DATA public cities postgres
3967; 0 16501 TABLE DATA public orders postgres
```

Κάθε αντικείμενο είναι μια γραμμή. Ο πίνακας και τα δεδομένα του είναι χωριστές εγγραφές, το ίδιο και κάθε index, constraint και δικαίωμα. Αυτό επιτρέπει να επαναφέρεις επιλεκτικά.

Τρεις ενότητες

Ένα dump χωρίζεται σε τρεις ενότητες, που το `--section` επιλέγει ξεχωριστά.

Ενότητα	Περιέχει
`pre-data`	τύποι, πίνακες, sequences, functions, views
`data`	τα δεδομένα των πινάκων, με `COPY`
`post-data`	indexes, constraints, triggers, foreign keys

Η σειρά έχει λόγο. Φορτώνεις τα δεδομένα σε πίνακες χωρίς indexes, που είναι πολύ γρηγορότερο. Τα indexes τα φτιάχνεις μία φορά στο τέλος. Για τον ίδιο λόγο το restore μιας μεγάλης βάσης κρατά πολύ περισσότερο από το dump. Το dump διαβάζει δεδομένα, το restore τα γράφει και ξαναχτίζει κάθε index.

Το plain format, `-Fp` και προεπιλογή, γράφει απλό SQL. Είναι χρήσιμο για να δεις τι ακριβώς θα εκτελεστεί. Εδώ το σχήμα ενός πίνακα, χωρίς σχόλια και κενές γραμμές.

TERMINAL

```
pg_dump --schema-only -t cities sqlbook | grep -v -E '^(--|$|SET|SELECT)'
```

ΕΞΟΔΟΣ

```
\restrict 1Fe9nL7rJnRfHbLCVjtaq4zXD8dWWQ1L2ZUwAc2KwIwAnwwnzolglESVpdZMluy
CREATE TABLE public.cities (
    id integer NOT NULL,
    name text NOT NULL,
    region text NOT NULL,
    population integer,
    CONSTRAINT cities_population_check CHECK ((population > 0))
);
ALTER TABLE public.cities OWNER TO postgres;
ALTER TABLE ONLY public.cities
    ADD CONSTRAINT cities_name_key UNIQUE (name);
ALTER TABLE ONLY public.cities
    ADD CONSTRAINT cities_pkey PRIMARY KEY (id);
\unrestrict 1Fe9nL7rJnRfHbLCVjtaq4zXD8dWWQ1L2ZUwAc2KwIwAnwwnzolglESVpdZMluy
```

Οι γραμμές `\restrict` και `\unrestrict` προστατεύουν το psql από ένα dump που προσπαθεί να εκτελέσει εντολές του psql. Το κλειδί τους είναι τυχαίο σε κάθε dump.

Restore σε άλλο server

Στήνουμε δεύτερο server με όνομα `restore` και επαναφέρουμε το dump εκεί. Το `--create` φτιάχνει τη βάση με το όνομα και τις ρυθμίσεις της αρχικής. Γι' αυτό συνδεόμαστε αρχικά στη βάση `postgres`.

TERMINAL

```
pglab init restore 5502
pglab start restore
pg_restore -p 5502 --create -d postgres sqlbook.dump
```

ΕΞΟΔΟΣ

```
restore: νέο cluster στο $LAB/restore, θύρα 5502
restore: σε λειτουργία στη θύρα 5502
pg_restore: error: could not execute query: ERROR:  role "app" does not exist
Command was: ALTER TABLE public.reviews OWNER TO app;

pg_restore: error: could not execute query: ERROR:  role "app" does not exist
Command was: GRANT SELECT ON TABLE public.customers TO app;


pg_restore: warning: errors ignored on restore: 2
```

Τα δεδομένα επανήλθαν, με δύο σφάλματα. Ο ρόλος `app` δεν υπάρχει στον νέο server. Οι ρόλοι ανήκουν σε όλο το cluster και όχι σε μια βάση, οπότε το `pg_dump` δεν τους περιέχει. Το ίδιο ισχύει για τα tablespaces και για τις ρυθμίσεις του `ALTER ROLE ... SET`. Αυτά τα εξάγει το `pg_dumpall` με την επιλογή `--globals-only`.

TERMINAL

```
pg_dumpall --globals-only -f globals.sql
grep -E '^(CREATE|ALTER) ROLE' globals.sql
```

ΕΞΟΔΟΣ

```
CREATE ROLE app;
ALTER ROLE app WITH NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB LOGIN NOREPLICATION NOBYPASSRLS;
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN REPLICATION BYPASSRLS;
```

Η σωστή σειρά είναι πρώτα τα globals και μετά οι βάσεις. Ξαναστήνουμε τον `restore` από την αρχή και επαναφέρουμε με αυτή τη σειρά.

TERMINAL

```
pglab rm restore
pglab init restore 5502
pglab start restore
psql -q -p 5502 -d postgres -f globals.sql 2>&1 | grep ERROR
pg_restore -p 5502 --create -d postgres sqlbook.dump
psql -p 5502 -c "SELECT count(*) AS orders FROM orders"
```

ΕΞΟΔΟΣ

```
restore: σβήστηκε
restore: νέο cluster στο $LAB/restore, θύρα 5502
restore: σε λειτουργία στη θύρα 5502
psql:globals.sql:18: ERROR:  role "postgres" already exists
 orders
--------
    162
(1 row)
```

Το μόνο σφάλμα είναι ότι ο ρόλος `postgres` υπάρχει ήδη, κάτι αναμενόμενο και ακίνδυνο. Το restore τελείωσε χωρίς σφάλματα.

ΠΑΓΙΔΑ

Ένα backup με `pg_dump` χωρίς τα globals δεν επαναφέρεται σωστά σε νέο server. Οι ιδιοκτήτες και τα δικαιώματα χάνονται και η εφαρμογή δεν μπορεί να συνδεθεί. Κράτα πάντα και ένα `pg_dumpall --globals-only` δίπλα στα dumps.

Τα statistics δεν έρχονται μαζί

Η βάση στον `restore` έχει τα ίδια δεδομένα, αλλά ο planner δεν ξέρει τίποτα γι' αυτά.

TERMINAL

```
psql -At -p 5501 -c "SELECT count(*) FROM pg_stats WHERE schemaname = 'public'"
psql -At -p 5502 -c "SELECT count(*) FROM pg_stats WHERE schemaname = 'public'"
```

ΕΞΟΔΟΣ

```
71
0
```

Μέχρι να τρέξει ο autovacuum ένα `ANALYZE` σε κάθε πίνακα, τα plans βασίζονται σε εκτιμήσεις χωρίς δεδομένα. Μετά από κάθε restore τρέχεις `ANALYZE` αμέσως, ή, από την έκδοση 18, κάνεις το dump με `--statistics`, οπότε τα statistics επαναφέρονται μαζί με τα δεδομένα.

TERMINAL

```
pg_dump -Fc --statistics -f sqlbook_stats.dump sqlbook
pg_restore -l sqlbook_stats.dump | grep -c 'STATISTICS DATA'
```

ΕΞΟΔΟΣ

```
30
```

Επαναφορά ενός κομματιού

Το πιο συχνό restore δεν είναι ολόκληρη η βάση. Είναι «σβήσαμε κατά λάθος τις κριτικές, φέρε τες πίσω». Το `-t` επιλέγει πίνακες. Επαναφέρουμε τον `products` σε μια νέα άδεια βάση.

TERMINAL

```
createdb -p 5502 partial
pg_restore -p 5502 -d partial -t products sqlbook.dump
psql -p 5502 -d partial -At -c "SELECT count(*) FROM products"
psql -p 5502 -d partial -c "\d products" | tail -n 5
```

ΕΞΟΔΟΣ

```
30
Check constraints:
    "products_cost_check" CHECK (cost >= 0::numeric)
    "products_price_check" CHECK (price > 0::numeric)
    "products_stock_check" CHECK (stock >= 0)
```

Ο πίνακας ήρθε με τα δεδομένα και τα `CHECK` του, αλλά χωρίς primary key και χωρίς το identity. Αυτά είναι χωριστά αντικείμενα στον πίνακα περιεχομένων και το `-t` δεν τα φέρνει. Για πλήρη έλεγχο φτιάχνεις λίστα με το `-l`, κρατάς ή σχολιάζεις τις γραμμές που θέλεις και τη δίνεις στο `-L`.

TERMINAL

```
pg_restore -l sqlbook.dump | grep -E ' public reviews' | grep -v 'FK CONSTRAINT' > reviews.list
cat reviews.list
dropdb -p 5502 partial
createdb -p 5502 partial
pg_restore -p 5502 -d partial -L reviews.list --no-owner sqlbook.dump
psql -p 5502 -d partial -c "\d reviews" | grep -A 1 Indexes
```

ΕΞΟΔΟΣ

```
234; 1259 16582 TABLE public reviews app
233; 1259 16581 SEQUENCE public reviews_id_seq app
3973; 0 16582 TABLE DATA public reviews app
3988; 0 0 SEQUENCE SET public reviews_id_seq app
3793; 2606 16596 CONSTRAINT public reviews reviews_pkey app
Indexes:
    "reviews_pkey" PRIMARY KEY, btree (id)
```

Τώρα ήρθαν και το primary key και το sequence. Τα foreign keys τα αφήσαμε έξω από τη λίστα, αφού οι πίνακες `customers` και `products` δεν υπάρχουν σε αυτή τη βάση. Το `--no-owner` αφήνει ως ιδιοκτήτη τον χρήστη που κάνει το restore, αντί για τον `app` του αρχικού server.

Parallel dump και restore

Για μεγάλες βάσεις το directory format, `-Fd`, γράφει κάθε πίνακα σε χωριστό αρχείο. Το `-j` δουλεύει με πολλές συνδέσεις ταυτόχρονα. Το ίδιο `-j` επιταχύνει και το restore, που είναι συνήθως το πιο αργό από τα δύο.

TERMINAL

```
pg_dump -Fd -j 4 -f sqlbook.dir sqlbook
ls sqlbook.dir | head -n 5
pg_restore -p 5502 -j 4 --create -d postgres --no-owner --clean --if-exists sqlbook.dir
```

ΕΞΟΔΟΣ

```
3958.dat.gz
3960.dat.gz
3961.dat.gz
3963.dat.gz
3964.dat.gz
```

Το `--clean --if-exists` σβήνει πρώτα τη βάση αν υπάρχει. Το αρχείο `toc.dat` είναι ο πίνακας περιεχομένων και κάθε `.dat.gz` τα δεδομένα ενός πίνακα.

Όρια του pg_dump

Το `pg_dump` είναι σωστό εργαλείο για βάσεις μερικών GB, για αντιγραφή σε περιβάλλον δοκιμών και για upgrade σε νέα major έκδοση. Τρία πράγματα όμως το κάνουν ανεπαρκές ως μοναδικό backup μιας μεγάλης βάσης.

1. Επαναφέρει μόνο τη στιγμή του dump. Αν κάνεις dump κάθε βράδυ και η καταστροφή γίνει το απόγευμα, χάνεις τη δουλειά μιας μέρας.
2. Το restore ξαναχτίζει κάθε index, οπότε για μια βάση εκατοντάδων GB κρατά ώρες.
3. Όσο τρέχει, κρατά ένα `ACCESS SHARE` lock σε κάθε πίνακα. Ένα `ALTER TABLE` εκείνη την ώρα περιμένει και όπως είδαμε στον πρώτο τόμο, πίσω του περιμένουν όλοι.

Τα δύο πρώτα λύνονται με το physical backup του επόμενου κεφαλαίου.

ΣΗΜΕΙΩΣΗ

Σε upgrade χρησιμοποιείς το `pg_dump` της νέας έκδοσης για να διαβάσεις τον παλιό server. Κάθε `pg_dump` διαβάζει servers παλαιότερων εκδόσεων και γράφει dump που καταλαβαίνει η δική του έκδοση.

ΚΡΑΤΑ ΑΥΤΟ

Το `pg_dump` εξάγει μια βάση από ένα συνεπές snapshot χωρίς να τη σταματήσει. Το custom και το directory format διαβάζονται με `pg_restore`, που επαναφέρει όλα ή μέρος τους και δουλεύει παράλληλα με `-j`. Ρόλοι και tablespaces θέλουν `pg_dumpall --globals-only`. Τα statistics χρειάζονται `--statistics` στο dump ή `ANALYZE` μετά. Το `-t` φέρνει μόνο πίνακα και δεδομένα, για τα υπόλοιπα χρησιμοποιείς λίστα με `-l` και `-L`.

Ασκήσεις

ΑΣΚΗΣΗ 12.1 Το σχήμα μιας βάσης

Γράψε μόνο το σχήμα όλης της βάσης `sqlbook` σε αρχείο `schema.sql` και μέτρησε πόσες εντολές `CREATE TABLE` περιέχει και σε πόσες γραμμές εμφανίζεται `PRIMARY KEY`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
12
12
```

ΑΣΚΗΣΗ 12.2 Πόσα δεδομένα

Μέτρησε από τον πίνακα περιεχομένων του `sqlbook.dump` πόσες εγγραφές είναι δεδομένα πινάκων, πόσες είναι constraints και πόσες foreign keys.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
12
18
15
```

ΑΣΚΗΣΗ 12.3 Πληρωμές ως INSERT

Εξήγαγε μόνο τα δεδομένα του πίνακα `payments` ως εντολές `INSERT`, μία ανά γραμμή. Δείξε τις τρεις πρώτες.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
INSERT INTO public.payments VALUES (1, 1, 289.00, 'card', '2025-01-17 20:56:00+02');
INSERT INTO public.payments VALUES (2, 2, 38.00, 'paypal', '2025-01-19 19:04:00+02');
INSERT INTO public.payments VALUES (3, 3, 160.00, 'card', '2025-01-28 16:56:00+02');
```

ΑΣΚΗΣΗ 12.4 Συμπίεση

Φτιάξε dump custom format χωρίς συμπίεση και με συμπίεση `zstd` και σύγκρινε τα μεγέθη με το αρχικό `sqlbook.dump`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
182794 plain.dump
53070 sqlbook.dump
55717 zstd.dump
```

ΑΣΚΗΣΗ 12.5 Μία βάση με άλλο όνομα

Επανάφερε το `sqlbook.dump` στον `restore` σε νέα βάση με όνομα `sqlbook_copy`, χωρίς ιδιοκτήτες. Μέτρησε μετά τους πελάτες της.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
30
```

ΑΣΚΗΣΗ 12.6 Μόνο τα δικαιώματα

Δείξε από τον πίνακα περιεχομένων του `sqlbook.dump` τις εγγραφές δικαιωμάτων και ιδιοκτησίας που αφορούν τον ρόλο `app`, με `pg_restore -l` και `grep`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
3982; 0 0 ACL public TABLE customers postgres
234; 1259 16582 TABLE public reviews app
233; 1259 16581 SEQUENCE public reviews_id_seq app
3973; 0 16582 TABLE DATA public reviews app
3988; 0 0 SEQUENCE SET public reviews_id_seq app
3793; 2606 16596 CONSTRAINT public reviews reviews_pkey app
3808; 2606 16602 FK CONSTRAINT public reviews reviews_customer_id_fkey app
3809; 2606 16597 FK CONSTRAINT public reviews reviews_product_id_fkey app
```

ΚΕΦΑΛΑΙΟ 13

Physical backup και WAL archiving

Ένα physical backup είναι αντίγραφο των αρχείων του data directory. Δεν ξαναφτιάχνει τίποτα στο restore, οπότε επαναφέρει μια βάση εκατοντάδων GB στον χρόνο που χρειάζεται για να αντιγραφούν τα αρχεία. Μόνο του όμως δεν αρκεί. Χρειάζεται και το WAL και όταν κρατάς όλο το WAL, το backup παύει να είναι μια φωτογραφία και γίνεται μια ταινία.

Ο server του κεφαλαίου

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pgbench -i -s 5 -q >/dev/null 2>&1
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
```

Γιατί όχι απλώς cp

Ένα `cp -r` του data directory ενώ ο server δουλεύει δεν δίνει χρήσιμο αντίγραφο. Η αντιγραφή κρατά λίγα λεπτά και στο μεταξύ ο server γράφει σε αρχεία που έχουν ήδη αντιγραφεί και σε άλλα που δεν έχουν. Το αποτέλεσμα είναι αρχεία από διαφορετικές στιγμές, κάτι σαν σελίδες που κόπηκαν στη μέση.

Η λύση είναι το WAL. Αν ξέρεις από ποιο σημείο του WAL ξεκίνησε η αντιγραφή και κρατήσεις όλο το WAL ως το τέλος της, ένα recovery ξαναπαίζει κάθε αλλαγή και φέρνει όλα τα αρχεία στην ίδια στιγμή. Είναι ακριβώς το recovery μετά από πτώση του κεφαλαίου 6, πάνω σε αντίγραφο. Το `pg_basebackup` κάνει όλη αυτή τη δουλειά σωστά.

pg_basebackup

Το `pg_basebackup` συνδέεται στον server με το **replication protocol**, το ίδιο που θα χρησιμοποιεί ο standby του μέρους IV. Ο server ξεκινά ένα checkpoint, σημειώνει το LSN και στέλνει όλα τα αρχεία του, μαζί με το WAL που γράφεται όσο διαρκεί η αντιγραφή.

TERMINAL

```
pg_basebackup -D "$LAB/backup1" -X stream -c fast
ls "$LAB/backup1"
```

ΕΞΟΔΟΣ

```
backup_label
backup_manifest
base
global
pg_commit_ts
pg_dynshmem
pg_hba.conf
pg_ident.conf
pg_logical
pg_multixact
pg_notify
pg_replslot
pg_serial
pg_snapshots
pg_stat
pg_stat_tmp
pg_subtrans
pg_tblspc
pg_twophase
PG_VERSION
pg_wal
pg_xact
postgresql.auto.conf
postgresql.conf
```

Το `-X stream` στέλνει το WAL παράλληλα με τα αρχεία, από δεύτερη σύνδεση, ώστε το backup να περιέχει όλο το WAL που χρειάζεται. Το `-c fast` ζητά άμεσο checkpoint αντί να περιμένει το επόμενο. Με `-P` θα τύπωνε και την πρόοδο, κάτι χρήσιμο σε μεγάλες βάσεις.

Ο φάκελος μοιάζει με data directory, με δύο επιπλέον αρχεία. Το `backup_label` λέει από πού πρέπει να ξεκινήσει το recovery.

TERMINAL

```
cat "$LAB/backup1/backup_label"
```

ΕΞΟΔΟΣ

```
START WAL LOCATION: 0/6000028 (file 000000010000000000000006)
CHECKPOINT LOCATION: 0/6000080
BACKUP METHOD: streamed
BACKUP FROM: primary
START TIME: 2026-09-28 09:47:57 EEST
LABEL: pg_basebackup base backup
START TIMELINE: 1
```

Το `backup_manifest` είναι ένα JSON με κάθε αρχείο του backup, το μέγεθος και ένα checksum του. Το εργαλείο `pg_verifybackup` το χρησιμοποιεί για να ελέγξει ότι τίποτα δεν χάθηκε ή δεν αλλοιώθηκε.

TERMINAL

```
pg_verifybackup "$LAB/backup1"
```

ΕΞΟΔΟΣ

```
backup successfully verified
```

Για να τρέξει το `pg_basebackup` χρειάζονται τρία πράγματα, που η PostgreSQL έχει από προεπιλογή. Το `wal_level` πρέπει να είναι `replica`, να υπάρχει διαθέσιμος WAL sender στο `max_wal_senders` και μια γραμμή `replication` στο `pg_hba.conf` για τον χρήστη. Ο χρήστης θέλει το attribute `REPLICATION`. Ο `postgres` το έχει ως superuser. Σε production φτιάχνεις ξεχωριστό ρόλο για τα backups, όπως θα δούμε σε άσκηση.

Restore από το backup

Το restore ενός physical backup είναι απλό. Ο φάκελος είναι ήδη ένα data directory. Του δίνουμε άλλη θύρα, για να τρέξει δίπλα στον primary. Μετά τον ξεκινάμε.

TERMINAL

```
echo "port = 5502" >> "$LAB/backup1/postgresql.conf"
mv "$LAB/backup1" "$LAB/restored"
pglab start restored
grep -E 'backup recovery|consistent|redo done|ready' "$LAB/restored.log"
```

ΕΞΟΔΟΣ

```
restored: σε λειτουργία στη θύρα 5502
2026-09-28 09:47:58.360 EEST [77990] LOG:  starting backup recovery with redo LSN
0/6000028, checkpoint LSN 0/6000080, on timeline ID 1
2026-09-28 09:47:58.362 EEST [77990] LOG:  completed backup recovery with redo LSN
0/6000028 and end LSN 0/6000120
2026-09-28 09:47:58.362 EEST [77990] LOG:  consistent recovery state reached at 0/6000120
2026-09-28 09:47:58.362 EEST [77990] LOG:  redo done at 0/6000120 system usage: CPU:
user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
2026-09-28 09:47:58.371 EEST [77984] LOG:  database system is ready to accept connections
```

Ο server βρήκε το `backup_label`, ξεκίνησε recovery από το LSN του και σταμάτησε μόλις έφτασε σε **consistent** κατάσταση, δηλαδή στο σημείο όπου τελείωσε η αντιγραφή. Από εκεί και πέρα είναι ένας κανονικός server με τα δεδομένα του primary τη στιγμή του backup.

TERMINAL

```
psql -p 5502 -c "SELECT count(*) AS accounts FROM pgbench_accounts"
pglab rm restored
```

ΕΞΟΔΟΣ

```
 accounts
----------
   500000
(1 row)

restored: σβήστηκε
```

ΣΗΜΕΙΩΣΗ

Ένα physical backup είναι αντίγραφο ολόκληρου του cluster, όλων των βάσεων και των ρόλων μαζί. Δεν επαναφέρει έναν πίνακα ή μία βάση. Και επαναφέρεται μόνο σε server της ίδιας major έκδοσης, στην ίδια αρχιτεκτονική. Για τα υπόλοιπα υπάρχει το `pg_dump`.

WAL archiving

Το backup μας δείχνει τον primary τη στιγμή που τελείωσε η αντιγραφή. Αν κρατάμε και όλα τα segments του WAL που γράφονται από εκεί και πέρα, μπορούμε να ξεκινήσουμε από το backup και να ξαναπαίξουμε όσο WAL θέλουμε, ως οποιοδήποτε σημείο. Αυτό λέγεται **continuous archiving**. Κάθε φορά που γεμίζει ένα segment, ο server εκτελεί το `archive_command` για να το αντιγράψει κάπου ασφαλές.

TERMINAL

```
mkdir -p "$LAB/archive"
psql -q -c "ALTER SYSTEM SET archive_mode = on" \
        -c "ALTER SYSTEM SET archive_command = 'test ! -f $LAB/archive/%f && cp %p $LAB/archive/%f'"
pglab restart primary
```

ΕΞΟΔΟΣ

```
primary: σε λειτουργία στη θύρα 5501
```

Στο `archive_command` το `%p` είναι η διαδρομή του segment και το `%f` το όνομά του. Η εντολή πρέπει να επιστρέφει επιτυχία μόνο αν το αρχείο αποθηκεύτηκε και να μην αντικαθιστά αρχείο που υπάρχει ήδη. Το `test ! -f` το εξασφαλίζει. Το `archive_mode` θέλει restart, το `archive_command` αλλάζει με reload.

Ένα segment αρχειοθετείται όταν γεμίσει. Το `pg_switch_wal()` κλείνει το τρέχον segment αμέσως, κάτι χρήσιμο για δοκιμές.

TERMINAL

```
pgbench -n -t 2000 -c 2 >/dev/null 2>&1
psql -q -c "SELECT pg_switch_wal()"
sleep 2
ls "$LAB/archive"
```

ΕΞΟΔΟΣ

```
 pg_switch_wal
---------------
 0/93DFCD0
(1 row)

000000010000000000000007
000000010000000000000008
000000010000000000000009
```

Το view `pg_stat_archiver` λέει τι αρχειοθετήθηκε και αν απέτυχε κάτι.

SQL

```
SELECT archived_count, last_archived_wal, failed_count, last_failed_wal
FROM pg_stat_archiver;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 archived_count |    last_archived_wal     | failed_count | last_failed_wal
----------------+--------------------------+--------------+-----------------
              3 | 000000010000000000000009 |            0 | ∅
(1 row)
```

Όταν το archiving αποτυγχάνει

Τι γίνεται αν ο προορισμός δεν είναι διαθέσιμος, για παράδειγμα γέμισε ο δίσκος ή έπεσε το δίκτυο; Προσομοιώνουμε ένα archive όπου δεν μπορούμε να γράψουμε.

TERMINAL

```
chmod 500 "$LAB/archive"
for i in 1 2 3; do psql -q -c "SELECT pg_switch_wal()" >/dev/null; pgbench -n -t 500 >/dev/null 2>&1; done
sleep 3
grep 'archive command failed' "$LAB/primary.log" | tail -n 1
ls "$LAB/primary/pg_wal/archive_status" | tail -n 3
```

ΕΞΟΔΟΣ

```
2026-09-28 09:48:04.890 EEST [78034] LOG:  archive command failed with exit code 1
000000010000000000000009.done
00000001000000000000000A.ready
00000001000000000000000B.ready
```

Το `archive_command` αποτυγχάνει και ο server το ξαναδοκιμάζει ξανά και ξανά. Στο `archive_status` κάθε segment που περιμένει έχει ένα αρχείο `.ready`. Και το πιο σημαντικό, ο server **δεν σβήνει** κανένα segment που δεν αρχειοθετήθηκε. Το `pg_wal` μεγαλώνει χωρίς όριο, όσο κι αν είναι το `max_wal_size`, μέχρι να γεμίσει ο δίσκος και να σταματήσει ο server.

SQL

```
SELECT archived_count, failed_count, last_failed_wal
FROM pg_stat_archiver;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 archived_count | failed_count |     last_failed_wal
----------------+--------------+--------------------------
              3 |            5 | 00000001000000000000000A
(1 row)
```

Μόλις ο προορισμός γίνει ξανά διαθέσιμος, ο server αρχειοθετεί όσα περιμένουν, με τη σειρά.

TERMINAL

```
chmod 700 "$LAB/archive"
sleep 3
ls "$LAB/primary/pg_wal/archive_status" | grep -c ready || true
ls "$LAB/archive" | wc -l
```

ΕΞΟΔΟΣ

```
0
       5
```

ΠΑΓΙΔΑ

Ένα `failed_count` που μεγαλώνει είναι επείγον alert. Όσο αποτυγχάνει το archiving, το `pg_wal` γεμίζει και δεν έχεις backup για ό,τι γράφεται. Το αντίθετο λάθος είναι εξίσου κακό. Ένα `archive_command` που επιστρέφει επιτυχία χωρίς να αποθηκεύσει, όπως ένα σκέτο `true`, κάνει τον server να σβήνει WAL που δεν σώθηκε ποτέ.

Πόσο WAL μπορείς να χάσεις

Ένα segment αρχειοθετείται όταν γεμίσει τα 16 MB. Σε μια ήσυχη βάση αυτό μπορεί να πάρει ώρες και αν χαθεί ο server, χάνεται μαζί και ό,τι βρίσκεται στο μισογεμάτο segment. Το `archive_timeout` κλείνει το segment μετά από όσα δευτερόλεπτα ορίσεις, ακόμη κι αν δεν γέμισε. Με `archive_timeout = 60` χάνεις το πολύ ένα λεπτό δεδομένων. Αυτό το μέγιστο που μπορείς να χάσεις λέγεται **RPO**, recovery point objective. Στο μέρος IV θα δούμε πώς η replication το κατεβάζει σχεδόν στο μηδέν.

ΣΗΜΕΙΩΣΗ

Σε production δεν γράφεις `archive_command` με `cp`. Εργαλεία όπως τα pgBackRest, Barman και WAL-G κάνουν backup και archiving μαζί, με συμπίεση, κρυπτογράφηση, παράλληλη μεταφορά, αποθήκευση σε S3 και κανόνες διατήρησης. Χρησιμοποιούν τους ίδιους μηχανισμούς που είδαμε, οπότε ό,τι έμαθες εδώ ισχύει και εκεί.

ΚΡΑΤΑ ΑΥΤΟ

Ένα physical backup είναι αντίγραφο του data directory και του WAL που γράφτηκε όσο αντιγραφόταν. Το `pg_basebackup` το κάνει με το replication protocol και γράφει `backup_label` και `backup_manifest`, που ελέγχει το `pg_verifybackup`. Το restore είναι εκκίνηση server πάνω στο αντίγραφο. Με `archive_mode` και `archive_command` κρατάς κάθε segment του WAL. Ένα archiving που αποτυγχάνει γεμίζει το `pg_wal`, οπότε το `pg_stat_archiver` θέλει alert. Το `archive_timeout` ορίζει πόσα δεδομένα μπορείς να χάσεις το πολύ.

Ασκήσεις

ΑΣΚΗΣΗ 13.1 Backup σε tar

Πάρε base backup του `primary` σε μορφή tar με συμπίεση gzip στον φάκελο `backup_tar` και δείξε τα αρχεία που δημιουργήθηκαν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
backup_manifest
base.tar.gz
pg_wal.tar.gz
```

ΑΣΚΗΣΗ 13.2 Ρόλος για backups

Φτιάξε ρόλο `backup` με `LOGIN` και `REPLICATION`, χωρίς να είναι superuser. Πάρε με αυτόν base backup στον φάκελο `backup_role`. Δείξε τη γραμμή `LABEL` του `backup_label`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
LABEL: nightly
```

ΑΣΚΗΣΗ 13.3 Αλλοιωμένο backup

Πρόσθεσε ένα byte στο τέλος του αρχείου `PG_VERSION` του `backup_role` και έλεγξε το backup με `pg_verifybackup`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
pg_verifybackup: error: "PG_VERSION" has size 4 on disk but size 3 in the manifest
```

ΑΣΚΗΣΗ 13.4 Κατάσταση του archiving

Κλείσε το τρέχον segment με `pg_switch_wal`, περίμενε δύο δευτερόλεπτα και δείξε από το `pg_stat_archiver` αν το τελευταίο αρχειοθετημένο segment είναι αυτό που μόλις έκλεισε.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 archived | failed_count
----------+--------------
 t        |            5
(1 row)
```

ΑΣΚΗΣΗ 13.5 Το archive και το pg_wal

Μέτρησε πόσα αρχεία segments υπάρχουν στο archive και πόσα στο `pg_wal` του `primary`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
      12
8
```

ΑΣΚΗΣΗ 13.6 Χρόνος ανάμεσα σε segments

Όρισε `archive_timeout` σε 60 δευτερόλεπτα με `ALTER SYSTEM`, κάνε reload και δείξε την τιμή και το context της ρύθμισης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 pg_reload_conf
----------------
 t
(1 row)

      name       | setting | unit | context
-----------------+---------+------+---------
 archive_timeout | 60      | s    | sighup
(1 row)
```

ΚΕΦΑΛΑΙΟ 14

Point in time recovery

Κάποιος τρέχει ένα `DELETE` χωρίς `WHERE` στις τρεις το μεσημέρι. Το backup της προηγούμενης νύχτας θα έφερνε πίσω τις πληρωμές, αλλά θα έσβηνε όλη τη δουλειά του πρωινού. Με ένα base backup και το archive του WAL μπορείς να επαναφέρεις τη βάση όπως ήταν στις 14:59:59. Αυτό είναι το point in time recovery και είναι ο λόγος που κρατάμε το WAL.

