Horizontal scaling with PL/Proxy

Jan Urbański
jan@newrelic.com

New Relic

PGConf.Russia 2015, Moscow, February 7

Outline

1. Scaling PostgreSQL horizontally
   - Problem definition
   - Getting ready to scale
2. The PL/Proxy language
3. PgBouncer
4. Usage scenarios

PostgreSQL in the VPS world

- maximum capacity of available machines is limited
- however, the number of available machines is limitless
- need to be able to add resources without disrupting current operations
- hosts will fail: not if but when
- typical for VPS scenarios, but enforces good engineering practices even if you manage your own metal

Challenges

- normalisation goes out the window
- idea: independent parts of the application get independent database hosts
  - not friendly for developers, who need to manage the complexity inside the app
  - oftentimes, not effective: a single module's data outgrows the biggest available node
- plan for using multiple machines from the beginning

Outline

1 Scaling PostgreSQL horizontally
- Problem definition
- Getting ready to scale

2 The PL/Proxy language

3 PgBouncer

4 Usage scenarios

Stored procedure API layer

- route application data access through stored procedures

Stored procedure API layer

- route application data access through stored procedures

BAD

```
insert into orders (select * from parts join ...  where
tmpl = $1 and user_id = $2 ...)
```

Stored procedure API layer

- route application data access through stored procedures

BAD

```
insert into orders (select * from parts join ...  where
tmpl = $1 and user_id = $2 ...)
```

WORSE

```
Order.new(Parts.find(:tmpl_id =>
tmpl_id).includes(...).where(:user_id => user_id)).save!
```

Stored procedure API layer

- route application data access through stored procedures

BAD

```
insert into orders (select * from parts join ... where
tmpl = $1 and user_id = $2 ...)
```

WORSE

```
Order.new(Parts.find(:tmpl_id =>
tmpl_id).includes(...).where(:user_id => user_id)).save!
```

BETTER

```
select create_order(tmpl_id, user_id)
```

Stored procedure API layer cont.

- database people regain control over database access
- much bigger freedom to do schema changes
- defines a clean interface between developers and DBAs
- it's not an all or nothing proposition!
  - define a procedural API to the hottest part of the database
  - keep accessing the rest through evil ORMs or whatever else

Outline

1 Scaling PostgreSQL horizontally

2 The PL/Proxy language

- How PL/Proxy works
- Language syntax

3 PgBouncer

4 Usage scenarios

Proxy functions

- a language for writing remote procedure calls
- very simple syntax, just a few constructs
- only handles connection and distribution, the rest is built on top of existing mechanisms
- could mostly be reimplemented in any unsafe procedural language (PL/PerlU, PL/PythonU) or with dblink

Function execution

- user calls a PL/Proxy function
- the system determines the target host
- a persistent connection to that host is opened
- code is run on the remote side
- result is sent back to the original PL/Proxy function caller

Simple proxy function example

Execute function on remote host

```
create function create_order(tmpl_id int, account_id int)
   returns orders
   language plproxy
as $func$
connect 'host=10.0.10.1 dbname=orders';
$func$;
```

Determining code to run

- by default, an identically named procedure is called on the remote side
- arguments are passed to the remote procedure
- the result type is validated against the proxy function's result type
- this makes it completely transparent to the caller
- you can seamlessly (and gradually) substitute your regular stored procedures with PL/Proxy functions

Outline

1 Scaling PostgreSQL horizontally

2 The PL/Proxy language

- How PL/Proxy works
- Language syntax

3 PgBouncer

4 Usage scenarios

CONNECT

- `connect` specifies a libpq connection string
- several ways of specifying the string
  - a literal string
  - one of the arguments of the procedure
  - a function invocation
- useful for static partitioning or local testing

Simple proxy function example

Execute function on remote host

```
create function create_order(tmpl_id int, account_id int)
   returns orders
   language plproxy
as $func$
connect 'host=10.0.10.1 dbname=orders';
$func$;
```

CLUSTER and RUN ON

- hardcoding connection strings won't work if you have your data partitioned
- for partitioned setups, `cluster` and `run on` are the solution
- `cluster` allows specifying the set of hosts where the function might run
- `run on` takes a partitioning key, calculates the partition number and runs the function

