PGDay France 2023

Tour d’horizon des Connection Poolers

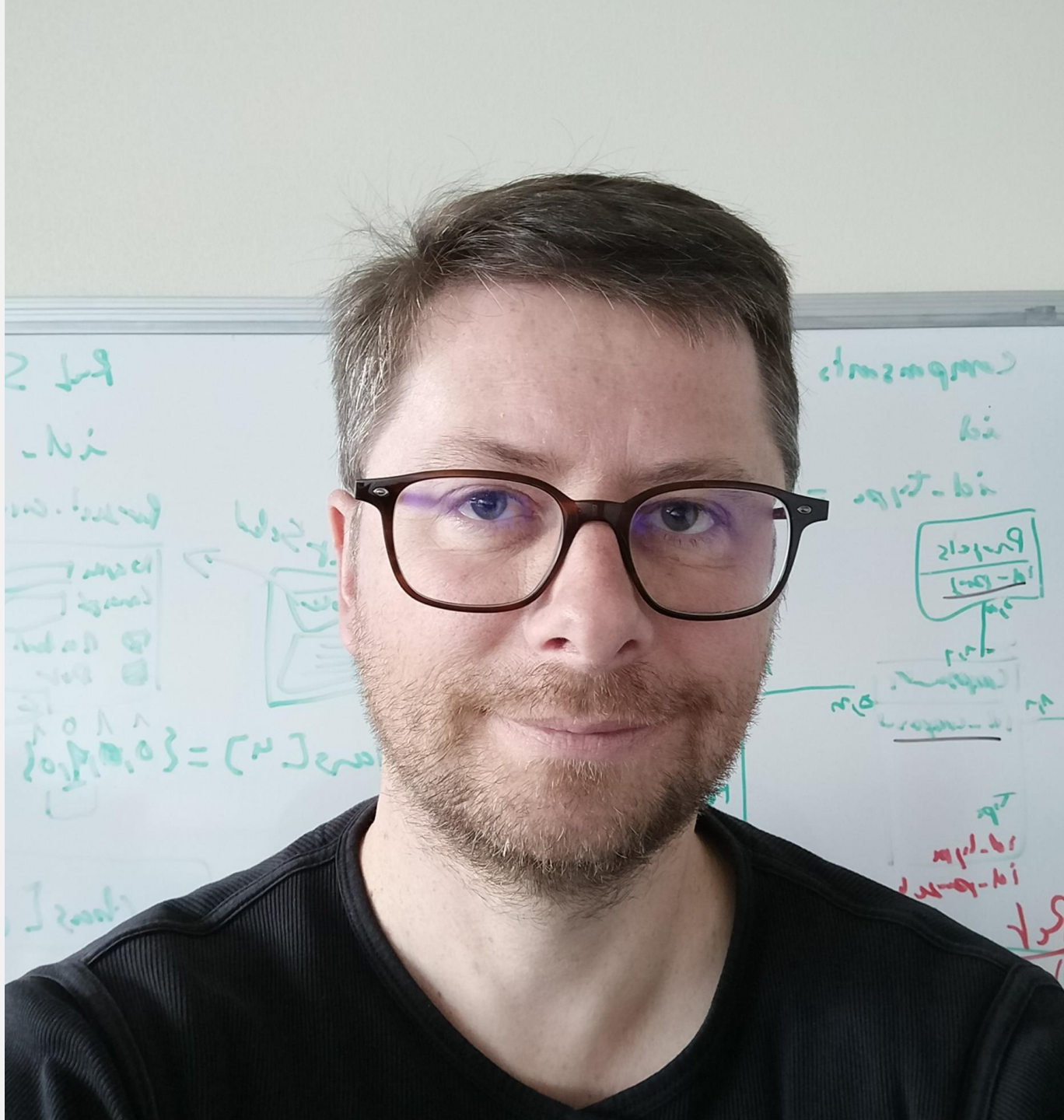
Frédéric Delacourt

Amiga 500 DemoMaker

Algorithm Maniac

Mad Scientist On Spare Time

PostgreSQL Expert @ Data Bene



Data Bene

Audit – Consulting – Conception

Assistance technique

Support – Services managés

Formation

Des services taillés au plus près des besoins réels des clients.

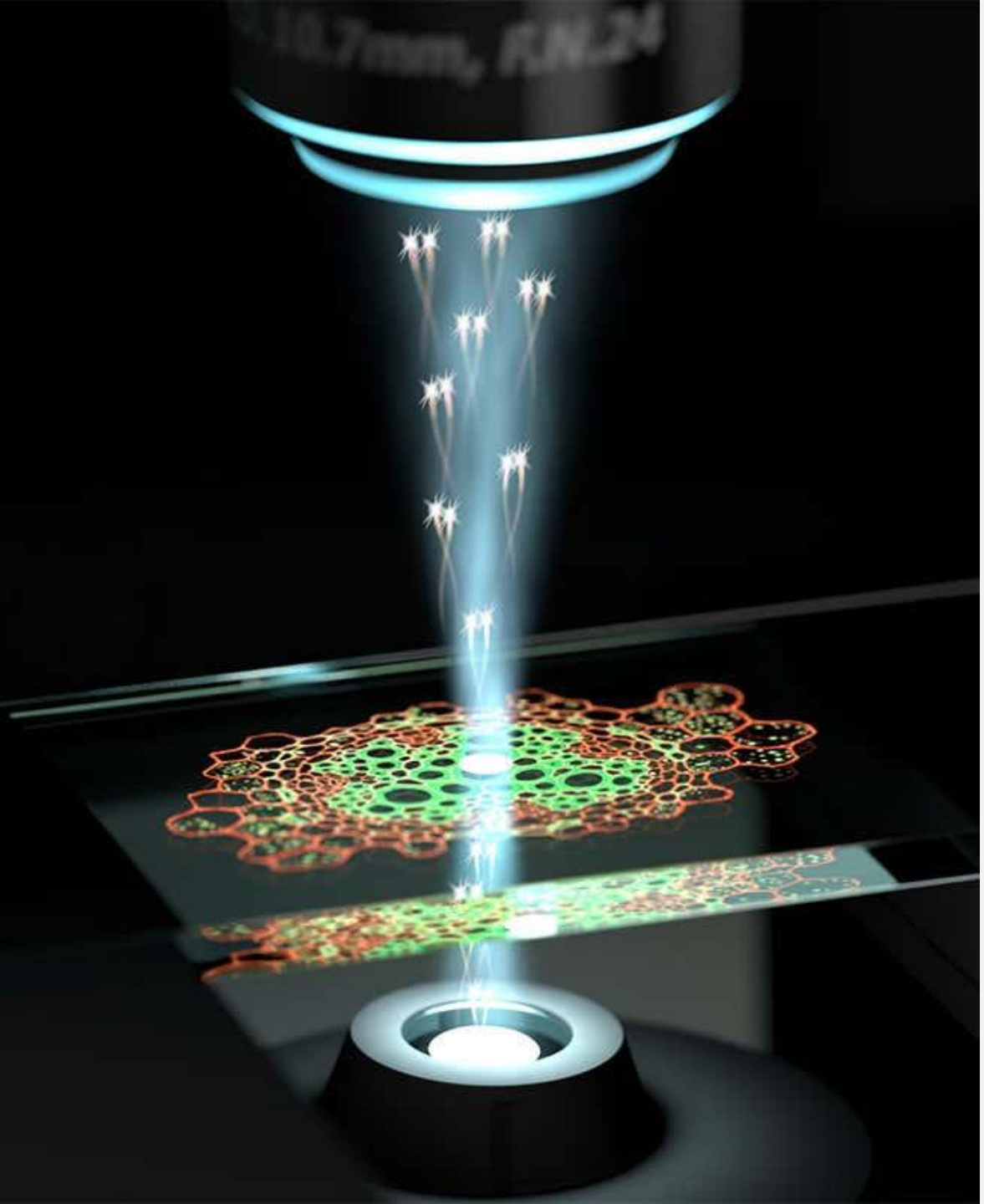
databene.io

DataBene

We are hiring!

Mad or regular (sane) scientists are welcome.

recrutement@data-bene.fr



Introduction

Connexions PostgreSQL

- Cycle de vie
- Paramètre `max_connections`
- Protocole de communication

Poolers de Connexions

Synthèse

Connexion PostgreSQL – Cycle de vie

Connecter – Interroger - Déconnecter

Connexion PostgreSQL – Méthodes

- Arguments

  sudo –iu postgres psql –p 5432 –d pgbench

  psql –h 127.0.0.1 –p 5432 –U postgres –d pgbench

- Chaîne de connexion

  psql ‘host=/tmp/s.5432 port=5432 user=postgres dbname=pgbench’

  psql ‘host=127.0.0.1 port=5432 user=postgres dbname=pgbench’

- Uniform Resource Identifier

  psql postgresql://localhost:5432/pgbench?user=postgres

Connexion PostgreSQL – Options intéressantes

- Multihosts
  - ‘host=h1,h2,h3 port=p1,p2,p3 dbname=xxx user=yyy’
  - postgres://h1,h2,h3:p1,p2,p3/dbname?user=yyy
- Connection target
  - ‘target_session_attrs=xxx‘
  - xxx = { `any | read-write | read-only | primary | standby | prefer-standby` }
  - JDBC::targetServerType = { `primary | secondary | preferSecondary` }
- Replication Protocol

  ‘replication=1 user=replication_user’

```
psql 'host=xxx port=5432 user=replicator replication=1' -c 'IDENTIFY_SYSTEM'
```

DataBene

Connexions PostgreSQL - Processus

1 connexion = 1 processus (backend)

Connexions PostgreSQL - Déconnexion

- Le frontend (client) peut fermer la session à tout moment
- Le backend (server) peut fermer la session à tout moment

- Le backend termine la session le plus tôt possible,
  quelle qu'en soit la raison.

PostgreSQL max_connections

Vers l’infini et au-delà (Buzz l’Éclair)

max_connections

- max_connections = 10 000 ? Oui c'est possible…

- Construction d'un snapshot
  - Par défaut le Transaction Isolation Level est READ COMMITTED
  - La prise de Snapshot vérifie l'état des 10 000 connexions pour chaque statement.
  - PostgreSQL 14+ écarte les connexions inactives (idle) du snapshot

- Supposons 10 000 connexions actives :
  - 1 connexion = 1 processus backend
  - Consommation mémoire
  - OS Scheduling

À votre avis, quel sera le premier facteur limitant ?

PostgreSQL max_connections

Quelle est la quantité de mémoire utilisée par une connexion (nouvelle) ?

Consommation mémoire – 1ère Expérience

```
$ psql 'host=xxx user=pgbench password=pgbench’ &

$ ps fauxwww | grep 192.168.245.134
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
postgres	1134	0.0	0.3	238460	13216	?	Ss	12:50	0:00	_ postgres: 15/main: postgres postgres 192.168.245.134(56080) idle
postgres	1135	0.0	0.2	238428	11904	?	Ss	12:50	0:00	_ postgres: 15/main: postgres postgres 192.168.245.134(56090) idle
postgres	1136	0.0	0.2	238428	11904	?	Ss	12:50	0:00	_ postgres: 15/main: postgres postgres 192.168.245.134(56092) idle
postgres	1137	0.0	0.2	238428	11912	?	Ss	12:50	0:00	_ postgres: 15/main: postgres postgres 192.168.245.134(56108) idle
postgres	1138	0.0	0.2	238428	11968	?	Ss	12:50	0:00	_ postgres: 15/main: postgres postgres 192.168.245.134(56120) idle
postgres	1139	0.0	0.2	238428	11904	?	Ss	12:50	0:00	_ postgres: 15/main: postgres postgres 192.168.245.134(56134) idle
postgres	1140	0.0	0.2	238428	11920	?	Ss	12:50	0:00	_ postgres: 15/main: postgres postgres 192.168.245.134(56148) idle
postgres	1141	0.0	0.2	238428	11968	?	Ss	12:50	0:00	_ postgres: 15/main: postgres postgres 192.168.245.134(56156) idle
postgres	1142	0.0	0.2	238428	11968	?	Ss	12:50	0:00	_ postgres: 15/main: postgres postgres 192.168.245.134(56160) idle

Les colonnes (**VSZ**,**RSS**) indiquent les quantités de mémoire **virtuelle** et **physique** consommées par chaque processus.
PostgreSQL 15, en configuration par défaut => chaque connexion consomme **~240Mo**, **~12Mo**.

Consommation mémoire – 2ème Expérience

Session Host Server

```
$ free -k
              total        used        free
Mem:        4017720       97484     3794428
Swap:        998396           0      998396
```

DataBene

Consommation mémoire – 2ème Expérience

Session Host Server

```
$ free -k
              total        used        free
Mem:        4017720       97484     3794428
Swap:        998396           0      998396
```

Session Host Client

```
for i in `seq 1 500`
do
   psql 'host=192.168.245.200 user=postgres' &
done
```

Consommation mémoire – 2ème Expérience

Session Host Server

```
$ free -k
              total        used        free
Mem:        4017720       97484     3794428
Swap:        998396           0      998396
```

```
$ free -k
              total        used        free
Mem:        4017720      845640     3043024
Swap:        998396           0      998396
```

Session Host Client

```
for i in `seq 1 500`
do
   psql 'host=192.168.245.200 user=postgres' &
done
```

PostgreSQL 15 => une connexion impose une consommation de (845640 - 97484) / 500 = **1.5Mo** (mémoire physique).

DataBene

PostgreSQL max_connections

Quelle est la quantité de mémoire utilisée par une connexion (active) ?

Consommation mémoire – Backends actifs

- Exécution de requêtes
  - La **work_mem** peut être utilisée lors des tris, des hashages ou par le tuplestore.
  - Une requête peut opérer plusieurs hashages, tirs ou tuplestore simultanément.
  - Le parallélisme peut survenir (Parallel workers)…
- Routines de maintenance
  - `(auto)VACUUM` jusqu'à min(**maintenance_work_mem**, 1Go) par backend
  - `INDEX`/`REINDEX` consomment **maintenance_work_mem** par backend
- `huge_pages = off | shared_buffers * max_connections`
  - VmPTE => 8 byte (pointeur) par page mémoire de 4kio
  - `shared_buffers` = 32Gio, `max_connections` = 1024 => $VmPTE_{max}$ = 64Gio

PostgreSQL max_connections

Combien de processus puis-je exécuter ?

PostgreSQL Processes - Paramètres

- max_connections,
- max_autovacuum_workers,
- max_wal_senders,
- max_worker_processes,
  - max_parallel_maintenance_workers,
  - max_parallel_workers,
  - max_parallel_workers_per_gather,
- …

En augmentation avec les versions de PostgreSQL.

PostgreSQL Processes - Types

- Core Processes (8)