Ο server του κεφαλαίου

Στήνουμε archiving όπως στο προηγούμενο κεφάλαιο και παίρνουμε base backup. Αυτή τη φορά γράφουμε τις ρυθμίσεις στο τέλος του `postgresql.conf` και όχι με `ALTER SYSTEM`. Οι servers του recovery θα είναι αντίγραφα αυτού του φακέλου και θα αλλάζουμε τις ρυθμίσεις τους με νέες γραμμές στο τέλος του ίδιου αρχείου. Το `postgresql.auto.conf` διαβάζεται μετά και θα τις ακύρωνε.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pgbench -i -s 5 -q >/dev/null 2>&1
mkdir -p "$LAB/archive"
cat >> "$LAB/primary/postgresql.conf" <<EOF
archive_mode = on
archive_command = 'test ! -f $LAB/archive/%f && cp %p $LAB/archive/%f'
EOF
pglab restart primary
pg_basebackup -D "$LAB/base" -X stream -c fast
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
primary: σε λειτουργία στη θύρα 5501
```

Το heredoc χωρίς εισαγωγικά, `<<EOF`, αντικαθιστά το `$LAB` με τη διαδρομή του εργαστηρίου πριν γράψει, ενώ το `%f` και το `%p` μένουν όπως είναι για τον server.

Το λάθος

Μετά το backup η εφαρμογή δουλεύει κανονικά για λίγο. Ο pgbench παίζει τον ρόλο της κανονικής κίνησης. Κρατάμε την ώρα λίγο πριν από το λάθος σε μια μεταβλητή του shell. Σε ένα πραγματικό περιστατικό θα τη βρίσκαμε στο log ή θα τη μαθαίναμε από όποιον έκανε το λάθος.

TERMINAL

```
pgbench -n -t 1000 >/dev/null 2>&1
psql -At -c "SELECT count(*) FROM payments"
sleep 1
before_mistake=$(psql -At -c "SELECT now()")
echo "$before_mistake"
sleep 1
psql -c "DELETE FROM payments"
```

ΕΞΟΔΟΣ

```
155
2026-09-28 09:48:19.399261+03
DELETE 155
```

Και η δουλειά συνεχίζεται, γιατί κανείς δεν το πρόσεξε αμέσως.

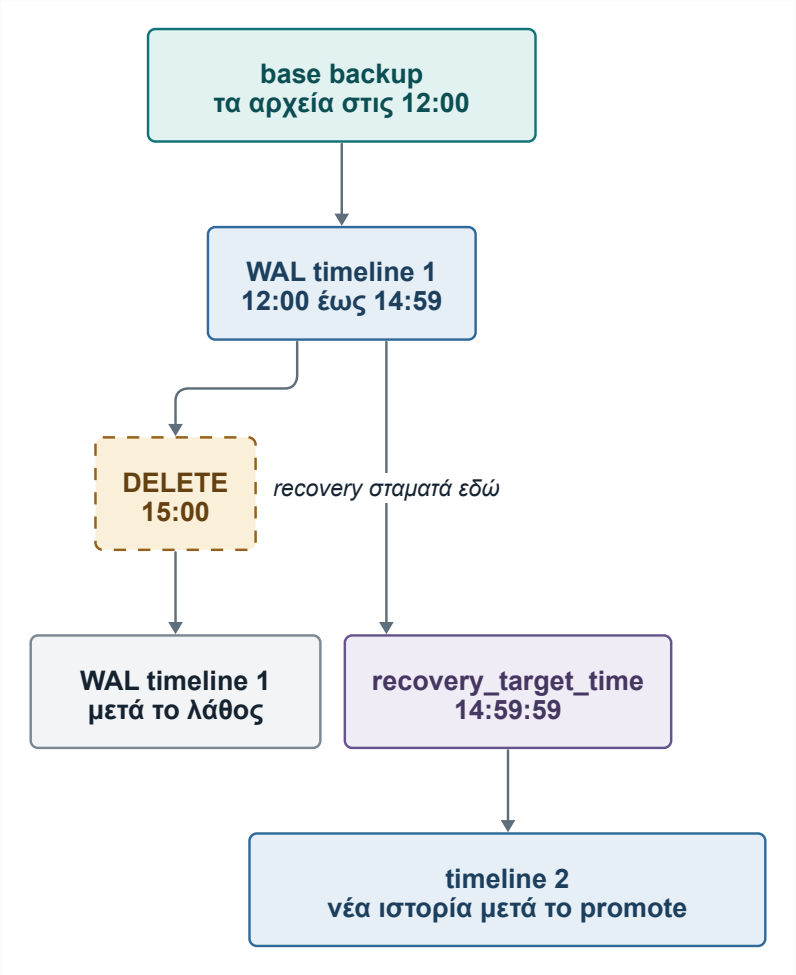
TERMINAL

```
pgbench -n -t 1000 >/dev/null 2>&1
psql -q -c "SELECT pg_switch_wal()"
sleep 2
```

ΕΞΟΔΟΣ

```
 pg_switch_wal
---------------
 0/8626FC0
(1 row)
```

Το `pg_switch_wal` εξασφαλίζει ότι το segment με το λάθος είναι στο archive. Σε πραγματικό server αυτό το κάνει το `archive_timeout` ή η συνεχής κίνηση.



Σχήμα 14.1 · Το recovery ξεκινά από το base backup, ξαναπαίζει το WAL ως τη στιγμή πριν από το λάθος και μετά το promote συνεχίζει σε νέο timeline.

Recovery μέχρι μια στιγμή

Δεν αγγίζουμε τον primary. Φτιάχνουμε έναν νέο server από το base backup στη θύρα 5502 και του λέμε τρία πράγματα. Από πού θα βρίσκει τα segments του WAL, ως ποια στιγμή θα κάνει recovery και τι θα κάνει όταν φτάσει εκεί.

TERMINAL

```
cp -Rp "$LAB/base" "$LAB/pitr"
cat >> "$LAB/pitr/postgresql.conf" <<EOF
port = 5502
archive_mode = off
restore_command = 'cp $LAB/archive/%f %p'
recovery_target_time = '$before_mistake'
recovery_target_action = 'pause'
EOF
touch "$LAB/pitr/recovery.signal"
pglab start pitr
```

ΕΞΟΔΟΣ

```
pitr: σε λειτουργία στη θύρα 5502
```

Το `cp -Rp` κρατά τα δικαιώματα των αρχείων. Ο server αρνείται να ξεκινήσει αν το data directory είναι ανοιχτό σε άλλους χρήστες. Το `restore_command` είναι το αντίστροφο του `archive_command`, ο server το εκτελεί για κάθε segment που χρειάζεται, με το `%f` για το όνομα και το `%p` για τη διαδρομή όπου πρέπει να το αντιγράψει. Το αρχείο `recovery.signal` λέει στον server ότι πρέπει να κάνει **targeted recovery** και όχι να ξεκινήσει κανονικά. Το `archive_mode = off` εμποδίζει τον νέο server να γράφει στο ίδιο archive με τον primary όσο κάνουμε δοκιμές. Χωρίς αυτό, ο `pitr` μετά το recovery θα έγραφε δικά του αρχεία στο archive του primary. Ένα επόμενο recovery από το ίδιο archive θα μπορούσε να ακολουθήσει τη δική του ιστορία αντί για του primary.

TERMINAL

```
grep -E 'point-in-time|restored log file|recovery stopping|pausing' "$LAB/pitr.log"
```

ΕΞΟΔΟΣ

```
2026-09-28 09:48:23.327 EEST [78545] LOG:  restored log file "000000010000000000000006" from archive
2026-09-28 09:48:23.328 EEST [78545] LOG:  starting point-in-time recovery to 2026-09-28 09:48:19.399261+03
2026-09-28 09:48:23.355 EEST [78545] LOG:  restored log file "000000010000000000000007" from archive
2026-09-28 09:48:23.372 EEST [78545] LOG:  recovery stopping before commit of transaction 1831, time
2026-09-28 09:48:20.459659+03
2026-09-28 09:48:23.372 EEST [78545] LOG:  pausing at the end of recovery
```

Ο server πήρε από το archive όσα segments χρειαζόταν, ξαναέπαιξε το WAL και σταμάτησε **πριν** από το commit του πρώτου transaction μετά την ώρα που του δώσαμε. Αυτό το transaction είναι το `DELETE`. Τώρα είναι σε παύση, σε λειτουργία μόνο για ανάγνωση. Ελέγχουμε.

SQL · ΘΥΡΑ 5502

```
SELECT pg_is_in_recovery() AS in_recovery,
       pg_get_wal_replay_pause_state() AS replay,
       (SELECT count(*) FROM payments) AS payments;
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
 in_recovery | replay | payments
-------------+--------+----------
 t           | paused |      155
(1 row)
```

Οι 155 πληρωμές είναι εκεί. Αν η στιγμή δεν ήταν σωστή, θα μπορούσαμε να σβήσουμε τον `pitr` και να δοκιμάσουμε με άλλη ώρα, χωρίς κανένα ρίσκο για τον primary. Ικανοποιημένοι, τελειώνουμε το recovery με `pg_promote()`, που κάνει τον server κανονικό, με δυνατότητα εγγραφής.

SQL · ΘΥΡΑ 5502

```
SELECT pg_promote();
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
 pg_promote
------------
 t
(1 row)
```

TERMINAL

```
sleep 1
grep -E 'selected new timeline|archive recovery complete|ready to accept' "$LAB/pitr.log"
```

ΕΞΟΔΟΣ

```
2026-09-28 09:48:23.357 EEST [78539] LOG:  database system is ready to accept read-only connections
2026-09-28 09:48:24.399 EEST [78545] LOG:  selected new timeline ID: 2
2026-09-28 09:48:24.435 EEST [78545] LOG:  archive recovery complete
2026-09-28 09:48:24.455 EEST [78539] LOG:  database system is ready to accept connections
```

Timelines

Ο `pitr` ξεκίνησε από το ίδιο παρελθόν με τον primary αλλά από το σημείο του recovery και μετά έχει άλλη ιστορία. Ο primary έσβησε τις πληρωμές, ο `pitr` όχι. Αν και οι δύο έγραφαν WAL με τα ίδια ονόματα αρχείων, θα μπλέκονταν στο archive. Γι' αυτό κάθε promote ξεκινά νέο **timeline**. Τα ονόματα των segments αρχίζουν με τον αριθμό του timeline και ένα αρχείο `.history` καταγράφει από ποιο σημείο διακλαδώθηκε.

TERMINAL

```
ls "$LAB/pitr/pg_wal" | grep -v archive_status
cat "$LAB/pitr/pg_wal/00000002.history"
```

ΕΞΟΔΟΣ

```
000000010000000000000006
000000010000000000000007
00000002.history
000000020000000000000007
000000020000000000000008
summaries
1   0/7CF4700   before 2026-09-28 09:48:20.459659+03
```

Το history λέει ότι το timeline 2 ξεκίνησε από το timeline 1 σε ένα LSN και γράφει και τον λόγο, το target του recovery. Αν κάποτε χρειαστεί recovery μέσα από το timeline 2, ο server θα ακολουθήσει τον σωστό δρόμο μέσα από το archive. Τα timelines θα ξαναβρούμε στο failover του κεφαλαίου 19.

Άλλοι τρόποι να ορίσεις τη στιγμή

Η ώρα δεν είναι πάντα ο καλύτερος στόχος. Αν το λάθος και άλλα transactions έγιναν στο ίδιο δευτερόλεπτο, θέλεις μεγαλύτερη ακρίβεια.

Ρύθμιση	Σταματά
`recovery_target_time`	στο πρώτο commit μετά από αυτή την ώρα
`recovery_target_xid`	στο commit ενός συγκεκριμένου transaction
`recovery_target_lsn`	σε ένα LSN
`recovery_target_name`	σε ένα restore point με όνομα
`recovery_target = 'immediate'`	μόλις το backup γίνει consistent

Το `recovery_target_inclusive`, αληθές από προεπιλογή για xid και lsn, ορίζει αν το ίδιο το target εφαρμόζεται ή σταματάμε ακριβώς πριν. Για ώρα, η παύση γίνεται πριν από το πρώτο commit που είναι αυστηρά μετά από αυτή.

Το πιο χρήσιμο στην πράξη είναι το restore point. Πριν από κάθε επικίνδυνη δουλειά, ένα migration ή ένα μαζικό `UPDATE`, φτιάχνεις ένα σημείο με όνομα.

SQL

```
SELECT pg_create_restore_point('before_price_update');
```

ΑΠΟΤΕΛΕΣΜΑ

```
 pg_create_restore_point
-------------------------
 0/9000090
(1 row)
```

Αν κάτι πάει στραβά, κάνεις recovery με `recovery_target_name = 'before_price_update'` και ξέρεις ακριβώς πού θα σταματήσει. Θα το κάνουμε σε άσκηση.

> **ΠΑΓΙΔΑ**
> Το point in time recovery φέρνει πίσω **όλο** το cluster σε μια στιγμή. Χάνονται και οι σωστές αλλαγές που έγιναν μετά, όπως τα 1.000 transactions του pgbench μετά το λάθος μας. Γι' αυτό συχνά δεν αντικαθιστάς τον primary με τον server του recovery. Τον κρατάς δίπλα, αντιγράφεις από εκεί μόνο τις γραμμές που χάθηκαν και τις ξαναβάζεις στον primary.

Αυτό κάνουμε εδώ. Ο primary έχει όλα τα σωστά transactions και καμία πληρωμή. Ο `pitr` έχει τις πληρωμές. Με ένα `pg_dump` μόνο των δεδομένων του πίνακα τις μεταφέρουμε.

TERMINAL

```
pg_dump -p 5502 --data-only -t payments sqlbook | psql -q -p 5501 >/dev/null
psql -At -p 5501 -c "SELECT count(*) FROM payments"
```

ΕΞΟΔΟΣ

```
155
```

ΚΡΑΤΑ ΑΥΤΟ

Base backup και archive του WAL επαναφέρουν το cluster σε οποιαδήποτε στιγμή μετά το backup. Αντιγράφεις το backup, ορίζεις `restore_command` και ένα `recovery_target_*`, φτιάχνεις `recovery.signal` και ξεκινάς. Με `recovery_target_action = 'pause'` ελέγχεις το αποτέλεσμα πριν από το `pg_promote()`. Κάθε promote ξεκινά νέο timeline. Φτιάχνε restore point πριν από επικίνδυνες αλλαγές. Συχνά είναι καλύτερο να φέρεις από τον server του recovery μόνο ό,τι χάθηκε.

Ασκήσεις

ΑΣΚΗΣΗ 14.1 Recovery σε restore point

Άλλαξε στον primary όλες τις τιμές των προϊόντων στο διπλάσιο, κλείσε το segment και φτιάξε από το `base` νέο server `rp` στη θύρα 5503 με recovery ως το restore point `before_price_update` και promote στο τέλος. Δείξε την τιμή του προϊόντος 1 στον primary και στον `rp`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 pg_switch_wal
---------------
 0/9016E60
(1 row)

rp: σε λειτουργία στη θύρα 5503
33.80
16.90
```

ΑΣΚΗΣΗ 14.2 Πού σταμάτησε

Δείξε από το log του `rp` τη γραμμή όπου σταμάτησε το recovery και τη γραμμή με το νέο timeline.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
2026-09-28 09:48:28.734 EEST [78645] LOG:  recovery stopping at restore point
"before_price_update", time 2026-09-28 09:48:25.569355+03
2026-09-28 09:48:28.742 EEST [78645] LOG:  selected new timeline ID: 2
```

ΑΣΚΗΣΗ 14.3 Στοιχεία του pg_control

Με το `pg_controldata` δείξε για τον `rp` την κατάσταση του cluster και το timeline του τελευταίου checkpoint.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Database cluster state:               in production
Latest checkpoint's TimeLineID:       2
```

ΑΣΚΗΣΗ 14.4 Recovery ως το τέλος του backup

Φτιάξε από το `base` server `early` στη θύρα 5504 με `recovery_target = 'immediate'` και promote και μέτρησε τις γραμμές του `pgbench_history`, όπου κάθε transaction του pgbench γράφει μία γραμμή.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
early: σε λειτουργία στη θύρα 5504
0
2000
```

ΑΣΚΗΣΗ 14.5 Ο pitr μετά το promote

Στον `pitr`, που είναι πλέον κανονικός server, δείξε αν είναι σε recovery και ποια είναι η τρέχουσα θέση του στο WAL μαζί με το όνομα του segment.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 in_recovery |    lsn    |         segment
-------------+-----------+--------------------------
 f           | 0/7D32000 | 000000020000000000000007
(1 row)
```

ΚΕΦΑΛΑΙΟ 15

Incremental backups και έλεγχος των backups

Ένα πλήρες backup κάθε νύχτα αντιγράφει ξανά και ξανά τα ίδια δεδομένα, αφού τα περισσότερα δεν αλλάζουν από μέρα σε μέρα. Από την έκδοση 17 η PostgreSQL παίρνει incremental backups, που περιέχουν μόνο όσες σελίδες άλλαξαν. Στο δεύτερο μισό του κεφαλαίου απαντάμε στην πιο σημαντική ερώτηση του μέρους. Πώς ξέρεις ότι ένα backup θα δουλέψει τη μέρα που θα το χρειαστείς;

Ο server του κεφαλαίου

Θέλουμε λίγο μεγαλύτερη βάση, ώστε οι διαφορές στα μεγέθη να φαίνονται. Ο pgbench με `-s 10` φτιάχνει ένα εκατομμύριο λογαριασμούς.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pgbench -i -s 10 -q >/dev/null 2>&1
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
```

Περίληψη του WAL

Για να ξέρει ποιες σελίδες άλλαξαν από το προηγούμενο backup, ο server χρειάζεται μια περίληψη του WAL. Η διεργασία **WAL summarizer** διαβάζει το WAL καθώς γράφεται και κρατά, για κάθε κομμάτι του, ποιες σελίδες ποιων αρχείων άλλαξαν. Οι περιλήψεις μπαίνουν στο `pg_wal/summaries` που είδαμε στο κεφάλαιο 9. Ενεργοποιείται με τη ρύθμιση `summarize_wal`, με reload.

SQL

```
ALTER SYSTEM SET summarize_wal = on;
SELECT pg_reload_conf();
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SYSTEM

 pg_reload_conf
----------------
 t
(1 row)
```

SQL

```
SELECT summarized_tli, summarized_lsn, pending_lsn
FROM pg_get_wal_summarizer_state();
```

ΑΠΟΤΕΛΕΣΜΑ

```
 summarized_tli | summarized_lsn | pending_lsn
----------------+----------------+-------------
              1 | 0/178D0A0      | 0/97D4638
(1 row)
```

Ο summarizer διαβάζει το WAL και γράφει περιλήψεις κατά διαστήματα. Το `pending_lsn` είναι ως πού έχει διαβάσει και το `summarized_lsn` ως πού έχει γράψει περιλήψεις. Μόλις τον ενεργοποιήσαμε, οπότε δουλεύει ακόμη πάνω στο WAL της φόρτωσης. Δεν χρειάζεται να τον περιμένεις. Ένα incremental backup περιμένει μόνο του να φτάσουν οι περιλήψεις ως το σημείο που χρειάζεται.

Πλήρες και incremental backup

Πρώτα ένα πλήρες backup, όπως στο κεφάλαιο 13.

TERMINAL

```
pg_basebackup -D "$LAB/full" -c fast
```

Η εφαρμογή δουλεύει για λίγο. Μετά παίρνουμε incremental backup. Η επιλογή `--incremental` δέχεται το `backup_manifest` του προηγούμενου backup. Από εκεί ο server μαθαίνει το LSN του και στέλνει μόνο τις σελίδες που άλλαξαν από τότε.

TERMINAL

```
pgbench -n -t 2000 >/dev/null 2>&1
pg_basebackup -D "$LAB/incr1" -c fast --incremental="$LAB/full/backup_manifest"
du -sh "$LAB/full" "$LAB/incr1"
```

ΕΞΟΔΟΣ

```
198M    ~/pglab/full
 50M    ~/pglab/incr1
```

Το incremental είναι ένα τέταρτο του πλήρους. Το μεγαλύτερο μέρος του είναι το WAL της αντιγραφής και οι σελίδες του `pgbench_accounts` που άγγιξαν τα 2.000 transactions, σκόρπιες σε όλο τον πίνακα. Σε

μια πραγματική βάση, όπου μεγάλο μέρος των δεδομένων είναι ιστορικό που δεν αλλάζει, η διαφορά είναι πολύ μεγαλύτερη.

Τα αρχεία που άλλαξαν μερικώς αποθηκεύονται με πρόθεμα `INCREMENTAL`. Κάθε τέτοιο αρχείο κρατά μόνο τις σελίδες που άλλαξαν και το μέγεθος του αρχικού.

TERMINAL

```
find "$LAB/incr1" -name 'INCREMENTAL.*' | wc -l
```

ΕΞΟΔΟΣ

```
     927
```

Ένα incremental μπορεί να βασίζεται σε άλλο incremental. Έτσι φτιάχνεται μια αλυσίδα, για παράδειγμα ένα πλήρες την Κυριακή και ένα incremental κάθε άλλη μέρα πάνω στο προηγούμενο.

TERMINAL

```
pgbench -n -t 1000 >/dev/null 2>&1
pg_basebackup -D "$LAB/incr2" -c fast --incremental="$LAB/incr1/backup_manifest"
du -sh "$LAB/incr2"
```

ΕΞΟΔΟΣ

```
 37M    ~/pglab/incr2
```

pg_combinebackup

Ένα incremental backup δεν ξεκινά μόνο του. Το `pg_combinebackup` ενώνει την αλυσίδα, από το πλήρες ως το τελευταίο incremental με τη σειρά, σε ένα πλήρες backup έτοιμο για restore.

TERMINAL

```
pg_combinebackup "$LAB/full" "$LAB/incr1" "$LAB/incr2" -o "$LAB/combined"
pg_verifybackup "$LAB/combined"
du -sh "$LAB/combined"
```

ΕΞΟΔΟΣ

```
backup successfully verified
198M    ~/pglab/combined
```

Το αποτέλεσμα έχει το ίδιο μέγεθος με ένα πλήρες backup και δικό του manifest, που ελέγχει το `pg_verifybackup`. Το ξεκινάμε όπως κάθε restore.

TERMINAL

```
echo "port = 5502" >> "$LAB/combined/postgresql.conf"
pglab start combined
psql -At -p 5501 -c "SELECT count(*) FROM pgbench_history"
psql -At -p 5502 -c "SELECT count(*) FROM pgbench_history"
```

ΕΞΟΔΟΣ

```
combined: σε λειτουργία στη θύρα 5502
3000
3000
```

Τα 3.000 transactions των δύο γύρων του pgbench είναι όλα εκεί. Η αλυσίδα έχει ένα ρίσκο. Αν χαθεί ή αλλοιωθεί ένας κρίκος, όλα τα επόμενα incrementals είναι άχρηστα. Γι' αυτό οι αλυσίδες είναι κοντές, με πλήρες backup κάθε λίγες μέρες.

Έλεγχος των backups

Ένα backup μπορεί να αποτύχει με πολλούς τρόπους χωρίς να το μάθεις. Ένα αρχείο δεν μεταφέρθηκε ολόκληρο, ο δίσκος του backup αλλοίωσε μια σελίδα, το archive έχει ένα κενό, ή τα δεδομένα ήταν ήδη χαλασμένα στον primary. Κάθε επίπεδο θέλει δικό του έλεγχο.

Έλεγχος	Εργαλείο	Τι πιάνει
αρχεία του backup	`pg_verifybackup`	αρχεία που λείπουν ή άλλαξαν, WAL που λείπει
checksums σελίδων	`pg_checksums --check`	αλλοιωμένες σελίδες, σε σταματημένο cluster
δομή πινάκων και indexes	`pg_amcheck`	σπασμένα indexes και heap, σε server που τρέχει
ότι επαναφέρεται	ένα πραγματικό restore	όλα τα παραπάνω και όσα δεν σκέφτηκες

Το `pg_amcheck` χρησιμοποιεί την επέκταση `amcheck` για να ελέγξει ότι κάθε B-tree είναι σωστά ταξινομημένο και ότι το heap έχει έγκυρες γραμμές. Το `--install-missing` εγκαθιστά την επέκταση αν λείπει. Όταν όλα είναι σωστά, δεν τυπώνει τίποτα.

TERMINAL

```
pg_amcheck -p 5502 --install-missing sqlbook
echo "exit code: $?"
```

ΕΞΟΔΟΣ

```
exit code: 0
```

Το `pg_checksums` διαβάζει κάθε σελίδα κάθε αρχείου και ελέγχει το checksum της. Θέλει τον server σταματημένο.

TERMINAL

```
pglab stop combined
pg_checksums --check -D "$LAB/combined"
```

ΕΞΟΔΟΣ

```
combined: σταμάτησε
Checksum operation completed
Files scanned:   1324
Blocks scanned:  23263
Bad checksums:  0
Data checksum version: 1
```

ΠΑΓΙΔΑ

Ο μόνος έλεγχος που αποδεικνύει ότι ένα backup επαναφέρεται είναι ένα restore. Βάλε ένα script να επαναφέρει αυτόματα το τελευταίο backup σε ένα δοκιμαστικό μηχάνημα κάθε εβδομάδα, να ξεκινά τον server, να τρέχει `pg_amcheck` και ένα query που ελέγχει ότι υπάρχουν πρόσφατα δεδομένα. Μέτρα και πόσο κράτησε. Αυτός ο χρόνος είναι το πραγματικό σου RTO.

Μια στρατηγική

Δύο νούμερα ορίζουν τι χρειάζεσαι. Το **RPO** είναι πόσα δεδομένα μπορείς να χάσεις. Το **RTO** είναι πόσο χρόνο μπορείς να μείνεις εκτός λειτουργίας. Με όσα είδαμε ως τώρα, μια λογική στρατηγική για μια βάση λίγων εκατοντάδων GB είναι η εξής.

1. Συνεχές archiving του WAL με `archive_timeout` ενός λεπτού. Το RPO είναι ένα λεπτό.
2. Πλήρες backup κάθε Κυριακή και incremental κάθε άλλη νύχτα.
3. Τα backups και το archive σε άλλο μηχάνημα και σε άλλη τοποθεσία από τον primary. Ένα backup στον ίδιο δίσκο χάνεται μαζί του.
4. Διατήρηση για δύο εβδομάδες και ό,τι WAL χρειάζεται για recovery από το πιο παλιό backup.
5. Αυτόματο restore και έλεγχος κάθε εβδομάδα.

Ένα `pg_dump` της βάσης κάθε νύχτα δίπλα σε αυτά δεν κάνει κακό. Είναι ο πιο εύκολος τρόπος να φέρεις πίσω έναν πίνακα και διαβάζεται από κάθε έκδοση. Όσο για το RTO, ένα restore από backup εκατοντάδων GB κρατά ώρες. Αν χρειάζεσαι λεπτά, χρειάζεσαι ήδη έναν server που περιμένει έτοιμος. Αυτό είναι το θέμα του επόμενου μέρους.

ΚΡΑΤΑ ΑΥΤΟ

Με `summarize_wal` ο server κρατά περιλήψεις του WAL και το `pg_basebackup --incremental` στέλνει μόνο τις σελίδες που άλλαξαν από ένα προηγούμενο backup. Το `pg_combinebackup` ενώνει την αλυσίδα σε πλήρες backup. Το `pg_verifybackup` ελέγχει τα αρχεία, το `pg_checksums` τις σελίδες και το `pg_amcheck` τη δομή. Μόνο ένα πραγματικό restore αποδεικνύει ότι το backup δουλεύει. Ο χρόνος του είναι το RTO σου.

Ασκήσεις

ΑΣΚΗΣΗ 15.1 Διαθέσιμες περιλήψεις

Δείξε πόσες περιλήψεις WAL υπάρχουν και από ποιο ως ποιο LSN καλύπτουν συνολικά, από το `pg_available_wal_summaries()`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 summaries | from_lsn  |  to_lsn
-----------+-----------+-----------
         9 | 0/1000028 | 0/F000028
(1 row)
```

ΑΣΚΗΣΗ 15.2 Πόσο χώρο γλιτώνεις

Δείξε σε KB το μέγεθος του `full`, του `incr1` και του `incr2`. Υπολόγισε μετά με το `awk` τι ποσοστό του πλήρους είναι τα δύο incrementals μαζί.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
202264
51348
38368
44%
```

ΑΣΚΗΣΗ 15.3 Αλυσίδα χωρίς αρχή

Δοκίμασε να ενώσεις με `pg_combinebackup` μόνο το `incr1` και το `incr2`, χωρίς το πλήρες, σε φάκελο `broken`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
pg_combinebackup: error: backup at "~/pglab/incr1" is an incremental backup, but the
first backup should be a full backup
```

ΑΣΚΗΣΗ 15.4 Έλεγχος του primary

Τρέξε `pg_amcheck` σε όλες τις βάσεις του `primary`, με έλεγχο και των γραμμών του heap στα δεδομένα των indexes. Δείξε τον κωδικό εξόδου.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
exit code: 0
```

ΑΣΚΗΣΗ 15.5 Checksums στο πλήρες backup

Δοκίμασε να ελέγξεις τα checksums του φακέλου `full` με `pg_checksums --check` και δείξε με `pg_controldata` την κατάσταση που γράφει το `pg_control` του.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Database cluster state:               in production
pg_checksums: error: cluster must be shut down
```

ΜΕΡΟΣ IV

Replication και διαθεσιμότητα

Ένα backup σε φέρνει πίσω μετά από ώρες. Ένας standby που ακολουθεί τον primary σε φέρνει πίσω σε δευτερόλεπτα. Εδώ στήνουμε streaming replication, βλέπουμε τι την καθυστερεί και τι την εμποδίζει, κάνουμε failover και φτάνουμε στη logical replication, που αντιγράφει μεμονωμένους πίνακες ανάμεσα σε διαφορετικούς servers.

