Database Tools by Skype

Asko Oja

Stored Procedure API

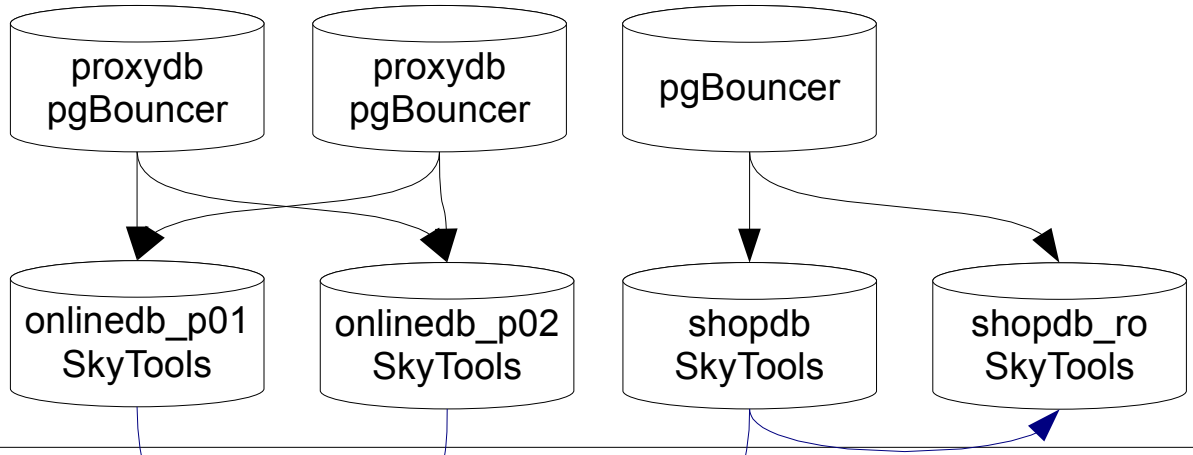
- Skype has had from the beginning the requirement that all database access must be implemented through stored procedures.
- Using function-based database access has more general good points:
  - It's good to have SQL statements that operate on data near to tables. That makes life of DBA's easier.
  - It makes it possible to optimize and reorganize tables transparently to the application.
  - Enables DBA's to move parts of database into another database without changing application interface.
  - Easier to manage security if you don't have to do it on table level. In most cases you need to control what user can do and on which data not on what tables.
  - All transactions can be made in autocommit mode. That means absolutely minimal amount of roundtrips (1) for each query and also each transaction takes shortest possible amount of time on server - remember that various locks that transactions acquire are release on COMMIT.

Keeping Online Databases Slim and Fit

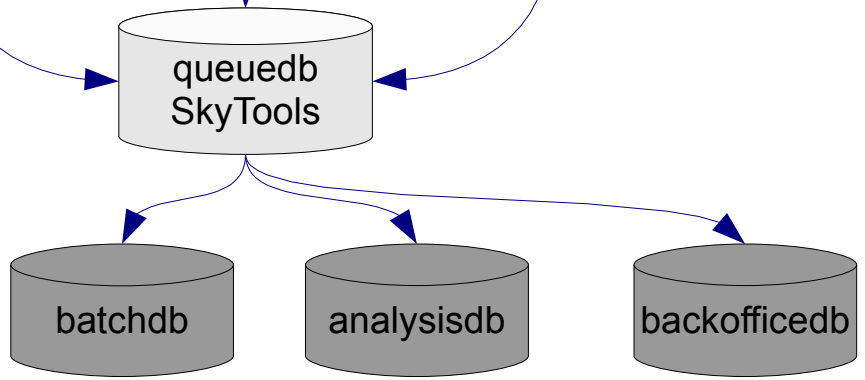
- Rule 1: Minimize number of connections. Channel all incoming queries through optimal number of database connections using pgBouncer
- Rule 2: Minimize number of indexes constraints that use up performance and resources. For example backoffice applications need quite often more and different indexes than are needed for online queries.
- Rule 3: Keep as little data as possible in online database. Move all data that is not need into second or third tier databases. Use remote calls to access it if needed.
- Rule 4: Keep transactions short. Using functions is the best way to do this as you can use functions in autocommit mode and still do complex SQL operations.
- Rule 5: Keep external dependancies to the minimum then you have less things that can get broken and affect online database. For examples run batch jobs against second and third tier databases.