```
   Postmaster   + Logger   + Startup
 + Checkpointer + BgWriter
 + WAL Writer   + Archiver + [Stats Collector]Before PG15
```

- Backend Processes (> 100)

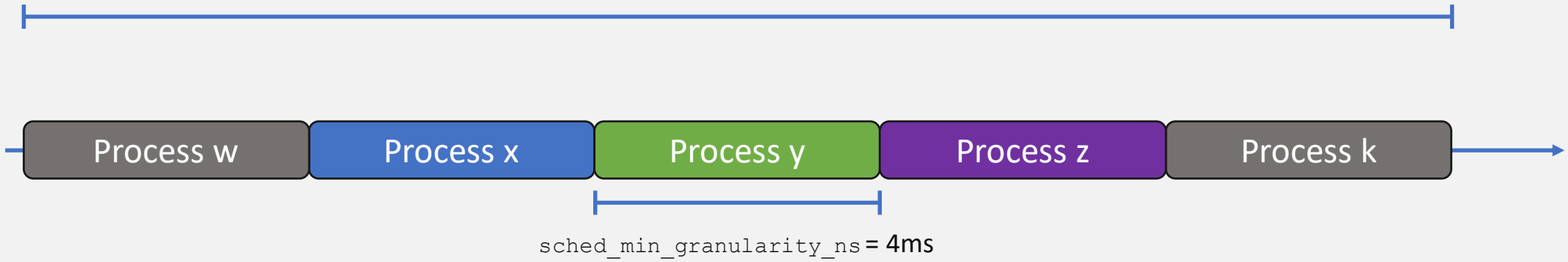
```
   max_connections (100)
 + autovacuum_max_workers (3)
 + max_wal_senders (5/10)
 + max_worker_processes (8)
```

Questionner l'activité de ces processus:

- Toujours actifs ?
- CPU bound vs IO bound ?

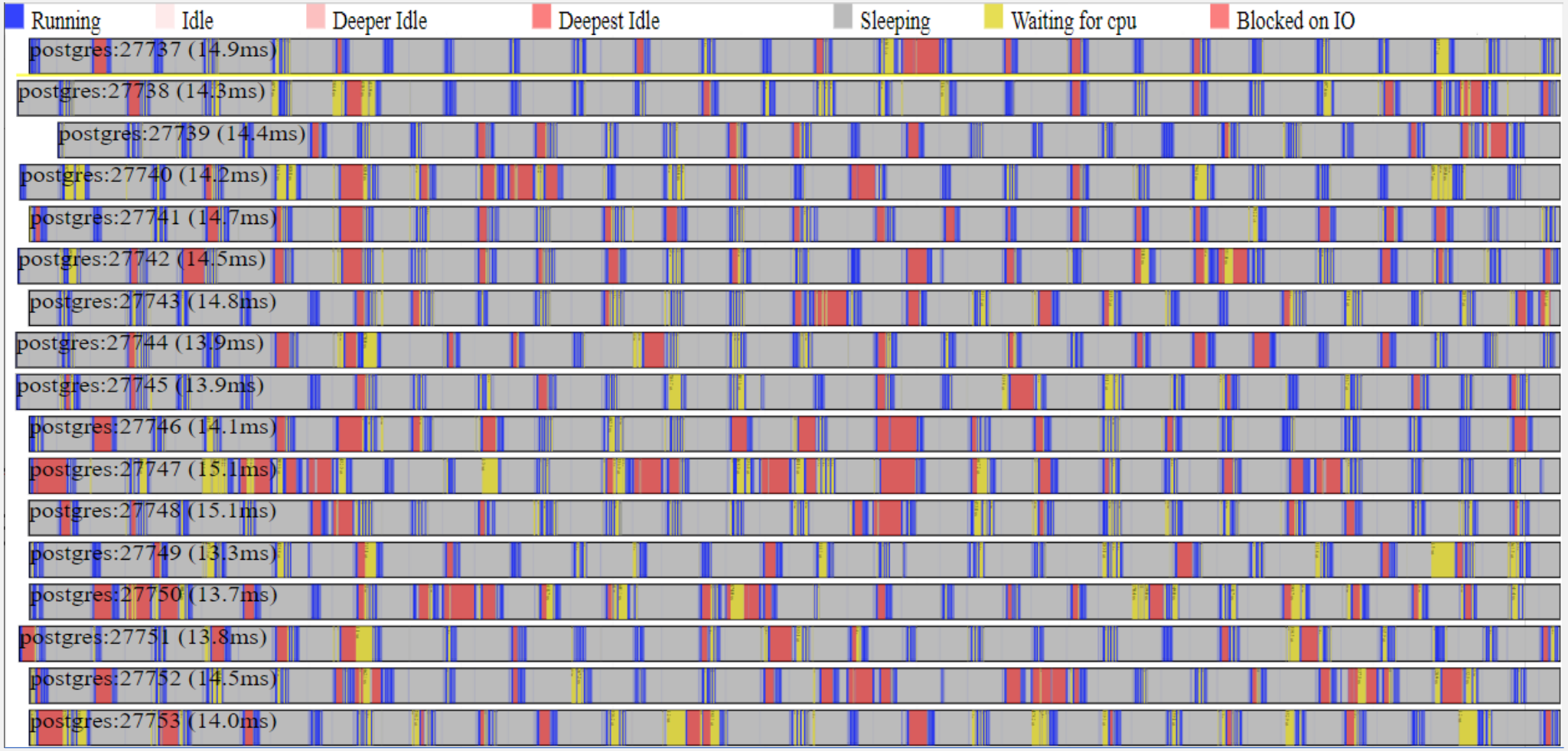
Linux – Completely Fair Scheduler

Scheduling period = `sched_latency_ns` = 20ms
nr_running * `sched_min_granularity_ns`



Dans cet exemple, la **runqueue** est dimensionnée pour exécuter 5 processus « CPU bound » par scheduling period de 20ms.

Linux – Completely Fair Scheduler - Saturation



DataBene

Point d’équilibre de support de Charge

Active Processes# = 2 x CPU Core# + Simultaneous Disk IO#

Only Blocking IOs Are Counted

Protocole PostgreSQL

Examiner le protocole de communication

PostgreSQL :

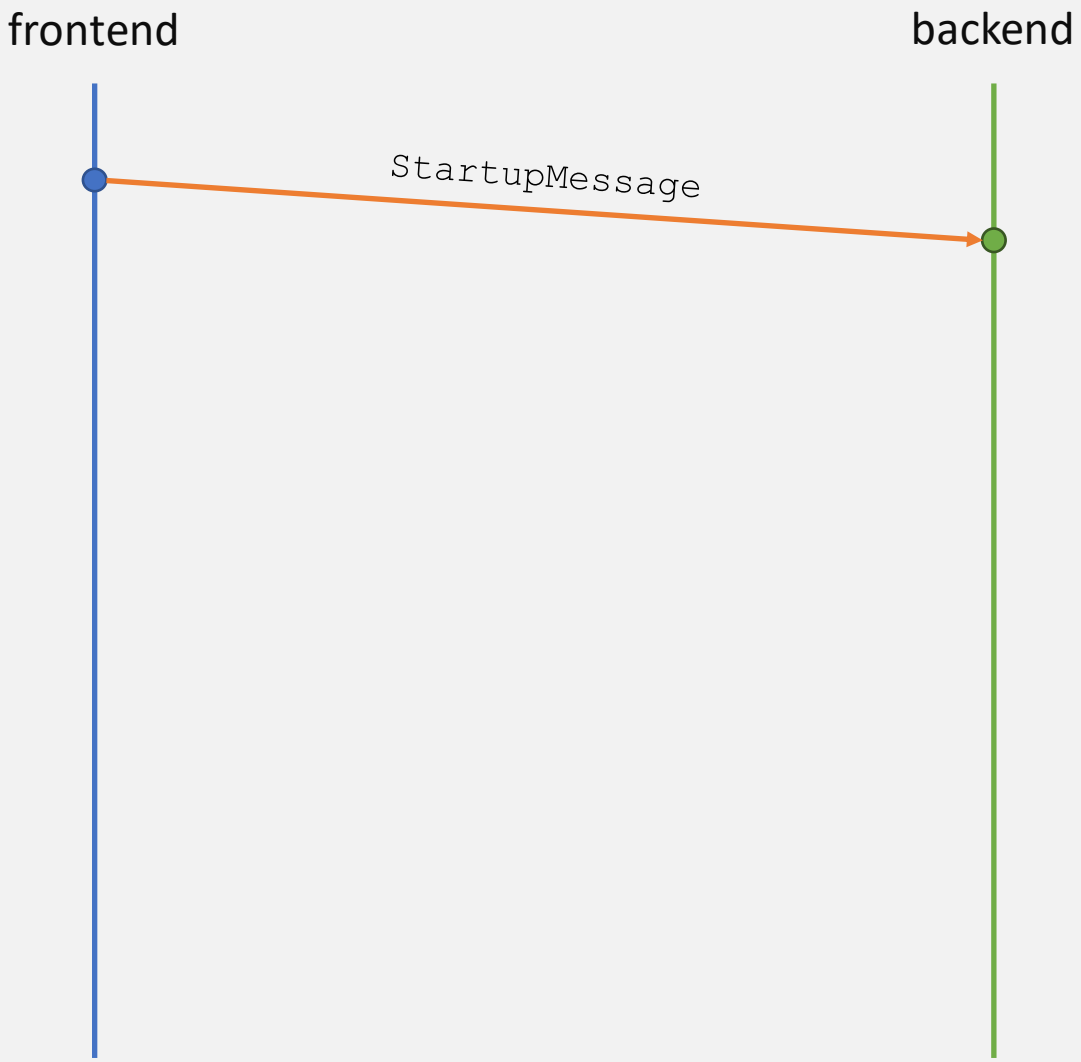
- Écouter tcp/5432 en **SSL désactivé**
- `'host=127.0.0.1 port=5432 dbname=pgbench sslmode=disable'`
- `«jdbc:postgresql://localhost:5432/pgbench?sslmode=disable»`

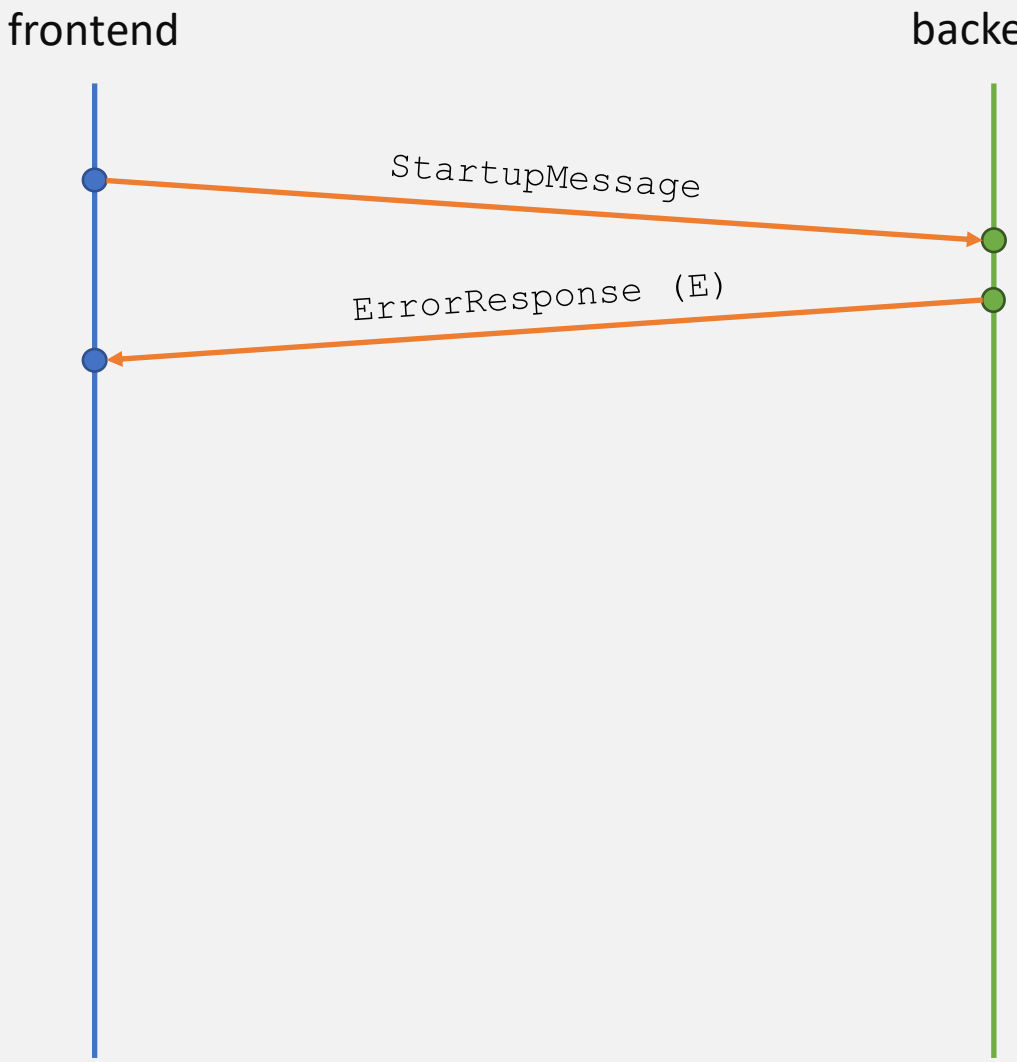
Linux :

- `sudo tshark -i lo -f 'tcp port 5432' -d tcp.port==5432,pgsql -V`
- `sudo tshark -i lo -f 'tcp port 5432' -d tcp.port==5432,pgsql -w /tmp/pg_protocol.pcap`
- `sudo tshark -V -r /tmp/pg_protocol.pcap`

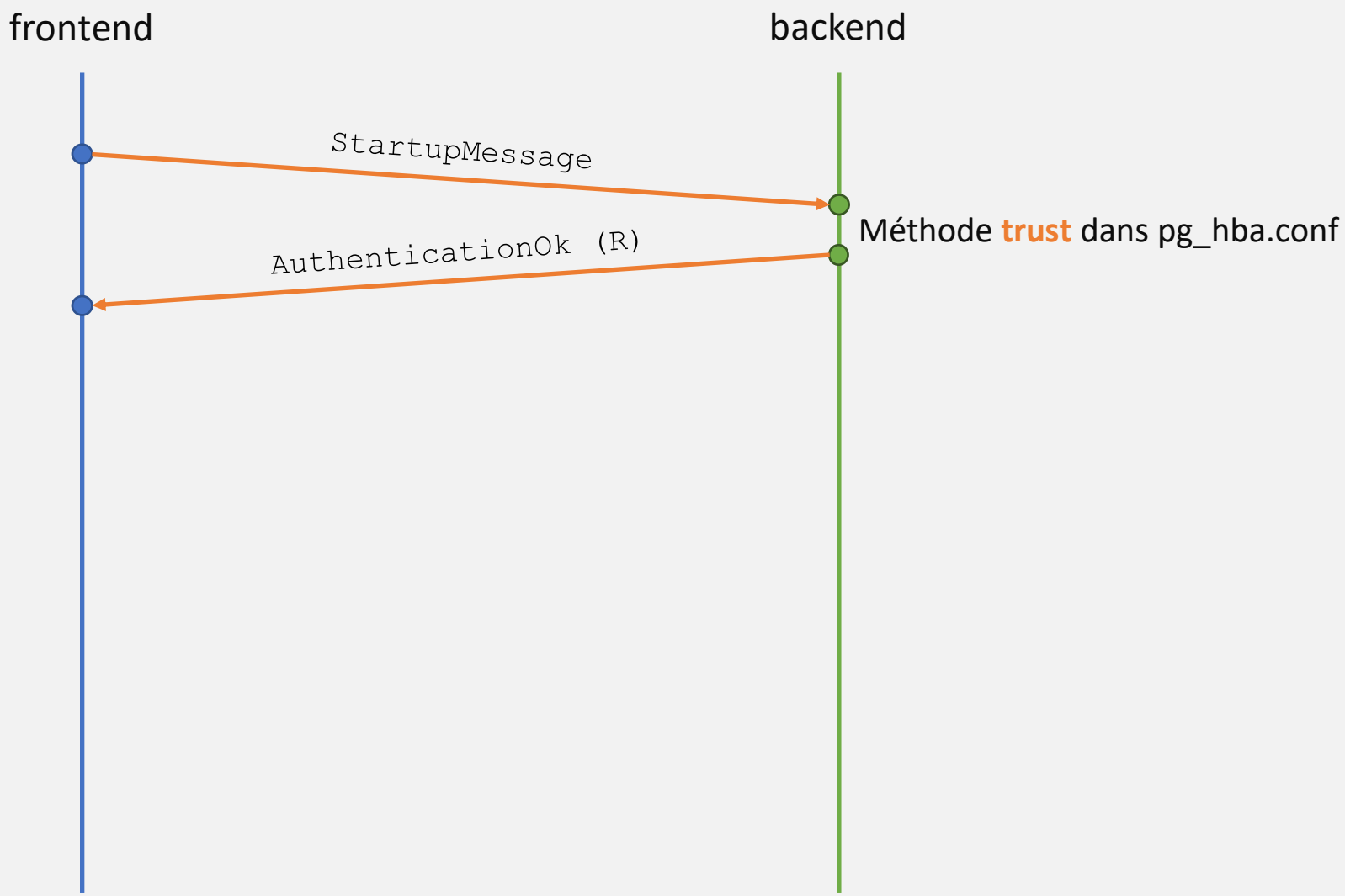
Protocole PostgreSQL

Messages – Start-up and Authentication


frontend
backend
StartupMessage

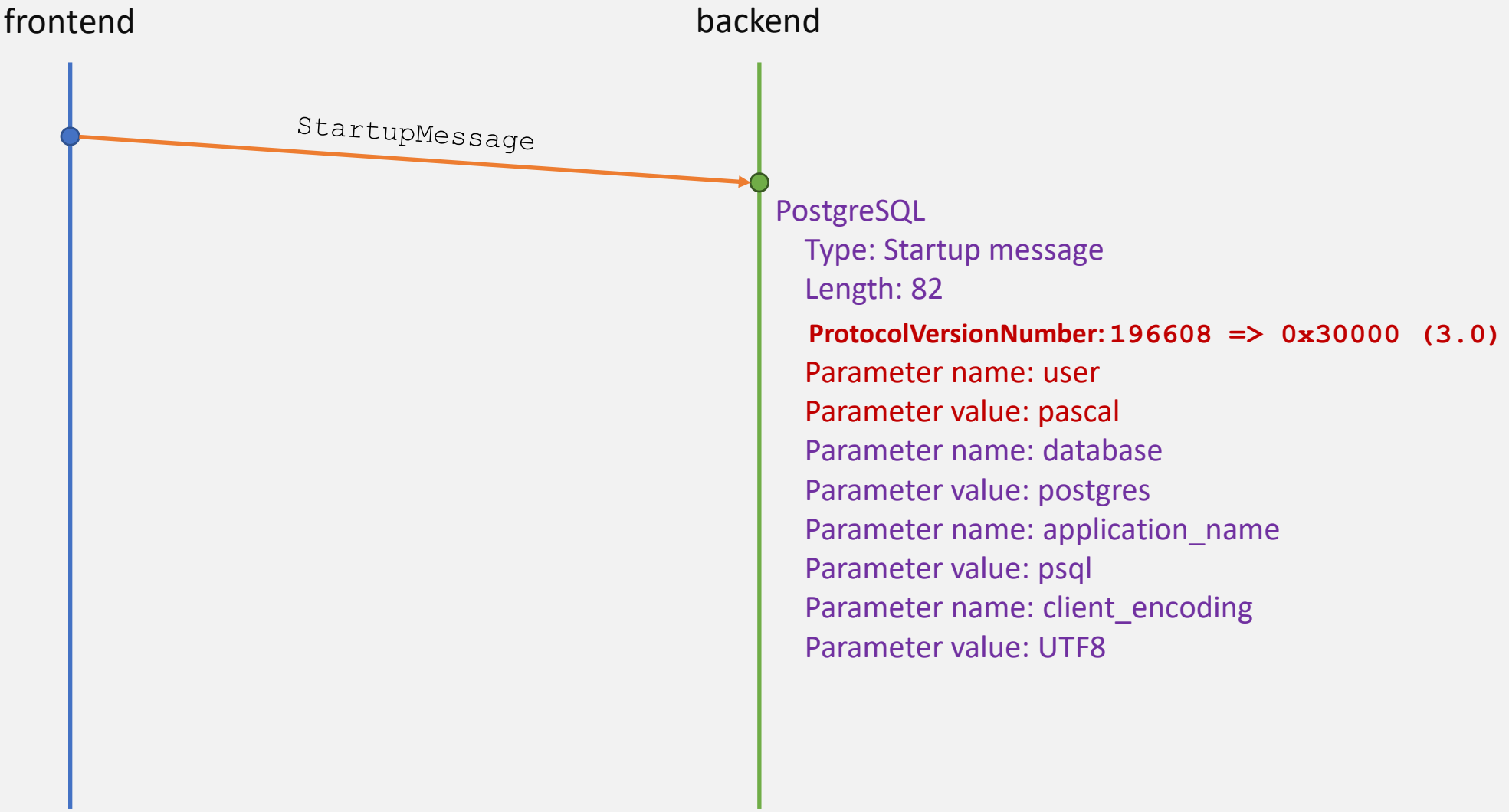


Pas de correspondance ou **reject** dans pg_hba.conf


frontend
backend
StartupMessage
AuthenticationOk (R)
Méthode trust dans pg_hba.conf

Protocole PostgreSQL

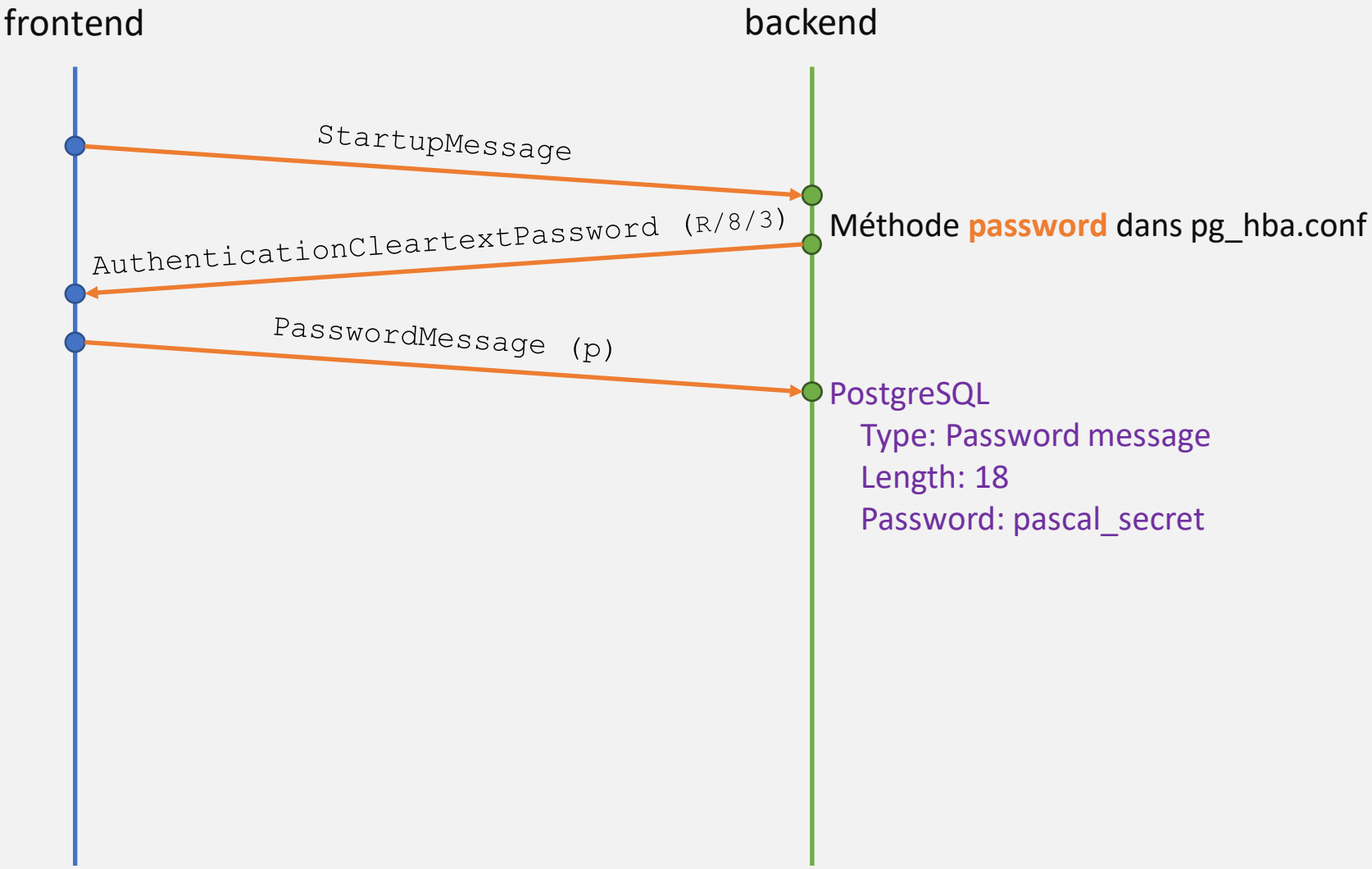
Authentication – (clear)password method

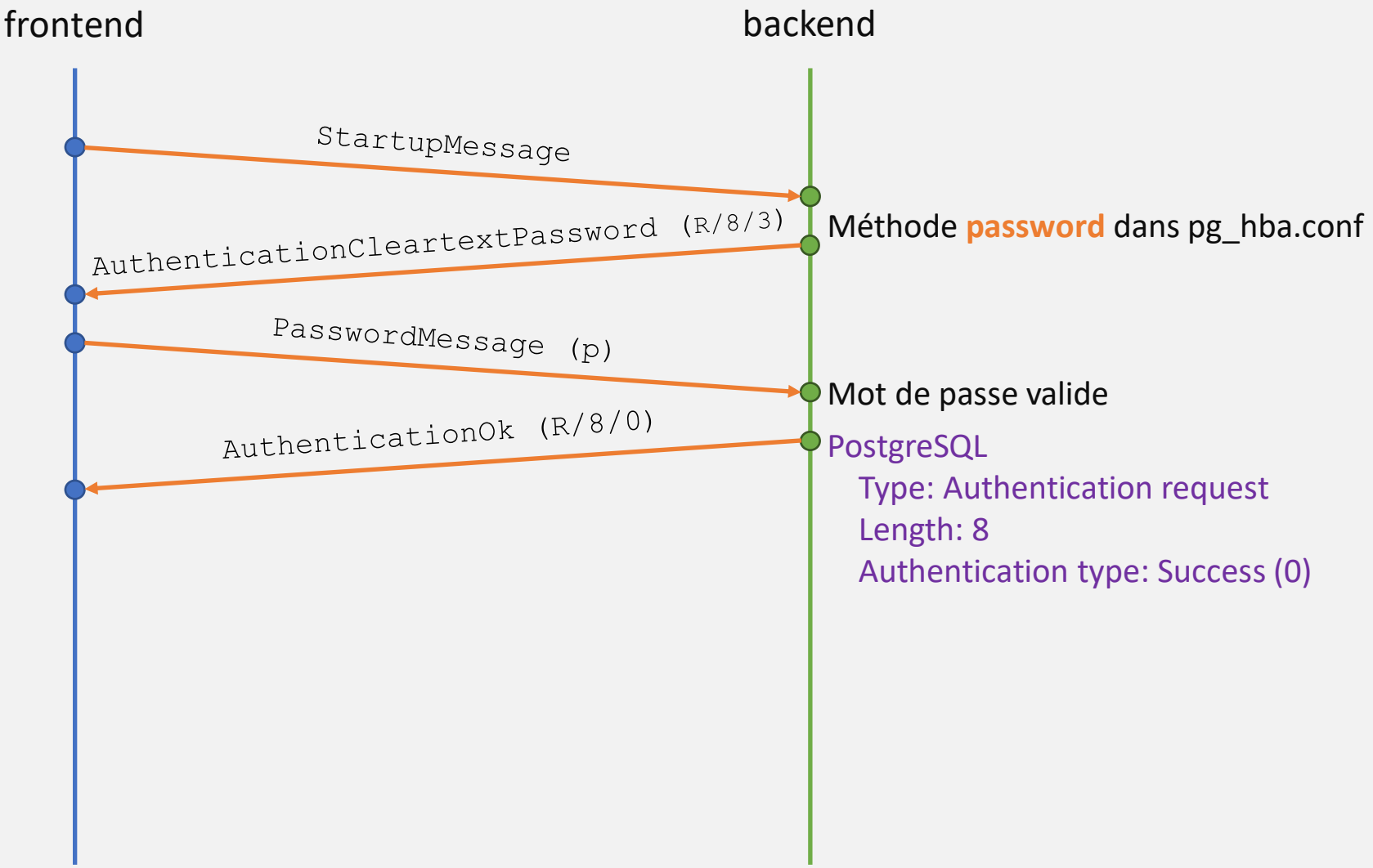

frontend
backend
StartupMessage
PostgreSQL
Type: Startup message
Length: 82
ProtocolVersionNumber: 196608 => 0x30000 (3.0)
Parameter name: user
Parameter value: pascal
Parameter name: database
Parameter value: postgres
Parameter name: application_name
Parameter value: psql
Parameter name: client_encoding
Parameter value: UTF8