ΚΕΦΑΛΑΙΟ 16

Streaming replication

Ο standby είναι ένας server σε διαρκές recovery. Αντί να διαβάζει segments από ένα archive, λαμβάνει το WAL του primary μέσα από μια σύνδεση καθώς γράφεται και το ξαναπαίζει αμέσως. Σε κανονικές συνθήκες βρίσκεται κλάσματα του δευτερολέπτου πίσω από τον primary και δέχεται queries ανάγνωσης.

Ο server του κεφαλαίου

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pgbench -i -s 5 -q >/dev/null 2>&1
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
```

Ένας standby σε δύο εντολές

Ένας standby ξεκινά από ένα base backup του primary. Με την επιλογή `-R`, το `pg_basebackup` γράφει στο backup και όσα χρειάζεται για να γίνει standby.

TERMINAL

```
pg_basebackup -D "$LAB/standby" -R -X stream -c fast
ls "$LAB/standby" | grep signal
grep primary_conninfo "$LAB/standby/postgresql.auto.conf" | tr -d "'" | tr ' ' '\n' |
  grep -E '^(user|host|port|dbname)='
```

ΕΞΟΔΟΣ

```
standby.signal
user=postgres
host=localhost
port=5501
dbname=sqlbook
```

Το αρχείο `standby.signal` λέει στον server να μείνει σε recovery για πάντα, σε αντίθεση με το `recovery.signal` του προηγούμενου κεφαλαίου, που σταματά σε έναν στόχο. Το `primary_conninfo` στο

`postgresql.auto.conf` λέει πού θα βρει τον primary. Είναι ένα connection string όπως αυτά που δίνεις στο `psql`. Το `pg_basebackup` γράφει εκεί κάθε παράμετρο σύνδεσης και τις περισσότερες με τις προεπιλογές τους, οπότε το `tr` και το `grep` κρατούν μόνο τις τέσσερις που μας ενδιαφέρουν. Δίνουμε στον standby άλλη θύρα και τον ξεκινάμε.

TERMINAL

```
echo "port = 5502" >> "$LAB/standby/postgresql.conf"
pglab start standby
grep -E 'entering standby|started streaming|ready to accept' "$LAB/standby.log"
```

ΕΞΟΔΟΣ

```
standby: σε λειτουργία στη θύρα 5502
2026-09-28 09:48:44.991 EEST [79133] LOG:  entering standby mode
2026-09-28 09:48:44.993 EEST [79127] LOG:  database system is ready to accept read-only connections
2026-09-28 09:48:45.034 EEST [79134] LOG:  started streaming WAL from primary at 0/7000000 on timeline 1
```

Ο standby ξεκίνησε recovery, έφτασε σε consistent κατάσταση, άνοιξε για συνδέσεις μόνο ανάγνωσης και άρχισε να λαμβάνει WAL από τον primary.

Τι βλέπει ο primary

Για κάθε standby ο primary έχει μια διεργασία **walsender**, που του στέλνει το WAL. Το view `pg_stat_replication` έχει μία γραμμή ανά walsender.

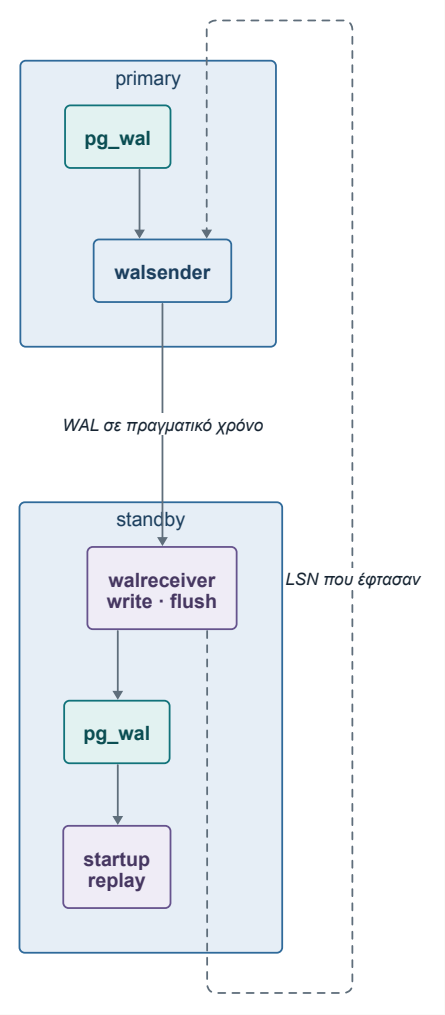
SQL

```
SELECT application_name, state, sync_state,
       sent_lsn, write_lsn, flush_lsn, replay_lsn
FROM pg_stat_replication;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 application_name |   state   | sync_state | sent_lsn  | write_lsn | flush_lsn | replay_lsn
------------------+-----------+------------+-----------+-----------+-----------+------------
 walreceiver      | streaming | async      | 0/7000000 | 0/7000000 | 0/7000000 | 0/7000000
(1 row)
```

Τα τέσσερα LSN είναι τα στάδια του ταξιδιού του WAL. Το `sent` είναι ό,τι έστειλε ο primary, το `write` ό,τι έγραψε ο standby στο λειτουργικό, το `flush` ό,τι έφτασε στον δίσκο του και το `replay` ό,τι ξαναπαίχτηκε και είναι ορατό σε queries. Τώρα που δεν γίνεται τίποτα, είναι όλα ίδια. Το `async` στο `sync_state` σημαίνει ότι ο primary δεν περιμένει τον standby στο commit. Αυτό αλλάζει στο κεφάλαιο 18.



Σχήμα 16.1 · Ο walsender στέλνει WAL καθώς γράφεται. Ο walreceiver το γράφει στο pg_wal του standby και η διεργασία startup το ξαναπαίζει.

Στον standby τα ίδια φαίνονται από την άλλη πλευρά. Η διεργασία **walreceiver** λαμβάνει το WAL και η διεργασία **startup** το ξαναπαίζει.

SQL · ΘΥΡΑ 5502

```
SELECT pg_is_in_recovery() AS standby, status, sender_host, sender_port
FROM pg_stat_wal_receiver;

SELECT backend_type FROM pg_stat_activity
WHERE backend_type IN ('walreceiver', 'startup');
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
 standby |  status   | sender_host | sender_port
---------+-----------+-------------+-------------
 t       | streaming | localhost   |        5501
(1 row)

 backend_type
--------------
 startup
 walreceiver
(2 rows)
```

Οι αλλαγές ταξιδεύουν

Ό,τι γράφεται στον primary, δεδομένα, πίνακες, indexes, ρόλοι, φτάνει στον standby. Η replication είναι physical. Αντιγράφει το WAL, άρα όλο το cluster, σελίδα προς σελίδα.

SQL

```
CREATE TABLE announcements (id integer PRIMARY KEY, body text);
INSERT INTO announcements VALUES (1, 'Νέα συλλογή βιβλίων');
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 1
```

SQL · ΘΥΡΑ 5502

```
SELECT * FROM announcements;
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
 id |        body
----+--------------------
  1 | Νέα συλλογή βιβλίων
(1 row)
```

Ο standby δέχεται μόνο αναγνώσεις. Κάθε προσπάθεια εγγραφής αποτυγχάνει.

SQL · ΘΥΡΑ 5502

```
INSERT INTO announcements VALUES (2, 'Από τον standby');
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
ERROR:  cannot execute INSERT in a read-only transaction
```

Όσοι πίνακες είναι unlogged, όπως είδαμε στο κεφάλαιο 9, δεν γράφουν WAL, άρα δεν φτάνουν ποτέ στον standby. Εκεί υπάρχουν μόνο ως άδεια κελύφη που δεν μπορείς να διαβάσεις.

Πόσο πίσω είναι

Το lag μετριέται με δύο τρόπους. Σε bytes, ως απόσταση ανάμεσα στο τρέχον LSN του primary και στο `replay_lsn`. Και σε χρόνο, με τις στήλες `write_lag`, `flush_lag` και `replay_lag`, που λένε πόσο κράτησε κάθε στάδιο για το πρόσφατο WAL. Φορτώνουμε τον primary με το pgbench και μετράμε όσο δουλεύει.

TERMINAL

```
pgbench -n -c 4 -T 3 >/dev/null 2>&1 &
sleep 1.5
psql -c "SELECT pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn)) AS behind,
                replay_lag
         FROM pg_stat_replication"
wait
```

ΕΞΟΔΟΣ

```
   behind   |   replay_lag
------------+-----------------
 8888 bytes | 00:00:00.000123
(1 row)
```

Σε ένα τοπικό εργαστήριο ο standby είναι λίγα KB και χιλιοστά του δευτερολέπτου πίσω. Σε πραγματικό δίκτυο, με έναν standby σε άλλο datacenter, ίσως αρκετές δεκάδες ms. Στον ίδιο τον standby η συνάρτηση `pg_last_xact_replay_timestamp()` δίνει την ώρα του τελευταίου commit που ξαναπαίχτηκε. Σε έναν ήσυχο primary όμως αυτή η ώρα μένει πίσω απλώς επειδή δεν γίνονται commits, οπότε δεν είναι καλό μέτρο μόνη της.

Σύνδεση στον σωστό server

Η εφαρμογή πρέπει να γράφει στον primary και μπορεί να διαβάζει από τον standby. Το libpq, η βιβλιοθήκη που χρησιμοποιεί το psql και οι περισσότεροι drivers, δέχεται πολλούς servers σε ένα connection string και την παράμετρο `target_session_attrs`, που λέει ποιον θέλεις. Οι λίστες του `host` και του `port` αντιστοιχούν θέση προς θέση, οπότε το `localhost` γράφεται δύο φορές. Με `read-write` το libpq δοκιμάζει τους servers με τη σειρά και κρατά τον πρώτο που δέχεται εγγραφές.

TERMINAL

```
psql "host=localhost,localhost port=5502,5501 target_session_attrs=read-write" \
  -At -c "SELECT current_setting('port')"
psql "host=localhost,localhost port=5501,5502 target_session_attrs=standby" \
  -At -c "SELECT current_setting('port')"
```

ΕΞΟΔΟΣ

```
5501
5502
```

Ο standby μπήκε πρώτος στην πρώτη λίστα, αλλά απορρίφθηκε. Η δεύτερη ζήτησε standby και τον βρήκε, αν και ήταν δεύτερος. Όταν ο primary αλλάξει μετά από failover, η ίδια ρύθμιση στέλνει την εφαρμογή στον νέο χωρίς αλλαγή στη διαμόρφωση. Υπάρχουν και οι τιμές `read-only`, `primary` και `prefer-standby`.

ΣΗΜΕΙΩΣΗ

Στο εργαστήριο η replication συνδέεται ως `postgres` χωρίς κωδικό, χάρη στο `trust` του `pg_hba.conf`. Σε production φτιάχνεις ρόλο με `REPLICATION` και κωδικό, τον βάζεις σε γραμμή `replication` του `pg_hba.conf` με `scram-sha-256`. Ο κωδικός μπαίνει σε ένα αρχείο `.pgpass` στον standby αντί για το `primary_conninfo`.

ΚΡΑΤΑ ΑΥΤΟ

Ένας standby είναι base backup με `standby.signal` και `primary_conninfo`, που τα γράφει το `pg_basebackup -R`. Ο walsender του primary στέλνει WAL, ο walreceiver του standby το λαμβάνει και η διεργασία startup το ξαναπαίζει. Το `pg_stat_replication` δείχνει τα στάδια sent, write, flush και replay και το lag. Ο standby δέχεται μόνο αναγνώσεις. Το `target_session_attrs` στέλνει κάθε σύνδεση στον σωστό server.

Ασκήσεις

ΑΣΚΗΣΗ 16.1 Από τον primary στον standby

Φτιάξε στον primary πίνακα `rep_test` με 1.000 γραμμές από το `generate_series`, περίμενε μισό δευτερόλεπτο και μέτρησε τις γραμμές στον standby.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
1000
```

ΑΣΚΗΣΗ 16.2 Τα LSN του standby

Στον standby δείξε το τελευταίο LSN που έλαβε, το τελευταίο που ξαναέπαιξε και αν είναι ίσα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 received  | replayed  | caught_up
-----------+-----------+-----------
 0/C7B3C98 | 0/C7B3C98 | t
(1 row)
```

ΑΣΚΗΣΗ 16.3 Σύνδεση μόνο σε standby

Με connection string που έχει και τους δύο servers, με τον primary πρώτο, σύνδεσε το psql σε server που είναι σε recovery και δείξε τη θύρα και το `pg_is_in_recovery()`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
5502|t
```

ΑΣΚΗΣΗ 16.4 Cascading replication

Φτιάξε δεύτερο standby, τον `standby2` στη θύρα 5503, που παίρνει WAL από τον `standby` αντί για τον primary. Δείξε μετά από τον `standby` ποιος συνδέεται σε αυτόν.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
standby2: σε λειτουργία στη θύρα 5503
 application_name |   state
------------------+-----------
 walreceiver      | streaming
(1 row)
```

ΑΣΚΗΣΗ 16.5 DDL στον standby

Δοκίμασε να φτιάξεις index στον πίνακα `announcements` από τον standby.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  cannot execute CREATE INDEX in a read-only transaction
```

ΚΕΦΑΛΑΙΟ 17

Replication slots, lag και conflicts

Ο standby του προηγούμενου κεφαλαίου δούλευε επειδή ήταν πάντα συνδεδεμένος. Στην πράξη ένας standby κάνει restart, χάνει το δίκτυο ή μένει πίσω. Τότε εμφανίζονται τρία προβλήματα. Ο primary σβήνει WAL που ο standby δεν πρόλαβε, ο primary κρατά WAL για έναν standby που δεν θα επιστρέψει. Και τα queries του standby συγκρούονται με το WAL που πρέπει να ξαναπαιχτεί.

Ο server του κεφαλαίου

Ένας primary με μικρό `max_wal_size`, ώστε να ανακυκλώνει WAL γρήγορα. Δίπλα του ένας standby χωρίς τίποτα επιπλέον.

TERMINAL

```
pglab init primary 5501
echo "max_wal_size = '64MB'" >> "$LAB/primary/postgresql.conf"
pglab start primary
pglab data 5501
pgbench -i -s 5 -q >/dev/null 2>&1
pg_basebackup -D "$LAB/standby" -R -X stream -c fast
echo "port = 5502" >> "$LAB/standby/postgresql.conf"
pglab start standby
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
standby: σε λειτουργία στη θύρα 5502
```

Όταν το WAL δεν υπάρχει πια

Ο standby σταματά, για παράδειγμα για αναβάθμιση του λειτουργικού. Στο μεταξύ ο primary δουλεύει και γράφει αρκετό WAL για αρκετά checkpoints.

TERMINAL

```
pglab stop standby
for i in 1 2; do psql -q -c "UPDATE pgbench_accounts SET abalance = abalance + 1"; done
psql -q -c "CHECKPOINT"
pglab start standby
sleep 2
grep 'has already been removed' "$LAB/standby.log" | tail -n 1
```

ΕΞΟΔΟΣ

```
standby: σταμάτησε
standby: σε λειτουργία στη θύρα 5502
2026-09-28 09:49:08.704 EEST [79706] FATAL:  could not receive data from WAL stream:
ERROR:  requested WAL segment 000000010000000000000007 has already been removed
```

Ο standby ζήτησε το segment από το οποίο έπρεπε να συνεχίσει, αλλά ο primary το είχε ήδη ανακυκλώσει, αφού κανείς δεν του είπε ότι το χρειάζεται κάποιος. Ο standby θα ξαναδοκιμάζει για πάντα και δεν θα προχωρήσει ποτέ. Αν υπάρχει archive, ο standby μπορεί να πάρει τα segments από εκεί με `restore_command`, όπως στο κεφάλαιο 14. Έτσι συνεχίζει κανονικά. Χωρίς archive, πρέπει να ξαναφτιαχτεί από νέο base backup.

Replication slots

Ένα **replication slot** είναι μια δέσμευση στον primary. Θυμάται μέχρι πού έχει λάβει WAL ένας συγκεκριμένος standby και ο primary δεν σβήνει κανένα segment από εκείνο το σημείο και μετά. Ξαναφτιάχνουμε τον standby με slot. Το `-C -S` του `pg_basebackup` φτιάχνει το slot και γράφει το όνομά του στο `primary_slot_name` του standby.

TERMINAL

```
pglab rm standby
pg_basebackup -D "$LAB/standby" -R -C -S standby_slot -X stream -c fast
echo "port = 5502" >> "$LAB/standby/postgresql.conf"
pglab start standby
grep primary_slot_name "$LAB/standby/postgresql.auto.conf"
```

ΕΞΟΔΟΣ

```
standby: σβήστηκε
standby: σε λειτουργία στη θύρα 5502
primary_slot_name = 'standby_slot'
```

SQL

```
SELECT slot_name, slot_type, active, restart_lsn, wal_status
FROM pg_replication_slots;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  slot_name   | slot_type | active | restart_lsn | wal_status
--------------+-----------+--------+-------------+------------
 standby_slot | physical  | t      | 0/29000000  | reserved
(1 row)
```

Το slot είναι ενεργό και το `restart_lsn` είναι το σημείο από το οποίο ο primary κρατά WAL. Επαναλαμβάνουμε το πείραμα.

TERMINAL

```
pglab stop standby
for i in 1 2; do psql -q -c "UPDATE pgbench_accounts SET abalance = abalance + 1"; done
psql -q -c "CHECKPOINT"
du -sh "$LAB/primary/pg_wal"
```

ΕΞΟΔΟΣ

```
standby: σταμάτησε
592M    ~/pglab/primary/pg_wal
```

SQL

```
SELECT slot_name, active, wal_status,
       pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)) AS retained
FROM pg_replication_slots;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  slot_name   | active | wal_status | retained
--------------+--------+------------+----------
 standby_slot | f      | extended   | 588 MB
(1 row)
```

Ο primary κράτησε όλο το WAL, αρκετά πάνω από το `max_wal_size`. Το `wal_status` έγινε `extended`, που σημαίνει ότι κρατά WAL πέρα από το κανονικό όριο για χάρη του slot. Όταν ο standby επιστρέψει, συνεχίζει κανονικά.

TERMINAL

```
pglab start standby
sleep 3
psql -c "SELECT application_name, state, replay_lsn = pg_current_wal_lsn() AS caught_up
         FROM pg_stat_replication"
```

ΕΞΟΔΟΣ

```
standby: σε λειτουργία στη θύρα 5502
 application_name |   state   | caught_up
------------------+-----------+-----------
 walreceiver      | streaming | t
(1 row)
```

Ο standby που δεν γύρισε ποτέ

Το slot δεν ξέρει αν ο standby θα επιστρέψει. Αν σβήσεις έναν standby και ξεχάσεις το slot του, ο primary θα κρατά WAL για πάντα, μέχρι να γεμίσει ο δίσκος. Επιπλέον, όπως είδαμε στο κεφάλαιο 10, ένα slot κρατά πίσω και τον ορίζοντα του `VACUUM`. Είναι από τις πιο συχνές αιτίες πτώσης σε production.

Η ρύθμιση `max_slot_wal_keep_size` βάζει όριο. Αν ένα slot χρειάζεται περισσότερο WAL, ο primary το ακυρώνει και σβήνει το WAL. Ο standby θα πρέπει να ξαναφτιαχτεί, αλλά ο primary θα ζήσει.

TERMINAL

```
psql -q -c "ALTER SYSTEM SET max_slot_wal_keep_size = '100MB'" -c "SELECT pg_reload_conf()"
pglab stop standby
for i in 1 2; do psql -q -c "UPDATE pgbench_accounts SET abalance = abalance + 1"; done
psql -q -c "CHECKPOINT"
grep 'invalidating' "$LAB/primary.log" | tail -n 1
```

ΕΞΟΔΟΣ

```
 pg_reload_conf
----------------
 t
(1 row)

standby: σταμάτησε
2026-09-28 09:49:24.257 EEST [79352] LOG:  invalidating obsolete replication slot "standby_slot"
```

SQL

```
SELECT slot_name, active, wal_status, invalidation_reason
FROM pg_replication_slots;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  slot_name   | active | wal_status | invalidation_reason
--------------+--------+------------+---------------------
 standby_slot | f      | lost       | wal_removed
(1 row)
```

Το slot είναι `lost` και ο λόγος είναι `wal_removed`. Ο standby δεν μπορεί να συνεχίσει από αυτό, οπότε σβήνουμε το slot και ξαναφτιάχνουμε τον standby. Από την έκδοση 18 υπάρχει και το `idle_replication_slot_timeout`, που ακυρώνει ένα slot αν μείνει ανενεργό περισσότερο από όσο ορίζει.

TERMINAL

```
psql -q -c "SELECT pg_drop_replication_slot('standby_slot')"
pglab rm standby
pg_basebackup -D "$LAB/standby" -R -C -S standby_slot -X stream -c fast
echo "port = 5502" >> "$LAB/standby/postgresql.conf"
pglab start standby
```

ΕΞΟΔΟΣ

```
 pg_drop_replication_slot
--------------------------

(1 row)

standby: σβήστηκε
standby: σε λειτουργία στη θύρα 5502
```

> **ΠΑΓΙΔΑ**
> Κάθε slot θέλει παρακολούθηση. Ένα slot με `active = false` για πολλή ώρα ή με `wal_status` διαφορετικό από `reserved` είναι alert. Το `max_slot_wal_keep_size` πρέπει να είναι μικρότερο από τον ελεύθερο χώρο του δίσκου. Η προεπιλογή του, `-1`, σημαίνει χωρίς όριο.

Conflicts στον standby

Ένα query στον standby διαβάζει από ένα snapshot, όπως στον primary. Ο primary όμως δεν ξέρει τι διαβάζει ο standby. Αν ένα `VACUUM` στον primary σβήσει γραμμές που το query του standby χρειάζεται ακόμη, το WAL του `VACUUM` φτάνει στον standby και δεν μπορεί να εφαρμοστεί χωρίς να χαλάσει το query. Ο standby έχει δύο επιλογές. Να καθυστερήσει το replay ή να ακυρώσει το query. Περιμένει το πολύ `max_standby_streaming_delay`, 30 δευτερόλεπτα από προεπιλογή. Μετά ακυρώνει. Για το πείραμα το κάνουμε ένα δευτερόλεπτο.

SQL · ΘΥΡΑ 5502

```
ALTER SYSTEM SET max_standby_streaming_delay = '1s';
SELECT pg_reload_conf();
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
ALTER SYSTEM

 pg_reload_conf
----------------
 t
(1 row)
```

Η συνεδρία Β στον standby ανοίγει ένα μακρύ transaction που διαβάζει τους tellers και μετά δουλεύει για πέντε δευτερόλεπτα.

SQL · ΣΥΝΕΔΡΙΑ Β · ΘΥΡΑ 5502

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT count(*) FROM pgbench_tellers;
SELECT pg_sleep(5);
SELECT sum(tbalance) FROM pgbench_tellers;
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Β · ΘΥΡΑ 5502

```
-- η εντολή περιμένει
```

Στο μεταξύ ο primary αλλάζει όλους τους tellers και τρέχει `VACUUM`, που σβήνει τις παλιές εκδόσεις.

SQL

```
UPDATE pgbench_tellers SET tbalance = tbalance + 1;
VACUUM pgbench_tellers;
```

ΑΠΟΤΕΛΕΣΜΑ

```
UPDATE 50

VACUUM
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ B

```
BEGIN

 count
-------
    50
(1 row)

ERROR:  canceling statement due to conflict with recovery
DETAIL:  User query might have needed to see row versions that must be removed.

ERROR:  current transaction is aborted, commands ignored until end of transaction block

ROLLBACK
```

Το query του standby ακυρώθηκε με `conflict with recovery`. Το `DETAIL` λέει ότι χρειαζόταν εκδόσεις γραμμών που αφαιρέθηκαν. Ο standby μετρά τέτοιες ακυρώσεις ανά αιτία.

SQL · ΘΥΡΑ 5502

```
SELECT datname, confl_snapshot, confl_lock
FROM pg_stat_database_conflicts
WHERE datname = 'sqlbook';
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
 datname | confl_snapshot | confl_lock
---------+----------------+------------
 sqlbook |              1 |          0
(1 row)
```

hot_standby_feedback

Η ρύθμιση `hot_standby_feedback` στον standby στέλνει στον primary το παλαιότερο snapshot που χρησιμοποιεί. Ο primary το σέβεται σαν να ήταν ένα δικό του ανοιχτό transaction και το `VACUUM` δεν σβήνει γραμμές που χρειάζεται ο standby.

SQL · ΘΥΡΑ 5502

```
ALTER SYSTEM SET hot_standby_feedback = on;
SELECT pg_reload_conf();
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
ALTER SYSTEM

 pg_reload_conf
----------------
 t
(1 row)
```

SQL · ΣΥΝΕΔΡΙΑ C · ΘΥΡΑ 5502

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT count(*) FROM pgbench_tellers;
SELECT pg_sleep(5);
SELECT sum(tbalance) FROM pgbench_tellers;
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ C · ΘΥΡΑ 5502

```
-- η εντολή περιμένει
```

SQL

```
SELECT slot_name, xmin FROM pg_replication_slots;
UPDATE pgbench_tellers SET tbalance = tbalance + 1;
VACUUM (VERBOSE) pgbench_tellers;
```

ΑΠΟΤΕΛΕΣΜΑ

```
  slot_name   | xmin
--------------+------
 standby_slot |  838
(1 row)

UPDATE 50

INFO:  vacuuming "sqlbook.public.pgbench_tellers"
INFO:  finished vacuuming "sqlbook.public.pgbench_tellers": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total), 0 eagerly scanned
tuples: 0 removed, 100 remain, 50 are dead but not yet removable
removable cutoff: 838, which was 1 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
visibility map: 0 pages set all-visible, 0 pages set all-frozen (0 were all-visible)
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 21 hits, 0 reads, 0 dirtied
WAL usage: 1 records, 0 full page images, 299 bytes, 0 buffers full
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ C

```
BEGIN

 count
-------
    50
(1 row)

 pg_sleep
----------

(1 row)

 sum
-----
  50
(1 row)

COMMIT
```

Αυτή τη φορά το query τελείωσε. Το `VACUUM` άφησε τις νεκρές γραμμές ως `not yet removable`, γιατί ο primary ήξερε το snapshot του standby. Επειδή ο standby χρησιμοποιεί slot, ο primary κρατά αυτή την πληροφορία στη στήλη `xmin` του slot. Χωρίς slot θα την έβλεπες στο `backend_xmin` του `pg_stat_replication`. Το slot τη θυμάται και όταν ο standby αποσυνδεθεί, οπότε ένα εγκαταλελειμμένο slot με feedback εμποδίζει το `VACUUM` σε όλη τη βάση. Το κόστος είναι στον primary. Ένα query αναφοράς μιας ώρας στον standby κρατά bloat μιας ώρας στον primary. Η επιλογή είναι ανάμεσα σε ακυρωμένα queries στον standby, bloat στον primary ή μεγάλο replay lag με μεγαλύτερο `max_standby_streaming_delay`. Για standby που υπάρχει μόνο για failover, τα μικρά όρια και η ακύρωση είναι σωστά. Για standby αναφορών, το `hot_standby_feedback` με `statement_timeout` στον ρόλο των αναφορών είναι συνήθως ο καλύτερος συμβιβασμός.

ΚΡΑΤΑ ΑΥΤΟ

Χωρίς slot ο primary σβήνει WAL που ένας standby δεν πρόλαβε και ο standby χρειάζεται archive ή νέο base backup. Ένα slot κρατά το WAL, αλλά ένα slot χωρίς standby γεμίζει τον δίσκο, οπότε βάζεις `max_slot_wal_keep_size` και παρακολουθείς το `pg_replication_slots`. Ένα query στον standby μπορεί να ακυρωθεί από το replay του `VACUUM`. Το `max_standby_streaming_delay` ορίζει πόσο περιμένει ο standby και το `hot_standby_feedback` μεταφέρει το κόστος στον primary ως bloat.

Ασκήσεις

ΑΣΚΗΣΗ 17.1 Πόσο WAL κρατά κάθε slot

Δείξε για κάθε slot το όνομα, αν είναι ενεργό, το `wal_status` και πόσο WAL μπορεί ακόμη να γραφτεί πριν ακυρωθεί, από τη στήλη `safe_wal_size`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  slot_name   | active | wal_status |  safe
--------------+--------+------------+--------
 standby_slot | t      | reserved   | 112 MB
(1 row)
```

ΑΣΚΗΣΗ 17.2 Slot χωρίς standby

Φτιάξε με `pg_create_physical_replication_slot` ένα slot `spare` που κρατά WAL από τώρα και δείξε το `restart_lsn` και το `active` του. Σβήσε το μετά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 slot_name | reserved
-----------+----------
 spare     | t
(1 row)

 slot_name | active | has_restart_lsn
-----------+--------+-----------------
 spare     | f      | t
(1 row)

 pg_drop_replication_slot
--------------------------

(1 row)
```

ΑΣΚΗΣΗ 17.3 Conflicts ανά αιτία

Δείξε στον standby όλες τις στήλες `confl_` του `pg_stat_database_conflicts` για τη βάση `sqlbook`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 confl_tablespace | confl_lock | confl_snapshot | confl_bufferpin | confl_deadlock | confl_active_logicalslot
------------------+------------+----------------+-----------------+----------------+--------------------------
                0 |          0 |              1 |               0 |              0 |                        0
(1 row)
```

ΑΣΚΗΣΗ 17.4 Lag σε bytes για κάθε στάδιο

Δείξε από τον primary για κάθε standby σε bytes πόσο πίσω είναι το write, το flush και το replay από το τρέχον LSN.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 application_name | write_behind | flush_behind | replay_behind
------------------+--------------+--------------+---------------
 walreceiver      |            0 |            0 |             0
(1 row)
```

ΑΣΚΗΣΗ 17.5 Κλείσιμο του feedback

Δείξε στον primary το `xmin` του slot. Κλείσε μετά το `hot_standby_feedback` στον standby και δείξε το ξανά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
t
 pg_reload_conf
----------------
 t
(1 row)

f
```

ΚΕΦΑΛΑΙΟ 18

Synchronous replication

Με την asynchronous replication το commit επιστρέφει μόλις το WAL γραφτεί στον δίσκο του primary. Αν ο primary χαθεί ένα κλάσμα του δευτερολέπτου μετά, τα τελευταία commits μπορεί να μην έχουν φτάσει στον standby. Η synchronous replication κάνει το commit να περιμένει και τον standby. Κερδίζεις την εγγύηση ότι κανένα commit δεν χάνεται σε failover και πληρώνεις με καθυστέρηση σε κάθε commit και με έναν νέο τρόπο να κολλήσει ο server.

Ο server του κεφαλαίου

Δύο standbys αυτή τη φορά. Ο καθένας έχει όνομα με τη ρύθμιση `cluster_name`. Ο walreceiver χρησιμοποιεί αυτό το όνομα ως `application_name` όταν συνδέεται στον primary και με αυτό ο primary τους ξεχωρίζει.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
for n in 1 2; do
  pg_basebackup -D "$LAB/standby$n" -R -X stream -c fast
  printf "port = %s\ncluster_name = 'standby%s'\n" "550$((n + 1))" "$n" >> "$LAB/standby$n/postgresql.conf"
  pglab start "standby$n"
done
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
standby1: σε λειτουργία στη θύρα 5502
standby2: σε λειτουργία στη θύρα 5503
```

SQL

```
SELECT application_name, state, sync_state FROM pg_stat_replication ORDER BY 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 application_name |   state   | sync_state
------------------+-----------+------------
 standby1         | streaming | async
 standby2         | streaming | async
(2 rows)
```

Ένας synchronous standby

Η ρύθμιση `synchronous_standby_names` στον primary λέει ποιοι standbys είναι synchronous. Αλλάζει με reload.