RUN ON cont

- `run on any` and `run on all` exist as well
  - with `run an all` the query is run in parallel on all partitions
  - results are combined and returned to the caller
- the partitioning key can also be specified using a function invocation
- built-in function `hashtext` creating stable hashes of text values

CLUSTER and RUN ON example

Partitioning

```
create function create_order(tmpl_id int, account_id int)
   returns orders
   language plproxy
as $func$
cluster ’appdata’;
run on account_id;
$func$;
```

Partitioning internals

- a cluster is a list of connection strings
- PL/Proxy requires the number of partitions to be a power of 2
  - annoying, but not that much
  - you can use the same connection strings for several partitions
  - changing the number of partitions is a pain, plan ahead and start with 32 partitions
- the partitioning key needs to be an integer (`int4` or `int8`)
- the target partition is determined with a simple `mod`

New Relic.

Defining clusters

- a legacy procedure-based approach
    - procedures in other languages to return partition lists and config
    - need to manage several of them, additional warts regarding caching
    - much easier to use the foreign server interface

Defining clusters cont

- PL/Proxy now provides a foreign data wrapper
- use `create server` to define clusters
- use a number of options called `p0`, `p1`, `p2`, ... with values being connection strings
- user mappings can supply additional libpq parameters

Foreign data wrapper configuration

Defining a cluster

```
create server appdata foreign data wrapper plproxy options (
    p0 'dbname=appdata1 host=10.0.10.1',
    p1 'dbname=appdata2 host=10.0.10.2'
);

create user mapping for webserver server appdata options (
    password 'tiger'
);
```

SPLIT

- `split` is a way to write queries that need to access more than one partition
- the PL/Proxy procedure should receive equal-length arrays of arguments
- an array of the same length should be passed to `run on`
- for each `run on` element, the specified partition gets a call with an array of corresponding arguments
- once all queries are complete, result are stitched together and returned

SPLIT example

Accessing multiple partitions

```
create function latest_orders(tmpl_ids int[],
                              account_ids int[])
   returns setof orders
   language plproxy
as $func$
cluster 'appdata';
split all; -- shorthand for "split tmpl_ids, account_ids;"
run on account_ids;
$func$;
```

Limitations

- no transactional guarantees!
- changing the partitioning key is a huge hassle
  - but then again, in which partitioning technology it isn't?
- eventually, a connection will be open from every backend to every partition
- to avoid keeping lots of backends running, use PgBouncer

Outline

1 Scaling PostgreSQL horizontally

2 The PL/Proxy language

3 PgBouncer

- Using PgBouncer with PL/Proxy

4 Usage scenarios

What is PgBouncer?

- a connection pooler for PostgreSQL, implementing the Postgres protocol
- sibling project to PL/Proxy
  - in fact, they used to be bundled together, now they're both standalone projects
- very useful even if you're not using PL/Proxy
  - helps with web apps that don't support persistent connections
  - has a bunch tricks that make operating a Postgres cluster simpler

How does PgBouncer work?

- configure a list of database that the pooler will handle
- PgBouncer listens for Postgres protocol connections and parses the startup packet
- it then proxies queries to the appropriate database, possibly reusing previously opened connections
- no forking, no backend startup overhead, can handle hundreds of connections per second

PgBouncer operating modes

- reusing connections breaks some features
  - transactions
  - session parameter changes, prepared plans
  - the list goes on...
- the pooler can use one of several modes
  - session mode, connections reused only if client disconnects
  - transaction mode, connections reused when client commits
  - statement mode, like transaction mode, but transactions are disabled
- statement mode is meant to be used with PL/Proxy

PgBouncer tricks

- set timeout on idle in transaction connections
- runtime config changes
- pausing access to a given database
  - starts queueing new queries to the database
  - waits while all active queries are finished
  - disconnects from the database
  - allows restarting the database without clients noticing
- online restart
  - start a new pooler process, transfer active TCP connections
  - allows restarting the pooler without clients noticing