```
# TYPE  DATABASE     USER         ADDRESS  METHOD
host    all          raymond      samenet  reject
host    all          pascal       samenet  password
host    replication  replicator   samenet  md5
host    all          all          samenet  md5
```

Méthode **password** dans pg_hba.conf

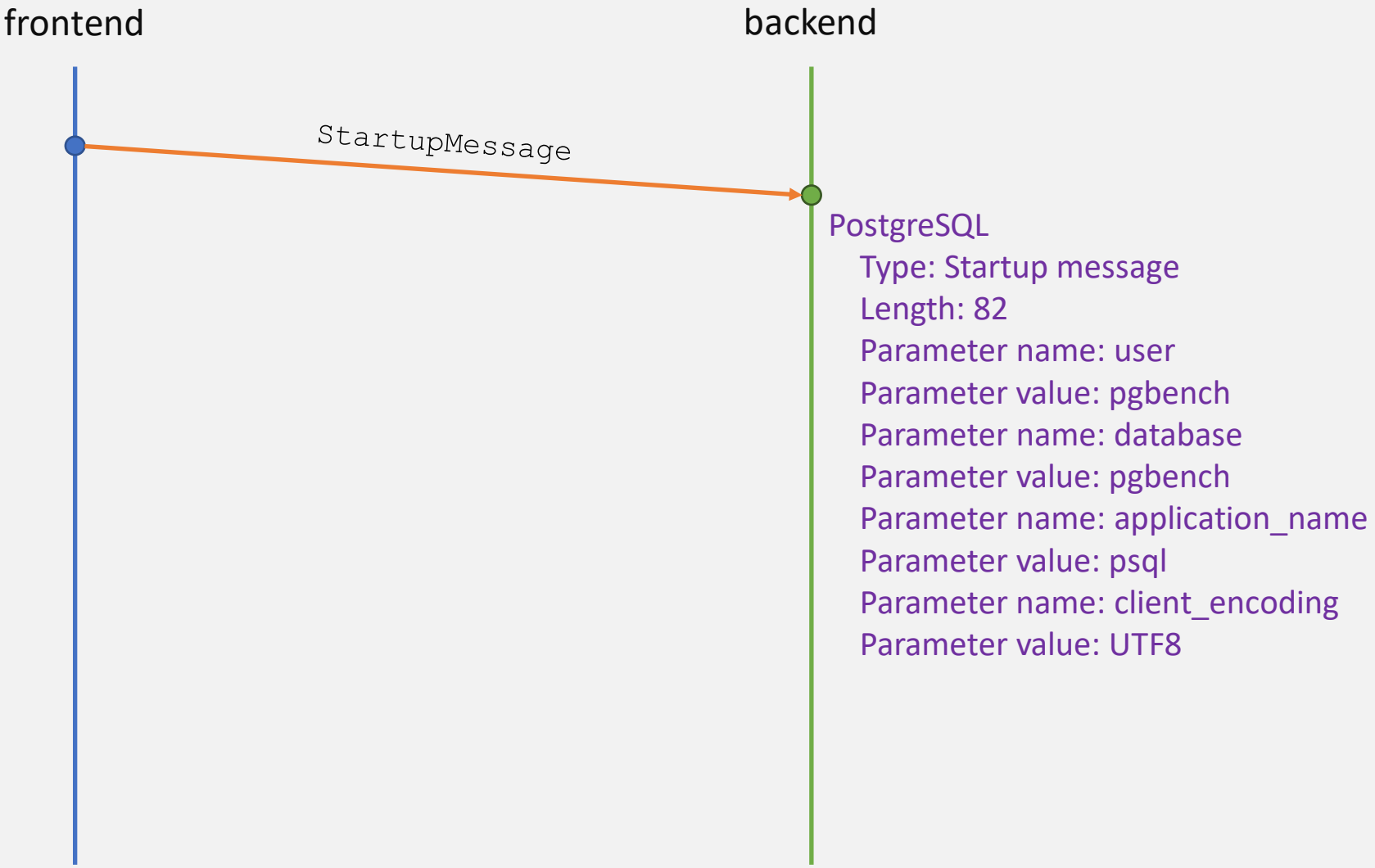
PostgreSQL
- Type: Authentication request
- Length: 8
- Authentication type: Plaintext password (3)


frontend
backend
StartupMessage
AuthenticationCleartextPassword (R/8/3)
Méthode password dans pg_hba.conf
PasswordMessage (p)
PostgreSQL
Type: Password message
Length: 18
Password: pascal_secret


frontend
backend
StartupMessage
Méthode password dans pg_hba.conf
AuthenticationCleartextPassword (R/8/3)
PasswordMessage (p)
Mot de passe valide
AuthenticationOk (R/8/0)
PostgreSQL
Type: Authentication request
Length: 8
Authentication type: Success (0)

Protocole PostgreSQL

Authentication – md5 method

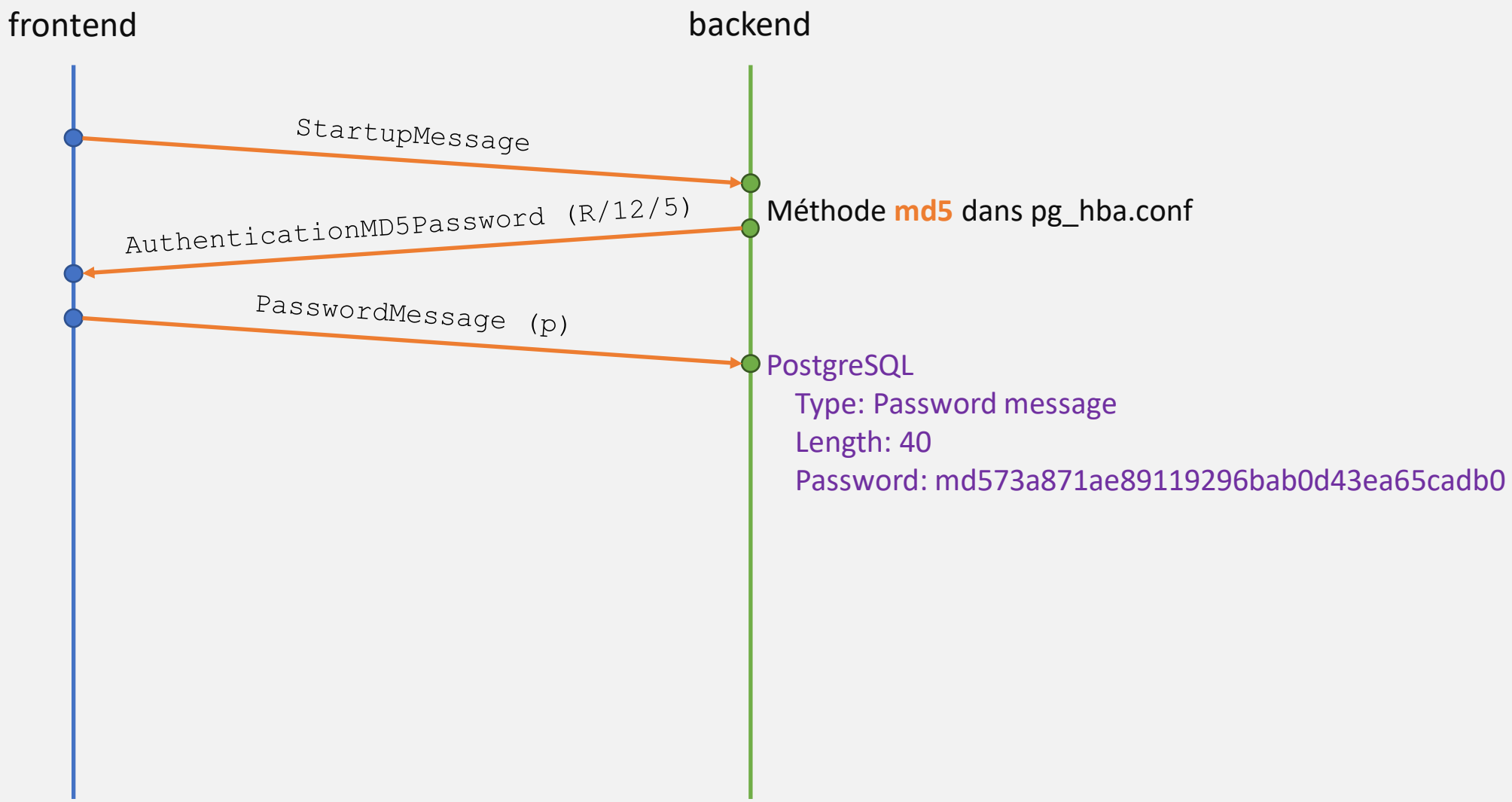

frontend
backend
StartupMessage
PostgreSQL
Type: Startup message
Length: 82
Parameter name: user
Parameter value: pgbench
Parameter name: database
Parameter value: pgbench
Parameter name: application_name
Parameter value: psql
Parameter name: client_encoding
Parameter value: UTF8

Méthode **md5** dans pg_hba.conf

PostgreSQL
- Type: Authentication request
- Length: 12
- Authentication type: MD5 password (5)
- Salt value: 272e110e

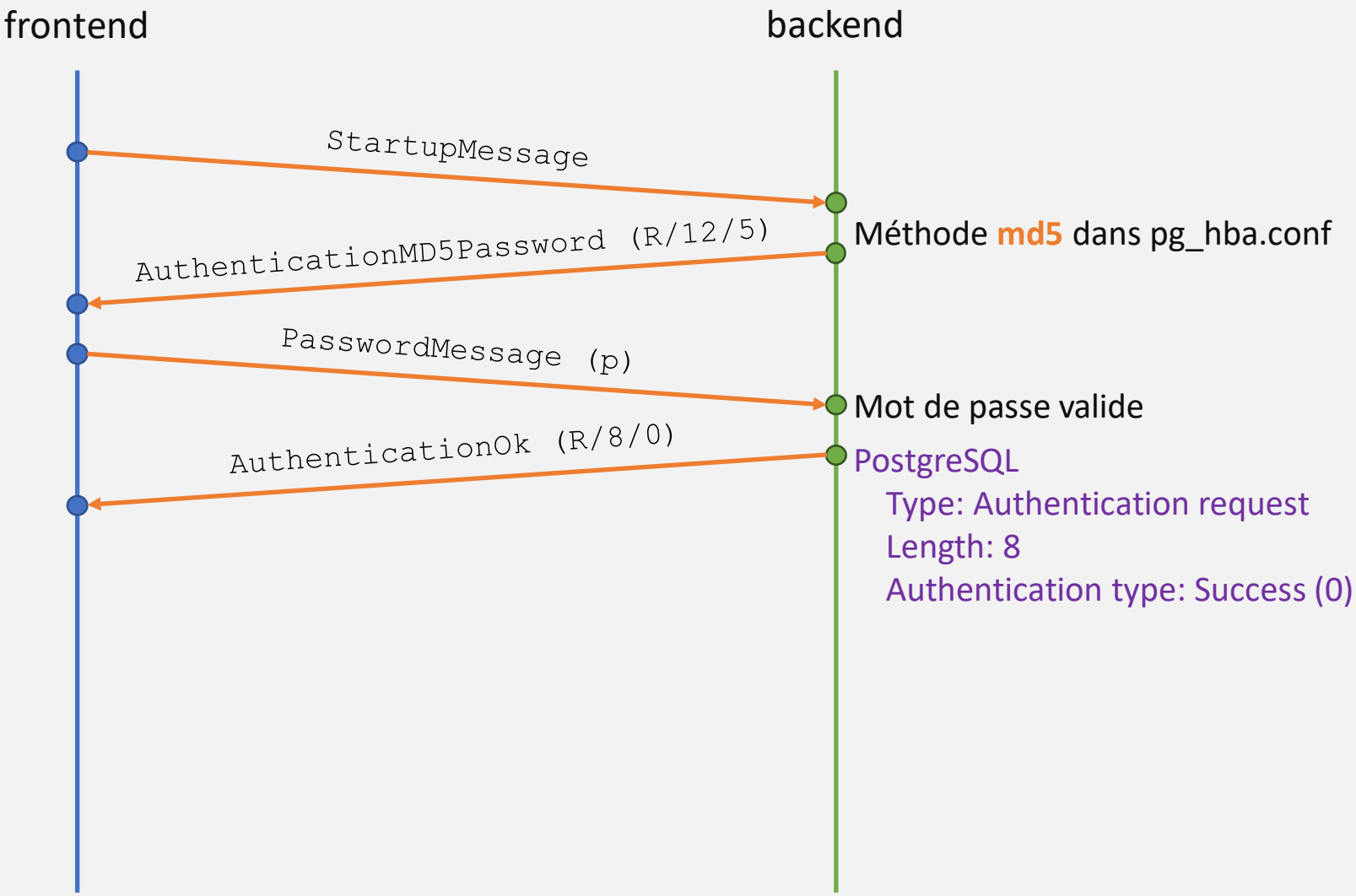
```
# TYPE  DATABASE     USER         ADDRESS  METHOD
host    all          raymond      samenet  reject
host    all          pascal       samenet  password
host    replication  replicator   samenet  md5
host    all          all          samenet  md5
```

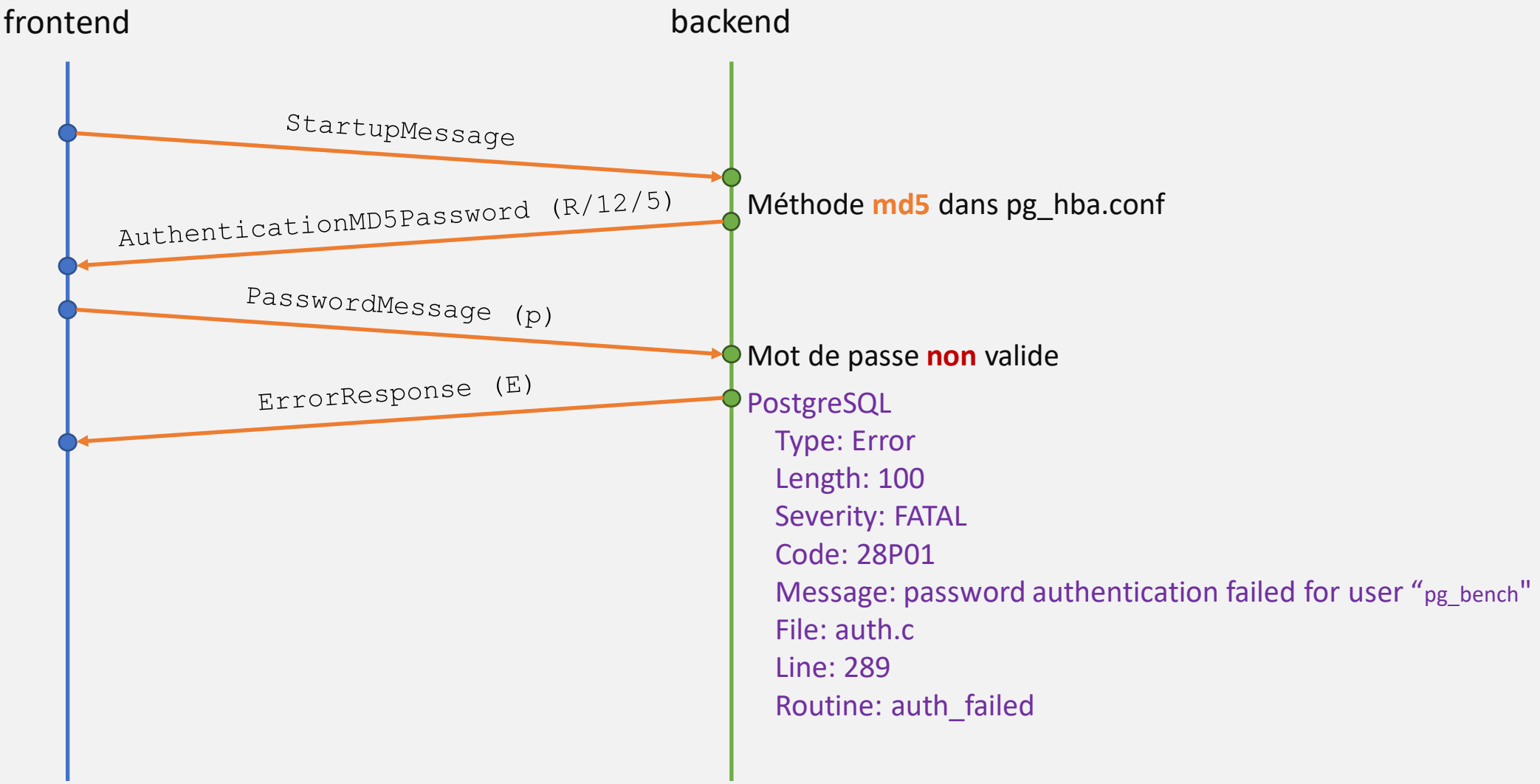
DataBene