SQL

```
ALTER SYSTEM SET synchronous_standby_names = 'standby1';
SELECT pg_reload_conf();
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SYSTEM

 pg_reload_conf
----------------
 t
(1 row)
```

SQL

```
SELECT application_name, sync_state FROM pg_stat_replication ORDER BY 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 application_name | sync_state
------------------+------------
 standby1         | sync
 standby2         | async
(2 rows)
```

Ο `standby1` είναι `sync` και ο `standby2` μένει `async`. Από εδώ και πέρα κάθε commit στον primary περιμένει ώσπου ο `standby1` να γράψει το WAL του commit στον δίσκο του. Όσο ο `standby1` απαντά γρήγορα, η διαφορά είναι μια καθυστέρηση ίση με έναν γύρο στο δίκτυο.

Όταν ο synchronous standby λείπει

Σταματάμε τον `standby1` και προσπαθούμε να γράψουμε.

TERMINAL

```
pglab stop standby1
```

ΕΞΟΔΟΣ

```
standby1: σταμάτησε
```

SQL · ΣΥΝΕΔΡΙΑ Β

```
INSERT INTO cities VALUES (12, 'Κέρκυρα', 'Ιόνια Νησιά', 100000);
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Β

```
-- η εντολή περιμένει
```

Το `INSERT` δεν τελειώνει. Από μια τρίτη σύνδεση βλέπουμε τι περιμένει.

SQL

```
SELECT application_name, state, wait_event_type, wait_event
FROM pg_stat_activity
WHERE wait_event = 'SyncRep';
```

ΑΠΟΤΕΛΕΣΜΑ

```
 application_name | state  | wait_event_type | wait_event
------------------+--------+-----------------+------------
 book-b           | active | IPC             | SyncRep
(1 row)
```

Η σύνδεση περιμένει στο `SyncRep`. Και το πιο παράξενο, η γραμμή είναι ήδη γραμμένη και στο WAL του primary. Ο primary απλώς δεν λέει στον client ότι το commit πέτυχε μέχρι να το επιβεβαιώσει ο standby. Αν ο standby δεν επιστρέψει ποτέ, η σύνδεση θα περιμένει για πάντα. Μόλις επιστρέψει, το commit ολοκληρώνεται.

TERMINAL

```
pglab start standby1
```

ΕΞΟΔΟΣ

```
standby1: σε λειτουργία στη θύρα 5502
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ Β

```
INSERT 0 1
```

ΠΑΓΙΔΑ

Ένα synchronous σύστημα με έναν μόνο standby έχει χειρότερη διαθεσιμότητα από έναν μόνο server. Αν πέσει είτε ο primary είτε ο standby, οι εγγραφές σταματούν. Γι' αυτό θέλεις τουλάχιστον δύο standbys και να αρκεί ένας από αυτούς.

Ακύρωση ενός commit που περιμένει

Τι γίνεται αν ακυρώσεις ένα commit που περιμένει τον standby; Σταματάμε ξανά τον `standby1` και η συνεδρία B γράφει.

TERMINAL

```
pglab stop standby1
```

ΕΞΟΔΟΣ

```
standby1: σταμάτησε
```

SQL · ΣΥΝΕΔΡΙΑ B

```
INSERT INTO cities VALUES (13, 'Χαλκίδα', 'Στερεά Ελλάδα', 60000);
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ B

```
-- η εντολή περιμένει
```

SQL

```
SELECT pg_cancel_backend(pid) AS cancelled
FROM pg_stat_activity WHERE wait_event = 'SyncRep';
```

ΑΠΟΤΕΛΕΣΜΑ

```
 cancelled
-----------
 t
(1 row)
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ B

```
WARNING:  canceling wait for synchronous replication due to user request
DETAIL:  The transaction has already committed locally, but might not have been replicated to the standby.
INSERT 0 1
```

SQL

```
SELECT id, name FROM cities WHERE id >= 12 ORDER BY id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |  name
----+---------
 12 | Κέρκυρα
 13 | Χαλκίδα
(2 rows)
```

Η ακύρωση δεν αναίρεσε το `INSERT`. Το προειδοποιητικό μήνυμα το λέει καθαρά. Το transaction έκανε ήδη commit τοπικά, αλλά ίσως δεν έφτασε στον standby. Η γραμμή είναι στον primary και ορατή σε όλους. Η εγγύηση της synchronous replication ισχύει μόνο αν ο client περιμένει να τελειώσει το commit. Μια εφαρμογή που βάζει timeout στο commit και θεωρεί ότι απέτυχε, θα κάνει λάθος.

TERMINAL

```
pglab start standby1
```

ΕΞΟΔΟΣ

```
standby1: σε λειτουργία στη θύρα 5502
```

Quorum

Η σύνταξη `ANY n (λίστα)` ζητά επιβεβαίωση από οποιουσδήποτε n της λίστας. Με `ANY 1 (standby1, standby2)` κάθε commit περιμένει όποιον από τους δύο απαντήσει πρώτος και ο server συνεχίζει να γράφει όσο ζει έστω και ένας.

SQL

```
ALTER SYSTEM SET synchronous_standby_names = 'ANY 1 (standby1, standby2)';
SELECT pg_reload_conf();
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SYSTEM

 pg_reload_conf
----------------
 t
(1 row)
```

SQL

```
SELECT application_name, sync_state FROM pg_stat_replication ORDER BY 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 application_name | sync_state
------------------+------------
 standby1         | quorum
 standby2         | quorum
(2 rows)
```

TERMINAL

```
pglab stop standby1
psql -c "INSERT INTO cities VALUES (14, 'Σέρρες', 'Κεντρική Μακεδονία', 70000)"
pglab start standby1
```

```
ΕΞΟΔΟΣ

standby1: σταμάτησε
INSERT 0 1
standby1: σε λειτουργία στη θύρα 5502
```

Το `INSERT` τελείωσε αμέσως, αφού ο `standby2` το επιβεβαίωσε. Η άλλη σύνταξη, `FIRST n (λίστα)`, είναι σειρά προτεραιότητας. Περιμένει τους n πρώτους της λίστας που είναι συνδεδεμένοι και αν ένας λείπει, τον επόμενο.

Πόσο synchronous

Το `synchronous_commit` ορίζει τι ακριβώς περιμένει ένα commit. Μπορεί να αλλάξει ανά transaction, οπότε δεν πληρώνουν όλες οι εγγραφές το ίδιο κόστος.

Τιμή	Το commit περιμένει
`off`	τίποτα, ούτε τον δίσκο του primary
`local`	τον δίσκο του primary, όχι τον standby
`remote_write`	τον standby να γράψει το WAL στο λειτουργικό του
`on`	τον standby να γράψει το WAL στον δίσκο του
`remote_apply`	τον standby να ξαναπαίξει το WAL, ώστε η αλλαγή να είναι ορατή εκεί

Χωρίς `synchronous_standby_names` τα `remote_*` και το `on` είναι ίδια με το `local`. Το `remote_apply` λύνει ένα κλασικό πρόβλημα. Ο χρήστης αποθηκεύει κάτι, η εφαρμογή τον στέλνει σε μια σελίδα που διαβάζει από τον standby. Η αλλαγή του δεν υπάρχει ακόμη εκεί. Με `remote_apply`, όταν επιστρέψει το commit, η αλλαγή είναι ήδη ορατή σε κάθε synchronous standby.

```
TERMINAL

psql -q -c "SET synchronous_commit = remote_apply" \
        -c "INSERT INTO cities VALUES (15, 'Δράμα', 'Ανατολική Μακεδονία', 58000)"
psql -p 5502 -At -c "SELECT name FROM cities WHERE id = 15"
psql -p 5503 -At -c "SELECT name FROM cities WHERE id = 15"
```

```
ΕΞΟΔΟΣ

Δράμα
Δράμα
```

Η αλλαγή φάνηκε αμέσως και στους δύο standbys. Εδώ είναι τοπικοί και θα την έβλεπαν σχεδόν αμέσως ούτως ή άλλως. Σε πραγματικό δίκτυο, ή με έναν standby απασχολημένο, η διαφορά είναι αυτή ανάμεσα σε «το βλέπω» και «δεν το βλέπω ακόμη».

Ένα `SET LOCAL synchronous_commit = local` σε ένα transaction που γράφει δεδομένα χωρίς αξία, όπως στατιστικά επισκέψεων, το απαλλάσσει από την αναμονή του standby. Οι σημαντικές εγγραφές συνεχίζουν να περιμένουν.

ΚΡΑΤΑ ΑΥΤΟ

Με `synchronous_standby_names` κάθε commit περιμένει και standbys, με ανοχή που ορίζεις με `FIRST n` ή `ANY n`. Αν δεν απαντήσει κανείς, οι εγγραφές σταματούν στο `SyncRep`, οπότε θέλεις πάντα περισσότερους standbys από όσους περιμένεις. Ένα commit που ακυρώνεται την ώρα που περιμένει έχει ήδη γίνει τοπικά. Το `synchronous_commit` ορίζει ανά transaction τι περιμένει το commit και το `remote_apply` κάνει την αλλαγή ορατή στον standby πριν επιστρέψει.

Ασκήσεις

ΑΣΚΗΣΗ 18.1 Σειρά προτεραιότητας

Όρισε `synchronous_standby_names` σε `FIRST 1 (standby2, standby1)`, κάνε reload και δείξε για κάθε standby το `sync_priority` και το `sync_state`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 pg_reload_conf
----------------
 t
(1 row)

 application_name | sync_priority | sync_state
------------------+---------------+------------
 standby2         |             1 | sync
 standby1         |             2 | potential
(2 rows)
```

ΑΣΚΗΣΗ 18.2 Commit που δεν περιμένει

Σταμάτησε και τους δύο standbys. Κάνε ένα `INSERT` στον `cities` με `synchronous_commit = local` για αυτό το transaction και δείξε ότι τελείωσε. Ξεκίνα μετά τους standbys.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
standby1: σταμάτησε
standby2: σταμάτησε
BEGIN
SET
INSERT 0 1
COMMIT
standby1: σε λειτουργία στη θύρα 5502
standby2: σε λειτουργία στη θύρα 5503
```

ΑΣΚΗΣΗ 18.3 Ο επόμενος στη σειρά

Με το `FIRST 1 (standby2, standby1)` της πρώτης άσκησης, σταμάτησε τον `standby2`, δείξε το `sync_state` του `standby1` και πρόσθεσε μια πόλη. Ξεκίνα μετά ξανά τον `standby2`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
standby2: σταμάτησε
 application_name | sync_state
------------------+------------
 standby1         | sync
(1 row)

INSERT 0 1
standby2: σε λειτουργία στη θύρα 5503
```

ΑΣΚΗΣΗ 18.4 Πίσω σε asynchronous

Άδειασε το `synchronous_standby_names`, κάνε reload και δείξε το `sync_state` των standbys.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 pg_reload_conf
----------------
 t
(1 row)

 application_name | sync_state
------------------+------------
 standby1         | async
 standby2         | async
(2 rows)
```

ΑΣΚΗΣΗ 18.5 Ορατό στον standby αμέσως

Με `synchronous_standby_names = 'ANY 2 (standby1, standby2)'` και `synchronous_commit = remote_apply` για ένα transaction, πρόσθεσε μια πόλη και διάβασέ την αμέσως από τον `standby2`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 pg_reload_conf
----------------
 t
(1 row)

Ξάνθη
```

ΚΕΦΑΛΑΙΟ 19

Failover, timelines και pg_rewind

Ο λόγος ύπαρξης ενός standby είναι η στιγμή που ο primary χάνεται. Τότε ο standby γίνεται primary, η εφαρμογή συνδέεται σε αυτόν και ο παλιός primary, όταν επιστρέψει, πρέπει να γίνει standby χωρίς να χαλάσει τίποτα. Κάθε βήμα έχει μια παγίδα. Η χειρότερη είναι δύο servers που πιστεύουν ταυτόχρονα ότι είναι ο primary.

Ο server του κεφαλαίου

Ο primary κρατά 256 MB WAL με το `wal_keep_size`, για λόγο που θα φανεί στο `pg_rewind`.

TERMINAL

```
pglab init primary 5501
echo "wal_keep_size = '256MB'" >> "$LAB/primary/postgresql.conf"
pglab start primary
pglab data 5501
pgbench -i -s 5 -q >/dev/null 2>&1
pg_basebackup -D "$LAB/standby" -R -X stream -c fast
echo "port = 5502" >> "$LAB/standby/postgresql.conf"
pglab start standby
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
standby: σε λειτουργία στη θύρα 5502
```

Ο primary χάνεται

Η εφαρμογή γράφει. Ο primary πέφτει απότομα, όπως θα έπεφτε αν χαλούσε το μηχάνημά του.

TERMINAL

```
pgbench -n -t 500 >/dev/null 2>&1
psql -At -c "SELECT count(*) FROM pgbench_history"
pg_ctl -D "$LAB/primary" -m immediate stop
```

ΕΞΟΔΟΣ

```
500
waiting for server to shut down.... done
server stopped
```

Ο standby το καταλαβαίνει αμέσως, αφού χάθηκε η σύνδεση replication. Συνεχίζει να προσπαθεί να συνδεθεί. Δεν αλλάζει ρόλο μόνος του. Η PostgreSQL δεν αποφασίζει ποτέ μόνη της ότι ο primary πέθανε, γιατί από την πλευρά του standby ένας νεκρός primary και ένα κομμένο δίκτυο φαίνονται ίδια.

TERMINAL

```
sleep 1
grep -E 'could not connect|FATAL' "$LAB/standby.log" | tail -n 1
```

ΕΞΟΔΟΣ

```
2026-09-28 09:49:58.331 EEST [80848] FATAL:  streaming replication receiver
"walreceiver" could not connect to the primary server: connection to server at
"localhost" (::1), port 5501 failed: Connection refused
```

Promote

Η απόφαση είναι δική σου ή ενός εργαλείου που παρακολουθεί τους servers. Το **promote** κάνει τον standby primary. Γίνεται με `pg_ctl promote` ή με `SELECT pg_promote()` από SQL.

TERMINAL

```
pg_ctl -D "$LAB/standby" promote
psql -p 5502 -At -c "SELECT pg_is_in_recovery(), count(*) FROM pgbench_history"
grep -E 'promote|selected new timeline|ready to accept connections' "$LAB/standby.log"
```

ΕΞΟΔΟΣ

```
waiting for server to promote.... done
server promoted
f|500
2026-09-28 09:49:59.451 EEST [80828] LOG:  received promote request
2026-09-28 09:49:59.453 EEST [80828] LOG:  selected new timeline ID: 2
2026-09-28 09:49:59.478 EEST [80821] LOG:  database system is ready to accept connections
```

Ο standby έγινε primary στο timeline 2 και έχει και τα 500 transactions, αφού τα είχε λάβει πριν πέσει ο primary. Με asynchronous replication αυτό δεν είναι εγγυημένο. Τα τελευταία commits μπορεί να χάνονταν και αυτό είναι ακριβώς ό,τι εξασφαλίζει η synchronous replication του προηγούμενου κεφαλαίου.

Η εφαρμογή με `target_session_attrs=read-write` και τους δύο servers στο connection string βρίσκει τον νέο primary χωρίς καμία αλλαγή.

TERMINAL

```
psql "host=localhost,localhost port=5501,5502 target_session_attrs=read-write" \
  -At -c "SELECT current_setting('port')"
```

ΕΞΟΔΟΣ

```
5502
```

Split brain

Ο παλιός primary επισκευάζεται και κάποιος τον ξεκινά, χωρίς να ξέρει ότι έγινε failover. Ξεκινά κανονικά ως primary.

TERMINAL

```
pglab start primary
psql -At -c "SELECT pg_is_in_recovery()"
psql -q -c "INSERT INTO cities VALUES (20, 'Τρίκαλα', 'Θεσσαλία', 80000)"
psql -p 5502 -q -c "INSERT INTO cities VALUES (20, 'Καβάλα', 'Ανατολική Μακεδονία', 70000)"
psql -At -c "SELECT name FROM cities WHERE id = 20"
psql -p 5502 -At -c "SELECT name FROM cities WHERE id = 20"
```

ΕΞΟΔΟΣ

```
primary: σε λειτουργία στη θύρα 5501
f
Τρίκαλα
Καβάλα
```

Τώρα υπάρχουν δύο primaries με διαφορετική πόλη για το ίδιο id. Αυτό λέγεται **split brain**. Μια εφαρμογή που συνδέεται με τη σωστή σειρά στο connection string θα γράφει στον έναν, μια άλλη εφαρμογή ή ένα cron job με παλιά ρύθμιση στον άλλο. Οι δύο ιστορίες δεν ενώνονται ποτέ αυτόματα. Κάποιος πρέπει να διαλέξει ποια κρατά και να μεταφέρει με το χέρι ό,τι χρειάζεται από την άλλη.

Γι' αυτό κάθε failover περιλαμβάνει **fencing**, εξασφάλιση ότι ο παλιός primary δεν μπορεί να ξαναγράψει. Σβήνεις το μηχάνημα, τον βγάζεις από το δίκτυο ή τουλάχιστον αλλάζεις τη ρύθμισή του ώστε να μην ξεκινήσει ως primary. Στο εργαστήριο τον σταματάμε.

TERMINAL

```
pglab stop primary
```

ΕΞΟΔΟΣ

```
primary: σταμάτησε
```

pg_rewind

Ο παλιός primary πρέπει να γίνει standby του νέου. Δεν μπορεί απλώς να ξεκινήσει ως standby, γιατί η ιστορία του απέκλινε. Έχει την εγγραφή για τα Τρίκαλα και ίσως ό,τι άλλο πρόλαβε να γράψει πριν

πέσει, που ο νέος primary δεν πήρε ποτέ. Θα μπορούσες να τον σβήσεις και να τον ξαναφτιάξεις από νέο base backup, αλλά για μια βάση εκατοντάδων GB αυτό κρατά ώρες.

Το `pg_rewind` βρίσκει το σημείο όπου χωρίστηκαν οι δύο ιστορίες, από το history του timeline 2. Αντιγράφει από τον νέο primary μόνο τις σελίδες που άλλαξαν από εκεί και πέρα σε οποιονδήποτε από τους δύο. Χρειάζεται τα data checksums ή το `wal_log_hints`, ώστε κάθε αλλαγή σελίδας να φαίνεται στο WAL. Τα checksums είναι ενεργά από προεπιλογή στην έκδοση 18. Χρειάζεται επίσης στο `pg_wal` του παλιού primary όλο το WAL από το τελευταίο κοινό checkpoint μέχρι σήμερα. Αυτό είναι ο λόγος του `wal_keep_size` στην αρχή του κεφαλαίου. Χωρίς αυτό ο παλιός primary θα είχε ήδη ανακυκλώσει τα segments και το `pg_rewind` θα χρειαζόταν `--restore-target-wal`, για να τα πάρει από το archive με το `restore_command`. Με `-R` γράφει και τις ρυθμίσεις standby, όπως το `pg_basebackup -R`.

TERMINAL

```
pg_rewind -D "$LAB/primary" --source-server="host=localhost port=5502 user=postgres dbname=postgres" -R
```

ΕΞΟΔΟΣ

```
pg_rewind: servers diverged at WAL location 0/771A9B8 on timeline 1
pg_rewind: rewinding from last common checkpoint at 0/6000080 on timeline 1
pg_rewind: Done!
```

Το `pg_rewind` αντιγράφει λίγα αρχεία, όχι ολόκληρο το cluster. Ανάμεσά τους όμως είναι και τα αρχεία ρυθμίσεων του νέου primary, που τα αντικαθιστά όλα. Ο παλιός primary έχει τώρα το `postgresql.conf` του 5502, με τη θύρα 5502. Σε πραγματικούς servers με ίδιες ρυθμίσεις αυτό δεν φαίνεται, αλλά αν κάθε server έχει κάτι δικό του στο `postgresql.conf`, πρέπει να το ξαναγράψεις μετά το rewind. Διορθώνουμε τη θύρα και ξεκινάμε τον παλιό primary ως standby του νέου.

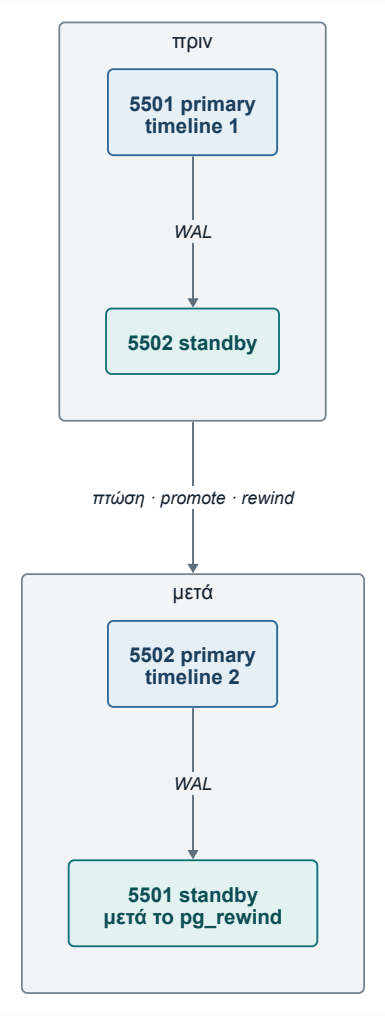
TERMINAL

```
grep '^port' "$LAB/primary/postgresql.conf"
echo "port = 5501" >> "$LAB/primary/postgresql.conf"
pglab start primary
sleep 2
psql -At -c "SELECT pg_is_in_recovery(), (SELECT name FROM cities WHERE id = 20)"
psql -p 5502 -c "SELECT application_name, state FROM pg_stat_replication"
```

ΕΞΟΔΟΣ

```
port = 5501
port = 5502
primary: σε λειτουργία στη θύρα 5501
t|Καβάλα
 application_name |   state
------------------+-----------
 walreceiver      | streaming
(1 row)
```

Η εγγραφή για τα Τρίκαλα χάθηκε. Ο server της θύρας 5501 έχει πια την ιστορία του νέου primary και τα Τρίκαλα υπάρχουν μόνο στο log. Αν ήταν σημαντικά δεδομένα, θα έπρεπε να τα είχες σώσει πριν από το `pg_rewind`.



Σχήμα 19.1 · Μετά το failover ο πρώην standby είναι primary σε νέο timeline και ο πρώην primary τον ακολουθεί ως standby.

Προγραμματισμένη εναλλαγή

Όταν αλλάζεις primary σκόπιμα, για αναβάθμιση ή συντήρηση του μηχανήματος, η διαδικασία λέγεται **switchover** και είναι απλούστερη. Σταματάς τον primary κανονικά. Στο κανονικό σταμάτημα ο primary στέλνει στον standby όλο το WAL ως το τελευταίο checkpoint πριν κλείσει. Ο standby έχει όλα τα commits και ο παλιός primary δεν έχει τίποτα που λείπει από τον standby. Κάνεις promote τον standby και ο παλιός primary μπαίνει standby χωρίς `pg_rewind`, αφού οι ιστορίες δεν απέκλιναν. Θα το κάνουμε σε άσκηση.

Εργαλεία για αυτόματο failover

Τα βήματα αυτού του κεφαλαίου, ανίχνευση, απόφαση, fencing, promote και rewind, σε production τα κάνει ένα εργαλείο. Το πιο διαδεδομένο είναι το **Patroni**. Τρέχει δίπλα σε κάθε PostgreSQL και κρατά σε ένα σύστημα όπως το etcd ή το Consul ποιος είναι ο primary, με ένα lock που λήγει. Ο primary ανανεώνει το lock κάθε λίγα δευτερόλεπτα. Αν δεν μπορεί, σταματά μόνος του να γράφει. Αν το lock

λήξει, ένας standby το παίρνει και κάνει promote. Έτσι δύο servers δεν μπορούν να είναι primary ταυτόχρονα, γιατί μόνο ένας κρατά το lock.

Εργαλείο	Τι κάνει
Patroni	αυτόματο failover με etcd, Consul ή ZooKeeper, rewind, ρύθμιση standbys
repmgr	εντολές για switchover και failover, με έναν daemon για αυτόματο
pg_auto_failover	ένας monitor node αποφασίζει για ένα ζεύγος primary και standby
CloudNativePG	operator για Kubernetes που κάνει τα ίδια με Kubernetes objects

Οι managed υπηρεσίες, όπως το Amazon RDS, κάνουν το ίδιο πίσω από ένα DNS όνομα που αλλάζει προς τον νέο primary.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Η PostgreSQL δεν κάνει ποτέ failover μόνη της. Το promote γίνεται με `pg_ctl promote` ή `pg_promote()` και ξεκινά νέο timeline. Ο παλιός primary πρέπει να απομονωθεί, αλλιώς έχεις split brain. Το `pg_rewind` τον ξαναφέρνει ως standby αντιγράφοντας μόνο ό,τι άλλαξε μετά τη διακλάδωση. Ό,τι είχε γράψει μόνο αυτός χάνεται. Ένα switchover με κανονικό σταμάτημα του primary δεν χρειάζεται rewind. Σε production όλα αυτά τα κάνει ένα εργαλείο όπως το Patroni.

Ασκήσεις

ΑΣΚΗΣΗ 19.1 Timeline στο pg_control

Δείξε με το `pg_controldata` το timeline του τελευταίου checkpoint για τον server της θύρας 5502 και το αρχείο history που έφτιαξε το promote.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Latest checkpoint's TimeLineID:       1
Latest checkpoint's TimeLineID:       2
1   0/771A9B8   no recovery target specified
```

ΑΣΚΗΣΗ 19.2 Ποιος είναι primary

Δείξε για τον 5502 αν είναι σε recovery και το timeline του από τα πρώτα οκτώ ψηφία του ονόματος του τρέχοντος segment. Για τον 5501 δείξε αν είναι σε recovery και το `received_tli` του `pg_stat_wal_receiver`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
f|00000002
t|2
```

ΑΣΚΗΣΗ 19.3 Switchover

Κάνε switchover ώστε ο server της θύρας 5501 να ξαναγίνει primary. Σταμάτησε κανονικά τον 5502, κάνε promote τον 5501, πρόσθεσε στον 5502 `standby.signal` και `primary_conninfo` προς τον 5501 και ξεκίνα τον. Δείξε το `pg_stat_replication` του 5501.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
standby: σταμάτησε
waiting for server to promote.... done
server promoted
standby: σε λειτουργία στη θύρα 5502
   state   | sync_state
-----------+------------
 streaming | async
(1 row)
```

ΑΣΚΗΣΗ 19.4 Δοκιμή χωρίς αλλαγές

Σταμάτησε τον 5502 και τρέξε `pg_rewind` με `--dry-run` προς τον 5501. Δείξε τις δύο τελευταίες γραμμές της εξόδου και ξεκίνα ξανά τον 5502.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
standby: σταμάτησε
pg_rewind: source and target cluster are on the same timeline
pg_rewind: no rewind required
standby: σε λειτουργία στη θύρα 5502
```

ΑΣΚΗΣΗ 19.5 Χαμένα commits

Στον 5501 δείξε αν υπάρχει η εγγραφή με id 20 στον `cities` και τι όνομα έχει.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id |  name
----+--------
 20 | Καβάλα
(1 row)
```

ΚΕΦΑΛΑΙΟ 20

Logical replication

Η physical replication αντιγράφει ολόκληρο το cluster σελίδα προς σελίδα και ο standby είναι ένα πιστό αντίγραφο που μόνο διαβάζεται. Η logical replication στέλνει αλλαγές γραμμών, «πρόσθεσε αυτή τη γραμμή στον `orders`», για όσους πίνακες διαλέξεις. Ο παραλήπτης είναι κανονικός server που δέχεται εγγραφές, μπορεί να έχει άλλους πίνακες και άλλη major έκδοση. Είναι το εργαλείο για αναβαθμίσεις χωρίς downtime και για βάσεις αναφορών.

Ο server του κεφαλαίου

Η logical replication χρειάζεται στο WAL περισσότερη πληροφορία από την physical, για να ξαναφτιάξει τις γραμμές. Το `wal_level` πρέπει να είναι `logical`, κάτι που θέλει restart. Ο δεύτερος server, ο `reporting`, είναι ανεξάρτητο cluster, όχι αντίγραφο του primary.

TERMINAL

```
pglab init primary 5501
echo "wal_level = logical" >> "$LAB/primary/postgresql.conf"
pglab start primary
pglab data 5501
pglab init reporting 5502
pglab start reporting
createdb -p 5502 --template=template0 --locale-provider=icu --icu-locale=el-GR \
  --locale=en_US.UTF-8 sqlbook
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
reporting: νέο cluster στο $LAB/reporting, θύρα 5502
reporting: σε λειτουργία στη θύρα 5502
```

Publication και subscription

Στον primary, τον **publisher**, μια **publication** λέει ποιοι πίνακες και ποιες αλλαγές στέλνονται.

SQL

```
CREATE PUBLICATION sales_pub FOR TABLE orders, order_items;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE PUBLICATION
```

Η logical replication δεν αντιγράφει το σχήμα. Οι πίνακες πρέπει να υπάρχουν στον **subscriber** με τις ίδιες στήλες. Τους φτιάχνουμε από ένα dump του σχήματος.