Outline

1 Scaling PostgreSQL horizontally

2 The PL/Proxy language

3 PgBouncer

4 Usage scenarios

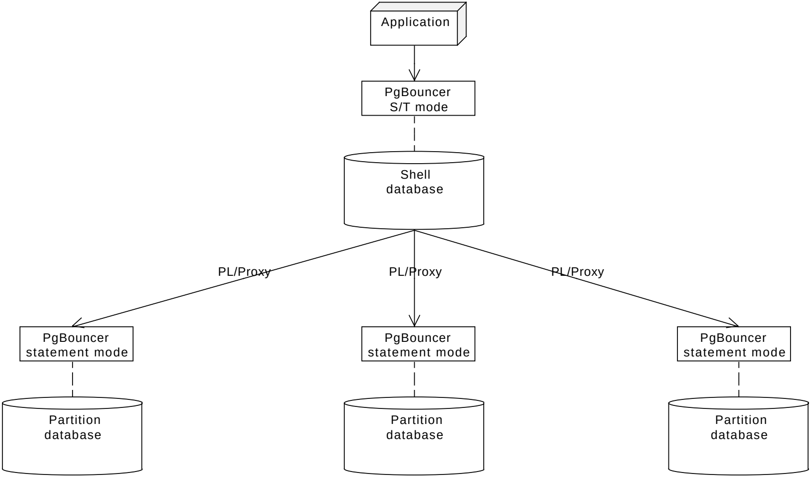
- AKA stories from the trenches

New Relic.

Setting up the cluster

- use a dedicated database as a “shell” with all the PL/Proxy functions
- run PgBouncer in statement mode on each partition host
- run PgBouncer on the shell host, too
    - if shell is 100% PL/Proxy, it can use statement mode
    - typically, the shell contains app data that didn’t need to be partitioned
    - in that case, use session or transaction mode
- partitions only get connections from the shell Postgres
- the shell only gets connections from PgBouncer
- be ruthless with `iptables` and `pg_hba.conf`

Setting up the cluster - diagram



Online reconfiguration

- changing cluster configuration is just an `alter server`
- changes applied immediately and atomically
    - it's even transactional!
- server settings can include things like TCP keepalives
- PL/Proxy triggers run as the table owner, be sure to add a user mapping for them

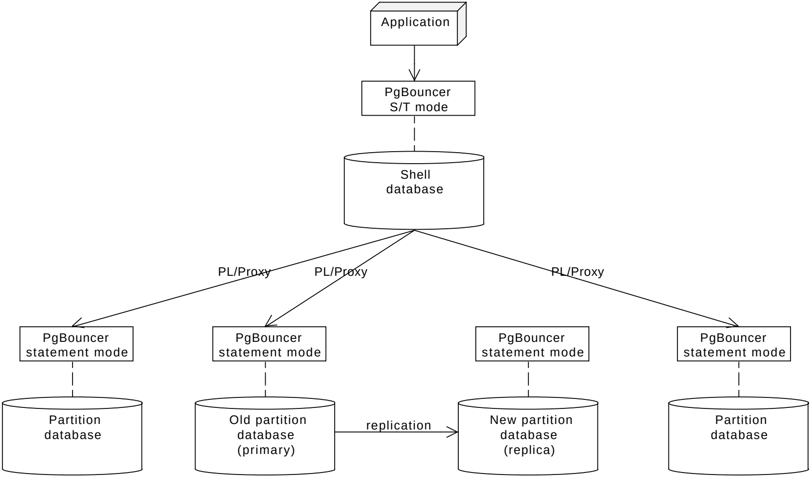
Upgrading hardware on partition host

Zero downtime database hardware upgrade:

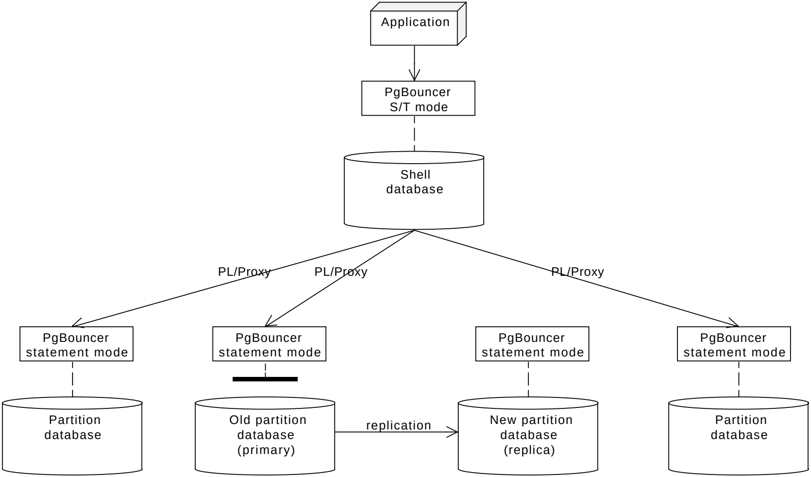
1. set up streaming replication to the new host
2. pause access to the old host via PgBouncer
3. promote the replica
4. change PgBouncer config on old host to point to new host
5. unpause PgBouncer on old host
6. alter PL/Proxy settings on shell to point to new host
7. once old host has no connections, decommission it

New Relic.