```
PasswordMessage = concat('md5', md5(concat(md5(concat(password, username)), random-salt)))
```

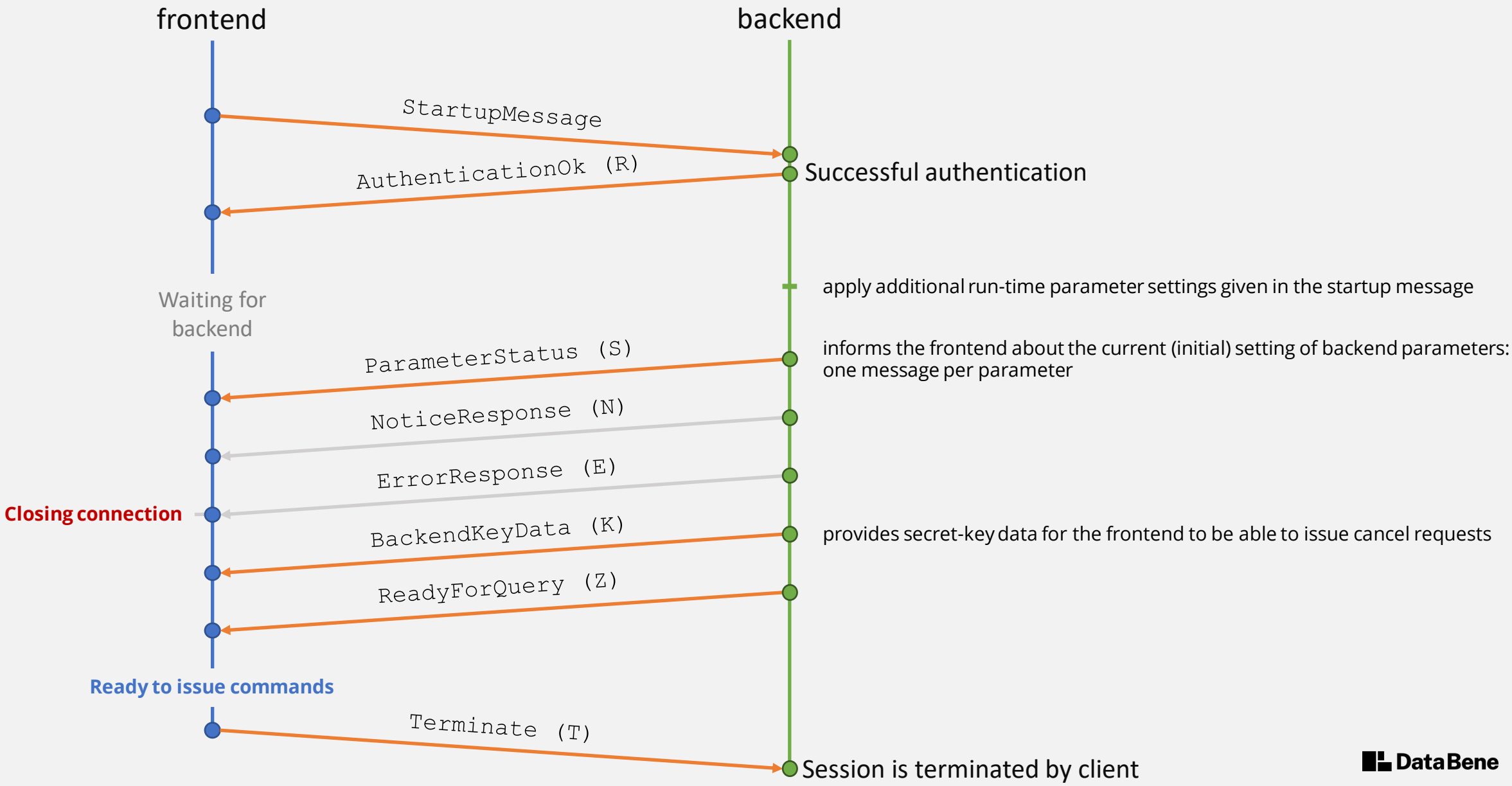
DataBene


frontend
backend
StartupMessage
Méthode md5 dans pg_hba.conf
AuthenticationMD5Password (R/12/5)
PasswordMessage (p)
Mot de passe valide
AuthenticationOk (R/8/0)
PostgreSQL
Type: Authentication request
Length: 8
Authentication type: Success (0)


frontend
backend
StartupMessage
AuthenticationMD5Password (R/12/5)
PasswordMessage (p)
ErrorResponse (E)
Méthode md5 dans pg_hba.conf
Mot de passe non valide
PostgreSQL
Type: Error
Length: 100
Severity: FATAL
Code: 28P01
Message: password authentication failed for user “pg_bench"
File: auth.c
Line: 289
Routine: auth_failed

Protocole PostgreSQL

Messages – Initialisation


frontend
backend
StartupMessage
AuthenticationOk (R)
Successful authentication
Waiting for backend
apply additional run-time parameter settings given in the startup message
ParameterStatus (S)
informs the frontend about the current (initial) setting of backend parameters: one message per parameter
NoticeResponse (N)
ErrorResponse (E)
Closing connection
BackendKeyData (K)
provides secret-key data for the frontend to be able to issue cancel requests
ReadyForQuery (Z)
Ready to issue commands
Terminate (T)
Session is terminated by client

Parameter Status - BackendKeyData - ReadyForQuery

PostgreSQL
Type: Parameter status
Length: 26
Parameter name: application_name
Parameter value: psql

PostgreSQL
Type: Parameter status
Length: 27
Parameter name: IntervalStyle
Parameter value: postgres

PostgreSQL
Type: Parameter status
Length: 34
Parameter name: **session_authorization**
Parameter value: pgbench

PostgreSQL
Type: Parameter status
Length: 25
Parameter name: **client_encoding**
Parameter value: UTF8

PostgreSQL
Type: Parameter status
Length: 21
Parameter name: **is_superuser**
Parameter value: off

PostgreSQL
Type: Parameter status
Length: 35
Parameter name: standard_conforming_strings
Parameter value: on

PostgreSQL
Type: Parameter status
Length: 23
Parameter name: **DateStyle**
Parameter value: ISO, MDY

PostgreSQL
Type: Parameter status
Length: 25
Parameter name: **server_encoding**
Parameter value: UTF8

PostgreSQL
Type: Parameter status
Length: 26
Parameter name: TimeZone
Parameter value: Europe/Paris

PostgreSQL
Type: Parameter status
Length: 25
Parameter name: integer_datetimes
Parameter value: on

PostgreSQL
Type: Parameter status
Length: 26
Parameter name: **server_version**
Parameter value: 9.4.26

PostgreSQL
Type: Backend key data
Length: 12
PID: 1133
Key: 482213844

PostgreSQL
Type: Ready for query
Length: 5
Status: Idle (73)

DataBene

Protocole PostgreSQL