Overall picture

Online databases
- Proxy db's
- pgBouncers
- OLTP db's
- read only db's

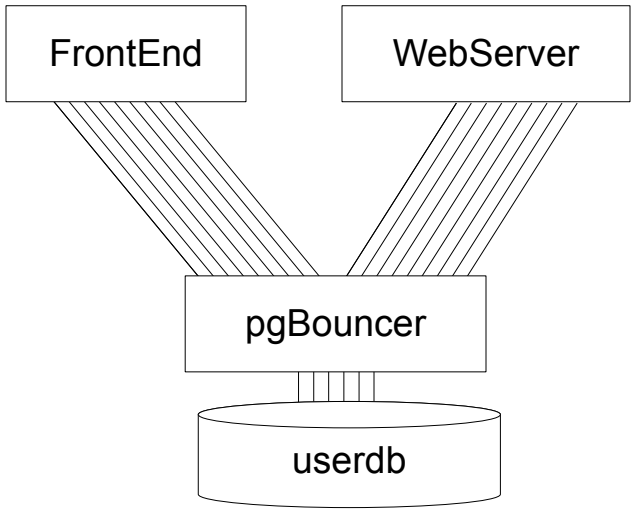


Support databases
- Queue db's
- Datamining
- Batchjobs
- Backoffice
- Greenplum



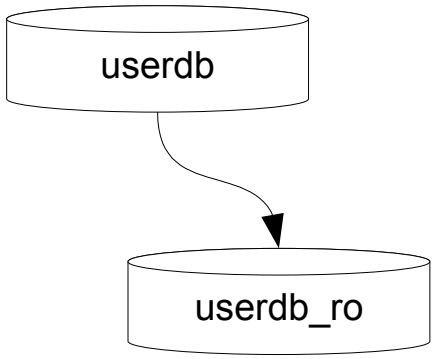
pgBouncer – PostgreSQL Connection pooler

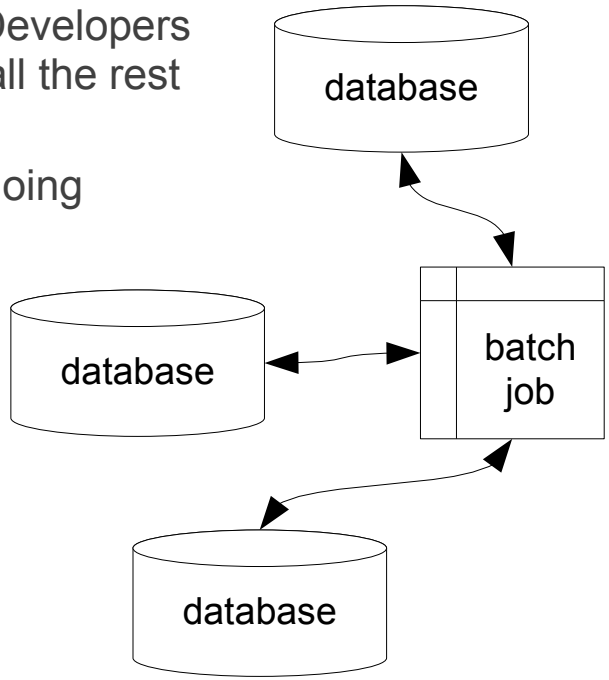
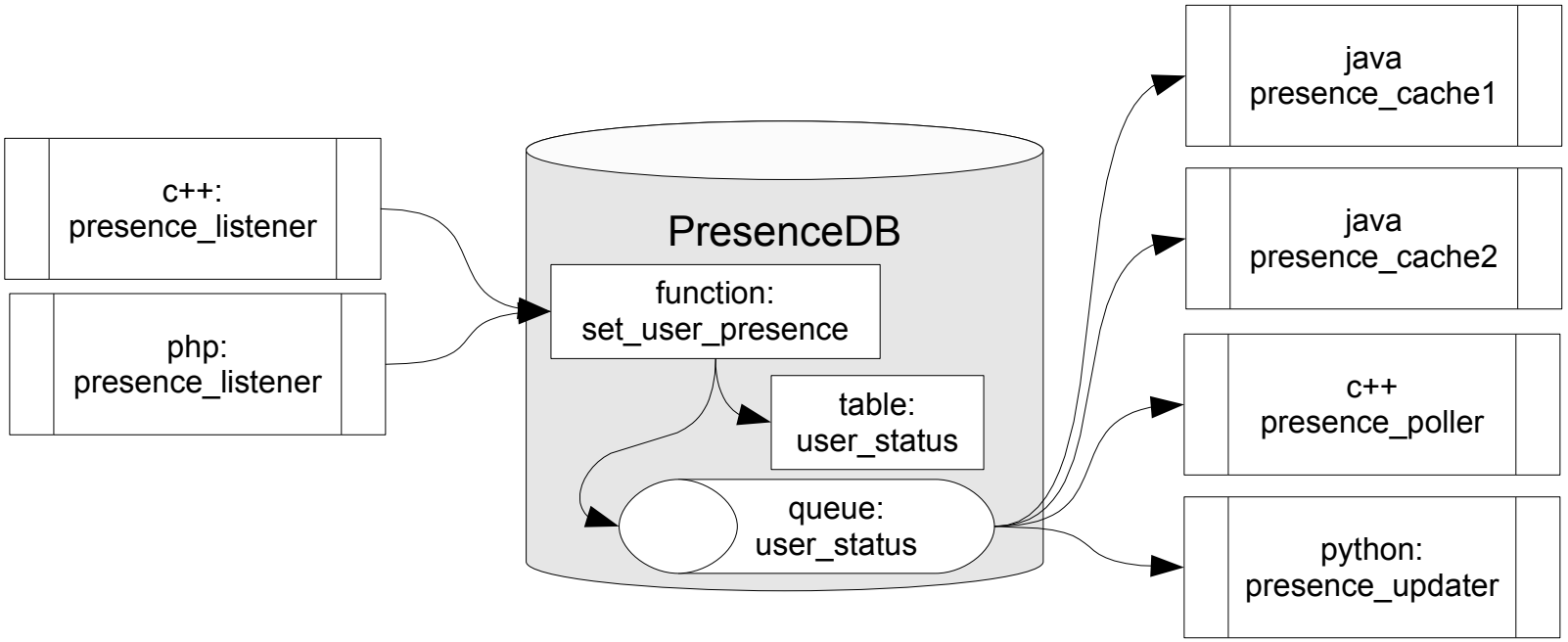
- pgBouncer is lightweight and robust connection pooler for PostgreSQL.
- Reduces thousands of incoming connections to only tens of connections in database.
- Low number of connections is important because each connection uses computer resources and each new connection is quite expensive as prepared plans have to be created each time from scratch.
- We are not using pgBouncer for load balancing.
- Can be used to redirect database calls (database aliases).