TERMINAL

```
pg_dump --schema-only -t orders -t order_items sqlbook | psql -q -p 5502 2>&1 | grep ERROR
psql -p 5502 -c "\dt"
```

ΕΞΟΔΟΣ

```
ERROR:  relation "public.products" does not exist
ERROR:  relation "public.customers" does not exist
ERROR:  relation "public.cities" does not exist
             List of tables
 Schema |    Name     | Type  |  Owner
--------+-------------+-------+----------
 public | order_items | table | postgres
 public | orders      | table | postgres
(2 rows)
```

Τα σφάλματα είναι τα foreign keys προς πίνακες που δεν αντιγράφουμε, τους πελάτες, τις πόλεις και τα προϊόντα. Το psql συνεχίζει μετά από κάθε σφάλμα, οπότε όλα τα υπόλοιπα, πίνακες, primary keys και το foreign key από τα items στις παραγγελίες, δημιουργήθηκαν κανονικά.

Στον subscriber η **subscription** συνδέεται στον publisher και ζητά μια ή περισσότερες publications.

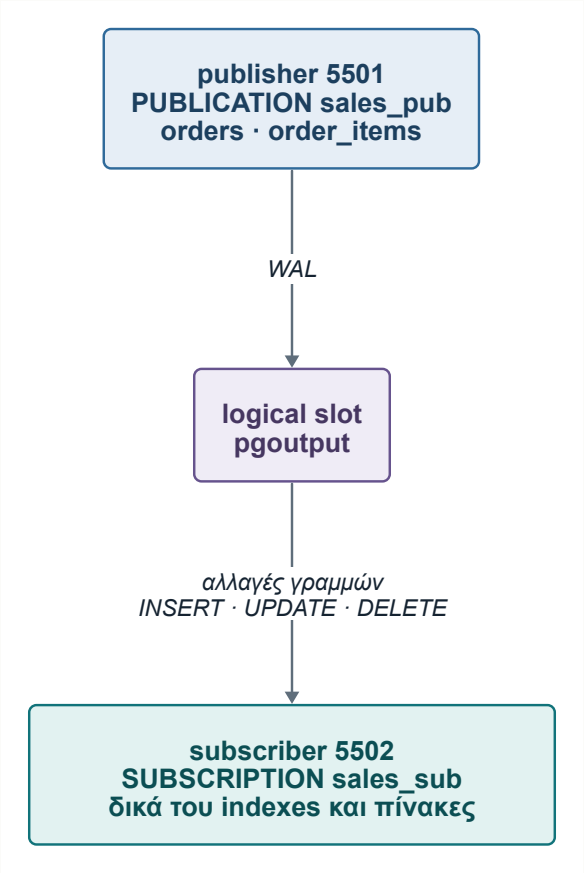
SQL · ΘΥΡΑ 5502

```
CREATE SUBSCRIPTION sales_sub
  CONNECTION 'host=localhost port=5501 user=postgres dbname=sqlbook'
  PUBLICATION sales_pub;
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
NOTICE:  created replication slot "sales_sub" on publisher
CREATE SUBSCRIPTION
```

Το μήνυμα λέει ότι στον publisher φτιάχτηκε ένα **logical replication slot**, με τον ίδιο ρόλο που είχαν τα slots του κεφαλαίου 17. Η subscription ξεκινά με αντιγραφή των υπαρχόντων δεδομένων κάθε πίνακα και μετά ακολουθεί τις αλλαγές.



Σχήμα 20.1 · Ο publisher αποκωδικοποιεί το WAL σε αλλαγές γραμμών για τους πίνακες της publication και τις στέλνει στον subscriber.

SQL · ΘΥΡΑ 5502

```
SELECT srrelid::regclass AS table_name, srsubstate AS state
FROM pg_subscription_rel ORDER BY 1;

SELECT (SELECT count(*) FROM orders) AS orders,
       (SELECT count(*) FROM order_items) AS items;
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
 table_name  | state
-------------+-------
 order_items | r
 orders      | r
(2 rows)

 orders | items
--------+-------
    162 |   310
(1 row)
```

Η κατάσταση `r`, ready, σημαίνει ότι η αρχική αντιγραφή τελείωσε. Τώρα κάθε αλλαγή στον primary φτάνει στον `reporting`.

SQL

```
INSERT INTO orders (customer_id, status, created_at) VALUES (5, 'pending', '2026-07-02 10:00');
UPDATE orders SET status = 'paid' WHERE id = 162;
DELETE FROM order_items WHERE order_id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 1

UPDATE 1

DELETE 1
```

SQL · ΘΥΡΑ 5502

```
SELECT id, status FROM orders WHERE id >= 162 ORDER BY id;
SELECT count(*) AS items_of_order_1 FROM order_items WHERE order_id = 1;
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
 id  | status
-----+---------
 162 | paid
 163 | pending
(2 rows)

 items_of_order_1
------------------
                0
(1 row)
```

Ο subscriber είναι κανονικός server

Ο `reporting` δέχεται εγγραφές. Μπορεί να έχει δικά του indexes για τις αναφορές του, δικούς του πίνακες και δικά του views. Αυτό δεν γίνεται σε έναν physical standby.

SQL · ΘΥΡΑ 5502

```
CREATE INDEX orders_created_idx ON orders (created_at);
CREATE TABLE daily_totals (day date PRIMARY KEY, orders integer);
INSERT INTO daily_totals
SELECT created_at::date, count(*) FROM orders GROUP BY 1;
SELECT count(*) AS days FROM daily_totals;
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
CREATE INDEX

CREATE TABLE

INSERT 0 135

 days
------
  135
(1 row)
```

Η ελευθερία αυτή έχει και κίνδυνο. Αν γράψεις στον subscriber σε πίνακα που αντιγράφεται, μπορεί να συγκρουστεί με αλλαγές που θα έρθουν.

Conflicts

Γράφουμε στον `reporting` μια παραγγελία με id 500 και μετά ο primary φτιάχνει τη δική του παραγγελία 500.

SQL · ΘΥΡΑ 5502

```
INSERT INTO orders (id, customer_id, status, created_at)
VALUES (500, 1, 'pending', '2026-07-03 09:00');
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
INSERT 0 1
```

SQL

```
INSERT INTO orders (id, customer_id, status, created_at)
VALUES (500, 2, 'paid', '2026-07-03 09:05');
INSERT INTO orders (customer_id, status, created_at) VALUES (6, 'pending', '2026-07-03 09:10');
```

ΑΠΟΤΕΛΕΣΜΑ

```
INSERT 0 1

INSERT 0 1
```

TERMINAL

```
sleep 2
grep -E 'conflict detected|duplicate key' "$LAB/reporting.log" | tail -n 2
```

ΕΞΟΔΟΣ

```
2026-09-28 09:50:13.529 EEST [81296] ERROR:  conflict detected on relation
"public.orders": conflict=insert_exists
2026-09-28 09:50:15.531 EEST [81338] ERROR:  conflict detected on relation
"public.orders": conflict=insert_exists
```

Ο worker που εφαρμόζει τις αλλαγές βρήκε ήδη γραμμή με το ίδιο primary key και σταμάτησε. Ξαναπροσπαθεί συνέχεια και αποτυγχάνει κάθε φορά και καμία επόμενη αλλαγή δεν περνά, ούτε η παραγγελία που ακολούθησε. Από την έκδοση 18 κάθε είδος conflict μετριέται χωριστά στο `pg_stat_subscription_stats`.

SQL · ΘΥΡΑ 5502

```
SELECT subname, apply_error_count, confl_insert_exists
FROM pg_stat_subscription_stats;
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
  subname  | apply_error_count | confl_insert_exists
-----------+-------------------+---------------------
 sales_sub |                 2 |                   2
(1 row)
```

Η λύση είναι να αποφασίσεις ποια γραμμή κρατάς. Εδώ σβήνουμε την τοπική και η αντιγραφή συνεχίζει από εκεί που σταμάτησε.

SQL · ΘΥΡΑ 5502

```
DELETE FROM orders WHERE id = 500;
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
DELETE 1
```

SQL · ΘΥΡΑ 5502

```
SELECT id, customer_id, status FROM orders
WHERE created_at >= '2026-07-03' ORDER BY created_at;
```

ΑΠΟΤΕΛΕΣΜΑ · ΘΥΡΑ 5502

```
 id  | customer_id | status
-----+-------------+---------
 500 |           2 | paid
 164 |           6 | pending
(2 rows)
```

ΠΑΓΙΔΑ

Ένα conflict σταματά όλη τη subscription και όσο είναι σταματημένη το slot του publisher κρατά WAL. Παρακολουθείς το `apply_error_count` και το log του subscriber. Και αποφεύγεις τις εγγραφές στον subscriber σε πίνακες που αντιγράφονται, εκτός αν τα ids δεν μπορούν να συγκρουστούν.

Replica identity

Για να εφαρμόσει ένα `UPDATE` ή ένα `DELETE`, ο subscriber πρέπει να βρει τη γραμμή. Χρησιμοποιεί το **replica identity** του πίνακα, από προεπιλογή το primary key. Ένας πίνακας χωρίς primary key δεν μπορεί να στείλει `UPDATE` και `DELETE`.

SQL

```
CREATE TABLE page_views (url text, viewed_at timestamptz);
INSERT INTO page_views VALUES ('/', now());
ALTER PUBLICATION sales_pub ADD TABLE page_views;
UPDATE page_views SET url = '/home';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 1

ALTER PUBLICATION

ERROR:  cannot update table "page_views" because it does not have a replica identity and publishes updates
HINT:  To enable updating the table, set REPLICA IDENTITY using ALTER TABLE.
```

Το `INSERT` πέρασε, το `UPDATE` όχι. Ο primary αρνείται να γράψει μια αλλαγή που ο subscriber δεν θα μπορεί να εφαρμόσει. Οι λύσεις είναι ένα primary key ή `ALTER TABLE ... REPLICA IDENTITY FULL`, που στέλνει ολόκληρη την παλιά γραμμή και ο subscriber την ψάχνει με όλες τις στήλες. Το `FULL` είναι αργό σε μεγάλους πίνακες χωρίς κατάλληλο index στον subscriber.

SQL

```
ALTER PUBLICATION sales_pub DROP TABLE page_views;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER PUBLICATION
```

Φίλτρα γραμμών και στηλών

Μια publication μπορεί να στέλνει μόνο κάποιες γραμμές με `WHERE` και μόνο κάποιες στήλες με λίστα στηλών. Έτσι ένας server αναφορών παίρνει μόνο όσα χρειάζεται και ένας πίνακας πελατών ταξιδεύει χωρίς email και τηλέφωνα.

SQL

```
CREATE PUBLICATION customers_pub
FOR TABLE customers (id, first_name, city_id, created_at) WHERE (newsletter);

SELECT pubname, tablename, attnames, rowfilter FROM pg_publication_tables
WHERE pubname = 'customers_pub';
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE PUBLICATION

    pubname    | tablename |              attnames              | rowfilter
---------------+-----------+------------------------------------+------------
 customers_pub | customers | {id,first_name,city_id,created_at} | newsletter
(1 row)
```

Το φίλτρο γραμμών σε μια publication που στέλνει `UPDATE` και `DELETE` μπορεί να αναφέρεται μόνο σε στήλες του replica identity ή σε στήλες που υπάρχουν στη λίστα, για τον ίδιο λόγο με πριν.

Αναβάθμιση με logical replication

Η logical replication δουλεύει ανάμεσα σε διαφορετικές major εκδόσεις. Γι' αυτό είναι ο τρόπος να αναβαθμίσεις μια μεγάλη βάση με διακοπή λίγων δευτερολέπτων.

1. Στήνεις τον νέο server με τη νέα έκδοση και το σχήμα από `pg_dump --schema-only`.
2. Φτιάχνεις publication για όλους τους πίνακες με `FOR ALL TABLES` και subscription στον νέο.
3. Περιμένεις την αρχική αντιγραφή και να μηδενιστεί το lag.
4. Σταματάς τις εγγραφές της εφαρμογής, περιμένεις να φτάσουν οι τελευταίες αλλαγές και ορίζεις τα sequences στον νέο server με `setval`, αφού οι τιμές των sequences δεν αντιγράφονται.
5. Γυρίζεις την εφαρμογή στον νέο server.

Τα κενά που πρέπει να θυμάσαι είναι λίγα. Το σχήμα, τα sequences και τα large objects δεν αντιγράφονται. Οι αλλαγές σχήματος πρέπει να γίνονται με το χέρι και στους δύο servers.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Η logical replication στέλνει αλλαγές γραμμών για συγκεκριμένους πίνακες. Θέλει `wal_level = logical`, publication στον publisher, subscription στον subscriber και τους πίνακες ήδη φτιαγμένους εκεί. Ο subscriber είναι κανονικός server που δέχεται εγγραφές και μια εγγραφή που συγκρούεται σταματά την αντιγραφή μέχρι να τη λύσεις. Τα `UPDATE` και `DELETE` χρειάζονται replica identity. Υπάρχουν φίλτρα γραμμών και στηλών. Σχήμα και sequences δεν αντιγράφονται.

Ασκήσεις

ΑΣΚΗΣΗ 20.1 Το slot της subscription

Δείξε στον primary τα slots με το όνομα, τον τύπο, το plugin και αν είναι ενεργά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 slot_name | slot_type |  plugin  | active
-----------+-----------+----------+--------
 sales_sub | logical   | pgoutput | t
(1 row)
```

ΑΣΚΗΣΗ 20.2 Νέος πίνακας στην publication

Πρόσθεσε τον `payments` στη `sales_pub`. Φτιάξε τον πίνακα στον `reporting` από dump του σχήματος χωρίς το foreign key, ανανέωσε τη subscription και μέτρησε τις πληρωμές εκεί.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
155
```

ΑΣΚΗΣΗ 20.3 Στήλη που λείπει

Πρόσθεσε στον primary στήλη `channel text` στον `orders` και μια παραγγελία με `channel = 'web'`. Δείξε το σφάλμα στο log του `reporting`, πρόσθεσε τη στήλη και εκεί και δείξε ότι η παραγγελία έφτασε.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
2026-09-28 09:50:25.555 EEST [81431] ERROR:  logical replication target relation
"public.orders" is missing replicated column: "channel"
web|1
```

ΑΣΚΗΣΗ 20.4 Παύση της subscription

Κάνε `DISABLE` τη subscription, δείξε στον primary αν το slot είναι ενεργό, κάνε την ξανά `ENABLE` και δείξε το slot ξανά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
sales_sub|f
sales_sub|t
```

ΑΣΚΗΣΗ 20.5 Κατάσταση της subscription

Στον `reporting` δείξε από το `pg_stat_subscription` τον worker που εφαρμόζει τις αλλαγές, το τελευταίο LSN που έλαβε και αν έχει pid.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  subname  | worker_type | running | receiving
-----------+-------------+---------+-----------
 sales_sub | apply       | t       | t
(1 row)
```

ΜΕΡΟΣ V

Συνδέσεις και αλλαγές σε production

Ένας server που αντέχει τον φόρτο μπορεί να γονατίσει από τις συνδέσεις ή από ένα `ALTER TABLE` στη λάθος στιγμή. Εδώ βάζουμε έναν connection pooler μπροστά από την PostgreSQL, αλλάζουμε σχήμα σε πίνακες που χρησιμοποιούνται και καθαρίζουμε bloat χωρίς να σταματήσουμε την εφαρμογή.

ΚΕΦΑΛΑΙΟ 21

Connection pooling με PgBouncer

Κάθε σύνδεση στην PostgreSQL είναι μια διεργασία, όπως είδαμε στο κεφάλαιο 6. Το άνοιγμά της κοστίζει χιλιοστά του δευτερολέπτου και η ύπαρξή της μνήμη. Μια εφαρμογή με δεκάδες servers και ένα pool συνδέσεων στον καθένα φτάνει εύκολα τις χιλιάδες συνδέσεις, από τις οποίες ελάχιστες δουλεύουν κάθε στιγμή. Ένας connection pooler όπως το PgBouncer μοιράζει λίγες πραγματικές συνδέσεις σε πολλούς clients.

Ο server του κεφαλαίου

Το PgBouncer είναι ξεχωριστό πρόγραμμα. Σε macOS εγκαθίσταται με `brew install pgbouncer` και σε Debian ή Ubuntu με `apt install pgbouncer`.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pgbench -i -s 5 -q >/dev/null 2>&1
pgbouncer --version | head -n 1
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
PgBouncer 1.26.0
```

Ρύθμιση

Το PgBouncer διαβάζει ένα αρχείο ini. Η ενότητα `[databases]` λέει σε ποιον server οδηγεί κάθε όνομα βάσης που ζητούν οι clients. Η ενότητα `[pgbouncer]` ρυθμίζει τον ίδιο τον pooler.

TERMINAL

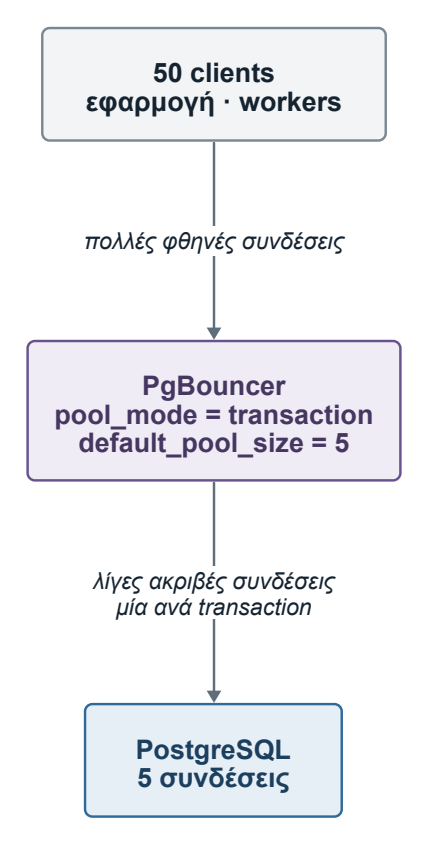
```
cat > "$LAB/pgbouncer.ini" <<EOF
[databases]
sqlbook = host=localhost port=5501 dbname=sqlbook
tiny = host=localhost port=5501 dbname=sqlbook pool_size=1

[pgbouncer]
listen_addr = localhost
```

```
listen_port = 6501
unix_socket_dir =
auth_type = trust
auth_file = $LAB/userlist.txt
admin_users = postgres
pool_mode = transaction
default_pool_size = 5
max_client_conn = 200
max_prepared_statements = 100
logfile = $LAB/pgbouncer.log
pidfile = $LAB/pgbouncer.pid
EOF
echo '"postgres" ""' > "$LAB/userlist.txt"
pgbouncer -d "$LAB/pgbouncer.ini"
sleep 1
tail -n 1 "$LAB/pgbouncer.log" | cut -c 1-80
```

ΕΞΟΔΟΣ

```
2026-09-28 09:50:39.200 EEST [81649] LOG process up: PgBouncer 1.26.0, libevent
```

Οι clients συνδέονται στη θύρα 6501 σαν να ήταν PostgreSQL. Το `sqlbook` και το `tiny` είναι δύο pools προς την ίδια βάση, το δεύτερο με μία μόνο σύνδεση, που θα χρησιμοποιήσουμε για ένα πείραμα. Το `default_pool_size` είναι πόσες συνδέσεις ανοίγει το PgBouncer προς τον server για κάθε ζεύγος βάσης και χρήστη και το `max_client_conn` πόσους clients δέχεται. Το `auth_file` έχει τους χρήστες που γνωρίζει. Με `auth_type = trust` δεν ελέγχει κωδικούς, όπως και ο server του εργαστηρίου. Το `-d` τρέχει το PgBouncer στο παρασκήνιο.



Σχήμα 21.1 · Οι clients ανοίγουν όσες συνδέσεις θέλουν στο PgBouncer. Ο server βλέπει μόνο τις λίγες συνδέσεις του pool.

Το κόστος μιας νέας σύνδεσης

Το `pgbench -C` ανοίγει νέα σύνδεση για κάθε transaction, όπως μια εφαρμογή χωρίς pool, για παράδειγμα ένα PHP script ή μια serverless function. Το τρέχουμε μία φορά κατευθείαν στον server και μία μέσω του PgBouncer.

TERMINAL

```
pgbench -n -C -c 4 -T 3 -p 5501 2>&1 | grep -E 'connection time|tps'
pgbench -n -C -c 4 -T 3 -p 6501 2>&1 | grep -E 'connection time|tps'
```

ΕΞΟΔΟΣ

```
average connection time = 2.262 ms
tps = 402.860409 (including reconnection times)
average connection time = 0.557 ms
tps = 1567.171777 (including reconnection times)
```

Μέσω του PgBouncer το transaction είναι πολλαπλάσια πιο γρήγορο. Μια νέα σύνδεση στο PgBouncer είναι φθηνή, αφού δεν ξεκινά διεργασία. Το PgBouncer δίνει στον client μια σύνδεση του server που είναι ήδη ανοιχτή.

Πολλοί clients, λίγες συνδέσεις

Πενήντα clients δουλεύουν ταυτόχρονα μέσω του PgBouncer. Η κονσόλα διαχείρισης είναι μια ψεύτικη βάση με όνομα `pgbouncer`, όπου το `SHOW POOLS` δείχνει την κατάσταση κάθε pool. Ο πίνακας έχει πολλές στήλες, οπότε ζητάμε από το psql έξοδο CSV, κρατάμε όσες μας ενδιαφέρουν με το `cut` και τις στοιχίζουμε με το `column`.

TERMINAL

```
pgbench -n -c 50 -T 3 -p 6501 >/dev/null 2>&1 &
sleep 1.5
psql -p 6501 -d pgbouncer --csv -c "SHOW POOLS" | cut -d, -f 1,3,4,7,10 | column -s, -t
psql -p 5501 -At -c "SELECT count(*) FROM pg_stat_activity
                     WHERE backend_type = 'client backend' AND application_name = 'pgbench'"
wait
```

ΕΞΟΔΟΣ

```
database   cl_active  cl_waiting  sv_active  sv_idle
pgbouncer  1          0           0          0
sqlbook    6          44          5          0
5
```

Από τους 50 clients του pool `sqlbook`, 5 έχουν σύνδεση και εκτελούν transaction, στη στήλη `cl_active`. Οι υπόλοιποι περιμένουν σειρά, στη στήλη `cl_waiting`. Ο server έχει μόνο 5 συνδέσεις του pgbench, `sv_active`. Η αναμονή στο PgBouncer είναι φθηνή και τα transactions του pgbench τελειώνουν σε κλάσματα του χιλιοστού, οπότε η σειρά προχωρά γρήγορα. Ο server δουλεύει με όσες συνδέσεις μπορεί να εξυπηρετήσει αποδοτικά, όχι με όσες ανοίγουν οι clients.

ΣΗΜΕΙΩΣΗ

Πόσες συνδέσεις χρειάζεται ο server; Λιγότερες απ' όσες νομίζεις. Αν τα queries είναι γρήγορα, μια σύνδεση που τρέχει κάτι χρησιμοποιεί έναν πυρήνα της CPU. Περισσότερες ενεργές συνδέσεις από δύο ή τρεις φορές τους πυρήνες απλώς περιμένουν η μία την άλλη, για CPU, για δίσκο ή για locks. Ένα pool λίγων δεκάδων συνδέσεων εξυπηρετεί χιλιάδες clients.

Τρεις τρόποι pooling

Το `pool_mode` λέει πότε μια σύνδεση του server επιστρέφει στο pool.

Mode	Η σύνδεση επιστρέφει	Τι δεν δουλεύει
`session`	όταν αποσυνδεθεί ο client	τίποτα, αλλά γλιτώνεις μόνο το κόστος της σύνδεσης
`transaction`	στο τέλος κάθε transaction	ό,τι κρατά κατάσταση στη σύνδεση πέρα από ένα transaction
`statement`	μετά από κάθε εντολή	transactions με πολλές εντολές

Το `transaction` είναι αυτό που δίνει τα μεγάλα οφέλη και αυτό που θέλει προσοχή. Ο client δεν έχει δική του σύνδεση. Κάθε transaction του μπορεί να τρέξει σε διαφορετική σύνδεση του server και κάθε σύνδεση του server εξυπηρετεί πολλούς clients στη σειρά.

Τι σπάει σε transaction mode

Ό,τι ζει στη σύνδεση και όχι στο transaction διαρρέει σε άλλους clients ή χάνεται. Το pool `tiny` έχει μία σύνδεση, οπότε δύο διαδοχικοί clients παίρνουν σίγουρα την ίδια. Ο πρώτος αλλάζει μια ρύθμιση με `SET`.

TERMINAL

```
psql -p 6501 -d tiny -c "SET work_mem = '999MB'"
psql -p 6501 -d tiny -c "SHOW work_mem"
```

ΕΞΟΔΟΣ

```
SET
 work_mem
----------
 999MB
(1 row)
```

Ο δεύτερος client, που δεν έκανε κανένα `SET`, βρήκε 999 MB `work_mem`. Σε μια πραγματική εφαρμογή αυτό σημαίνει ότι ένα `SET statement_timeout` ή ένα `SET search_path` ενός request επηρεάζει τυχαία άλλα requests. Η λύση είναι το `SET LOCAL` μέσα σε transaction, που χάνεται με το commit.

Τα υπόλοιπα που δεν δουλεύουν σε transaction mode είναι λίγα αλλά σημαντικά.

- Τα session advisory locks του πρώτου τόμου. Χρησιμοποιείς τα `pg_advisory_xact_lock`, που ελευθερώνονται στο τέλος του transaction.
- Το `LISTEN` του κεφαλαίου 4. Ο worker που ακούει συνδέεται απευθείας στον server.
- Προσωρινοί πίνακες που πρέπει να ζήσουν πέρα από ένα transaction και cursors `WITH HOLD`.
- Το `PREPARE` της SQL. Τα prepared statements του πρωτοκόλλου, αυτά που χρησιμοποιούν οι drivers, δουλεύουν από την έκδοση 1.21 του PgBouncer αν ορίσεις `max_prepared_statements`. Το PgBouncer θυμάται ποια statements έχει ετοιμάσει κάθε client και τα ετοιμάζει ξανά σε όποια σύνδεση του server χρειαστεί.

TERMINAL

```
pgbench -n -M prepared -c 4 -t 200 -p 6501 2>&1 | grep 'actually processed'
```

ΕΞΟΔΟΣ

```
number of transactions actually processed: 800/800
```

Το `-M prepared` κάνει το pgbench να χρησιμοποιεί prepared statements του πρωτοκόλλου. Όλα τα transactions πέρασαν, αν και κάθε client άλλαζε συνεχώς σύνδεση στον server.

Ποιον βλέπει ο server

Ο server βλέπει μόνο το PgBouncer. Στο `pg_stat_activity` κάθε σύνδεση έχει διεύθυνση του PgBouncer και ο πραγματικός client δεν φαίνεται. Το PgBouncer δείχνει τους δικούς του clients με `SHOW CLIENTS` και τις συνδέσεις προς τον server με `SHOW SERVERS`. Το `SHOW STATS` δίνει μετρητές ανά βάση.

TERMINAL

```
psql -p 6501 -d pgbouncer --csv -c "SHOW STATS" | cut -d, -f 1-4 | column -s, -t
```

ΕΞΟΔΟΣ

```
database   total_server_assignment_count  total_xact_count  total_query_count
pgbouncer  0                              2                 2
sqlbook    35876                          35876             251096
tiny       2                              2                 2
```

ΠΑΓΙΔΑ

Σε production το PgBouncer δεν έχει `auth_type = trust`. Με `scram-sha-256` και `auth_query` ρωτά την ίδια την PostgreSQL για τους κωδικούς, ώστε να μη χρειάζεται να συντηρείς αντίγραφο των χρηστών σε αρχείο. Και επειδή όλες οι συνδέσεις περνούν από αυτό, τρέχεις τουλάχιστον δύο, ή ένα σε κάθε server της εφαρμογής, ώστε να μην είναι το μοναδικό σημείο αποτυχίας.

ΚΡΑΤΑ ΑΥΤΟ

Κάθε σύνδεση στην PostgreSQL είναι διεργασία, οπότε πολλές συνδέσεις κοστίζουν και σπάνια βοηθούν. Το PgBouncer δέχεται χιλιάδες clients και τους μοιράζει σε λίγες συνδέσεις του server. Σε transaction mode κάθε transaction μπορεί να τρέξει σε άλλη σύνδεση, οπότε `SET`, session advisory locks, `LISTEN` και SQL `PREPARE` δεν δουλεύουν όπως περιμένεις. Τα prepared statements του πρωτοκόλλου θέλουν `max_prepared_statements`. Η κονσόλα `pgbouncer` με `SHOW POOLS` δείχνει ποιος περιμένει.

Ασκήσεις

ΑΣΚΗΣΗ 21.1 Pool σε session mode

Πρόσθεσε στο `pgbouncer.ini` ένα pool `sess` προς την ίδια βάση με `pool_mode = session`, ξαναφόρτωσε τις ρυθμίσεις με `RELOAD` από την κονσόλα και δείξε από το `SHOW DATABASES` το όνομα και το pool mode κάθε βάσης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
name       pool_mode
pgbouncer  statement
sess       session
sqlbook
tiny
```

ΑΣΚΗΣΗ 21.2 Η ρύθμιση που δεν διαρρέει

Μέσω του pool `tiny` άλλαξε το `work_mem` σε 64 MB με `SET LOCAL` μέσα σε transaction και δείξε από νέο client του ίδιου pool την τιμή του `work_mem`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
RESET
BEGIN
SET
 work_mem
----------
 64MB
(1 row)

COMMIT
4MB
```

ΑΣΚΗΣΗ 21.3 Συνδέσεις του PgBouncer στον server

Δείξε από τον server πόσες συνδέσεις έχει κάθε `application_name` και από ποια διεύθυνση, μόνο για client backends.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 application_name | client_addr | count
------------------+-------------+-------
 pgbench          | 127.0.0.1   |     3
 pgbench          | ::1         |     2
 psql             | ::1         |     1
(3 rows)
```

ΑΣΚΗΣΗ 21.4 Παύση για συντήρηση

Κάνε `PAUSE sqlbook` από την κονσόλα, ξεκίνα στο παρασκήνιο ένα query μέσω του PgBouncer, δείξε ότι περιμένει με το `SHOW POOLS` και κάνε `RESUME sqlbook`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
database  cl_active  cl_waiting
sqlbook   0          1
πέρασε
```

ΑΣΚΗΣΗ 21.5 Pool που δεν φτάνει

Τρέξε pgbench με 20 clients μέσω του pool `tiny` για δύο δευτερόλεπτα και δείξε τη μέση καθυστέρηση. Κάνε το ίδιο μέσω του `sqlbook`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
latency average = 5.844 ms
latency average = 1.977 ms
```

ΚΕΦΑΛΑΙΟ 22

Migrations χωρίς downtime

Ένα migration που τρέχει σε δευτερόλεπτα στο laptop μπορεί να σταματήσει την εφαρμογή για λεπτά σε production. Δεν φταίει το μέγεθος του πίνακα μόνο. Φταίει ποιο lock παίρνει η εντολή, πόσο το κρατά και ποιος περιμένει πίσω της. Σε αυτό το κεφάλαιο μαθαίνουμε να βλέπουμε το lock κάθε αλλαγής και να σπάμε κάθε επικίνδυνη αλλαγή σε βήματα που δεν μπλοκάρουν.

Ο server του κεφαλαίου

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pgbench -i -s 10 -q >/dev/null 2>&1
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
```

Η ουρά των locks

Στο κεφάλαιο 38 του πρώτου τόμου είδαμε ότι τα περισσότερα `ALTER TABLE` θέλουν `ACCESS EXCLUSIVE`, που συγκρούεται ακόμη και με το `SELECT`. Το χειρότερο όμως φαίνεται μόνο με τρεις συνδέσεις. Η Α διαβάζει τον πίνακα σε ένα transaction που δεν έχει κλείσει, όπως μια αναφορά ή ένα ξεχασμένο `BEGIN`.