Messages – Canceling et Termination

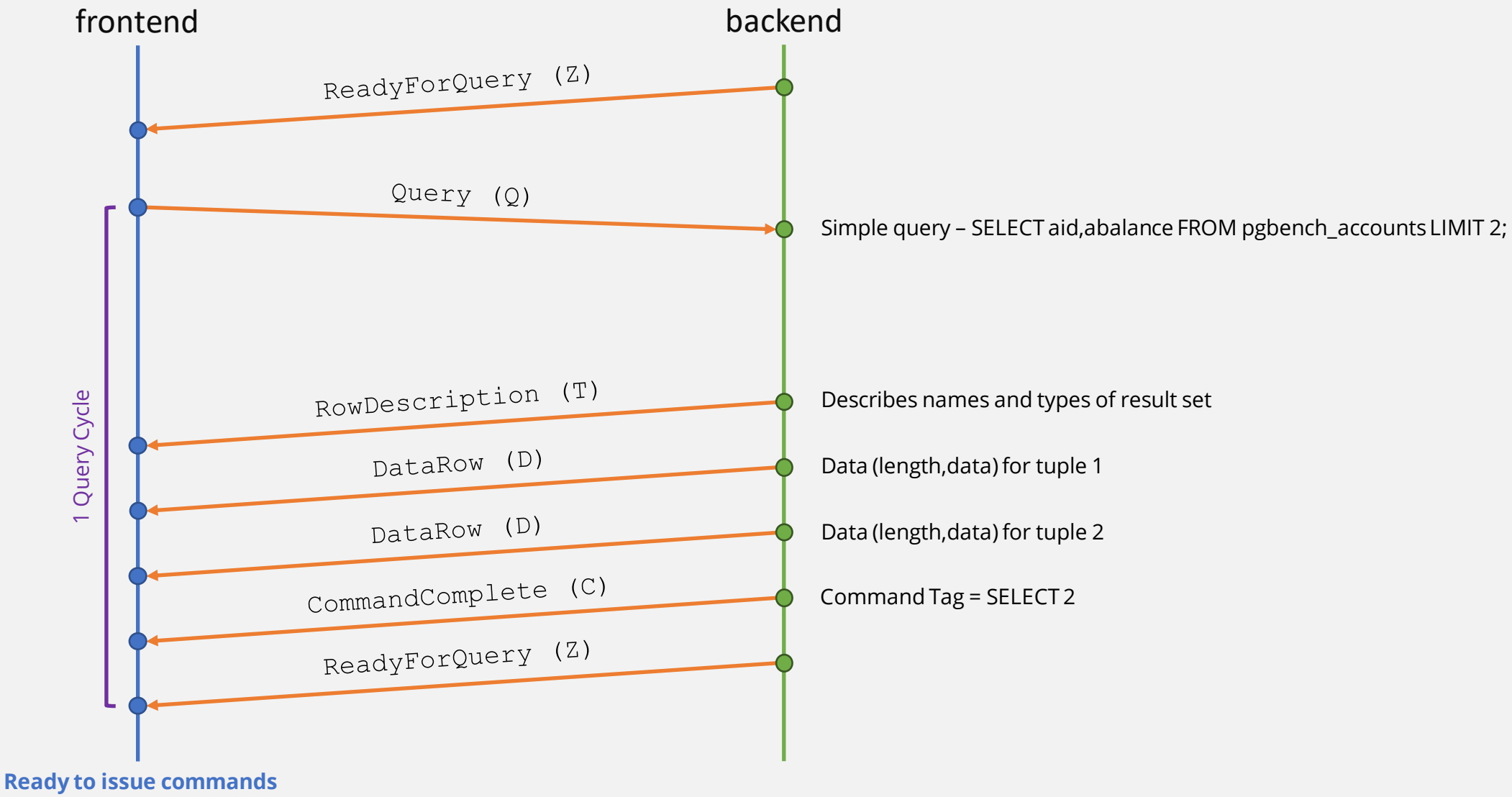
Annulation et Terminaison depuis le frontend

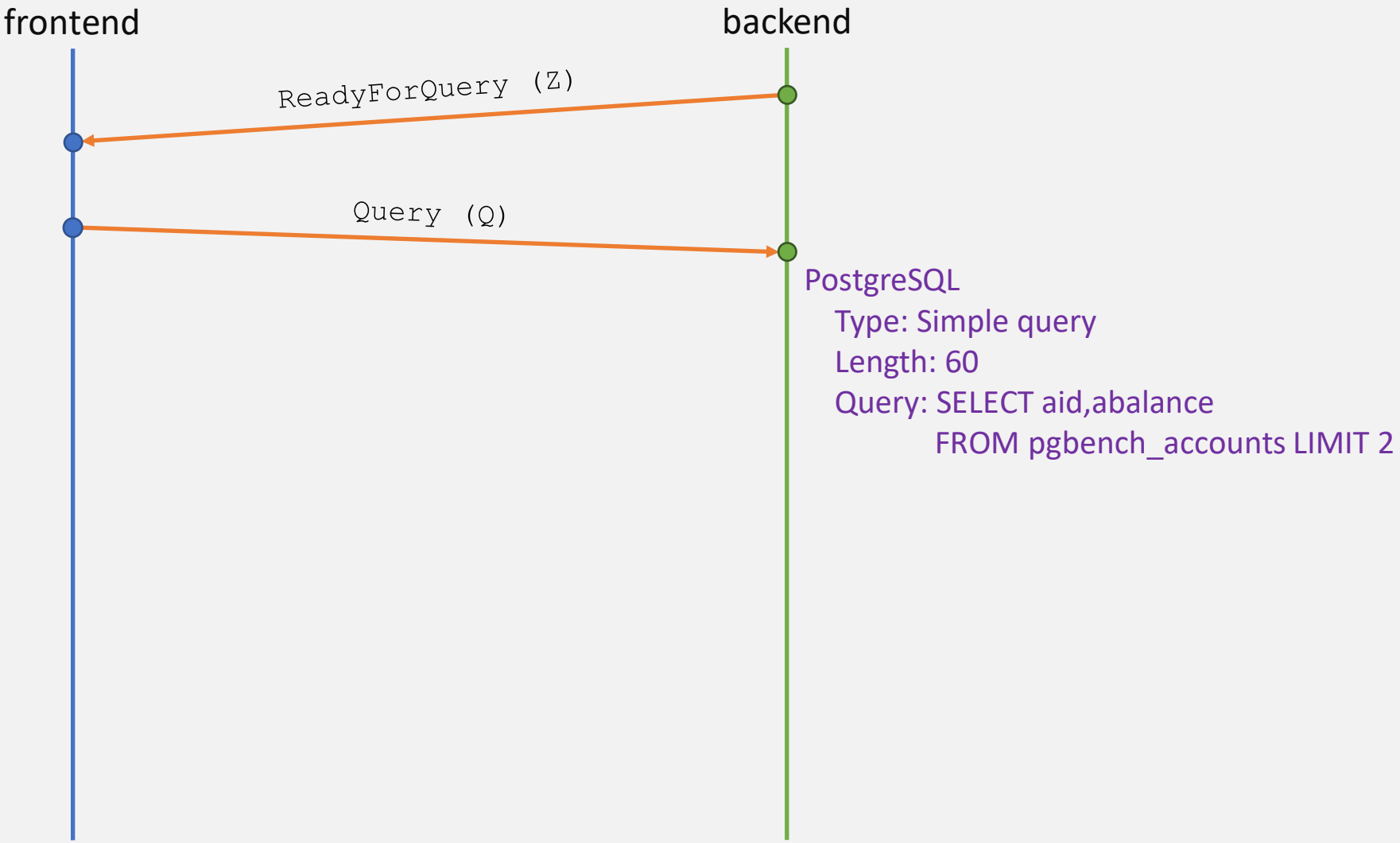
- Annuler une commande (e.g.; une requête) :
  - Via une nouvelle connexion à PostgreSQL,
  - Le message `CancelRequest` (0x04d2162e,pid,**key**) remplace `StartupMessage`,
  - Pour des raisons de sécurité, aucun message ou information n'est renvoyé au frontend.

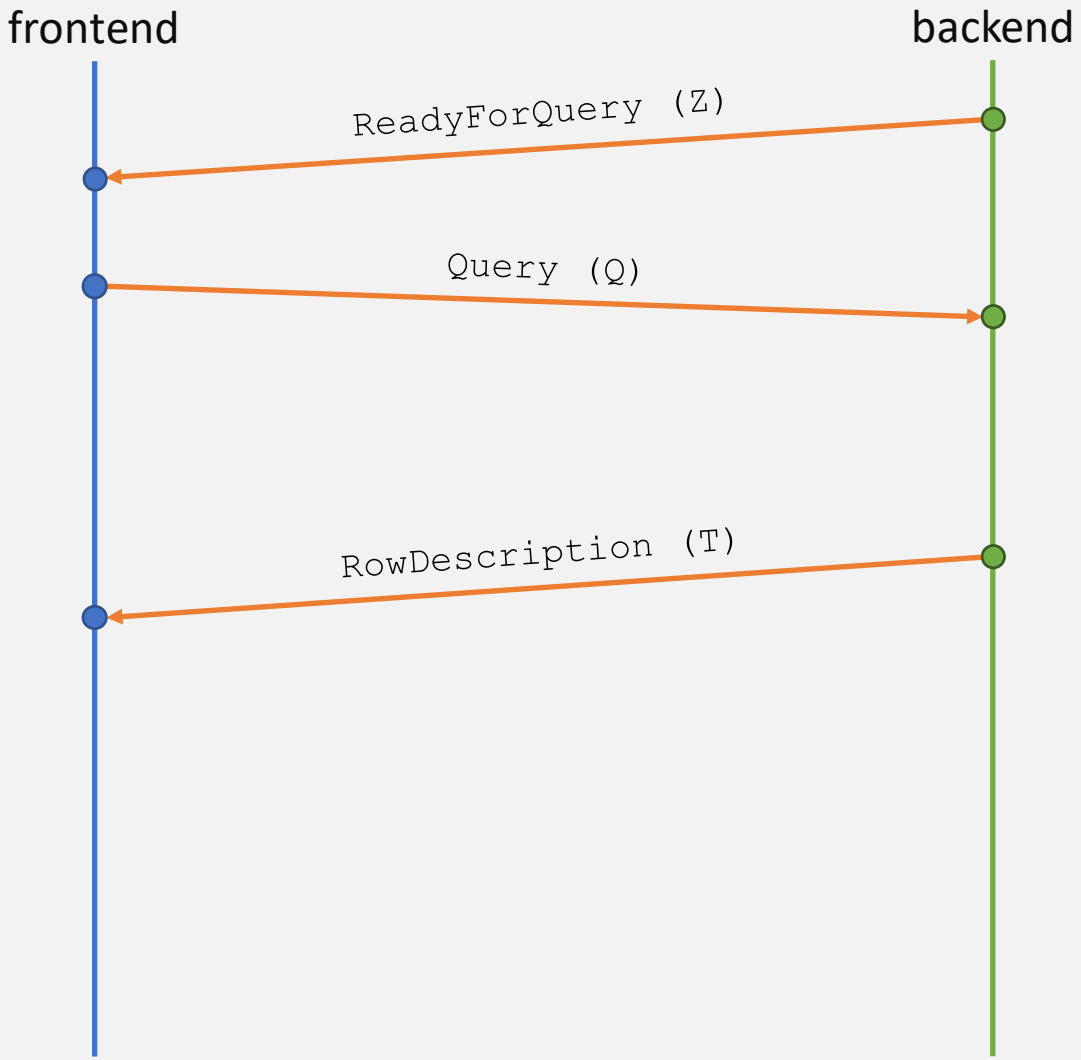
- Terminer une session
  - Le **frontend** envoie le message `Terminate` au **backend**,
  - Le frontend ferme la connexion,
  - Le backend ferme la connexion puis met un terme à son exécution.

Protocole PostgreSQL

Messages – Simple query


frontend
backend
ReadyForQuery (Z)
Query (Q)
Simple query – SELECT aid,abalance FROM pgbench_accounts LIMIT 2;
1 Query Cycle
RowDescription (T)
Describes names and types of result set
DataRow (D)
Data (length,data) for tuple 1
DataRow (D)
Data (length,data) for tuple 2
CommandComplete (C)
Command Tag = SELECT 2
ReadyForQuery (Z)
Ready to issue commands


frontend
backend
ReadyForQuery (Z)
Query (Q)
PostgreSQL
Type: Simple query
Length: 60
Query: SELECT aid,abalance
FROM pgbench_accounts LIMIT 2

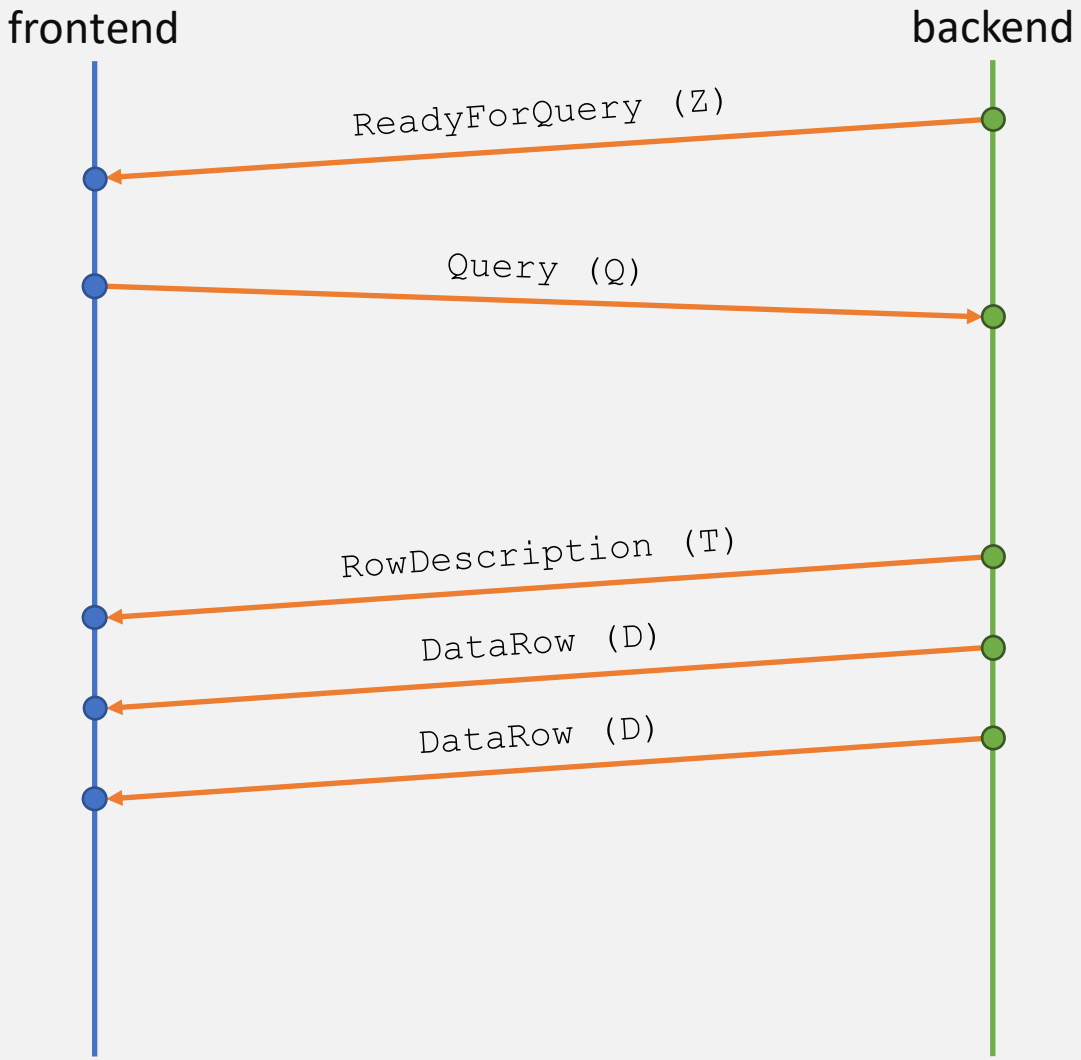


Describes names and types of result set.

PostgreSQL
- Type: Row description
- Length: 55
- Field count: 2

Column name: aid
- Table OID: 16416
- Column index: 1
- Type OID: 23
- Column length: 4
- Type modifier: -1
- Format: Text (0)

Column name: abalance
- Table OID: 16416
- Column index: 3
- Type OID: 23
- Column length: 4
- Type modifier: -1
- Format: Text (0)



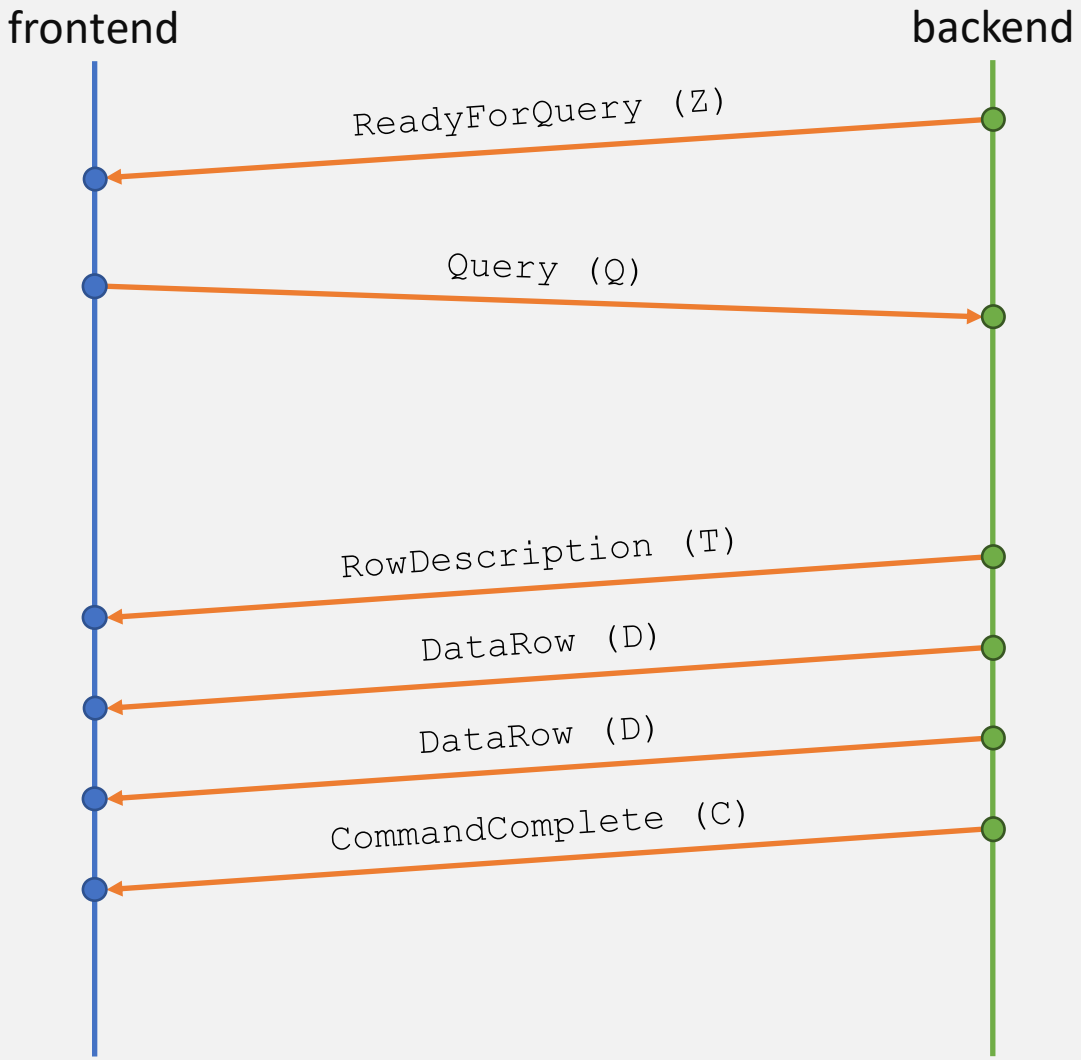
Tuple Data (result set of 2 tuples).

PostgreSQL
- Type: Data row
- Length: 16
- Field count: 2
  - Column length: 1
  - Data: 32
  - Column length: 1
  - Data: 30

PostgreSQL
- Type: Data row
- Length: 16
- Field count: 2
  - Column length: 1
  - Data: 33
  - Column length: 1
  - Data: 30

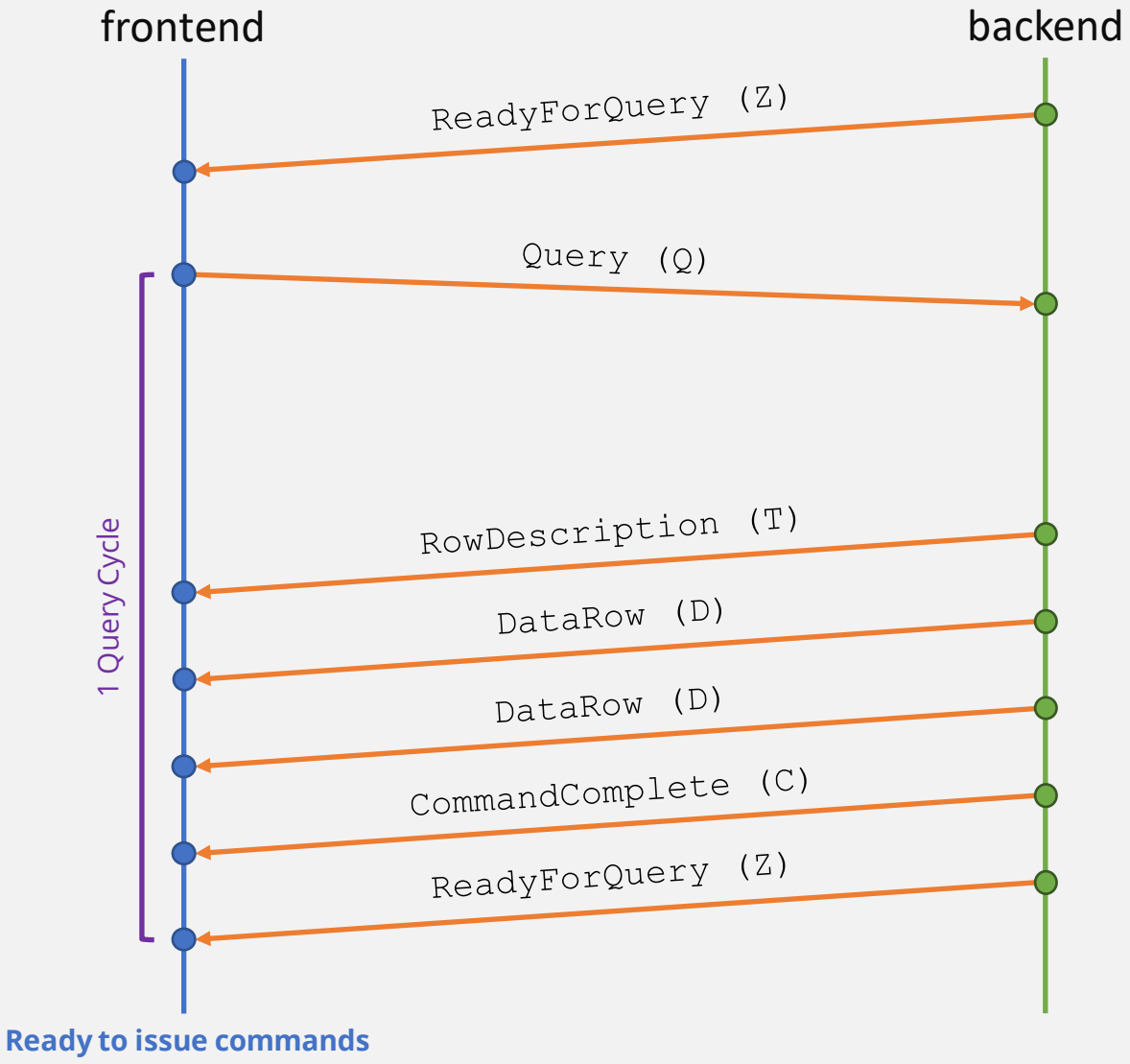
```
 aid | abalance
-----+----------
   2 |         0
   3 |         0
(2 rows)
```

DataBene



Query is complete.

PostgreSQL
- Type: Command completion
- Length: 13
- Tag: SELECT 2



The Backend is ready to run another command

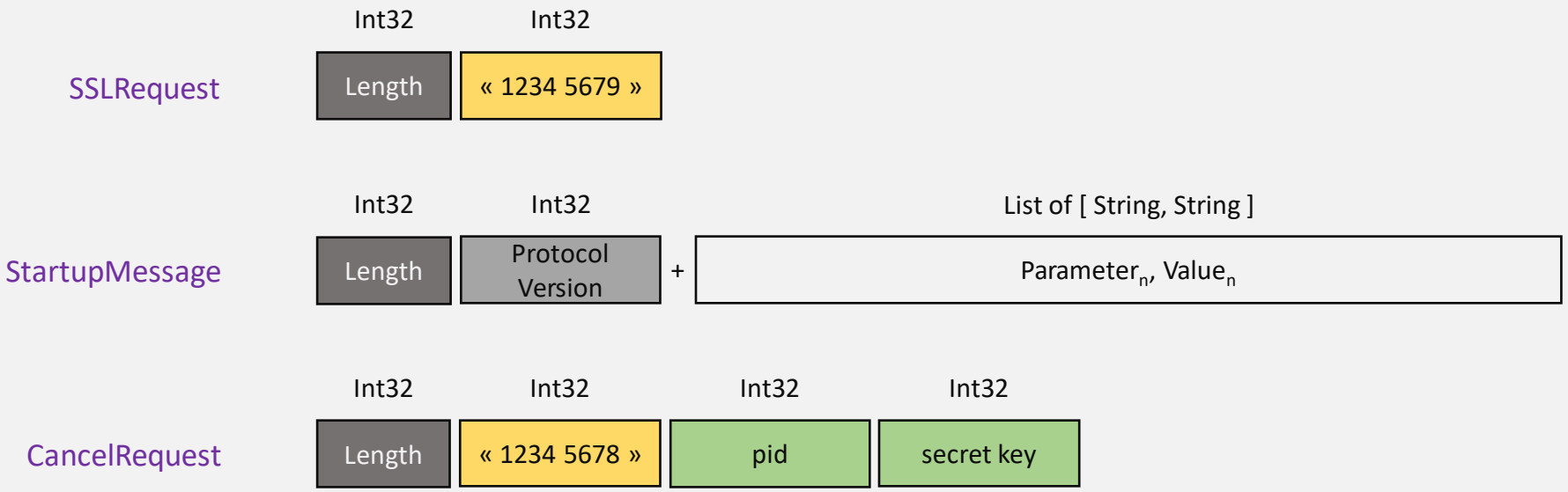
PostgreSQL
- Type: Ready for query
- Length: 5
- Status: Idle (73) | Transaction | Error

DataBene

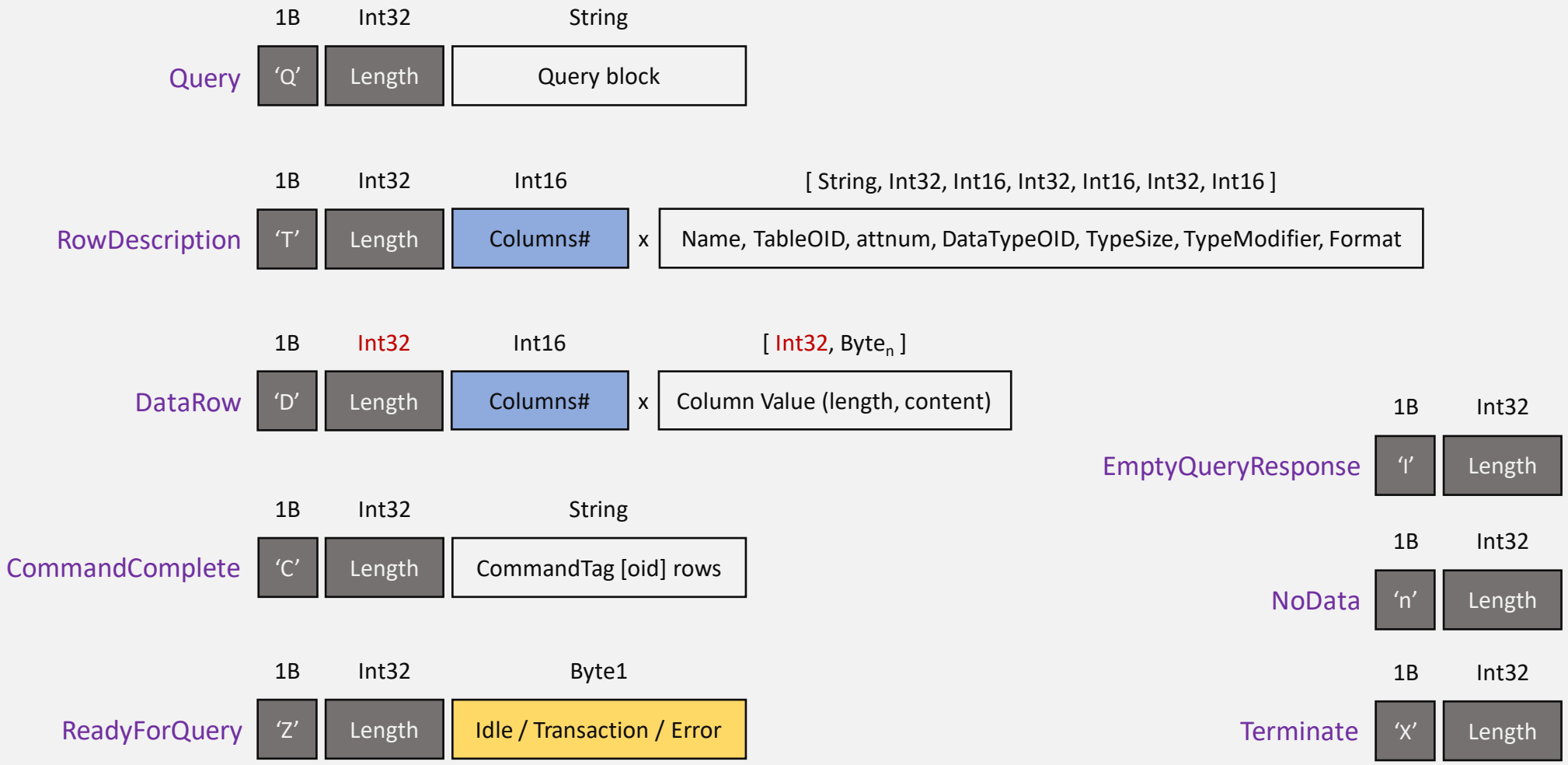
Protocole PostgreSQL

Messages – Structures

Messages d'établissement de connexion

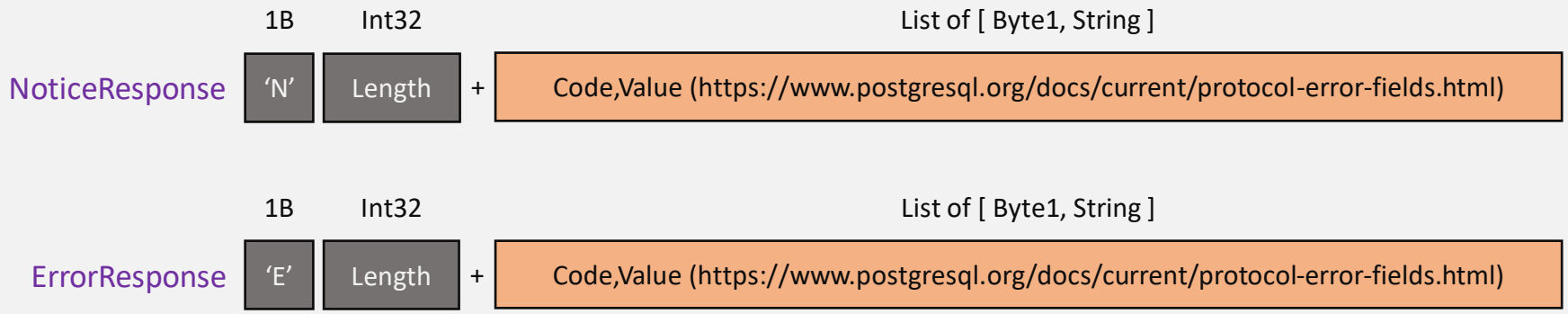


Messages de traitement



DataBene

Messages d’erreurs



Protocole PostgreSQL

Messages – Simple query - multistatements

Simple query – Multiple Statements