plProxy – Remote Call Language

- PL/Proxy is compact language for remote calls between PostgreSQL databases.
- With PL/Proxy user can create proxy functions that have same signature as remote functions to be called. The function body describes how the remote connection should be acquired.




- ```
  CREATE FUNCTION get_user_email(username text) RETURNS text AS $$
      CONNECT 'dbname=userdb_ro';
  $$ LANGUAGE plproxy;
  ```
  ```

SkyTools: Batch Job Framework

- Written in python and contain most everything we have found useful in our everyday work with databases and PostgreSQL.
- Framework provides database connectivity, logging, stats management, encoding, decoding etc for batch jobs. Developers need only to take care of business logic in batch jobs all the rest is handled by batch jobs.
- SkyTools contains 10s of reusable generic scripts for doing various data related tasks.
- PgQ that adds event queues to PostgreSQL.
- Londiste replication.
- Walmgr for wal based log shipping.



PgQ: PostgreSQL Queues

- **PgQ** is PostgreSQL based event processing system. It is part of SkyTools package that contains several useful implementations on this engine.
- **Event** - atomic piece of data created by Producers. In PgQ event is one record in one of tables that services that queue. PgQ guarantees that each event is seen at least once but it is up to consumer to make sure that event is processed no more than once if that is needed.
- **Batch** - PgQ is designed for efficiency and high throughput so events are grouped into batches for bulk processing.
- **Queue** - Event are stored in queue tables i.e queues. Several producers can write into same queue and several consumers can read from the queue. Events are kept in queue until all the consumers have seen them. Queue can contain any number of event types it is up to Producer and Consumer to agree on what types of events are passed and how they are encoded
- **Producer** - applications that pushes event into queue. Producer can be written in any language that is able to run stored procedures in PostgreSQL.
- **Consumer** - application that reads events from queue. Consumers can be written in any language that can interact with PostgreSQL.

PgQ: PostgreSQL Queues Illustration

- Database that keeps track of user status (online, offline, busy, etc.
- Producer can be anybody who can call stored procedure
- The same goes about consumers



PgQ: Features

- **Transactional.** Event insertion is committed or rolled back together with the other things that transaction changes.
- **Efficient**. Usually database based queue implementations are not efficient but we managed to solve it because PostgreSQL makes transaction state visible to users.
- **Fast**. Events are processed in batches which gives low per event overhead.
- **Flexible**. Each queue can have several producers and consumers and any number of event types handled in it.
- **Robust**. PgQ guarantees that each consumers sees event at least once. There several methods how the processed events can be tracked depending on business needs.
- Ideally suited for all kinds of batch processing.

plProxy

plProxy: Installation

- Download PL/Proxy from http://pgfoundry.org/projects/plproxy and extract.
- Build PL/Proxy by running make and make install inside of the plproxy directory. If your having problems make sure that pg_config from the postgresql bin directory is in your path.
- To install PL/Proxy in a database execute the commands in the plproxy.sql file. For example psql -f $SHAREDIR/contrib/plproxy.sql mydatabase
- Steps 1 and 2 can be skipped if your installed pl/proxy from a packaging system such as RPM.
- Create a test function to validate that plProxy is working as expected.
- ```
  CREATE FUNCTION get_user_email(username text)
  RETURNS text AS $$
     CONNECT 'dbname=userdb';
  $$ LANGUAGE plproxy;
  ```
  ```