SQL · ΣΥΝΕΔΡΙΑ Α

```
BEGIN;
SELECT count(*) FROM pgbench_branches;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Α

```
BEGIN

 count
-------
    10
(1 row)
```

Η Β είναι το migration. Περιμένει την Α.

SQL · ΣΥΝΕΔΡΙΑ Β

```
ALTER TABLE pgbench_branches ADD COLUMN region text;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ Β

```
-- η εντολή περιμένει
```

Η C είναι ένα απλό query της εφαρμογής. Δεν συγκρούεται με την Α, αφού και οι δύο μόνο διαβάζουν. Κι όμως περιμένει.

SQL · ΣΥΝΕΔΡΙΑ C

```
SELECT count(*) FROM pgbench_branches;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ C

```
-- η εντολή περιμένει
```

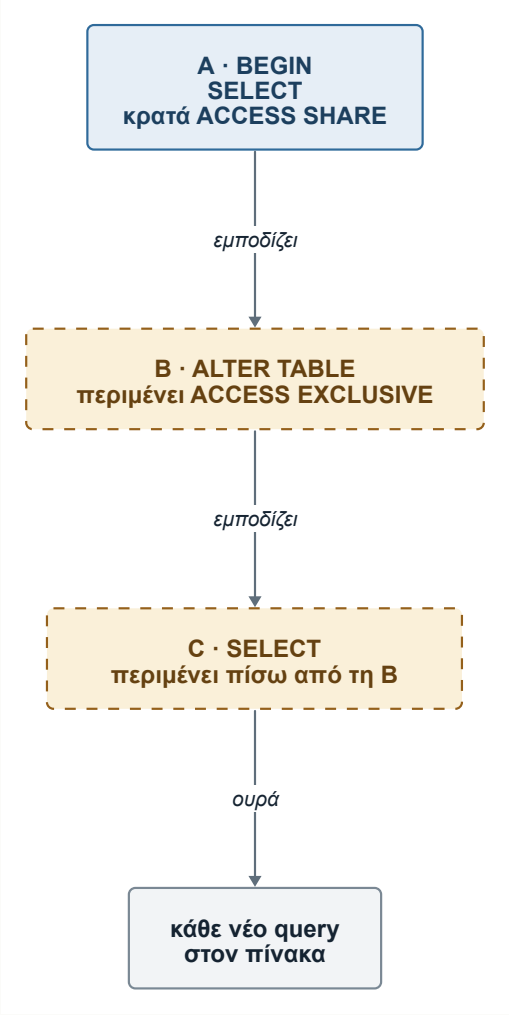
SQL · ΣΥΝΕΔΡΙΑ D

```
SELECT application_name, wait_event_type, pg_blocking_pids(pid) AS blocked_by,
       left(query, 45) AS query
FROM pg_stat_activity
WHERE application_name LIKE 'book-%' AND pid <> pg_backend_pid()
ORDER BY application_name;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ D

```
 application_name | wait_event_type | blocked_by |                     query
------------------+-----------------+------------+-----------------------------------------------
 book-a           | Client          | {}         | SELECT count(*) FROM pgbench_branches
 book-b           | Lock            | {83200}    | ALTER TABLE pgbench_branches ADD COLUMN regio
 book-c           | Lock            | {83201}    | SELECT count(*) FROM pgbench_branches
(3 rows)
```

Τα locks μπαίνουν σε ουρά. Η C ζητά lock που είναι συμβατό με της Α αλλά όχι με αυτό που περιμένει η Β και μπαίνει πίσω από τη Β. Κάθε νέο query στον πίνακα κάνει το ίδιο. Όσο η Α δεν τελειώνει, η εφαρμογή έχει σταματήσει σε αυτόν τον πίνακα. Τα requests της συσσωρεύονται μέχρι να εξαντληθούν οι συνδέσεις.



Σχήμα 22.1 · Η B περιμένει την A και όλα τα επόμενα queries περιμένουν τη B, ακόμη κι αν δεν συγκρούονται με την A.

SQL · ΣΥΝΕΔΡΙΑ A

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ A

```
COMMIT
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ B

```
ALTER TABLE
```

ΑΠΟΤΕΛΕΣΜΑ ΣΤΗ ΣΥΝΕΔΡΙΑ C

```
 count
-------
    10
(1 row)
```

lock_timeout και επανάληψη

Η άμυνα είναι το `lock_timeout`. Αν το migration δεν πάρει το lock του μέσα σε λίγο χρόνο, εγκαταλείπει. Η ουρά πίσω του ελευθερώνεται. Μετά ξαναδοκιμάζει. Επαναλαμβάνουμε το σενάριο με ένα script που δοκιμάζει τρεις φορές.

SQL · ΣΥΝΕΔΡΙΑ A

```
BEGIN;
SELECT count(*) FROM pgbench_branches;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ A

```
BEGIN

 count
-------
    10
(1 row)
```

TERMINAL

```
for attempt in 1 2 3; do
  if psql -q -c "SET lock_timeout = '500ms'" -c "ALTER TABLE pgbench_branches ADD COLUMN manager text"; then
    echo "έγινε στην προσπάθεια $attempt"; break
  fi
  echo "προσπάθεια $attempt απέτυχε"; sleep 1
done
```

ΕΞΟΔΟΣ

```
ERROR:  canceling statement due to lock timeout
προσπάθεια 1 απέτυχε
ERROR:  canceling statement due to lock timeout
προσπάθεια 2 απέτυχε
ERROR:  canceling statement due to lock timeout
προσπάθεια 3 απέτυχε
```

SQL · ΣΥΝΕΔΡΙΑ A

```
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ · ΣΥΝΕΔΡΙΑ A

```
COMMIT
```

Κάθε προσπάθεια μπλόκαρε την εφαρμογή για μισό δευτερόλεπτο το πολύ. Όσο η Α ήταν ανοιχτή, το migration απέτυχε. Σε production θα ξαναδοκίμαζε αργότερα ή θα ειδοποιούσε κάποιον. Τα εργαλεία migrations των frameworks δεν το κάνουν μόνα τους. Ορίζεις `lock_timeout` στη σύνδεση του migration και τη λογική επανάληψης εσύ.

Ποιο lock παίρνει μια εντολή

Δεν χρειάζεται να αποστηθίσεις πίνακες. Τρέχεις την εντολή μέσα σε transaction, κοιτάς το `pg_locks` και κάνεις `ROLLBACK`. Το ίδιο κόλπο δείχνει και αν η εντολή ξαναγράφει ολόκληρο τον πίνακα, αφού τότε αλλάζει το αρχείο του, το `relfilenode`.

SQL

```
CREATE FUNCTION lock_of(p_table regclass)
RETURNS text
LANGUAGE sql
RETURN (SELECT string_agg(mode, ', ') FROM pg_locks
        WHERE relation = p_table AND pid = pg_backend_pid());

SELECT pg_relation_filenode('pgbench_accounts') AS filenode_before;

BEGIN;
ALTER TABLE pgbench_accounts ADD COLUMN note text;
SELECT lock_of('pgbench_accounts') AS lock, pg_relation_filenode('pgbench_accounts') AS filenode;
ROLLBACK;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE FUNCTION

 filenode_before
-----------------
           16650
(1 row)

BEGIN

ALTER TABLE

        lock         | filenode
---------------------+----------
 AccessExclusiveLock |    16650
(1 row)

ROLLBACK
```

Το `ADD COLUMN` χωρίς default παίρνει `ACCESS EXCLUSIVE`, αλλά μόνο για μια στιγμή. Αλλάζει τον κατάλογο και όχι τα δεδομένα, οπότε το filenode μένει ίδιο. Με default σταθερή τιμή ισχύει το ίδιο από την έκδοση 11. Η τιμή αποθηκεύεται στον κατάλογο και οι παλιές γραμμές τη δείχνουν χωρίς να ξαναγραφτούν. Με default που αλλάζει σε κάθε γραμμή όμως, όπως το `clock_timestamp()`, ο πίνακας ξαναγράφεται.

SQL

```
BEGIN;
ALTER TABLE pgbench_accounts ADD COLUMN created timestamptz DEFAULT clock_timestamp();
SELECT lock_of('pgbench_accounts') AS lock, pg_relation_filenode('pgbench_accounts') AS filenode;
ROLLBACK;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

ALTER TABLE

              lock               | filenode
---------------------------------+----------
 ShareLock, AccessExclusiveLock |    16666
(1 row)

ROLLBACK
```

Νέο filenode, άρα ένα εκατομμύριο γραμμές ξαναγράφτηκαν με το `ACCESS EXCLUSIVE` κρατημένο όλη αυτή την ώρα. Σε πίνακα εκατοντάδων GB, αυτό είναι ώρες downtime. Το ίδιο κάνουν οι περισσότερες αλλαγές τύπου στήλης.

Αλλαγή	Lock	Ξαναγράφει τον πίνακα
`ADD COLUMN` χωρίς default ή με σταθερό default	`ACCESS EXCLUSIVE`, στιγμιαία	όχι
`ADD COLUMN` με volatile default	`ACCESS EXCLUSIVE`	ναι
`ALTER COLUMN TYPE`	`ACCESS EXCLUSIVE`	συνήθως ναι
`SET NOT NULL`	`ACCESS EXCLUSIVE`	όχι, αλλά σαρώνει όλο τον πίνακα
`ADD FOREIGN KEY`	`SHARE ROW EXCLUSIVE`	όχι, αλλά σαρώνει όλο τον πίνακα
`CREATE INDEX`	`SHARE`, μπλοκάρει εγγραφές	όχι
`CREATE INDEX CONCURRENTLY`	`SHARE UPDATE EXCLUSIVE`	όχι
`VALIDATE CONSTRAINT`	`SHARE UPDATE EXCLUSIVE`	όχι

Το `SHARE UPDATE EXCLUSIVE` είναι το lock που θέλεις. Επιτρέπει αναγνώσεις και εγγραφές και συγκρούεται μόνο με άλλες αλλαγές σχήματος και με το `VACUUM`.

Constraints σε δύο βήματα

Ένα `SET NOT NULL` ή ένα νέο foreign key πρέπει να ελέγξει κάθε γραμμή και το κάνει κρατώντας ισχυρό lock. Η λύση είναι να χωρίσεις την προσθήκη από τον έλεγχο. Με `NOT VALID` το constraint προστίθεται αμέσως, χωρίς έλεγχο των υπαρχουσών γραμμών, αλλά ισχύει για κάθε νέα γραμμή. Το `VALIDATE CONSTRAINT` ελέγχει μετά τις παλιές, με `SHARE UPDATE EXCLUSIVE`, ενώ η εφαρμογή δουλεύει.

Από την έκδοση 18 αυτό ισχύει και για το `NOT NULL`.

SQL

```
ALTER TABLE pgbench_accounts ADD CONSTRAINT filler_not_null NOT NULL filler NOT VALID;

BEGIN;
ALTER TABLE pgbench_accounts VALIDATE CONSTRAINT filler_not_null;
SELECT lock_of('pgbench_accounts') AS lock;
COMMIT;

SELECT conname, contype, convalidated FROM pg_constraint
WHERE conrelid = 'pgbench_accounts'::regclass AND conname = 'filler_not_null';
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

BEGIN

ALTER TABLE

           lock
--------------------------
 ShareUpdateExclusiveLock
(1 row)

COMMIT

     conname     | contype | convalidated
-----------------+---------+--------------
 filler_not_null | n       | t
(1 row)
```

Το ίδιο για ένα foreign key. Το `pgbench_accounts.bid` δείχνει στο `pgbench_branches`, αλλά δεν έχει foreign key.

SQL

```
ALTER TABLE pgbench_accounts
  ADD CONSTRAINT accounts_bid_fkey FOREIGN KEY (bid) REFERENCES pgbench_branches (bid) NOT VALID;

BEGIN;
ALTER TABLE pgbench_accounts VALIDATE CONSTRAINT accounts_bid_fkey;
SELECT lock_of('pgbench_accounts') AS accounts_lock, lock_of('pgbench_branches') AS branches_lock;
COMMIT;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

BEGIN

ALTER TABLE

               accounts_lock               |         branches_lock
-------------------------------------------+-------------------------------
 AccessShareLock, ShareUpdateExclusiveLock | AccessShareLock, RowShareLock
(1 row)

COMMIT
```

Κανένα από τα locks δεν μπλοκάρει εγγραφές. Ο πίνακας αναφοράς πήρε `RowShareLock`, το ίδιο lock που παίρνει ένα `SELECT ... FOR UPDATE`.

CREATE INDEX CONCURRENTLY

Ένα κανονικό `CREATE INDEX` μπλοκάρει κάθε εγγραφή στον πίνακα για όσο χτίζεται το index. Το `CONCURRENTLY` δεν μπλοκάρει, με τρία κόστη. Διαβάζει τον πίνακα δύο φορές, δεν τρέχει μέσα σε transaction και, αν αποτύχει, αφήνει πίσω του ένα index που δεν είναι έγκυρο.

Δοκιμάζουμε ένα unique index στο `bbalance` των branches. Όλα τα branches ξεκινούν με υπόλοιπο μηδέν, οπότε υπάρχουν διπλές τιμές.

SQL

```
CREATE UNIQUE INDEX CONCURRENTLY branches_balance_key ON pgbench_branches (bbalance);
```

ΑΠΟΤΕΛΕΣΜΑ

```
ERROR:  could not create unique index "branches_balance_key"
DETAIL:  Key (bbalance)=(0) is duplicated.
```

SQL

```
SELECT indexrelid::regclass AS index_name, indisvalid
FROM pg_index WHERE indrelid = 'pgbench_branches'::regclass;
```

ΑΠΟΤΕΛΕΣΜΑ

```
      index_name       | indisvalid
-----------------------+------------
 pgbench_branches_pkey | t
 branches_balance_key  | f
(2 rows)
```

Το index υπάρχει αλλά με `indisvalid = false`. Ο planner δεν το χρησιμοποιεί, όμως κάθε εγγραφή στον πίνακα το ενημερώνει. Ένα script migrations που απέτυχε έτσι και δεν το πρόσεξε κανείς αφήνει ένα άχρηστο index που κοστίζει. Το σβήνεις, διορθώνεις τα δεδομένα και ξαναδοκιμάζεις.

SQL

```
DROP INDEX CONCURRENTLY branches_balance_key;
```

ΑΠΟΤΕΛΕΣΜΑ

```
DROP INDEX
```

Expand και contract

Για αλλαγές που δεν γίνονται χωρίς rewrite, όπως μια αλλαγή τύπου ή ένα rename που χρησιμοποιεί η εφαρμογή, η τεχνική είναι να κάνεις την αλλαγή σε πολλά μικρά deploys, όπου κάθε έκδοση της εφαρμογής δουλεύει με το σχήμα πριν και μετά.

1. **Expand.** Προσθέτεις τη νέα στήλη, χωρίς default, στιγμιαία.
2. Η εφαρμογή γράφει και στις δύο στήλες, ή ένας trigger αντιγράφει την παλιά στη νέα σε κάθε `INSERT` και `UPDATE`.
3. **Backfill.** Γεμίζεις τη νέα στήλη για τις παλιές γραμμές σε μικρά κομμάτια, με commit μετά από κάθε κομμάτι.
4. Η εφαρμογή αρχίζει να διαβάζει από τη νέα στήλη.
5. **Contract.** Σταματάς να γράφεις στην παλιά και τη σβήνεις.

Το βήμα 3 είναι αυτό που χρειάζεται προσοχή. Ένα `UPDATE` ενός εκατομμυρίου γραμμών σε ένα transaction κρατά row locks σε όλες, γράφει τεράστιο WAL μονομιάς και αφήνει ένα εκατομμύριο νεκρές γραμμές. Μια procedure με `COMMIT` μέσα στον βρόχο το κάνει σε κομμάτια.

SQL

```
ALTER TABLE pgbench_accounts ADD COLUMN balance_cents bigint;

CREATE PROCEDURE backfill_cents(p_batch integer)
LANGUAGE plpgsql
AS $$
DECLARE
  last_id integer := 0;
  max_id integer;
BEGIN
  SELECT max(aid) INTO max_id FROM pgbench_accounts;
  WHILE last_id < max_id LOOP
    UPDATE pgbench_accounts SET balance_cents = abalance * 100
    WHERE aid > last_id AND aid <= last_id + p_batch;
    last_id := last_id + p_batch;
    COMMIT;
  END LOOP;
END;
$$;

CALL backfill_cents(100000);

SELECT count(*) FILTER (WHERE balance_cents IS NULL) AS missing FROM pgbench_accounts;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

CREATE PROCEDURE

CALL

 missing
---------
       0
(1 row)
```

Κάθε κομμάτι είναι δικό του transaction. Οι row locks κρατούν λίγο, ο autovacuum μπορεί να καθαρίζει ανάμεσα στα κομμάτια και οι replicas δεν δέχονται ένα τεράστιο κύμα WAL. Σε πραγματικό

σύστημα βάζεις και ένα μικρό `pg_sleep` ανάμεσα στα κομμάτια για να μην ανταγωνίζεσαι την εφαρμογή.

ΠΑΓΙΔΑ

Κάθε migration σε production τρέχει με `lock_timeout` και κάθε `CREATE INDEX` με `CONCURRENTLY`. Αν δεν ξέρεις ποιο lock παίρνει μια εντολή, το δοκιμάζεις σε transaction με `pg_locks` πριν τη βάλεις στο migration.

ΚΡΑΤΑ ΑΥΤΟ

Ένα `ALTER TABLE` που περιμένει lock μπλοκάρει κάθε query που έρχεται μετά από αυτό, οπότε κάθε migration θέλει `lock_timeout` και επανάληψη. Το lock και το rewrite μιας εντολής τα βλέπεις σε transaction με `pg_locks` και `relfilenode`. Constraints και `NOT NULL` προστίθενται με `NOT VALID` και ελέγχονται με `VALIDATE`, που δεν μπλοκάρει. Τα indexes χτίζονται με `CONCURRENTLY` και ελέγχεις το `indisvalid`. Οι μεγάλες αλλαγές γίνονται με expand, backfill σε κομμάτια και contract.

Ασκήσεις

ΑΣΚΗΣΗ 22.1 Το lock ενός index

Δείξε με τη `lock_of` ποιο lock παίρνει ένα κανονικό `CREATE INDEX` στο `pgbench_accounts(abalance)`, μέσα σε transaction που αναιρείται.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

   lock
-----------
 ShareLock
(1 row)
```

ΑΣΚΗΣΗ 22.2 Αλλαγή τύπου

Δείξε αν η αλλαγή του `pgbench_accounts.abalance` από `integer` σε `bigint` ξαναγράφει τον πίνακα, συγκρίνοντας το filenode πριν και μέσα στο transaction.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 before
--------
  16650
(1 row)

ALTER TABLE

 after |              lock
-------+--------------------------------
 16679 | ShareLock, AccessExclusiveLock
(1 row)
```

ΑΣΚΗΣΗ 22.3 Μεγαλύτερο varchar

Φτιάξε πίνακα `labels (code varchar(10))` με μία γραμμή και δείξε αν η αλλαγή σε `varchar(50)` και μετά σε `text` αλλάζει το filenode.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 1

 start
-------
 16683
(1 row)

ALTER TABLE

 after_varchar50
-----------------
           16683
(1 row)

ALTER TABLE

 after_text
------------
      16683
(1 row)
```

ΑΣΚΗΣΗ 22.4 Χωρίς ουρά

Φτιάξε `CHECK (abalance > -1000000)` στο `pgbench_accounts` ως `NOT VALID`, κάνε το `VALIDATE` και δείξε το `convalidated` πριν και μετά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

 convalidated
--------------
 f
(1 row)

ALTER TABLE

 convalidated
--------------
 t
(1 row)
```

ΑΣΚΗΣΗ 22.5 Άκυρα indexes

Βρες όλα τα indexes της βάσης που δεν είναι έγκυρα, με τον πίνακά τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 index_name | table_name
------------+------------
(0 rows)
```

ΚΕΦΑΛΑΙΟ 23

Bloat, REINDEX και pg_repack

Το VACUUM κάνει τον χώρο των νεκρών γραμμών διαθέσιμο για νέες γραμμές, αλλά σπάνια μικραίνει τα αρχεία. Μετά από ένα μαζικό DELETE ή μια περίοδο όπου ο autovacuum δεν προλάβαινε, ένας πίνακας μπορεί να πιάνει πολλαπλάσιο χώρο από όσο χρειάζεται. Το ίδιο ισχύει και για τα indexes του. Αυτό είναι το bloat. Εδώ το μετράμε και το αφαιρούμε με τρεις τρόπους, από τον πιο απλό και επικίνδυνο μέχρι αυτόν που δεν σταματά την εφαρμογή.

Ο server του κεφαλαίου

Το pg_repack είναι extension και εργαλείο γραμμής εντολών. Σε Debian ή Ubuntu εγκαθίσταται με το πακέτο postgresql-18-repack. Σε macOS χτίζεται από τον κώδικά του με make install, με το pg_config της PostgreSQL 18 στο PATH.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pgbench -i -s 10 -q >/dev/null 2>&1
psql -q -c "CREATE EXTENSION pgstattuple" -c "CREATE EXTENSION pg_repack"
pg_repack --version
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
pg_repack 1.5.3
```

Πώς φτιάχνεται bloat

Σβήνουμε τέσσερις στους πέντε λογαριασμούς, όπως θα έκανε μια εργασία που καθαρίζει παλιά δεδομένα. Μετά τρέχουμε VACUUM.

SQL

```
SELECT pg_size_pretty(pg_relation_size('pgbench_accounts')) AS table_size,
       pg_size_pretty(pg_relation_size('pgbench_accounts_pkey')) AS index_size;

DELETE FROM pgbench_accounts WHERE aid % 5 <> 0;
VACUUM pgbench_accounts;

SELECT pg_size_pretty(pg_relation_size('pgbench_accounts')) AS table_size,
       pg_size_pretty(pg_relation_size('pgbench_accounts_pkey')) AS index_size;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 table_size | index_size
------------+------------
 128 MB     | 21 MB
(1 row)

DELETE 800000

VACUUM

 table_size | index_size
------------+------------
 128 MB     | 21 MB
(1 row)
```

Τίποτα δεν μίκρυνε. Οι 200.000 λογαριασμοί που έμειναν είναι σκόρπιοι σε όλες τις σελίδες, ένας στους πέντε σε κάθε σελίδα. Το `pgstattuple` δείχνει πόσο κενός είναι ο πίνακας και το `pgstatindex` πόσο γεμάτες είναι οι σελίδες του index.

SQL

```
SELECT round(tuple_percent::numeric, 1) AS live_pct,
       round(free_percent::numeric, 1) AS free_pct
FROM pgstattuple('pgbench_accounts');

SELECT avg_leaf_density, leaf_pages FROM pgstatindex('pgbench_accounts_pkey');
```

ΑΠΟΤΕΛΕΣΜΑ

```
 live_pct | free_pct
----------+----------
     18.0 |     77.8
(1 row)

 avg_leaf_density | leaf_pages
------------------+------------
             18.2 |       2733
(1 row)
```

Οι ζωντανές γραμμές πιάνουν περίπου το ένα πέμπτο του πίνακα και ο υπόλοιπος χώρος είναι ελεύθερος. Τα φύλλα του index είναι γεμάτα κατά λιγότερο από ένα πέμπτο. Αν ο πίνακας θα ξαναγεμίσει σύντομα, αυτό δεν είναι πρόβλημα, ο χώρος θα ξαναχρησιμοποιηθεί. Αν όχι, κάθε sequential scan διαβάζει πέντε φορές περισσότερες σελίδες απ' όσο χρειάζεται. Η cache γεμίζει με κενό χώρο.

ΣΗΜΕΙΩΣΗ

Το `pgstattuple` διαβάζει ολόκληρο τον πίνακα. Σε πίνακα εκατοντάδων GB, το `pgstattuple_approx` δίνει εκτίμηση διαβάζοντας μόνο τις σελίδες που δεν είναι all visible στο visibility map.

Όταν το VACUUM μικραίνει τον πίνακα

Υπάρχει μία περίπτωση όπου το `VACUUM` επιστρέφει χώρο στο λειτουργικό. Αν οι σελίδες στο τέλος του αρχείου αδειάσουν εντελώς, τις κόβει. Ας σβήσουμε τους λογαριασμούς με τα μεγαλύτερα ids, που είναι στις τελευταίες σελίδες.

SQL

```
DELETE FROM pgbench_accounts WHERE aid > 800000;
VACUUM pgbench_accounts;
SELECT pg_size_pretty(pg_relation_size('pgbench_accounts')) AS table_size;
```

ΑΠΟΤΕΛΕΣΜΑ

```
DELETE 40000

VACUUM

 table_size
____________
 102 MB
(1 row)
```

Ο πίνακας μίκρυνε κατά ένα πέμπτο. Για να κόψει το τέλος, το `VACUUM` χρειάζεται για μια στιγμή `ACCESS EXCLUSIVE` lock. Αν δεν το πάρει αμέσως, παραιτείται από το κόψιμο αντί να περιμένει.

VACUUM FULL

Το `VACUUM FULL` γράφει έναν νέο πίνακα με μόνο τις ζωντανές γραμμές, ξαναχτίζει όλα τα indexes και σβήνει τα παλιά αρχεία. Είναι απλό και αποτελεσματικό, αλλά κρατά `ACCESS EXCLUSIVE` σε όλη τη διάρκεια. Κανείς δεν διαβάζει ή γράφει τον πίνακα μέχρι να τελειώσει.

SQL

```
CREATE TABLE accounts_copy AS SELECT * FROM pgbench_accounts;
DELETE FROM accounts_copy WHERE aid % 2 = 0;

SELECT pg_size_pretty(pg_relation_size('accounts_copy')) AS before;
VACUUM FULL accounts_copy;
SELECT pg_size_pretty(pg_relation_size('accounts_copy')) AS after;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SELECT 160000

DELETE 80000

 before
--------
 21 MB
(1 row)

VACUUM

 after
-------
 10 MB
(1 row)
```

Σε ένα αντίγραφο, σε μια βάση αναφορών ή σε παράθυρο συντήρησης, το `VACUUM FULL` είναι μια χαρά. Σε έναν πίνακα που χρησιμοποιεί η εφαρμογή, όχι.

REINDEX CONCURRENTLY

Συχνά ο πίνακας είναι εντάξει και μόνο τα indexes έχουν φουσκώσει. Ένα index B-tree που δέχεται πολλά `UPDATE` και `DELETE` χάνει σταδιακά την πυκνότητά του. Το `REINDEX INDEX CONCURRENTLY` χτίζει ένα νέο index δίπλα στο παλιό, με τον τρόπο του `CREATE INDEX CONCURRENTLY`. Τα αλλάζει με ένα σύντομο lock στο τέλος.

SQL

```
SELECT pg_size_pretty(pg_relation_size('pgbench_accounts_pkey')) AS before;
REINDEX INDEX CONCURRENTLY pgbench_accounts_pkey;
SELECT pg_size_pretty(pg_relation_size('pgbench_accounts_pkey')) AS after,
       avg_leaf_density
FROM pgstatindex('pgbench_accounts_pkey');
```

ΑΠΟΤΕΛΕΣΜΑ

```
 before
--------
 21 MB
(1 row)

REINDEX

  after  | avg_leaf_density
---------+------------------
 3536 kB |            89.87
(1 row)
```

Το index έγινε μικρότερο και η πυκνότητα των φύλλων του ξεπέρασε το 90. Ένα B-tree που χτίζεται από την αρχή γεμίζει κάθε φύλλο ως το `fillfactor` του, 90% από προεπιλογή.

pg_repack

Το `pg_repack` κάνει ό,τι το `VACUUM FULL` χωρίς να κρατά τον πίνακα κλειδωμένο. Φτιάχνει ένα αντίγραφο του πίνακα και έναν trigger που καταγράφει κάθε αλλαγή του αρχικού σε έναν πίνακα log. Γεμίζει το

αντίγραφο, χτίζει τα indexes του, εφαρμόζει όσες αλλαγές έγιναν στο μεταξύ και στο τέλος παίρνει για μια στιγμή `ACCESS EXCLUSIVE` για να αλλάξει το αρχικό με το αντίγραφο. Θέλει primary key ή unique index στον πίνακα.

Για να δούμε ότι η εφαρμογή δεν σταματά, τρέχουμε το pgbench όσο δουλεύει το `pg_repack`.

TERMINAL

```
psql -At -c "SELECT pg_size_pretty(pg_total_relation_size('pgbench_accounts'))"
pgbench -n -c 4 -T 6 2>&1 | grep -E 'processed|failed' > pgbench.out &
sleep 1
pg_repack -d sqlbook -t pgbench_accounts 2>&1
wait
cat pgbench.out
psql -At -c "SELECT pg_size_pretty(pg_total_relation_size('pgbench_accounts'))"
```

ΕΞΟΔΟΣ