```
INSERT INTO mytable VALUES(1);
INSERT INTO mytable VALUES(2);
```

Simple query – Multiple Statements

Par défaut

```
INSERT INTO mytable VALUES(1);
INSERT INTO mytable VALUES(2);
```

est traité comme

```
BEGIN;
INSERT INTO mytable VALUES(1);
INSERT INTO mytable VALUES(2);
COMMIT;
```

Simple query – Multiple Statements

Erreur à l'exécution du bloc de statements

```
INSERT INTO mytable VALUES(1);
SELECT 1/0;
INSERT INTO mytable VALUES(2);
```

L'erreur de division par zéro provoque le **rollback** du premier INSERT.

L'exécution du bloc de statements étant **abandonnée**, les statements suivants sont **ignorés**.

Simple query – Multiple Statements

Le comportement par défaut peut être modifié :

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELECT 1/0;
```

Implicit transaction block

Le premier INSERT est explicitement validé par l'ordre COMMIT.

Le second INSERT et le SELECT sont considérés faisant partie d'une même transaction implicite, donc l'erreur divide-by-zero annule le second INSERT, mais pas le premier.

Simple query – Multiple Statements

Analyse syntaxique avant exécution :

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELCT 1/0;
```

Puisque l'analyse syntaxique de tous les statements est faite avant l'execution du premier statement, la phase d'execution n'est pas initiée du fait de l'erreur de syntaxe.

Protocole PostgreSQL

Messages – Extended query

Extended Query – Prepared Statements

Parser
Analyzer
Rewritter

```
PREPARE fooplan (int, text, bool, numeric) AS
INSERT INTO foo VALUES($1, $2, $3, $4);
```

Planner
Executor

```
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

```
DEALLOCATE PREPARE fooplan;
```

Extended Query – Query => Parse/Bind/Execute

Le message Query est décomposé en plusieurs messages

Parser
Analyzer
Rewritter

- Le message Parse contient :
  - le nom du prepared statement (named vs unnamed)
  - une seule requête au format texte pouvant contenir des paramètres génériques $1,$2,..,$n
  - le type de chaque paramètre générique $1,$2,..,$n

Le message Parse crée ou modifie un prepared statement.

Planner

- Le message Bind contient :
  - le nom d'un prepared statement (named vs unnamed)
  - le nom d'un portal (named vs unnamed)
  - la valeur des paramètres
  - le format (textuel or binaire) des paramètres
  - le format (textuel or binaire) des résultats

Le message Bind déclenche la planification du prepared statement associé et construit un portal.

Executor

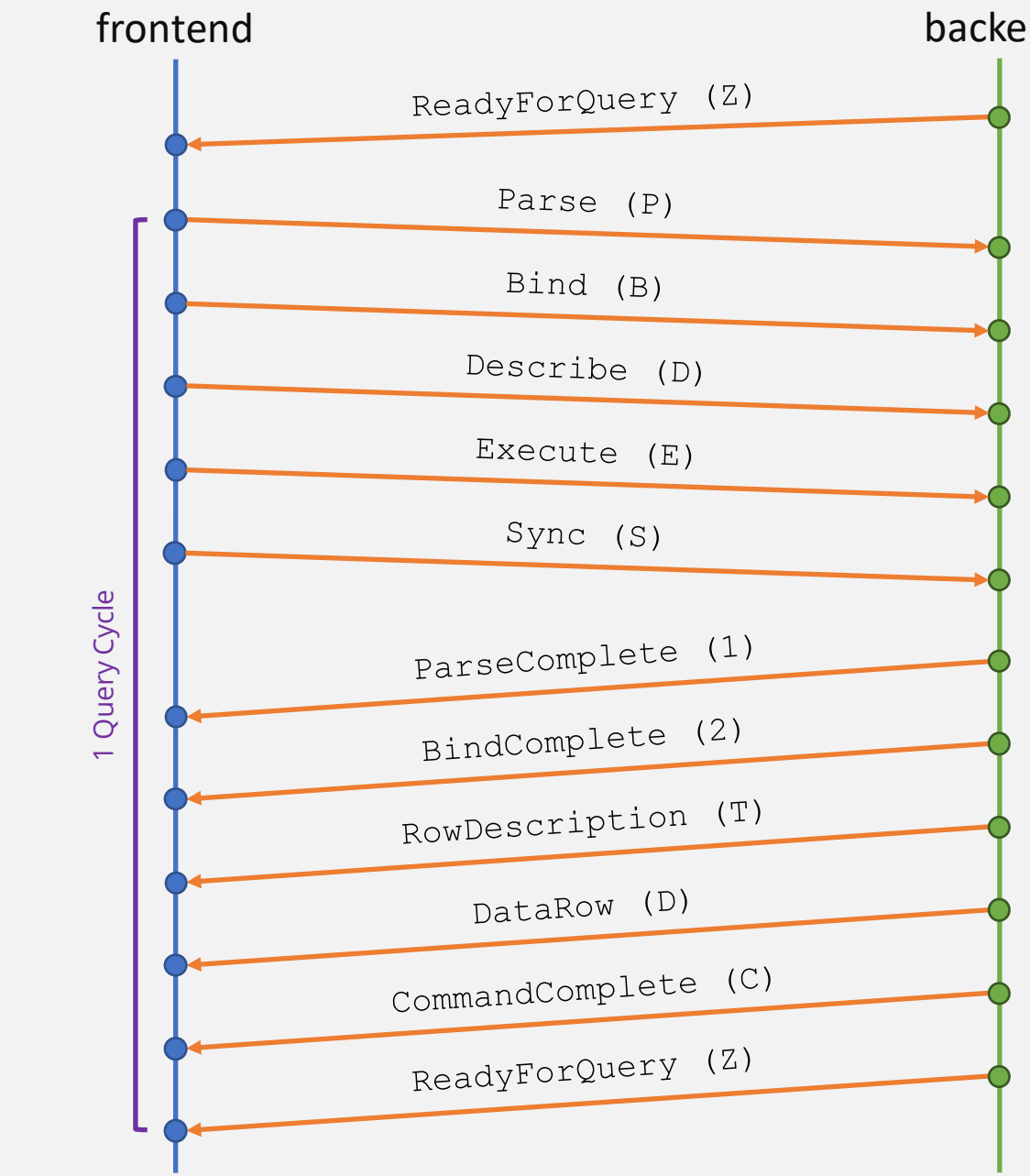
- Le message Execute contient :
  - le nom du portal (named vs unnamed)
  - le nombre maximal de lignes à retourner au frontend

Le message Execute déclenche l'exécution du portal associé.

Extended Query – Query => Parse/Bind/Execute

- Bénéfices
  - Gain de performance possible par factorisation des étapes
    - parser/analyzer/rewritter
    - planner
    - executor
  - Prévient les attaques par SQL Injection sur les valeurs des paramètres.

- Contraintes
  - Générer plusieurs messages au lieu d'un seul.
  - Les objets statement et portal existent dans le contexte d'une session.



Textual query + [placeholder data types] + [prepare statement name]

Placeholder replacements + [output mode] => create new or use existing portal
SELECT * FROM pgbench_accounts WHERE aid = $1; $1 = 5

Asks for output description of (un)named 'S' statement or 'P' portal ('P' here)

Executes (un)named portal

Frontend tells backend it is finished with the current statement (**`ReadyForQuery`**)

Describes names and types of result set

Describes names and types of result set

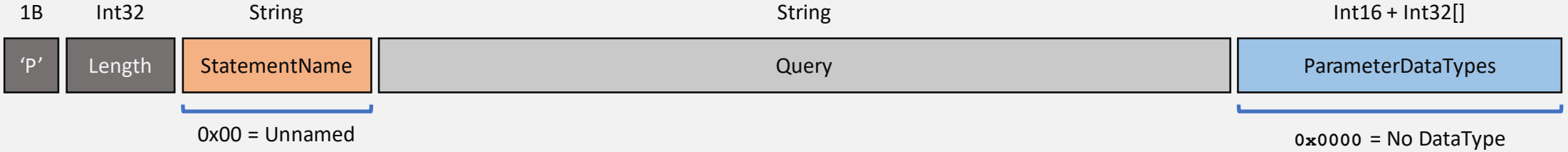
Data (length,data) for tuple 1

Command Tag = SELECT 2

DataBene

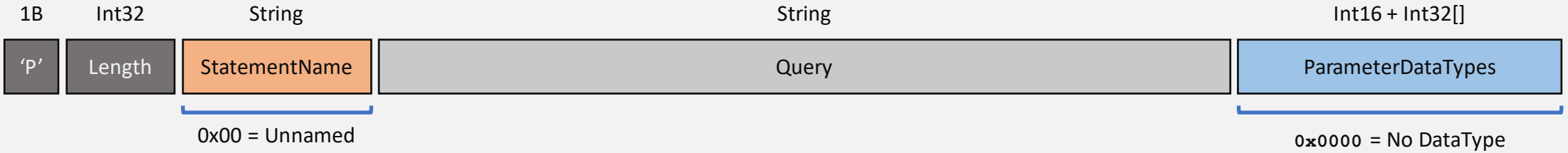
Extended Query – Message Structures

Parse

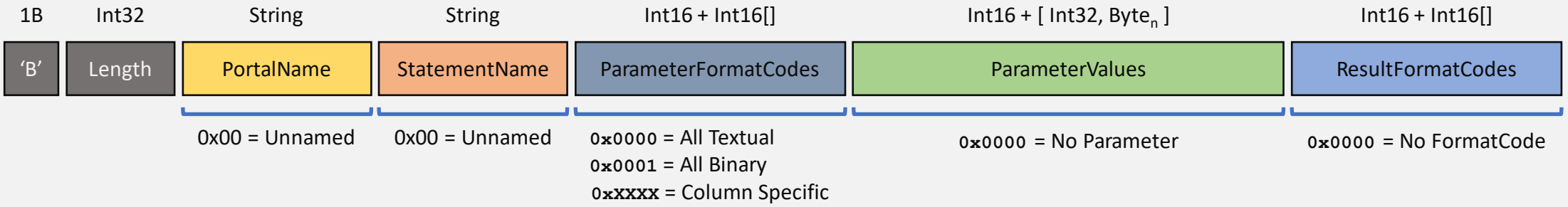


Extended Query – Message Structures

Parse



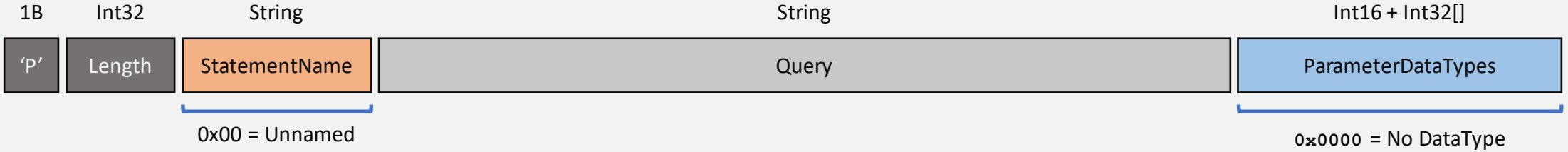
Bind



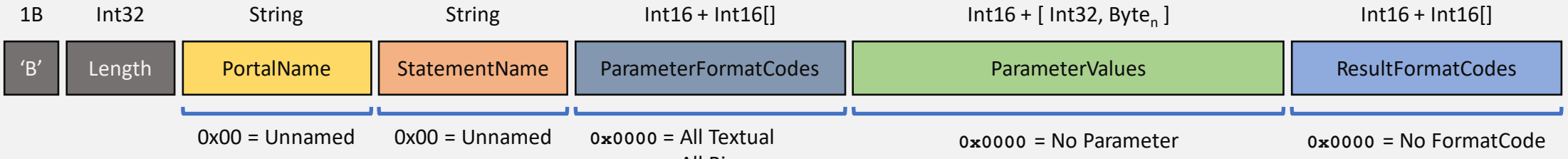
DataBene

Extended Query – Message Structures

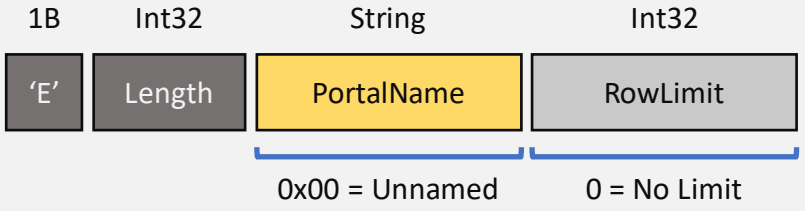
Parse



Bind



Execute



DataBene

Extended Query – Durée de vie des (un)named objects

- Les objets nommés
  - existent jusqu'à la fin de la session ou jusqu'à une suppression explicite de l'objet
  - doivent être explicitement supprimés avant de pouvoir être redéfinis par Parse ou Bind.

- Les objets sans nom
  - existent jusqu'au prochain Parse ou Bind les redéfinissant.
  - Les statements et portals sans nom (unnamed) sont systématiquement écrasés par les nouveaux messages parse/bind/execute liés à aucun objet nommé.
  - Les messages Query détruisent systématiquement les objets sans nom.

Protocole PostgreSQL

Messages – Extended query – JDBC

JDBC génère toujours [P]/B/D/E/S

J’ai observé que

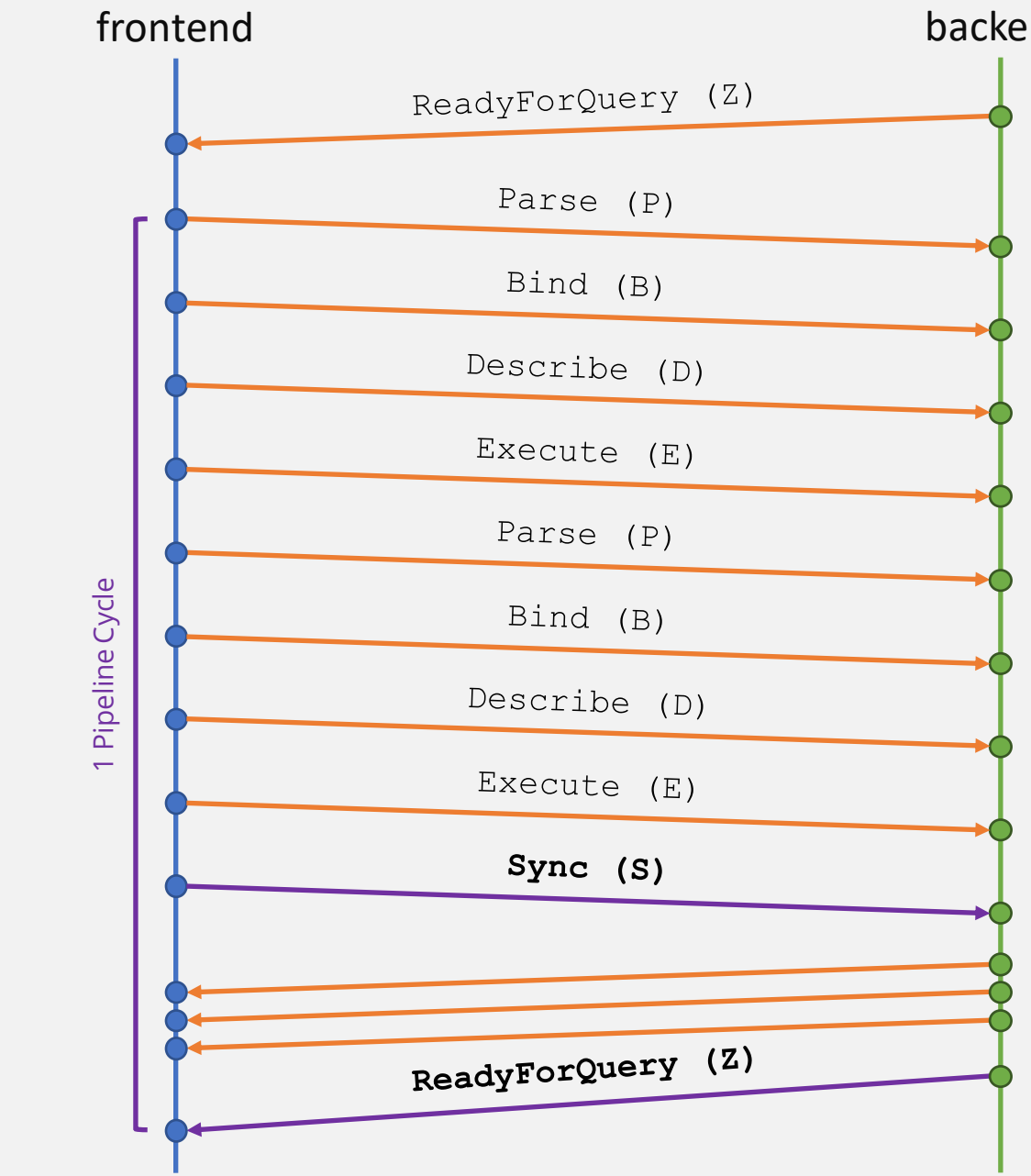
- conn.createStatement() + st.executeQuery()
- Ou conn.preparedStatement() + st.executeQuery()

produit

- `[Parse]` après PrepareThreshold messages, plus aucun message Parse
- `Bind`
- `Describe`
- `Execute`
- `Sync`

Protocole PostgreSQL

Messages – Extended query - Pipelining


frontend
backend
ReadyForQuery (Z)
Parse (P)
BEGIN
Bind (B)
Describe (D)
Execute (E)
Parse (P)
In the meanwhile enqueue all queries you want to pipeline: repeat (P/B/D/E).
Bind (B)
Describe (D)
Execute (E)
COMMIT
Sync (S)
Frontend tells backend it is finished with the current buffer.
ReadyForQuery (Z)
Success when COUNT(Sync) = COUNT(ReadyForQuery)
1 Pipeline Cycle
Ready to issue commands