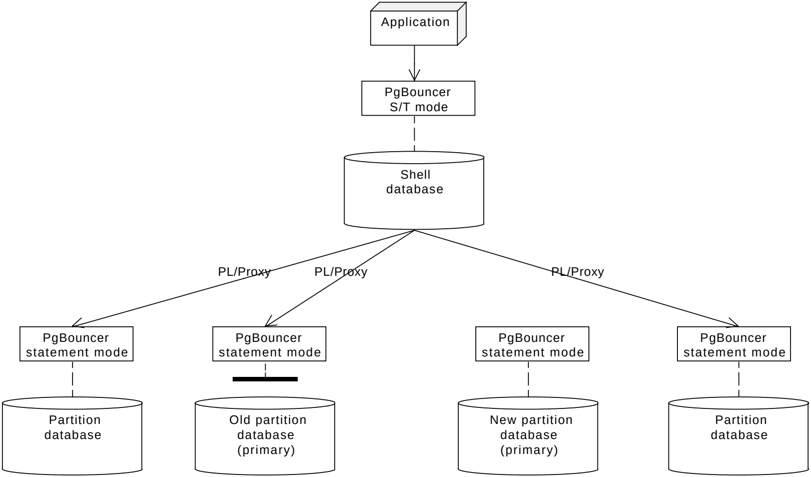
Upgrading hardware - diagram



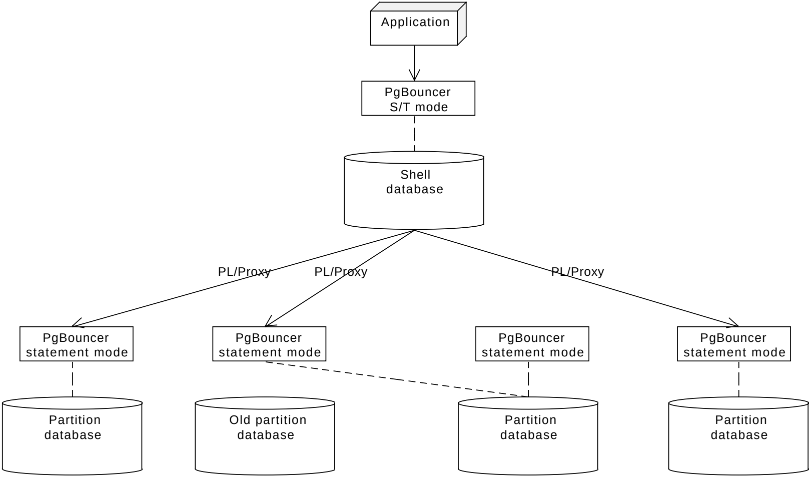
Upgrading hardware - diagram



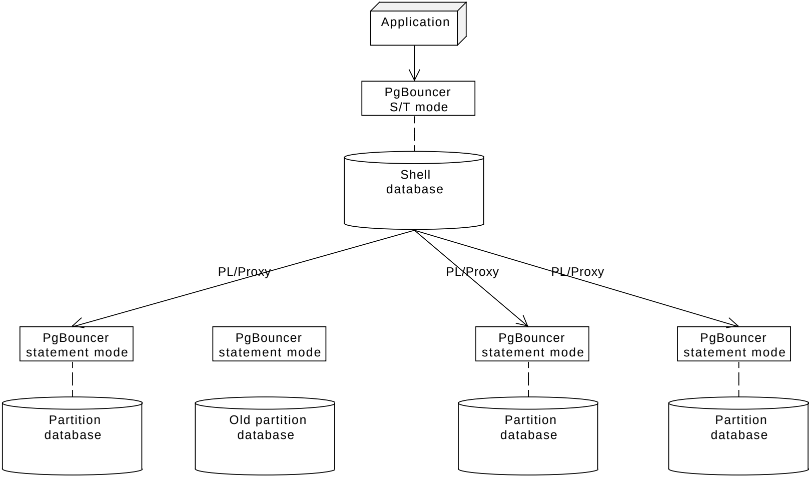
Upgrading hardware - diagram



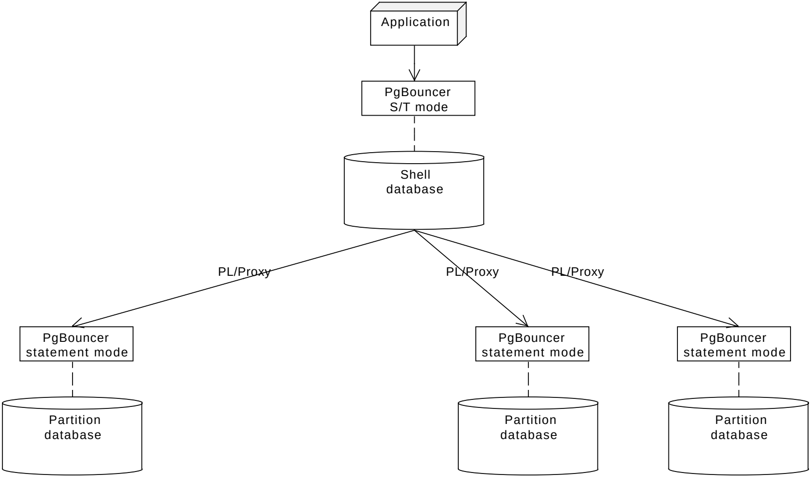