```
106 MB
INFO: repacking table "public.pgbench_accounts"
number of transactions actually processed: 76454
number of failed transactions: 0 (0.000%)
24 MB
```

Ο πίνακας μαζί με τα indexes του μίκρυνε και το pgbench δεν είχε κανένα αποτυχημένο transaction. Το κόστος του `pg_repack` είναι χώρος και I/O. Χρειάζεται ελεύθερο χώρο για ένα πλήρες αντίγραφο του πίνακα και των indexes του και γράφει πολύ WAL, που φτάνει και στους standbys. Το `-x` ξαναχτίζει μόνο τα indexes και το `-j` χτίζει πολλά indexes παράλληλα.

ΠΑΓΙΔΑ

Και το `pg_repack` θέλει δύο φορές `ACCESS EXCLUSIVE` για λίγο, στην αρχή για να στήσει τον trigger και στο τέλος για την αλλαγή. Αν ένα μακρύ transaction κρατά τον πίνακα, το `pg_repack` περιμένει και η ουρά του κεφαλαίου 22 ξαναγυρίζει. Οι νεότερες εκδόσεις του σκοτώνουν όποιον το εμποδίζει μετά από ένα όριο, εκτός αν δώσεις `--no-kill-backend`. Τρέχε το σε ώρες με λίγη κίνηση.

Πρόληψη

Ο καλύτερος τρόπος να μην χρειάζεσαι όλα αυτά είναι να μη φτιάχνεις bloat.

- Ο autovacuum να προλαβαίνει, με τις ρυθμίσεις ανά πίνακα του κεφαλαίου 10.
- Κανένα transaction ανοιχτό για ώρες, αφού κρατά πίσω τον ορίζοντα του `VACUUM`.
- Τα μαζικά `DELETE` σε κομμάτια, όπως το backfill του κεφαλαίου 22, ώστε ο autovacuum να καθαρίζει ανάμεσα.
- Για δεδομένα που σβήνονται ανά περίοδο, partitioning. Ένα `DROP` ή `DETACH` ενός παλιού partition δεν αφήνει καθόλου bloat.
- Χαμηλότερο `fillfactor` για πίνακες με πολλά `UPDATE`, ώστε να γίνονται HOT updates.

ΚΡΑΤΑ ΑΥΤΟ

Μετά από `DELETE` ή `UPDATE` ο χώρος γίνεται ελεύθερος με το `VACUUM` αλλά το αρχείο δεν μικραίνει, εκτός από κενές σελίδες στο τέλος του. Το `pgstattuple` και το `pgstatindex` μετρούν το bloat. Το `VACUUM FULL` το αφαιρεί με `ACCESS EXCLUSIVE` για όλη τη διάρκεια, το `REINDEX CONCURRENTLY` αφαιρεί bloat indexes χωρίς να μπλοκάρει και το `pg_repack` ξαναγράφει πίνακα με σύντομα locks στην αρχή και στο τέλος. Η πρόληψη είναι ένας autovacuum που προλαβαίνει και partitioning για δεδομένα που σβήνονται μαζικά.

Ασκήσεις

ΑΣΚΗΣΗ 23.1 Πόσο γεμάτοι είναι οι tellers

Δείξε από το `pgstattuple` το μέγεθος του `pgbench_tellers`, τις ζωντανές και τις νεκρές γραμμές και το ποσοστό ελεύθερου χώρου.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 table_len | tuple_count | dead_tuple_count | free_pct
-----------+-------------+------------------+----------
     49152 |         100 |              251 |     62.7
(1 row)
```

ΑΣΚΗΣΗ 23.2 Index μετά το pg_repack

Δείξε το `avg_leaf_density` και τον αριθμό των φύλλων του `pgbench_accounts_pkey` μετά το `pg_repack`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 avg_leaf_density | leaf_pages
------------------+------------
            91.33 |        438
(1 row)
```

ΑΣΚΗΣΗ 23.3 Όλα τα indexes ενός πίνακα

Ξαναχτίσε με `REINDEX TABLE CONCURRENTLY` όλα τα indexes του `pgbench_branches` και δείξε μετά τα indexes του πίνακα με το μέγεθός τους.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
REINDEX

      index_name       | size
-----------------------+-------
 pgbench_branches_pkey | 16 kB
(1 row)
```

ΑΣΚΗΣΗ 23.4 Τι θα έκανε το pg_repack

Τρέξε το `pg_repack` με `--dry-run` για τον πίνακα `accounts_copy` και δείξε την έξοδο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
INFO: Dry run enabled, not executing repack
WARNING: relation "public.accounts_copy" must have a primary key or not-null unique keys
```

ΑΣΚΗΣΗ 23.5 Μέγεθος χωρίς bloat

Υπολόγισε για το `pgbench_accounts` πόσος χώρος θα χρειαζόταν αν ο πίνακας ήταν γεμάτος, από το `tuple_len` του `pgstattuple`. Σύγκρινέ τον με το πραγματικό μέγεθος.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 actual | live_data | live_pct
--------+-----------+----------
 21 MB  | 18 MB     |     88.6
(1 row)
```

ΜΕΡΟΣ VI

Extensions

Μεγάλο μέρος της δύναμης της PostgreSQL δεν βρίσκεται στον πυρήνα της αλλά στις extensions. Κλείνουμε με τρεις που αλλάζουν το τι μπορεί να κάνει μια βάση. Πίνακες που ζουν σε άλλον server ή σε αρχείο, αναζήτηση με ομοιότητα vectors για εφαρμογές AI και γεωγραφικά δεδομένα.

ΚΕΦΑΛΑΙΟ 24

Foreign data wrappers

Ένας foreign table μοιάζει με πίνακα, αλλά τα δεδομένα του βρίσκονται αλλού, σε άλλον PostgreSQL server, σε ένα αρχείο CSV ή σε μια εντελώς διαφορετική βάση. Το query τον διαβάζει σαν κανονικό πίνακα και ο **foreign data wrapper** μεταφράζει την ανάγνωση σε ό,τι καταλαβαίνει η πηγή. Είναι ο πιο απλός τρόπος να ενώσεις δεδομένα από δύο συστήματα χωρίς να τα αντιγράψεις.

Ο server του κεφαλαίου

Δύο servers. Ο `primary` με τη βάση του καταστήματος και ο `warehouse` με τη βάση της αποθήκης, όπου ένας πίνακας κρατά το απόθεμα κάθε προϊόντος σε κάθε κέντρο διανομής.

TERMINAL

```
pglab init primary 5501
pglab start primary
pglab data 5501
pglab init warehouse 5502
pglab start warehouse
createdb -p 5502 warehouse
psql -p 5502 -d warehouse -q <<'SQL'
CREATE TABLE stock (
  product_id integer NOT NULL,
  center     text    NOT NULL,
  qty        integer NOT NULL,
  PRIMARY KEY (product_id, center)
);
INSERT INTO stock
SELECT p, c, (p * 7 + length(c) * 3) % 25
FROM generate_series(1, 30) AS p, unnest(ARRAY['Αθήνα', 'Θεσσαλονίκη']) AS c;
SQL
psql -p 5502 -d warehouse -At -c "SELECT count(*), sum(qty) FROM stock"
```

ΕΞΟΔΟΣ

```
primary: νέο cluster στο $LAB/primary, θύρα 5501
primary: σε λειτουργία στη θύρα 5501
θύρα 5501: η βάση sqlbook είναι έτοιμη
warehouse: νέο cluster στο $LAB/warehouse, θύρα 5502
warehouse: σε λειτουργία στη θύρα 5502
60|725
```

postgres_fdw

Ο wrapper για άλλους PostgreSQL servers είναι το `postgres_fdw`, που έρχεται με την PostgreSQL. Χρειάζονται τρία αντικείμενα. Ένας **server**, που λέει πού βρίσκεται η απομακρυσμένη βάση, ένα **user mapping**, που λέει ως ποιος χρήστης συνδεόμαστε εκεί. Τρίτο είναι οι ίδιοι οι foreign tables.

SQL

```
CREATE EXTENSION postgres_fdw;

CREATE SERVER warehouse_srv
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host 'localhost', port '5502', dbname 'warehouse');

CREATE USER MAPPING FOR postgres
  SERVER warehouse_srv
  OPTIONS (user 'postgres');
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE EXTENSION

CREATE SERVER

CREATE USER MAPPING
```

Αντί να δηλώσουμε τις στήλες με το χέρι, το `IMPORT FOREIGN SCHEMA` διαβάζει τον ορισμό των πινάκων από τον απομακρυσμένο server. Τους βάζουμε σε δικό τους schema, ώστε να ξεχωρίζουν.

SQL

```
CREATE SCHEMA wh;
IMPORT FOREIGN SCHEMA public LIMIT TO (stock) FROM SERVER warehouse_srv INTO wh;

SELECT center, sum(qty) AS units FROM wh.stock GROUP BY center ORDER BY center;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE SCHEMA

IMPORT FOREIGN SCHEMA

   center    | units
-------------+-------
 Αθήνα       |   355
 Θεσσαλονίκη |   370
(2 rows)
```

Και ένα join ανάμεσα σε τοπικό και απομακρυσμένο πίνακα, με τα προϊόντα που έχουν συνολικό απόθεμα κάτω από 10 τεμάχια.

```sql
SELECT p.id, p.name, sum(s.qty) AS units
FROM products AS p
JOIN wh.stock AS s ON s.product_id = p.id
GROUP BY p.id, p.name
HAVING sum(s.qty) < 10
ORDER BY units, p.id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 id |       name        | units
----+-------------------+-------
  6 | Ruby από την αρχή |     7
 24 | Σετ μαχαιριών     |     9
(2 rows)
```

Τι στέλνεται στον απομακρυσμένο server

Η απόδοση ενός foreign table εξαρτάται από το τι θα εκτελέσει ο απομακρυσμένος server και τι θα γίνει τοπικά, αφού μεταφερθούν οι γραμμές. Το `postgres_fdw` προσπαθεί να στείλει όσο περισσότερα μπορεί, το λεγόμενο **pushdown**. Το `EXPLAIN (VERBOSE)` δείχνει το ακριβές query που στέλνει, στη γραμμή `Remote SQL`.

```sql
EXPLAIN (VERBOSE, COSTS OFF)
SELECT center, sum(qty) FROM wh.stock WHERE qty > 5 GROUP BY center;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Foreign Scan
  Output: center, (sum(qty))
  Relations: Aggregate on (wh.stock)
  Remote SQL: SELECT center, sum(qty) FROM public.stock WHERE ((qty > 5)) GROUP BY 1
```

Ολόκληρο το query, με το `WHERE` και το `GROUP BY`, εκτελέστηκε στον `warehouse`. Ήρθαν μόνο δύο γραμμές. Δεν μπορούν όμως να σταλούν όλα. Μια function που ο απομακρυσμένος server ίσως δεν έχει, ή που δεν είναι `IMMUTABLE`, εφαρμόζεται τοπικά.

```sql
EXPLAIN (VERBOSE, COSTS OFF)
SELECT product_id FROM wh.stock WHERE qty > random() * 20;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Foreign Scan on wh.stock
  Output: product_id
  Filter: ((stock.qty)::double precision > (random() * '20'::double precision))
  Remote SQL: SELECT product_id, qty FROM public.stock
```

Τώρα το `Remote SQL` ζητά όλες τις γραμμές και το `Filter` εφαρμόζεται τοπικά. Για 60 γραμμές δεν έχει σημασία. Για 60 εκατομμύρια σημαίνει ότι όλες ταξιδεύουν στο δίκτυο. Ένα join με τοπικό πίνακα έχει το ίδιο πρόβλημα, αφού ο απομακρυσμένος server δεν βλέπει τον τοπικό πίνακα.

SQL

```
EXPLAIN (VERBOSE, COSTS OFF)
SELECT p.name, s.qty
FROM products AS p JOIN wh.stock AS s ON s.product_id = p.id
WHERE p.category_id = 9;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Hash Join
  Output: p.name, s.qty
  Inner Unique: true
  Hash Cond: (s.product_id = p.id)
  ->  Foreign Scan on wh.stock s
        Output: s.product_id, s.center, s.qty
        Remote SQL: SELECT product_id, qty FROM public.stock
  ->  Hash
        Output: p.name, p.id
        ->  Seq Scan on public.products p
              Output: p.name, p.id
              Filter: (p.category_id = 9)
```

Ο planner φέρνει όλο τον `stock` και κάνει το join τοπικά, γιατί δεν έχει statistics για τον απομακρυσμένο πίνακα και δεν ξέρει ότι μερικές δεκάδες τιμές του `product_id` θα αρκούσαν. Με την επιλογή `use_remote_estimate` ρωτά τον απομακρυσμένο server για εκτιμήσεις, με ένα `ANALYZE` του foreign table κρατά τοπικά statistics. Μπορεί τότε να επιλέξει parameterized scan, ένα query ανά τιμή του join. Για joins ανάμεσα σε δύο foreign tables του ίδιου server, το `postgres_fdw` στέλνει ολόκληρο το join στον απομακρυσμένο server.

Εγγραφές

Ένας foreign table του `postgres_fdw` δέχεται και `INSERT`, `UPDATE` και `DELETE`.

SQL

```
UPDATE wh.stock SET qty = qty + 10 WHERE product_id = 30;
SELECT product_id, center, qty FROM wh.stock WHERE product_id = 30 ORDER BY center;
```

ΑΠΟΤΕΛΕΣΜΑ

```
UPDATE 2

 product_id |   center    | qty
------------+-------------+-----
         30 | Αθήνα       |  10
         30 | Θεσσαλονίκη |  28
(2 rows)
```

ΠΑΓΙΔΑ

Ένα transaction που γράφει τοπικά και σε foreign table δεν είναι ατομικό ανάμεσα στους δύο servers. Το `postgres_fdw` κάνει commit στον απομακρυσμένο server λίγο πριν από το τοπικό commit. Αν το τοπικό αποτύχει μετά, η απομακρυσμένη αλλαγή έχει ήδη γίνει. Για μεταφορές που πρέπει να είναι σωστές και στα δύο μέρη, χρειάζεσαι σχεδιασμό σε επίπεδο εφαρμογής, όχι ένα join.

file_fdw

Το `file_fdw` διαβάζει ένα αρχείο του server σαν πίνακα. Κάθε query ξαναδιαβάζει το αρχείο, οπότε ένα CSV που ανανεώνει ένα άλλο πρόγραμμα φαίνεται πάντα ενημερωμένο. Φτιάχνουμε ένα CSV με τιμές ενός προμηθευτή.

TERMINAL

```
cat > "$LAB/supplier_prices.csv" <<'CSV'
sku,supplier_price
BK-LIT-001,9.80
BK-LIT-002,11.40
BK-PRG-001,31.00
EL-KB-001,61.50
CSV
psql -q -c "CREATE EXTENSION file_fdw" \
        -c "CREATE SERVER files FOREIGN DATA WRAPPER file_fdw" \
        -c "CREATE FOREIGN TABLE supplier_prices (sku text, supplier_price numeric)
            SERVER files OPTIONS (filename '$LAB/supplier_prices.csv', format 'csv', header 'true')"
```

Η διαδρομή του αρχείου είναι απόλυτη και αφορά τον server, όχι τον client. Γι’ αυτό το `file_fdw` χρειάζεται δικαιώματα superuser ή τον ρόλο `pg_read_server_files`.

SQL

```
SELECT sp.sku, p.name, p.cost, sp.supplier_price
FROM supplier_prices AS sp
LEFT JOIN products AS p ON p.sku = sp.sku
ORDER BY sp.sku;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    sku     |                name                | cost  | supplier_price
------------+------------------------------------+-------+----------------
 BK-LIT-001 | Βίος και πολιτεία του Αλέξη Ζορμπά |  9.10 |           9.80
 BK-LIT-002 | Το Κιβώτιο                         |  7.80 |          11.40
 BK-PRG-001 | Μαθαίνω Python                     | 17.50 |          31.00
 EL-KB-001  | ∅                                  |     ∅ |          61.50
(4 rows)
```

Ο κωδικός `EL-KB-001` δεν υπάρχει στα προϊόντα μας, οπότε το `LEFT JOIN` δίνει κενές στήλες. Αυτό το μοτίβο, ένα αρχείο ως πίνακας και ένα join για να δεις τι ταιριάζει, είναι ο πιο γρήγορος τρόπος να ελέγξεις ένα αρχείο πριν το φορτώσεις.

ΣΗΜΕΙΩΣΗ

Υπάρχουν wrappers για πολλά άλλα συστήματα, όπως MySQL, SQLite, MongoDB, Redis και αρχεία Parquet. Δεν έρχονται με την PostgreSQL και η ποιότητά τους διαφέρει. Για ad hoc queries σε άλλη PostgreSQL υπάρχει και το παλαιότερο `dblink`, που εκτελεί ένα query ως κείμενο και επιστρέφει γραμμές.

ΚΡΑΤΑ ΑΥΤΟ

Ένας foreign data wrapper κάνει δεδομένα άλλου συστήματος να φαίνονται ως πίνακες. Για άλλη PostgreSQL χρειάζεσαι `postgres_fdw`, server, user mapping και foreign tables, συνήθως με `IMPORT FOREIGN SCHEMA`. Το `EXPLAIN (VERBOSE)` δείχνει στο `Remote SQL` τι στέλνεται. Ό,τι δεν στέλνεται μεταφέρεται ολόκληρο και εφαρμόζεται τοπικά. Οι εγγραφές σε δύο servers δεν είναι ατομικές. Το `file_fdw` διαβάζει αρχεία του server σαν πίνακες.

Ασκήσεις

ΑΣΚΗΣΗ 24.1 Pushdown ενός LIMIT

Δείξε με `EXPLAIN (VERBOSE, COSTS OFF)` το `Remote SQL` για τα τρία μεγαλύτερα αποθέματα της Θεσσαλονίκης.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Foreign Scan on wh.stock
  Output: product_id, qty
  Remote SQL: SELECT product_id, qty FROM public.stock WHERE ((center = 'Θεσσαλονίκη')) ORDER BY qty DESC NULLS FIRST LIMIT 3::bigint
```

ΑΣΚΗΣΗ 24.2 Απόθεμα ανά κατηγορία

Δείξε για κάθε κατηγορία το συνολικό απόθεμα στα δύο κέντρα, με join ανάμεσα στα τοπικά `products` και `categories` και στο απομακρυσμένο `wh.stock`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
     category      | units
-------------------+-------
 Laptops           |   127
 Ήχος              |   122
 Περιφερειακά      |   110
 Λογοτεχνία        |   107
 Γραφείο           |    86
 Κουζίνα           |    60
 Βάσεις δεδομένων  |    59
 Προγραμματισμός   |    46
 Τεχνολογία        |    13
(9 rows)
```

ΑΣΚΗΣΗ 24.3 Νέο κέντρο διανομής

Πρόσθεσε μέσω του foreign table απόθεμα 5 τεμαχίων στο κέντρο `Πάτρα` για τα προϊόντα 1 έως 3 και μέτρησε από τον `warehouse` πόσες γραμμές έχει το νέο κέντρο.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
3
```

ΑΣΚΗΣΗ 24.4 Μέγεθος παρτίδας

Όρισε στον server `warehouse_srv` το `fetch_size` σε 500 και δείξε τις επιλογές του server από το `pg_foreign_server`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
ALTER SERVER

    srvname    |                          srvoptions
---------------+-------------------------------------------------------------
 warehouse_srv | {host=localhost,port=5502,dbname=warehouse,fetch_size=500}
(1 row)
```

ΑΣΚΗΣΗ 24.5 Το CSV αλλάζει

Πρόσθεσε μια γραμμή στο `supplier_prices.csv` και μέτρησε ξανά τις γραμμές του foreign table.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
5
```

ΚΕΦΑΛΑΙΟ 25

pgvector

Οι εφαρμογές με γλωσσικά μοντέλα αναζητούν κείμενα με βάση το νόημα, όχι τις λέξεις. Ένα μοντέλο μετατρέπει κάθε κείμενο σε ένα **embedding**, μια λίστα εκατοντάδων αριθμών, έτσι ώστε κείμενα με παρόμοιο νόημα να δίνουν κοντινά σημεία στον χώρο. Η αναζήτηση γίνεται «βρες τα σημεία που είναι πιο κοντά σε αυτό». Η επέκταση pgvector κάνει αυτή την αναζήτηση μέσα στην PostgreSQL, δίπλα στα υπόλοιπα δεδομένα σου.

Ο τύπος vector

Η επέκταση λέγεται `vector` και προσθέτει έναν τύπο με το ίδιο όνομα. Σε macOS εγκαθίσταται με `brew install pgvector` και σε Debian ή Ubuntu με το πακέτο `postgresql-18-pgvector`.

```sql
CREATE EXTENSION vector;
SELECT extversion FROM pg_extension WHERE extname = 'vector';
```

```
CREATE EXTENSION

 extversion
------------
 0.8.6
(1 row)
```

Τα embeddings τα φτιάχνει ένα μοντέλο έξω από τη βάση, για παράδειγμα ένα API ή ένα τοπικό μοντέλο. Η εφαρμογή τα αποθηκεύει. Ένα πραγματικό embedding έχει 384, 768 ή 1.536 διαστάσεις. Για να δούμε πώς δουλεύει ο τύπος ξεκινάμε με τρεις, που μπορούμε να τις καταλάβουμε. Δίνουμε με το χέρι σε μερικά προϊόντα τρεις βαθμούς, πόσο είναι τεχνολογία, κουζίνα και βιβλίο.

SQL

```
CREATE TABLE product_vectors (
  product_id integer PRIMARY KEY REFERENCES products (id),
  features   vector(3) NOT NULL
);

INSERT INTO product_vectors VALUES
  (1,  '[0.0, 0.0, 1.0]'),
  (5,  '[0.6, 0.0, 0.8]'),
  (7,  '[0.7, 0.0, 0.7]'),
  (13, '[1.0, 0.0, 0.0]'),
  (15, '[0.9, 0.1, 0.0]'),
  (22, '[0.1, 1.0, 0.0]'),
  (24, '[0.0, 0.9, 0.0]');

SELECT pv.product_id, p.name, pv.features
FROM product_vectors AS pv JOIN products AS p ON p.id = pv.product_id
ORDER BY pv.product_id;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 7

 product_id |                name                |  features
------------+------------------------------------+-------------
          1 | Βίος και πολιτεία του Αλέξη Ζορμπά | [0,0,1]
          5 | Μαθαίνω Python                     | [0.6,0,0.8]
          7 | Καθαρός κώδικας στην πράξη         | [0.7,0,0.7]
         13 | Laptop Student 15                  | [1,0,0]
         15 | Μηχανικό πληκτρολόγιο              | [0.9,0.1,0]
         22 | Καφετιέρα espresso                 | [0.1,1,0]
         24 | Σετ μαχαιριών                      | [0,0.9,0]
(7 rows)
```

Αποστάσεις

Η επέκταση έχει τρεις τελεστές απόστασης. Όσο μικρότερη η τιμή, τόσο πιο κοντά.

Τελεστής	Απόσταση	Πότε
`<->`	ευκλείδεια, η απόσταση σε ευθεία γραμμή	όταν μετράει και το μήκος του vector
`<=>`	συνημιτόνου, 1 μείον το συνημίτονο της γωνίας	η συνηθισμένη για embeddings κειμένου
`<#>`	αρνητικό εσωτερικό γινόμενο	για vectors με μήκος 1, ίδια σειρά με τη `<=>` και γρηγορότερη

Ποια προϊόντα μοιάζουν περισσότερο με ένα που είναι κυρίως τεχνολογία και λίγο βιβλίο; Το vector της ερώτησης γράφεται ως κείμενο και μετατρέπεται σε `vector`.

SQL

```
SELECT pv.product_id, p.name,
       round((pv.features <=> '[0.8, 0.0, 0.5]')::numeric, 3) AS distance
FROM product_vectors AS pv JOIN products AS p ON p.id = pv.product_id
ORDER BY pv.features <=> '[0.8, 0.0, 0.5]'
LIMIT 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 product_id |            name            | distance
------------+----------------------------+----------
          7 | Καθαρός κώδικας στην πράξη |    0.026
          5 | Μαθαίνω Python             |    0.067
         13 | Laptop Student 15          |    0.152
(3 rows)
```

Τα δύο πιο κοντινά είναι τα βιβλία προγραμματισμού, που έχουν και τα δύο χαρακτηριστικά. Ακολουθεί το laptop. Αυτό είναι ολόκληρη η ιδέα της σημασιολογικής αναζήτησης. Η μόνη διαφορά σε μια πραγματική εφαρμογή είναι ότι οι διαστάσεις είναι εκατοντάδες και δεν έχουν όνομα.

Ένα μεγαλύτερο σύνολο

Για να μετρήσουμε ταχύτητα και ακρίβεια χρειαζόμαστε πολλά vectors. Χωρίς μοντέλο, φτιάχνουμε συνθετικά. Δέκα κατηγορίες, καθεμία με ένα τυχαίο κέντρο στις 128 διαστάσεις. Γύρω από τα κέντρα φτιάχνουμε 50.000 αντικείμενα γύρω από τα κέντρα, με πολύ θόρυβο. Μοιάζει με ό,τι δίνει ένα πραγματικό μοντέλο, όπου τα κείμενα ενός θέματος μαζεύονται σε μια περιοχή του χώρου. Το `setseed` κάνει τους τυχαίους αριθμούς ίδιους σε κάθε εκτέλεση.

SQL

```
SELECT setseed(0.42);

CREATE TABLE centers AS
SELECT c AS category,
       ARRAY(SELECT random() * 2 - 1 FROM generate_series(1, 128) WHERE c > 0) AS v
FROM generate_series(1, 10) AS c;

CREATE TABLE items AS
SELECT i AS id, c.category,
       (SELECT array_agg(c.v[d] + (random() - 0.5) * 4 ORDER BY d)
        FROM generate_series(1, 128) AS d WHERE i > 0)::vector(128) AS embedding
FROM generate_series(1, 50000) AS i
JOIN centers AS c ON c.category = 1 + i % 10;

ALTER TABLE items ADD PRIMARY KEY (id);
```

ΑΠΟΤΕΛΕΣΜΑ

```
 setseed
---------

(1 row)

SELECT 10

SELECT 50000

ALTER TABLE
```

Τα `WHERE c > 0` και `WHERE i > 0` είναι πάντα αληθή. Κάνουν όμως το subquery να εξαρτάται από την εξωτερική γραμμή, ώστε η PostgreSQL να το εκτελεί ξανά για κάθε γραμμή, με νέους τυχαίους αριθμούς, αντί να το υπολογίσει μία φορά.

Χωρίς index η αναζήτηση υπολογίζει την απόσταση από κάθε γραμμή και κρατά τις δέκα μικρότερες. Είναι ακριβής, αλλά διαβάζει ολόκληρο τον πίνακα.

SQL

```
EXPLAIN (ANALYZE, COSTS OFF, BUFFERS OFF)
SELECT id FROM items
ORDER BY embedding <=> (SELECT embedding FROM items WHERE id = 7)
LIMIT 10;
```

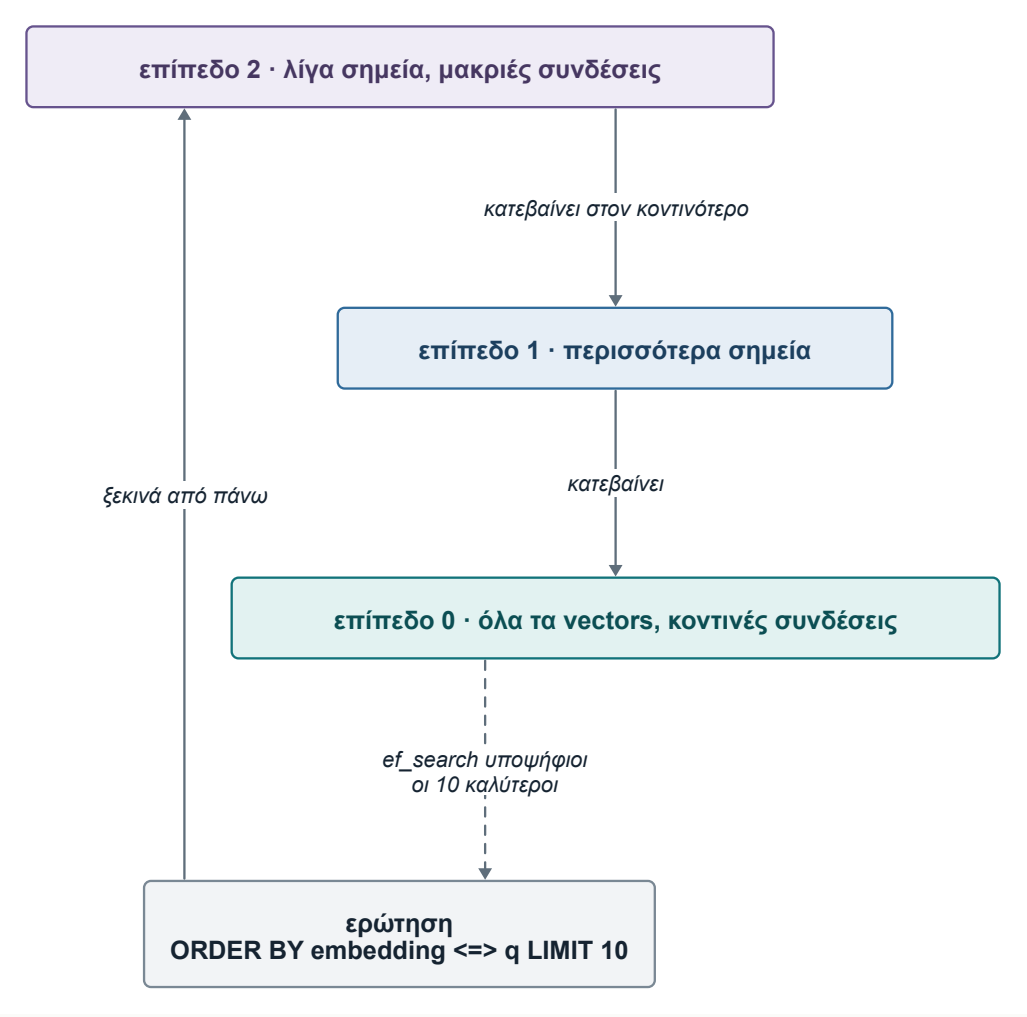
ΑΠΟΤΕΛΕΣΜΑ

```
Limit (actual time=10.892..10.893 rows=10.00 loops=1)
  InitPlan 1
    ->  Index Scan using items_pkey on items items_1 (actual time=0.015..0.016 rows=1.00 loops=1)
          Index Cond: (id = 7)
          Index Searches: 1
  ->  Sort (actual time=10.892..10.892 rows=10.00 loops=1)
        Sort Key: ((items.embedding <=> (InitPlan 1).col1))
        Sort Method: top-N heapsort  Memory: 25kB
        ->  Seq Scan on items (actual time=0.020..9.596 rows=50000.00 loops=1)
Planning Time: 0.111 ms
Execution Time: 10.905 ms
```

HNSW

Ένα B-tree δεν βοηθά εδώ, αφού δεν υπάρχει σειρά ανάμεσα σε vectors. Το pgvector έχει δύο είδη index για **approximate nearest neighbor** αναζήτηση, που βρίσκουν πολύ γρήγορα τους περισσότερους από τους πιο κοντινούς γείτονες, χωρίς εγγύηση ότι θα βρουν όλους.

Το **HNSW** είναι ένας γράφος σε επίπεδα. Κάθε vector συνδέεται με μερικούς κοντινούς του. Τα πάνω επίπεδα έχουν λίγα σημεία και μακριές συνδέσεις, τα κάτω όλα τα σημεία και κοντινές συνδέσεις. Η αναζήτηση ξεκινά από πάνω και κατεβαίνει, πηγαίνοντας κάθε φορά προς τον γείτονα που είναι πιο κοντά στην ερώτηση. Η κλάση τελεστών ορίζει την απόσταση και πρέπει να ταιριάζει με τον τελεστή του query.



Σχήμα 25.1 · Η αναζήτηση στο HNSW ξεκινά από το αραιό πάνω επίπεδο και κατεβαίνει προς τους κοντινότερους γείτονες ως το κάτω επίπεδο.

SQL

```
CREATE INDEX items_embedding_hnsw ON items USING hnsw (embedding vector_cosine_ops);

SELECT pg_size_pretty(pg_relation_size('items')) AS table_size,
       pg_size_pretty(pg_relation_size('items_embedding_hnsw')) AS index_size;

EXPLAIN (ANALYZE, COSTS OFF, BUFFERS OFF)
SELECT id FROM items
ORDER BY embedding <=> (SELECT embedding FROM items WHERE id = 7)
LIMIT 10;
```

```
ΑΠΟΤΕΛΕΣΜΑ

CREATE INDEX

 table_size | index_size
------------+------------
 28 MB      | 40 MB
(1 row)

Limit (actual time=0.565..0.572 rows=10.00 loops=1)
  InitPlan 1
    ->  Index Scan using items_pkey on items items_1 (actual time=0.003..0.003 rows=1.00 loops=1)
          Index Cond: (id = 7)
          Index Searches: 1
  ->  Index Scan using items_embedding_hnsw on items (actual time=0.564..0.571 rows=10.00 loops=1)
        Order By: (embedding <=> (InitPlan 1).col1)
        Index Searches: 1
Planning Time: 0.117 ms
Execution Time: 0.580 ms
```

Το `Index Scan` με `Order By` διάβασε ένα μικρό κομμάτι του γράφου αντί για όλο τον πίνακα. Το index είναι μεγαλύτερο από τον ίδιο τον πίνακα. Αυτό είναι συνηθισμένο για HNSW και για τον λόγο αυτό τα indexes vectors θέλουν αρκετή μνήμη ώστε να χωρούν στο `shared_buffers`. Το χτίσιμο είναι αργό σε μεγάλους πίνακες. Βοηθά ένα μεγάλο `maintenance_work_mem` και τα parallel workers του `max_parallel_maintenance_workers`.

Ακρίβεια

Πόσους από τους πραγματικούς δέκα κοντινότερους βρίσκει το index; Το μέτρο λέγεται **recall**. Το υπολογίζουμε για 20 ερωτήσεις. Πρώτα η ακριβής απάντηση, με το index απενεργοποιημένο για ένα transaction. Μετά η απάντηση με το index.