DataBene

Protocole PostgreSQL

Messages – Streaming Replication Protocol

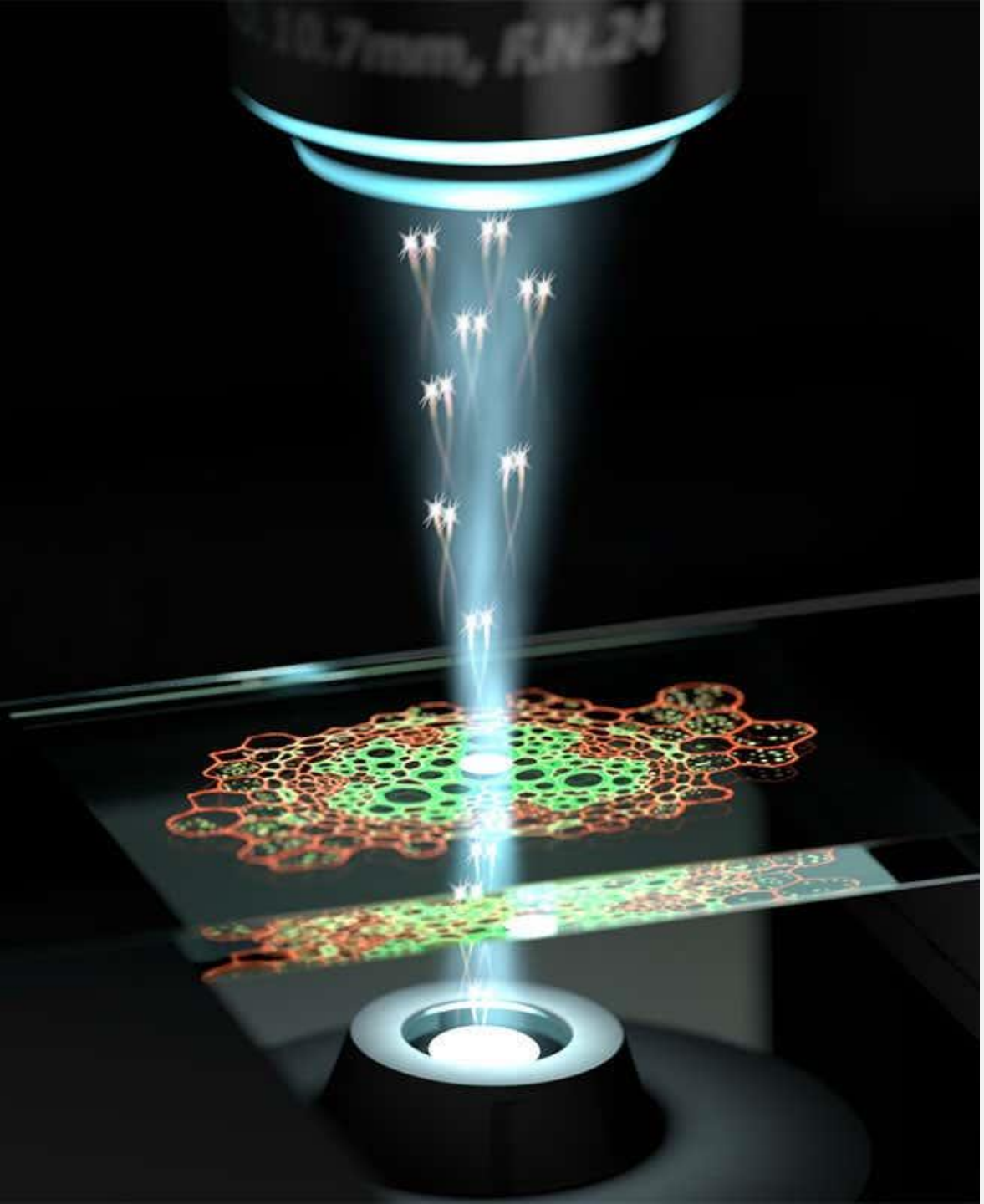
Streaming Replication

- Activé par la présence du paramètre `replication` dans `StartupMessage`.
- `replication` = **`on`** | **`yes`** | **`1`**
  => walsender en mode réplication physique
  => seulement des commandes de réplication
- `replication` = **`database`**
  => walsender en mode de réplication logique
  => commandes de réplication + commandes SQL
- Supporte uniquement le mode Simple Query

Replication Commands

```
IDENTIFY_SYSTEM

SHOW name

TIMELINE_HISTORY tli

CREATE_REPLICATION_SLOT slot_name [ TEMPORARY ] { PHYSICAL | LOGICAL output_plugin } [ ( option [, ...] ) ]

READ_REPLICATION_SLOT slot_name

START_REPLICATION [ SLOT slot_name ] [ PHYSICAL ] XXX/XXX [ TIMELINE tli ]
START_REPLICATION SLOT slot_name LOGICAL XXX/XXX [ ( option_name [ option_value ] [, ...] ) ]

DROP_REPLICATION_SLOT slot_name [ WAIT ]

BASE_BACKUP [ ( option [, ...] ) ]
```

Documentation: https://www.postgresql.org/docs/current/protocol-replication.html



Introduction

Connexions PostgreSQL

Poolers de Connexions

- À quoi servent-ils ?
- Éléments d’architecture
- Modes de Pooling
- Les poolers NextGen

Summary

Poolers de Connexions

À quoi servent-ils ?

Poolers de Connexions – Objectifs

Restreindre le nombre de processus actifs,
pour que le système opère dans sa zone de fonctionnement optimale.

La plupart des processus venant de `max_connections` (e.g.; sql backends)
le levier principal est de limiter le nombre de connexions actives.

Masquer voire supprimer les durées d'établissement des connexions.

Poolers de Connexions – Conséquences

- Les clients se connectent au pooler de connexions.
- Lorsque le pooler de connexions n'a plus de connexion disponible, les demandes clients sont insérées dans une file d'attente

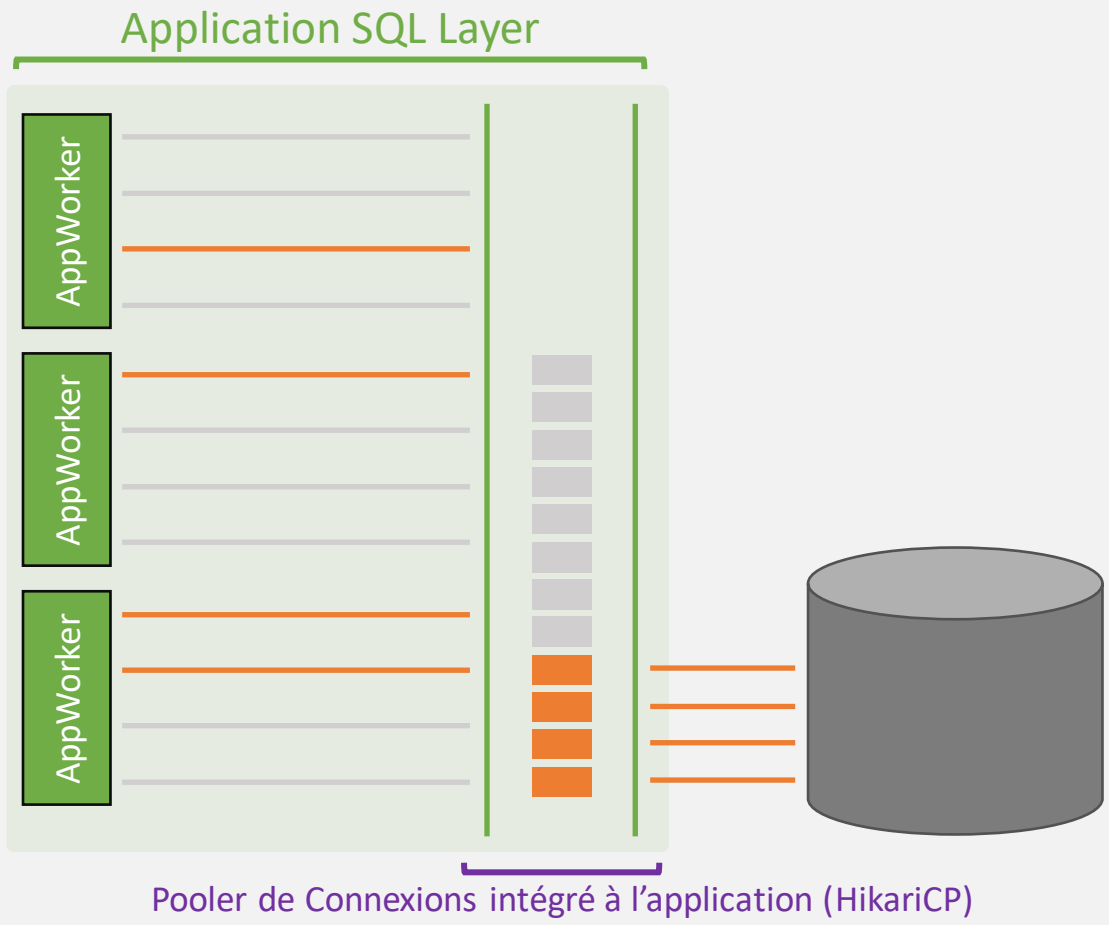
  => Les pics d'activité peuvent être absorbés avec ce mécanisme.
- Le pool de connexions peut agir comme un interrupteur pour autoriser ou interdire l'accès à un nœud PostgreSQL.

  => Lors d'un failover, le Fencing peut être assuré par le pooler.

Poolers de Connexions

Élément applicatif

Poolers de Connexions – Élément Applicatif

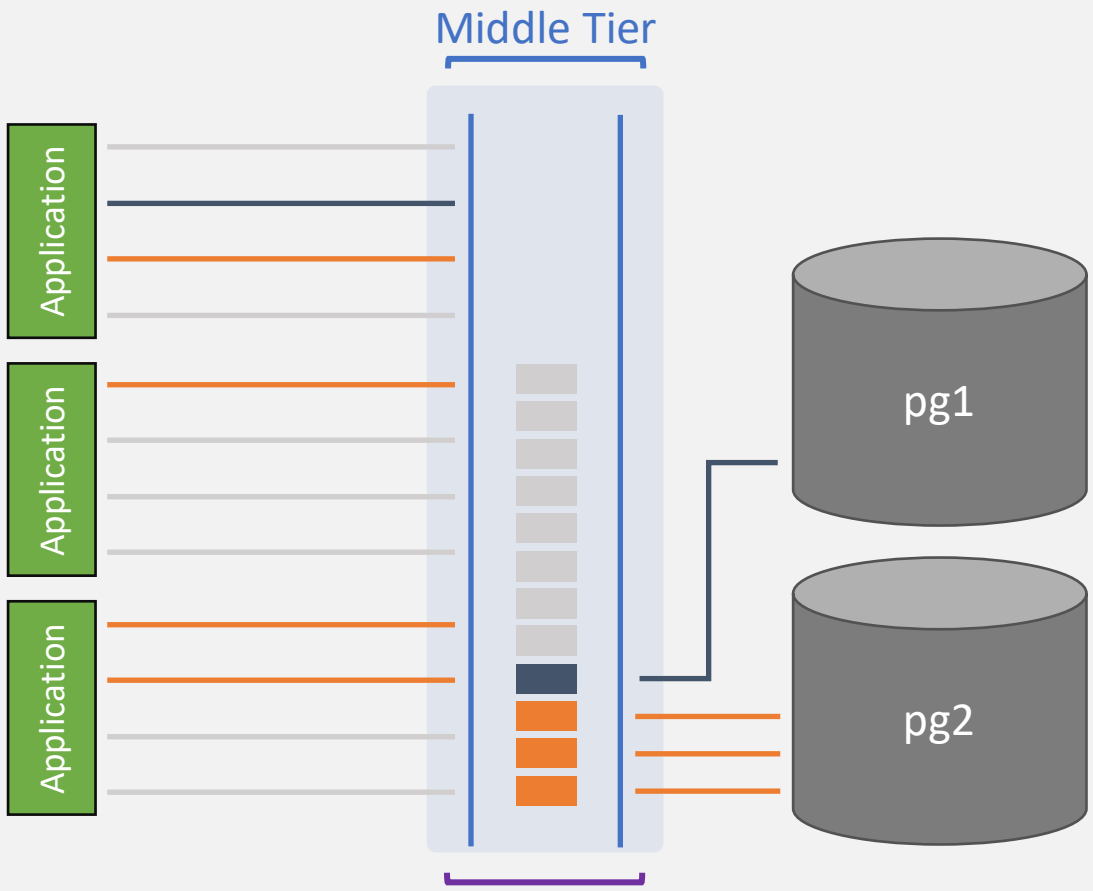


DataBene

Poolers de Connexions

Élément déporté

Connection Poolers – Middle Tier (Proxy)



Le pooler de connexions a son propre serveur.
Le pooler de connexion **proxy** se comporte comme un serveur PostgreSQL face au client et comme un client face au serveur.

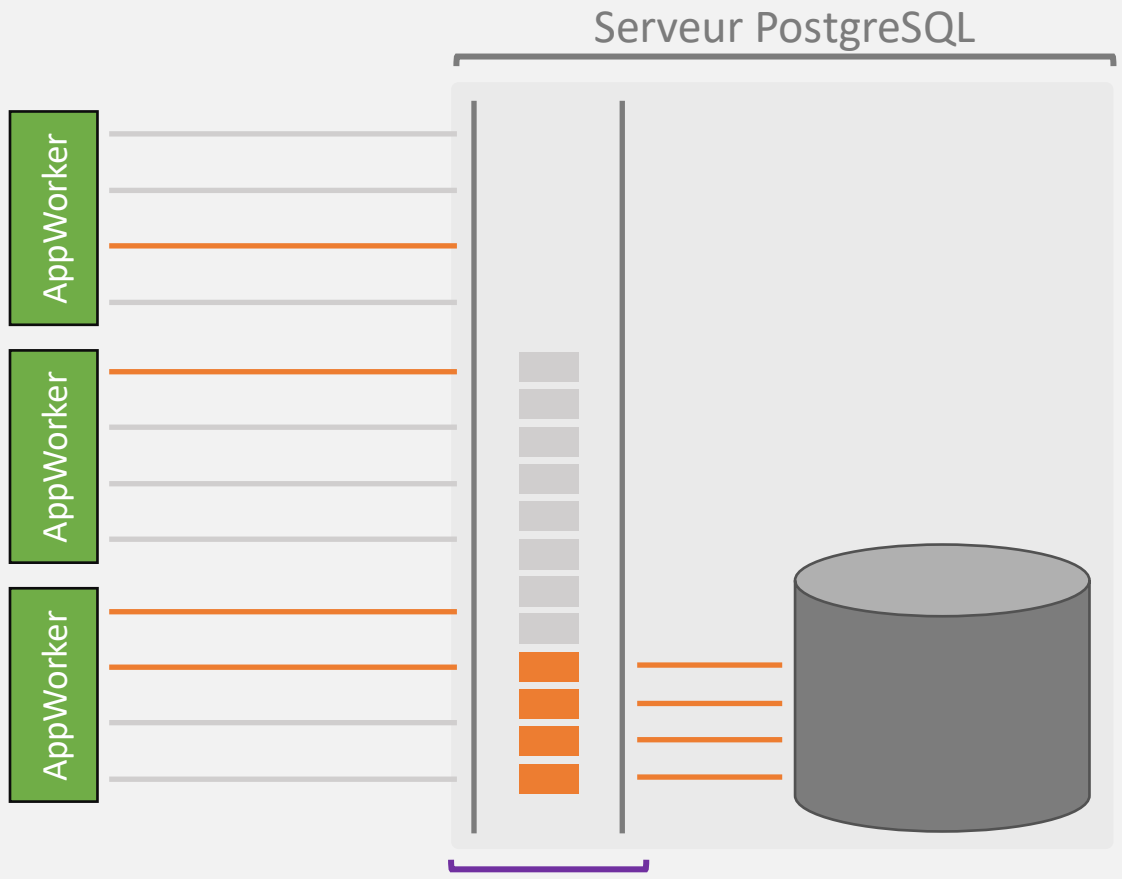
Connexion en attente
Connexion active pg1
Connexion active pg2

DataBene

Poolers de Connexions

Élément du serveur PostgreSQL

Connection Poolers – PostgreSQL Server Side



Le pooler de Connexions est hébergé sur le serveur PostgreSQL.
Il est le point d'accès publique à l'instance PostgreSQL.

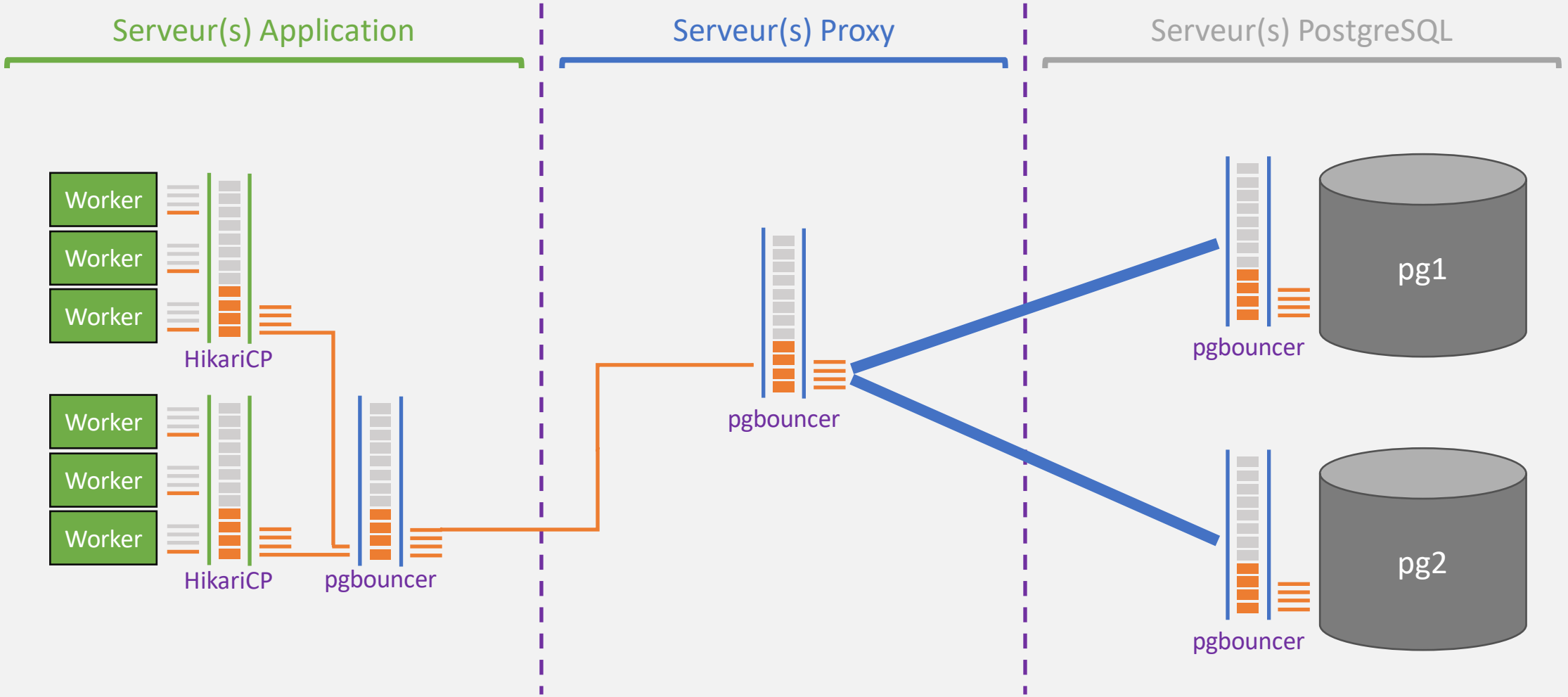
Connexion en attente
Connexion active

DataBene

Poolers de Connexions

Superposition fonctionnelle

Poolers de Connexions – Superposition fonctionnelle



DataBene

Poolers de Connexions

Pooling Modes

Pooling modes (pgbouncer)

- **Session pooling**
  - La connexion est affectée à un client au moment du CONNECT.
  - La connexion est retournée au pool lorsque au moment du DISCONNECT.
  - Toutes les fonctionnalités de PostgreSQL sont supportées.
- **Transaction pooling**
  - La connexion est affectée à un client pour la durée d'une transaction.
  - Lorsque pgbouncer détecte une fin de transaction, il retourne la connexion au pool.
  - Ce mode nécessite la cooperation de l'application qui s'interdit l'utilisation de fonctionnalités de niveau session ou juste incompatible avec ce mode.
- **Statement pooling**
  - La connexion est affectée à un client pour la durée d'execution d'une seule requête.
  - Autocommit obligatoire côté client (le multistatements est interdit).

pgbouncer – Support de Fonctionnalités

Feature	Session pooling	Transaction pooling
Startup parameters [1]	Yes	Yes
SET/RESET	Yes	Never
LISTEN/NOTIFY	Yes	Never
WITHOUT HOLD CURSOR	Yes	Yes
WITH HOLD CURSOR	Yes	Never
Protocol-level prepared plans	Yes	No [2]
PREPARE / DEALLOCATE	Yes	Never
ON COMMIT DROP temp tables	Yes	Yes
PRESERVE/DELETE ROWS temp tables	Yes	Never
Cached plan reset	Yes	Yes
LOAD statement	Yes	Never
Session-level advisory locks	Yes	Never

Startup parameters are: client_encoding, datestyle, timezone, and standard_conforming_strings. PgBouncer detects their changes and so it can guarantee they remain consistent for the client.