Upgrading hardware - diagram



Upgrading hardware - diagram



Upgrading hardware - diagram

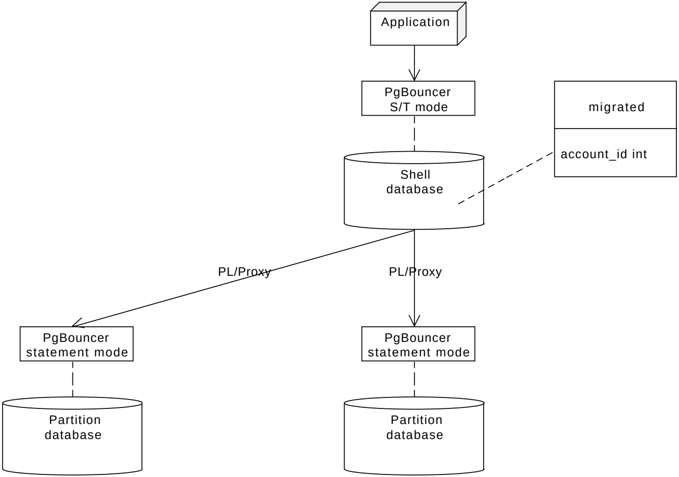


Adding a new partition

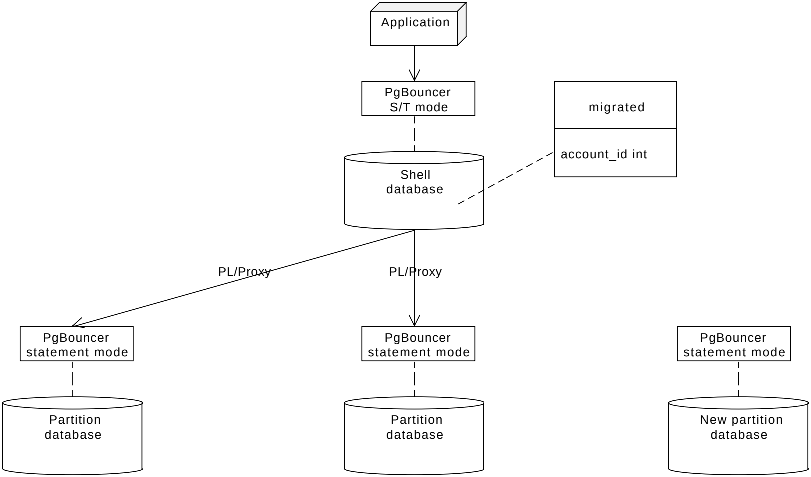
Splitting data from partition A to B:

1. create a `migrated` table to list already migrated IDs
2. write custom partitioning function
    1. calculate target partition
    2. return it if it's not A
    3. looks it up in `migrated`, return B if found
    4. return A
3. alter PL/Proxy functions to use the new function
4. kick off migration process, update `migrated` as you go
5. once all data is migrated, alter the foreign server config and restore original PL/Proxy partitioning function definition

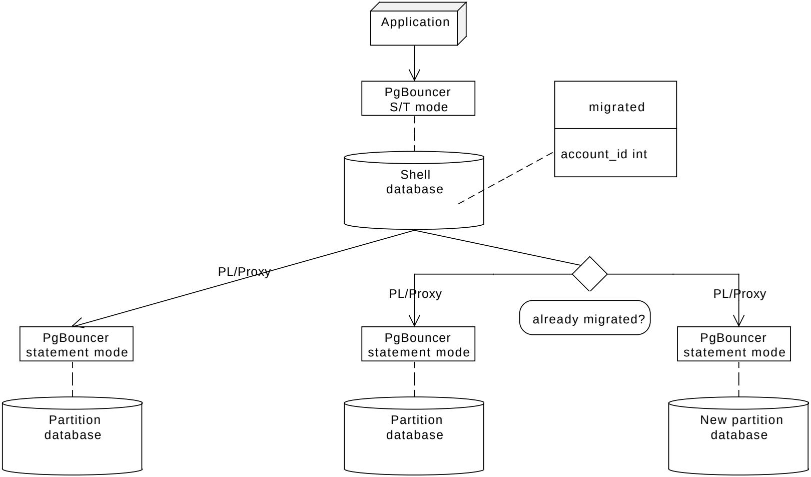
Adding a partition - diagram



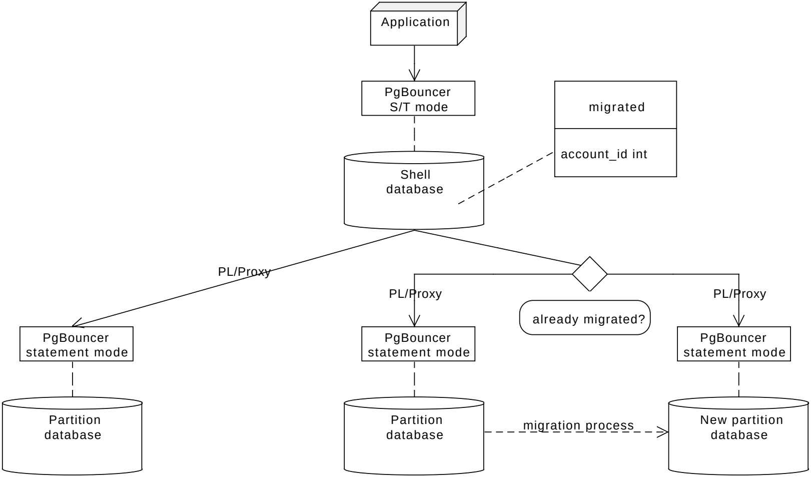
Adding a partition - diagram



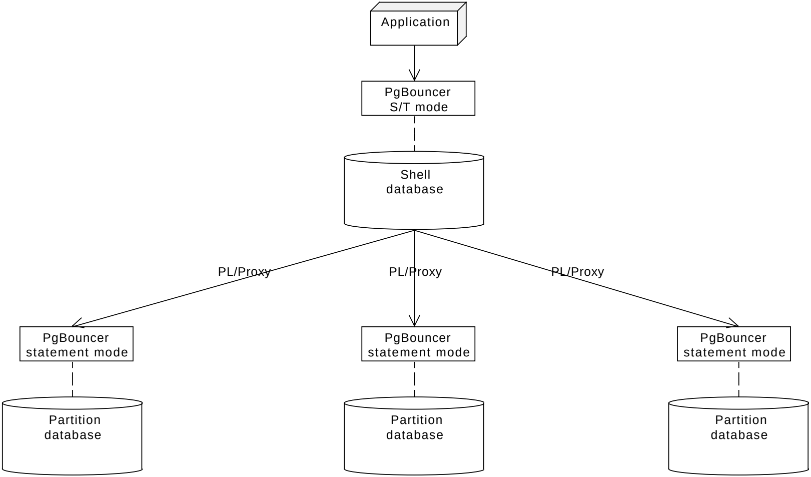
Adding a partition - diagram



Adding a partition - diagram



Adding a partition - diagram



Questions?