```sql
SQL

BEGIN;
SET LOCAL enable_indexscan = off;
CREATE TABLE exact_top AS
SELECT q.id AS query_id, n.id
FROM items AS q
CROSS JOIN LATERAL (SELECT id FROM items ORDER BY embedding <=> q.embedding LIMIT 10) AS n
WHERE q.id <= 20;
COMMIT;

CREATE FUNCTION recall_at_10()
RETURNS numeric
LANGUAGE sql
AS $$
  SELECT round(100.0 * count(*) / 200, 1)
  FROM items AS q
  CROSS JOIN LATERAL (SELECT id FROM items ORDER BY embedding <=> q.embedding LIMIT 10) AS n
  JOIN exact_top AS e ON e.query_id = q.id AND e.id = n.id
  WHERE q.id <= 20;
$$;

SELECT recall_at_10() AS recall_pct;
```

ΑΠΟΤΕΛΕΣΜΑ

```
BEGIN

SET

SELECT 200

COMMIT

CREATE FUNCTION

 recall_pct
------------
       88.5
(1 row)
```

Η ρύθμιση `hnsw.ef_search` ορίζει πόσους υποψήφιους κρατά η αναζήτηση στο κάτω επίπεδο, 40 από προεπιλογή. Μικρότερη τιμή είναι γρηγορότερη και λιγότερο ακριβής.

SQL

```
SET hnsw.ef_search = 10;
SELECT recall_at_10() AS recall_ef10;
SET hnsw.ef_search = 100;
SELECT recall_at_10() AS recall_ef100;
RESET hnsw.ef_search;
```

ΑΠΟΤΕΛΕΣΜΑ

```
SET

 recall_ef10
-------------
         66.5
(1 row)

SET

 recall_ef100
--------------
         91.5
(1 row)

RESET
```

Με `ef_search` 10 το index βρίσκει περίπου τα τρία τέταρτα των πραγματικών γειτόνων. Με την προεπιλογή βρίσκει τους εννιά στους δέκα και με 100 λίγο περισσότερους. Κάθε αύξηση κοστίζει χρόνο, αφού το index εξετάζει περισσότερους υποψήφιους. Το σωστό `ef_search` το βρίσκεις ακριβώς έτσι. Μετράς το recall σε ερωτήσεις με γνωστή απάντηση και διαλέγεις τη μικρότερη τιμή που δίνει ό,τι σου αρκεί. Οι παράμετροι του χτισίματος, `m` και `ef_construction` στο `WITH` του `CREATE INDEX`, ανεβάζουν επίσης την ακρίβεια με κόστος μέγεθος και χρόνο.

Αναζήτηση με φίλτρο

Σε μια εφαρμογή σχεδόν πάντα φιλτράρεις. «Τα πιο παρόμοια έγγραφα αυτού του πελάτη», «τα πιο παρόμοια προϊόντα αυτής της κατηγορίας». Εδώ το HNSW έχει μια παγίδα. Το index βρίσκει τους

`ef_search` πιο κοντινούς υποψήφιους και μετά εφαρμόζεται το `WHERE`. Αν κανείς από αυτούς δεν περνά το φίλτρο, το αποτέλεσμα είναι λιγότερες γραμμές από όσες ζήτησες, ή καμία.

```sql
SELECT count(*) AS returned
FROM (SELECT id FROM items
      WHERE category = 3
      ORDER BY embedding <=> (SELECT embedding FROM items WHERE id = 7)
      LIMIT 10) AS nearest;
```

```
ΑΠΟΤΕΛΕΣΜΑ

 returned
----------
        0
(1 row)
```

Το αντικείμενο 7 ανήκει στην κατηγορία 8. Οι 40 υποψήφιοι του index είναι όλοι από την κατηγορία 8 και κανείς δεν περνά το φίλτρο. Το query ζήτησε δέκα γραμμές και δεν πήρε καμία. Από την έκδοση 0.8 το pgvector κάνει **iterative scan**. Αν μετά το φίλτρο δεν έμειναν αρκετές γραμμές, συνεχίζει την αναζήτηση στο index.

```sql
SET hnsw.iterative_scan = relaxed_order;

SELECT count(*) AS returned
FROM (SELECT id FROM items
      WHERE category = 3
      ORDER BY embedding <=> (SELECT embedding FROM items WHERE id = 7)
      LIMIT 10) AS nearest;

RESET hnsw.iterative_scan;
```

```
ΑΠΟΤΕΛΕΣΜΑ

SET

 returned
----------
       10
(1 row)

RESET
```

Το `relaxed_order` μπορεί να επιστρέψει τις γραμμές σε ελαφρώς λάθος σειρά, που διορθώνεται με ένα εξωτερικό `ORDER BY`. Το `strict_order` κρατά αυστηρή σειρά με λίγο μεγαλύτερο κόστος. Για φίλτρα με λίγες τιμές, όπως η κατηγορία, μια άλλη λύση είναι ένα partial index ανά τιμή ή partitioning.

IVFFlat και halfvec

Το δεύτερο είδος index, το **IVFFlat**, χωρίζει τα vectors σε `lists` ομάδες γύρω από κέντρα και ψάχνει μόνο στις `ivfflat.probes` πιο κοντινές ομάδες. Χτίζεται πολύ γρηγορότερα από το HNSW και πιάνει λιγότερο χώρο, αλλά έχει χαμηλότερη ακρίβεια για την ίδια ταχύτητα. Επιπλέον τα κέντρα του

υπολογίζονται από τα δεδομένα που υπάρχουν όταν χτίζεται. Αν τα δεδομένα αλλάξουν πολύ, το ξαναχτίζεις. Για τις περισσότερες εφαρμογές το HNSW είναι η καλύτερη αφετηρία.

Ο τύπος `halfvec` αποθηκεύει κάθε διάσταση σε 2 bytes αντί για 4. Ένα embedding 1.536 διαστάσεων πιάνει 3 KB αντί για 6, με αμελητέα απώλεια ακρίβειας στην αναζήτηση. Το index ενός `vector` δέχεται ως 2.000 διαστάσεις, ενώ του `halfvec` ως 4.000.

> **ΚΡΑΤΑ ΑΥΤΟ**
> Το pgvector προσθέτει τον τύπο `vector` και τους τελεστές απόστασης `<->`, `<=>` και `<#>`. Τα embeddings τα φτιάχνει ένα μοντέλο έξω από τη βάση. Χωρίς index η αναζήτηση είναι ακριβής και διαβάζει όλο τον πίνακα. Ένα HNSW index με την κλάση τελεστών της απόστασής σου κάνει approximate αναζήτηση, με ακρίβεια που ελέγχεις με το `hnsw.ef_search` και μετράς ως recall. Με φίλτρα χρησιμοποιείς `hnsw.iterative_scan`. Το `halfvec` μισεί τον χώρο.

Ασκήσεις

ΑΣΚΗΣΗ 25.1 Τα πιο κοντινά σε ευθεία γραμμή

Βρες τα τρία προϊόντα του `product_vectors` που είναι πιο κοντά στο `[0.0, 1.0, 0.1]` με ευκλείδεια απόσταση, με την απόσταση στρογγυλεμένη σε τρία δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 product_id | distance
------------+----------
         22 |    0.141
         24 |    0.141
         15 |    1.277
(3 rows)
```

ΑΣΚΗΣΗ 25.2 Μέσος όρος vectors

Υπολόγισε με το `avg` το μέσο vector των αντικειμένων της κατηγορίας 3 και βρες την απόσταση συνημιτόνου του από το κέντρο της κατηγορίας στον `centers`, με τέσσερα δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 distance
----------
   0.0005
(1 row)
```

ΑΣΚΗΣΗ 25.3 Χρόνος χωρίς index

Με το index απενεργοποιημένο για ένα transaction, δείξε με `EXPLAIN (ANALYZE, COSTS OFF, BUFFERS OFF)` το plan της αναζήτησης των πέντε κοντινότερων στο αντικείμενο 100.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
SET

Limit (actual time=10.028..10.029 rows=5.00 loops=1)
  InitPlan 1
    ->  Bitmap Heap Scan on items items_1 (actual time=0.006..0.007 rows=1.00 loops=1)
          Recheck Cond: (id = 100)
          Heap Blocks: exact=1
          ->  Bitmap Index Scan on items_pkey (actual time=0.004..0.004 rows=1.00 loops=1)
                Index Cond: (id = 100)
                Index Searches: 1
  ->  Sort (actual time=10.027..10.028 rows=5.00 loops=1)
        Sort Key: ((items.embedding <=> (InitPlan 1).col1))
        Sort Method: top-N heapsort  Memory: 25kB
        ->  Seq Scan on items (actual time=0.012..8.780 rows=50000.00 loops=1)
Planning Time: 0.049 ms
Execution Time: 10.038 ms
```

ΑΣΚΗΣΗ 25.4 Index για ευκλείδεια απόσταση

Το index του κεφαλαίου είναι για `<=>`. Δείξε με `EXPLAIN (COSTS OFF)` αν χρησιμοποιείται σε αναζήτηση με `<->`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
Limit
  InitPlan 1
    ->  Index Scan using items_pkey on items items_1
          Index Cond: (id = 7)
  ->  Sort
        Sort Key: ((items.embedding <-> (InitPlan 1).col1))
        ->  Seq Scan on items
```

ΑΣΚΗΣΗ 25.5 Μισό μέγεθος

Δείξε το μέγεθος σε bytes ενός embedding του `items` ως `vector` και ως `halfvec`.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 vector_bytes | halfvec_bytes
--------------+---------------
          520 |           264
(1 row)
```

ΚΕΦΑΛΑΙΟ 26

PostGIS

Πόσο απέχει η αποθήκη από τον πελάτη, ποια σημεία παραλαβής είναι μέσα σε δέκα χιλιόμετρα, ποιο είναι το πιο κοντινό κατάστημα. Αυτές οι ερωτήσεις θέλουν γεωμετρία πάνω σε μια σφαίρα, indexes για σημεία και σχήματα και ένα κοινό λεξιλόγιο για συντεταγμένες. Το PostGIS τα προσθέτει όλα στην PostgreSQL και είναι από τις πιο ώριμες extensions που υπάρχουν.

Σημεία στον χάρτη

Το PostGIS εγκαθίσταται με `brew install postgis` σε macOS ή με το πακέτο `postgresql-18-postgis-3` σε Debian και Ubuntu.

```sql
CREATE EXTENSION postgis;
SELECT postgis_lib_version();
```

```
CREATE EXTENSION

 postgis_lib_version
---------------------
 3.6.4
(1 row)
```

Ένα σημείο στη Γη ορίζεται με γεωγραφικό μήκος και πλάτος. Η PostgreSQL δεν ξέρει μόνη της τι σημαίνουν δύο αριθμοί. Κάθε γεωμετρία του PostGIS έχει ένα **SRID**, τον αριθμό ενός συστήματος συντεταγμένων. Το 4326 είναι το WGS 84, το σύστημα του GPS, με μήκος και πλάτος σε μοίρες. Το PostGIS έχει δύο τύπους.

Τύπος	Υπολογισμοί	Μονάδες
`geometry`	σε επίπεδο, με την απλή γεωμετρία του σχολείου	του συστήματος, για το 4326 μοίρες
`geography`	πάνω στο ελλειψοειδές της Γης	μέτρα

Για αποστάσεις ανάμεσα σε πόλεις, το `geography` δίνει σωστά αποτελέσματα χωρίς σκέψη. Προσθέτουμε στις πόλεις του βιβλίου τη θέση τους. Προσοχή στη σειρά. Το `ST_MakePoint` παίρνει πρώτα το μήκος, που είναι ο άξονας x. Μετά παίρνει το πλάτος.

SQL

```
ALTER TABLE cities ADD COLUMN location geography(Point, 4326);

UPDATE cities AS c SET location = ST_MakePoint(v.lon, v.lat)::geography
FROM (VALUES
  ('Αθήνα', 23.7275, 37.9838), ('Θεσσαλονίκη', 22.9444, 40.6401),
  ('Πάτρα', 21.7346, 38.2466), ('Ηράκλειο', 25.1442, 35.3387),
  ('Λάρισα', 22.4191, 39.6390), ('Βόλος', 22.9420, 39.3622),
  ('Ιωάννινα', 20.8537, 39.6650), ('Χανιά', 24.0180, 35.5138),
  ('Καλαμάτα', 22.1142, 37.0389), ('Ρόδος', 28.2176, 36.4349),
  ('Κοζάνη', 21.7887, 40.3007)
) AS v(name, lon, lat)
WHERE c.name = v.name;

SELECT name, ST_AsText(location) AS wkt FROM cities ORDER BY id LIMIT 3;
```

ΑΠΟΤΕΛΕΣΜΑ

```
ALTER TABLE

UPDATE 11

    name     |          wkt
-------------+------------------------
 Αθήνα       | POINT(23.7275 37.9838)
 Θεσσαλονίκη | POINT(22.9444 40.6401)
 Πάτρα       | POINT(21.7346 38.2466)
(3 rows)
```

Το `ST_AsText` γράφει τη γεωμετρία σε **WKT**, well known text, την τυποποιημένη μορφή κειμένου για γεωμετρίες.

Αποστάσεις

SQL

```
SELECT round(ST_Distance(a.location, b.location)::numeric / 1000, 1) AS km
FROM cities AS a, cities AS b
WHERE a.name = 'Αθήνα' AND b.name = 'Θεσσαλονίκη';
```

ΑΠΟΤΕΛΕΣΜΑ

```
  km
-------
 302.5
(1 row)
```

Περίπου 300 km σε ευθεία, πάνω στην καμπύλη της Γης. Ας δούμε τι θα έδινε το ίδιο σε `geometry`.

SQL

```
SELECT ST_Distance(a.location::geometry, b.location::geometry) AS degrees
FROM cities AS a, cities AS b
WHERE a.name = 'Αθήνα' AND b.name = 'Θεσσαλονίκη';
```

ΑΠΟΤΕΛΕΣΜΑ

```
      degrees
-------------------
 2.769327589867253
(1 row)
```

Μια απόσταση σε μοίρες δεν σημαίνει τίποτα, αφού μια μοίρα μήκους είναι διαφορετική απόσταση σε διαφορετικό πλάτος. Για να δουλέψεις σε `geometry` με μέτρα, μετατρέπεις σε ένα προβολικό σύστημα με `ST_Transform`. Για την Ελλάδα είναι το ΕΓΣΑ87, SRID 2100, με συντεταγμένες σε μέτρα.

SQL

```
SELECT round(ST_Distance(ST_Transform(a.location::geometry, 2100),
                         ST_Transform(b.location::geometry, 2100))::numeric / 1000, 1) AS km
FROM cities AS a, cities AS b
WHERE a.name = 'Αθήνα' AND b.name = 'Θεσσαλονίκη';
```

ΑΠΟΤΕΛΕΣΜΑ

```
  km
-------
 302.4
(1 row)
```

Σχεδόν ίδιο με το `geography`. Τα προβολικά συστήματα είναι ακριβή μέσα στην περιοχή για την οποία φτιάχτηκαν και γρηγορότερα στους υπολογισμούς. Το `geography` είναι σωστό παντού, με λίγο μεγαλύτερο κόστος.

Μέσα σε μια ακτίνα

Ποιες πόλεις απέχουν λιγότερο από 150 km από τη Λάρισα; Η function `ST_DWithin` απαντά ναι ή όχι για μια απόσταση και σε αντίθεση με ένα `ST_Distance < 150000`, μπορεί να χρησιμοποιήσει index.

SQL

```
SELECT b.name, round(ST_Distance(a.location, b.location)::numeric / 1000) AS km
FROM cities AS a JOIN cities AS b ON b.id <> a.id
WHERE a.name = 'Λάρισα' AND ST_DWithin(a.location, b.location, 150000)
ORDER BY km;
```

ΑΠΟΤΕΛΕΣΜΑ

```
    name     | km
-------------+-----
 Βόλος       |  54
 Κοζάνη      |  91
 Θεσσαλονίκη | 120
 Ιωάννινα    | 134
(4 rows)
```

Πολλά σημεία και ένα GiST index

Για να φανεί η αξία ενός index χρειαζόμαστε πολλά σημεία. Φτιάχνουμε 100.000 σημεία παραλαβής με τυχαίες θέσεις γύρω από τις πόλεις, περισσότερα στις μεγαλύτερες.

SQL

```
SELECT setseed(0.26);

CREATE TABLE pickup_points AS
SELECT n AS id, c.name AS city,
       ST_Project(c.location, sqrt(random()) * 20000, radians(random() * 360)) AS location
FROM generate_series(1, 100000) AS n
JOIN LATERAL (
  SELECT name, location FROM cities
  WHERE n > 0
  ORDER BY population * random() DESC
  LIMIT 1
) AS c ON true;

ALTER TABLE pickup_points ADD PRIMARY KEY (id);

SELECT city, count(*) FROM pickup_points GROUP BY city ORDER BY count(*) DESC LIMIT 4;
```

ΑΠΟΤΕΛΕΣΜΑ

```
 setseed
---------

(1 row)

SELECT 100000

ALTER TABLE

    city     | count
-------------+-------
 Αθήνα       | 71339
 Θεσσαλονίκη | 19530
 Ηράκλειο    |  3570
 Πάτρα       |  2686
(4 rows)
```

Το `ST_Project` επιστρέφει το σημείο σε μια απόσταση και μια κατεύθυνση από ένα άλλο. Εδώ ως 20 km από την πόλη, σε τυχαία κατεύθυνση. Η πόλη κάθε σημείου διαλέγεται τυχαία, με μεγάλη προτίμηση στις πόλεις με μεγάλο πληθυσμό.

Τα σημεία και τα σχήματα θέλουν indexes που καταλαβαίνουν τον χώρο. Είναι τα **GiST** indexes του κεφαλαίου 35 του πρώτου τόμου. Κάθε κόμβος κρατά ένα ορθογώνιο που περιέχει όλα τα σημεία από κάτω του και η αναζήτηση κατεβαίνει μόνο στα ορθογώνια που τέμνουν την περιοχή της ερώτησης.

SQL

```
CREATE INDEX pickup_points_location_idx ON pickup_points USING gist (location);
ANALYZE pickup_points;

EXPLAIN (ANALYZE, COSTS OFF, BUFFERS OFF)
SELECT count(*) FROM pickup_points
WHERE ST_DWithin(location, (SELECT location FROM cities WHERE name = 'Βόλος'), 2000);
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE INDEX

ANALYZE

Aggregate (actual time=0.057..0.058 rows=1.00 loops=1)
  InitPlan 1
    ->  Seq Scan on cities (actual time=0.004..0.005 rows=1.00 loops=1)
          Filter: (name = 'Βόλος'::text)
          Rows Removed by Filter: 10
  ->  Bitmap Heap Scan on pickup_points (actual time=0.036..0.055 rows=7.00 loops=1)
        Filter: st_dwithin(location, (InitPlan 1).col1, '2000'::double precision, true)
        Rows Removed by Filter: 8
        Heap Blocks: exact=15
        ->  Bitmap Index Scan on pickup_points_location_idx (actual time=0.029..0.029 rows=15.00 loops=1)
              Index Cond: (location && _st_expand((InitPlan 1).col1, '2000'::double precision))
              Index Searches: 1
Planning Time: 0.132 ms
Execution Time: 0.088 ms
```

Το `Index Cond` με τον τελεστή `&&` είναι ο πρώτος, χονδρικός έλεγχος με τα ορθογώνια. Το `Filter` με το `ST_DWithin` είναι ο ακριβής έλεγχος στις γραμμές που πέρασαν τον πρώτο. Έτσι δουλεύουν σχεδόν όλα τα χωρικά queries.

Ο πιο κοντινός

Για «τα πέντε πιο κοντινά σημεία» δεν ξέρεις από πριν την ακτίνα. Ο τελεστής `<->` επιστρέφει απόσταση και ένα `ORDER BY` με αυτόν συνοδευόμενο από `LIMIT` χρησιμοποιεί το GiST index για αναζήτηση **k nearest neighbours**. Το index επιστρέφει τα σημεία κατά σειρά απόστασης και σταματά μόλις βρει πέντε. Είναι η ίδια ιδέα με το HNSW του προηγούμενου κεφαλαίου, αλλά ακριβής.

SQL

```
EXPLAIN (COSTS OFF)
SELECT id FROM pickup_points
ORDER BY location <-> ST_MakePoint(23.7275, 37.9838)::geography
LIMIT 5;

SELECT id, city,
       round(ST_Distance(location, ST_MakePoint(23.7275, 37.9838)::geography)) AS meters
FROM pickup_points
ORDER BY location <-> ST_MakePoint(23.7275, 37.9838)::geography
LIMIT 5;
```

ΑΠΟΤΕΛΕΣΜΑ

```
Limit
  ->  Index Scan using pickup_points_location_idx on pickup_points
        Order By: (location <-> '0101000020E61000000AD7A3703DBA374004E78C28EDFD4240'::geography)

  id   | city  | meters
-------+-------+--------
 84508 | Αθήνα |     53
  7835 | Αθήνα |     55
 88630 | Αθήνα |     77
 77745 | Αθήνα |    120
 87317 | Αθήνα |    128
(5 rows)
```

Σχήματα

Εκτός από σημεία, το PostGIS κρατά γραμμές και πολύγωνα. Μια ζώνη παράδοσης είναι ένα πολύγωνο. Εδώ ένας κύκλος 10 km γύρω από το κέντρο της Θεσσαλονίκης, που φτιάχνει το `ST_Buffer`. Το δεύτερο είναι ένα ορθογώνιο γύρω από το λεκανοπέδιο της Αττικής με το `ST_MakeEnvelope`.

SQL

```
CREATE TABLE delivery_zones (name text PRIMARY KEY, area geography(Polygon, 4326));

INSERT INTO delivery_zones
SELECT 'Θεσσαλονίκη κέντρο', ST_Buffer(location, 10000)
FROM cities WHERE name = 'Θεσσαλονίκη';

INSERT INTO delivery_zones
VALUES ('Λεκανοπέδιο', ST_MakeEnvelope(23.60, 37.90, 23.85, 38.08, 4326)::geography);

SELECT z.name, round(ST_Area(z.area)::numeric / 1000000) AS km2, count(p.id) AS points
FROM delivery_zones AS z
LEFT JOIN pickup_points AS p ON ST_Intersects(z.area, p.location)
GROUP BY z.name, z.area
ORDER BY z.name;
```

ΑΠΟΤΕΛΕΣΜΑ

```
CREATE TABLE

INSERT 0 1

INSERT 0 1

        name        | km2 | points
--------------------+-----+--------
 Θεσσαλονίκη κέντρο | 312 |   4773
 Λεκανοπέδιο        | 439 |  24968
(2 rows)
```

Το `ST_Area` σε `geography` δίνει τετραγωνικά μέτρα και το `ST_Intersects` ελέγχει αν δύο γεωμετρίες έχουν κοινό σημείο, εδώ αν το σημείο είναι μέσα στη ζώνη. Και αυτό χρησιμοποιεί το GiST index των σημείων.

Προς τον χάρτη

Μια εφαρμογή που δείχνει χάρτη στον browser θέλει συνήθως **GeoJSON**. Το `ST_AsGeoJSON` το φτιάχνει και με το `jsonb_build_object` του πρώτου τόμου φτιάχνεις ολόκληρο το αντικείμενο που περιμένουν βιβλιοθήκες όπως το Leaflet.

SQL

```
SELECT jsonb_pretty(jsonb_build_object(
         'type', 'Feature',
         'geometry', ST_AsGeoJSON(location)::jsonb,
         'properties', jsonb_build_object('name', name, 'population', population)
       )) AS feature
FROM cities WHERE name = 'Βόλος';
```

ΑΠΟΤΕΛΕΣΜΑ

```
           feature
-----------------------------
 {                           +
     "type": "Feature",      +
     "geometry": {           +
         "type": "Point",    +
         "coordinates": [    +
             22.942,         +
             39.3622         +
         ]                   +
     },                      +
     "properties": {         +
         "name": "Βόλος",    +
         "population": 144449+
     }                       +
 }
(1 row)
```

ΚΡΑΤΑ ΑΥΤΟ

Το PostGIS προσθέτει τους τύπους `geometry` και `geography`. Κάθε γεωμετρία έχει SRID, με το 4326 για μήκος και πλάτος. Το `ST_MakePoint` παίρνει πρώτα το μήκος. Το `geography` δίνει αποστάσεις και εμβαδά σε μέτρα πάνω στη Γη, ενώ για `geometry` σε μέτρα χρειάζεσαι `ST_Transform` σε προβολικό σύστημα όπως το 2100. Για ακτίνες χρησιμοποιείς `ST_DWithin` και για τους κοντινότερους `ORDER BY <->` με `LIMIT` και τα δύο με GiST index. Το `ST_AsGeoJSON` βγάζει τα δεδομένα για χάρτες.

Ασκήσεις

ΑΣΚΗΣΗ 26.1 Από τη Θεσσαλονίκη στο Ηράκλειο

Υπολόγισε σε km, με ένα δεκαδικό, την απόσταση Θεσσαλονίκης και Ηρακλείου.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  km
-------
 619.3
(1 row)
```

ΑΣΚΗΣΗ 26.2 Η πιο μακρινή πόλη

Για κάθε πόλη βρες την πιο μακρινή άλλη πόλη και την απόσταση σε km, χωρίς δεκαδικά, ταξινομημένες κατά απόσταση.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    name     |  farthest   | km
-------------+-------------+-----
 Ιωάννινα    | Ρόδος       | 739
 Ρόδος       | Ιωάννινα    | 739
 Κοζάνη      | Ρόδος       | 707
 Θεσσαλονίκη | Ρόδος       | 655
 Ηράκλειο    | Κοζάνη      | 625
 Λάρισα      | Ρόδος       | 621
 Πάτρα       | Ρόδος       | 608
 Χανιά       | Θεσσαλονίκη | 577
 Βόλος       | Ρόδος       | 566
 Καλαμάτα    | Ρόδος       | 549
 Αθήνα       | Ρόδος       | 434
(11 rows)
```

ΑΣΚΗΣΗ 26.3 Σημεία κοντά στην Πάτρα

Βρες τα τρία σημεία παραλαβής που είναι πιο κοντά στο κέντρο της Πάτρας, με την απόσταση σε μέτρα χωρίς δεκαδικά.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
  id   | meters
-------+--------
 83148 |    266
 68849 |    696
 16478 |    821
(3 rows)
```

ΑΣΚΗΣΗ 26.4 Πυκνότητα ανά πόλη

Μέτρησε για κάθε πόλη πόσα σημεία παραλαβής απέχουν λιγότερο από 5 km από το κέντρο της και δείξε τις τέσσερις με τα περισσότερα.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
    name     | points
-------------+--------
 Αθήνα       |   4473
 Θεσσαλονίκη |   1231
 Ηράκλειο    |    211
 Πάτρα       |    155
(4 rows)
```

ΑΣΚΗΣΗ 26.5 Σε ποια ζώνη

Για το σημείο παραλαβής 1, δείξε σε ποια ζώνη παράδοσης βρίσκεται, αν βρίσκεται σε κάποια, μαζί με την πόλη του.

ΑΝΑΜΕΝΟΜΕΝΟ ΑΠΟΤΕΛΕΣΜΑ

```
 id | city  | zone
----+-------+------
  1 | Αθήνα | ∅
(1 row)
```