DataBene

Partage de Connexions et Invariance

Session-pooling mode

- Lorsqu'un client obtient une connexion, il suppose un état de session identique à celui d'une session qui n'a pas eu d'activité.
- `server_reset_query = DISCARD ALL`

Transaction-pooling mode

- La restriction des fonctionnalités permet un « ménage » plus léger
- Unnamed Prepared Statements
- JDBC::prepareThreshold=0

`DISCARD { ALL | PLANS | SEQUENCES | TEMP }`

`DISCARD ALL` executes the following:

```
CLOSE ALL; -- cursors
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL; -- runtime parameters to default
DEALLOCATE ALL; -- prepared statements
UNLISTEN *; -- listen/notify
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
DISCARD SEQUENCES;
```

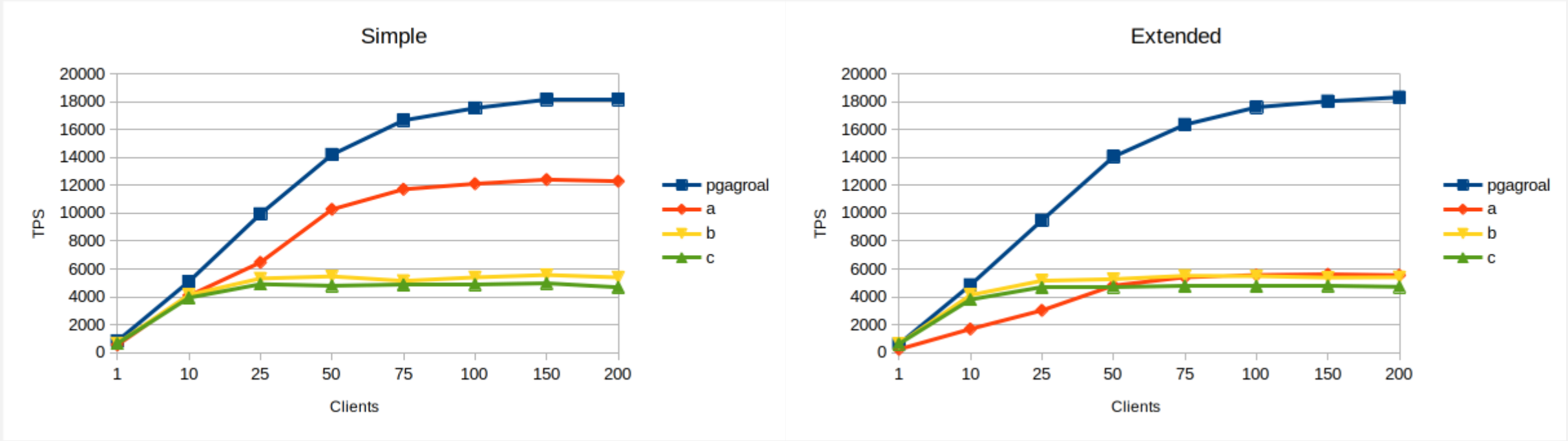
Poolers de Connexions- NextGen

pgbouncer a tracé la voie mais il ne suffit plus aux usages actuels

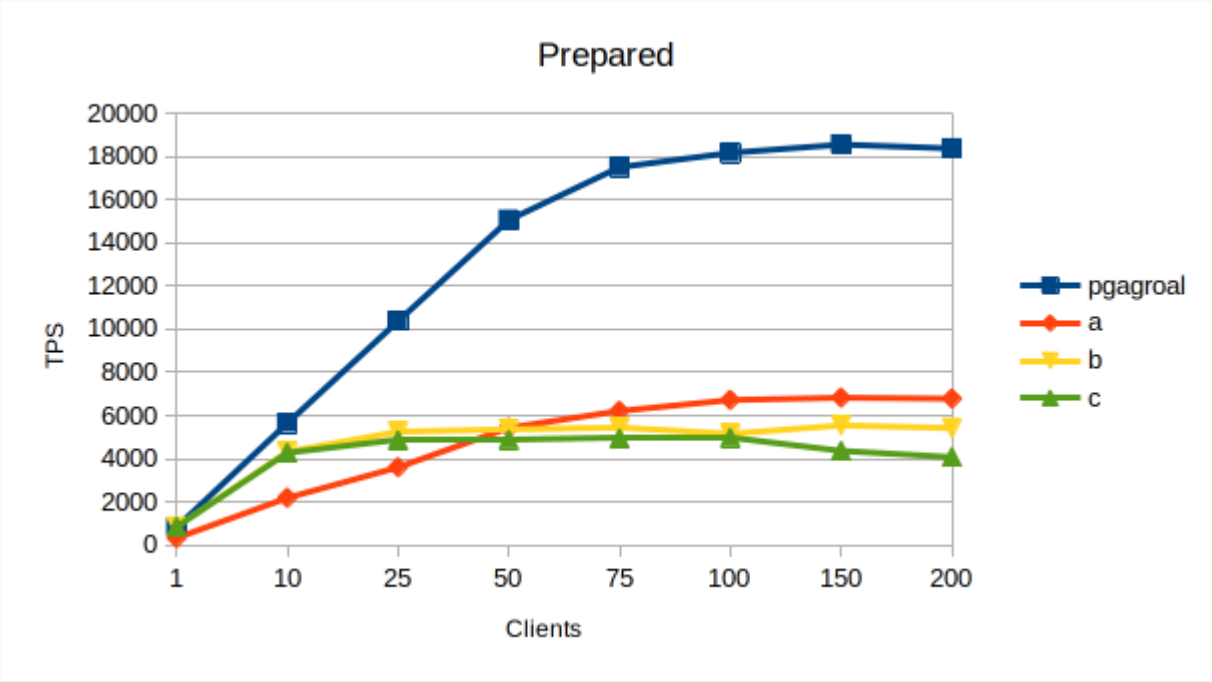
Connection Poolers – NextGen – pgagroal

- https://agroal.github.io/pgagroal/
- High performance (process model + shared memory + atomic ops)
- Enable / disable database access
- Graceful / fast shutdown
- Prometheus support
- Grafana 8 dashboard
- Remote management
- Authentication query support
- Failover support
- Transport Layer Security (TLS) v1.2+ support
- User vault

Connection Poolers – NextGen - pgagroal



Connection Poolers – NextGen - pgagroal



DataBene

Connection Poolers – NextGen - Odyssey

- https://github.com/yandex/odyssey

- Supports TLS
- Supports Multithreading
- Supports **Named Prepared Statements** in **Transaction-pooling** mode

Connection Poolers – NextGen - PgCat

- https://github.com/levkk/pgcat (PostgresML/pgCat)

- Supports Multithreading
- Pools can connect to several servers
- Supports Load Balancing and Server Isolation at the Pool Level
- Allows query routing to shards

- Inspired from pgbouncer and pgpool-II

Connection Poolers – NextGen - PgCat

Features	Comments
Transaction pooling	Identical to PgBouncer.
Session pooling	Identical to PgBouncer.
COPY support	Both COPY TO and COPY FROM are supported.
Query cancellation	Supported both in transaction and session pooling modes.
Load balancing of read queries	Using random between replicas. Primary is included when primary_reads_enabled is enabled (default).
Sharding	Transactions are sharded using SET SHARD TO and SET SHARDING KEY TO syntax extensions
~~Failover~~ (**server isolation**)	Replicas are tested with a health check. If a health check fails, remaining replicas are attempted
Statistics	Statistics available in the admin database (pgcat and pgbouncer) with SHOW STATS, SHOW POOLS and others.
Live configuration reloading	Reload supported settings with a SIGHUP to the process, e.g. kill -s SIGHUP $(pgrep pgcat) or RELOAD query issued to the admin database.
Client authentication	MD5 password authentication is supported, **SCRAM is on the roadmap**; one user is used to connect to Postgres with both SCRAM and MD5 supported.
Admin database	The admin database, similar to PgBouncer's, allows to query for statistics and reload the configuration.

NextGen – Supabase / supavisor

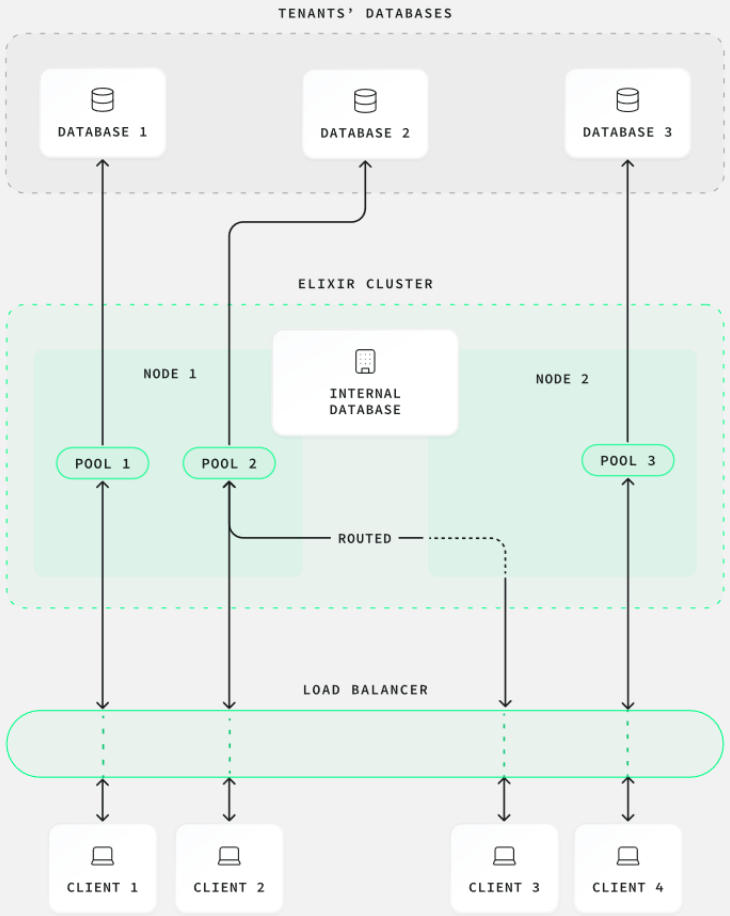
- https://github.com/supabase/supavisor

- Sources d’inspiration
  - pgBouncer
  - stolon
  - pgcat
  - odyssey
  - crunchy-proxy
  - pgpool
  - pgagroal

NextGen – Supabase / supavisor

- Supavisor is a scalable, cloud-native Postgres connection pooler.
- Multinodes connection pooler
- Zero-downtime scaling
- Handling modern connection demands: millions of connections over TCP and HTTP protocols.
- Efficiency: connection pooling to a dedicated cluster adjacent to tenant databases.

NextGen – Supabase / supavisor



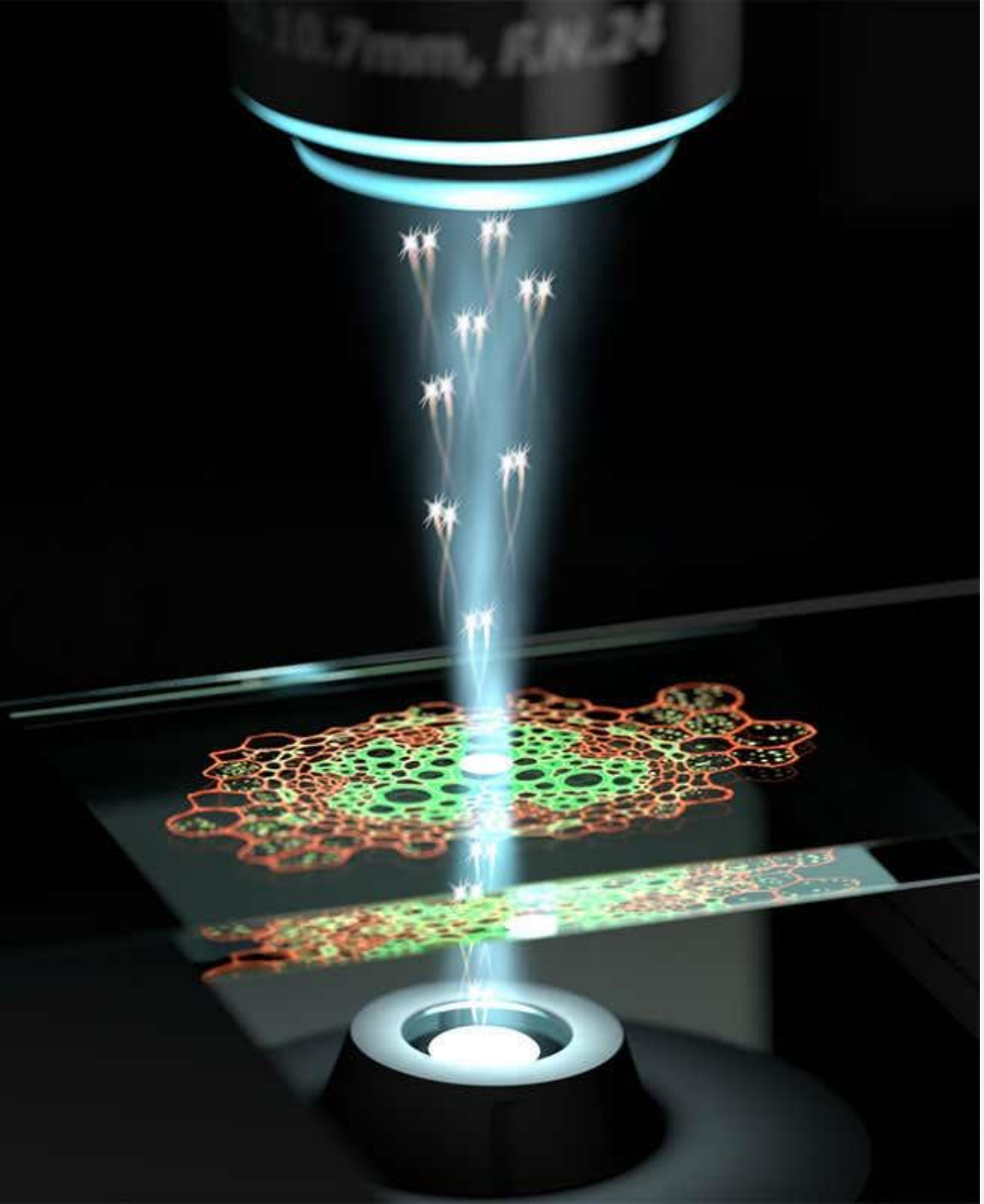
NextGen – Supabase / supavisor

- Fast
- Scalable
  - 1 million Postgres connections on a cluster
  - 250 000 idle connections on a single 16 core node with 64GB of ram
- Multi-tenant
- Pool mode support per tenant (Transaction)
- Cloud-native
- Observable
- Manageable
- Highly available
- Connection buffering for transparent database restarts or failovers

NextGen – Supabase / supavisor

Future work

- Load balancing over read-replica
- Query caching
- Session pooling
- Multi-protocol Postgres query interface
  - Postgres binary, HTTPS, Websocket
- Postgres high-availability management
  - Primary database election on primary failure
  - Health checks
  - Push button read-replica configuration

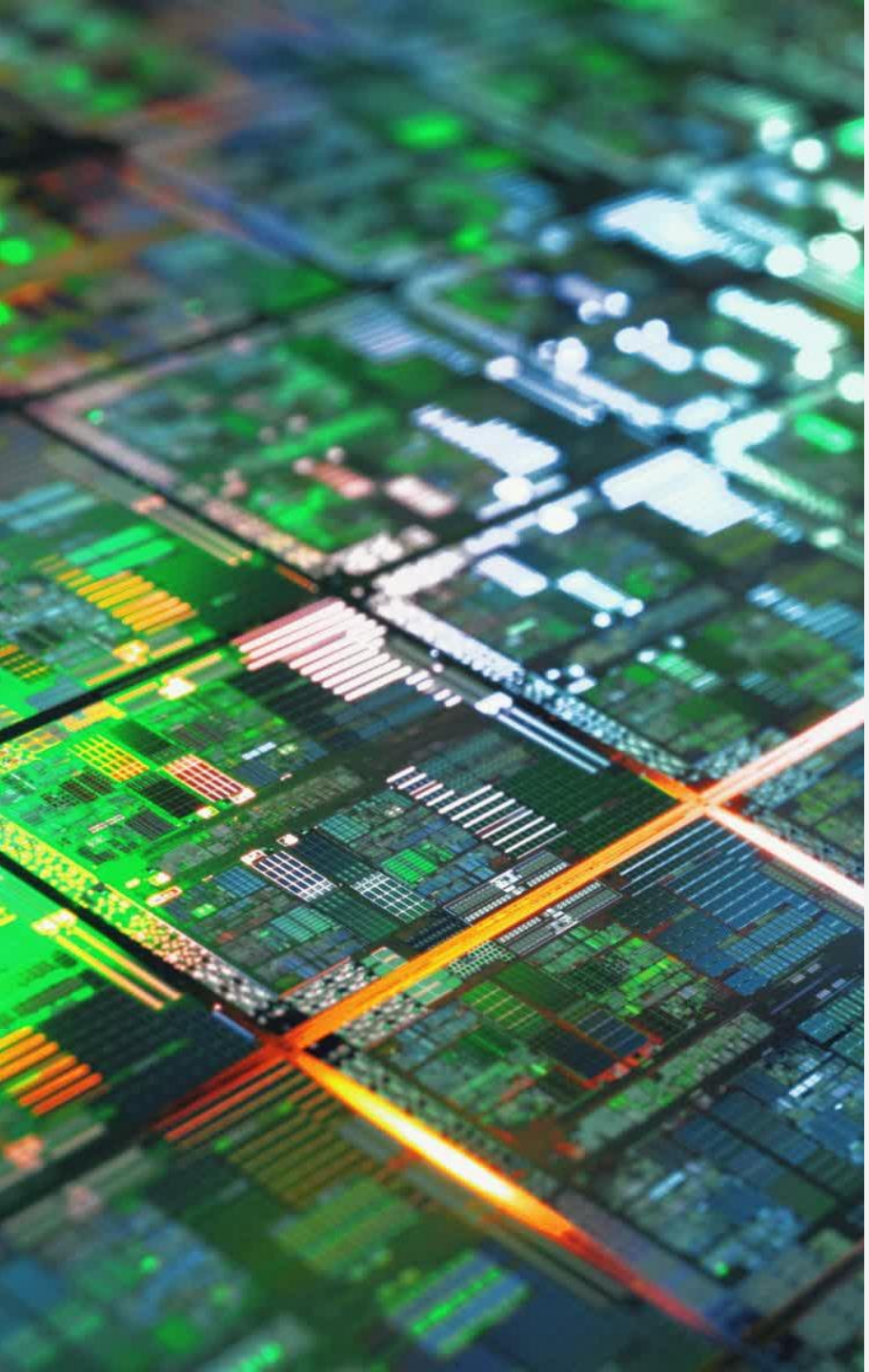


Introduction

Connexions PostgreSQL

Poolers de Connexions

Summary



Synthèse

Les poolers de connexions ont d'abord été conçus pour :

- des raisons de performance
  - durée d'établissement de connexion,
  - limitation du nombre de sessions actives,
  - mise en attente des demandes de connexions.

Ensuite, les fonctionnalités de « routing » et de « fencing » ont vu le jour.

L' étude du protocole de communication de PostgreSQL nous rend capable :

- de comprendre les défis qui s'imposent aux poolers de connexions (proxy),
- mais aussi d'envisager des solutions
- et même d'imaginer de nouvelles fonctionnalités.

Ce cheminement a partiellement été fait par les poolers de connexions NextGen qui proposent des améliorations significatives par rapport à leurs ainés.

Thank you!

Questions?

https://data-bene.io frederic.delacourt@data-bene.io