plProxy Language

- The language is similar to plpgsql - string quoting, comments, semicolon at the statements end.It contains only 4 statements: CONNECT, CLUSTER, RUN and SELECT.
- Each function needs to have either CONNECT or pair of CLUSTER + RUN statements to specify where to run the function.
- **CONNECT 'libpq connstr';** -- Specifies exact location where to connect and execute the query. If several functions have same connstr, they will use same connection.
- **CLUSTER 'cluster_name';** -- Specifies exact cluster name to be run on. The cluster name will be passed to plproxy.get_cluster_* functions.
- **CLUSTER cluster_func(..);** -- Cluster name can be dynamically decided upon proxy function arguments. cluster_func should return text value of final cluster name.

plProxy Language RUN ON ...

- **RUN ON ALL;** -- Query will be run on all partitions in cluster in parallel.
- **RUN ON ANY;** -- Query will be run on random partition.
- **RUN ON <NR>;** -- Run on partition number <NR>.
- **RUN ON partition_func(..);** -- Run partition_func() which should return one or more hash values. (int4) Query will be run on tagged partitions. If more than one partition was tagged, query will be sent in parallel to them.

```
CREATE FUNCTION get_user_email(text)
RETURNS text AS $$
   CLUSTER 'userdb';
   RUN ON get_hash($1);
$$ LANGUAGE plproxy;
```

Partition selection is done by taking lower bits from hash value.

hash & (n-1), where n is power of 2.

plProxy Configuration

- Schema **`plproxy`** and three functions are needed for plProxy
- **`plproxy.get_cluster_partitions(cluster_name text)`** – initializes plProxy connect strings to remote databases
- **`plproxy.get_cluster_version(cluster_name text)`** – used by plProxy to determine if configuration has changed and should be read again. Should be as fast as possible because it is called for every function call that goes through plProxy.
- **`plproxy.get_cluster_config(in cluster_name text, out key text, out val text)`** – can be used to change plProxy parameters like connection lifetime.

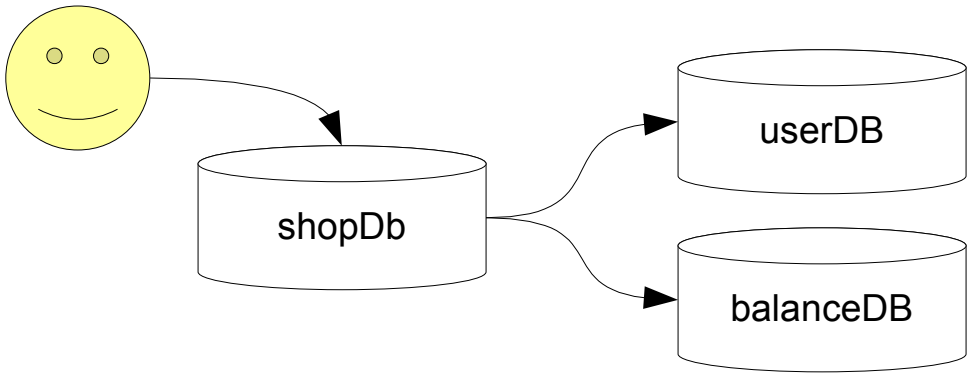
```
CREATE FUNCTION plproxy.get_cluster_version(i_cluster text)
RETURNS integer AS $$
    SELECT 1;
$$ LANGUAGE sql;

CREATE FUNCTION plproxy.get_cluster_config(cluster_name text,
                    OUT key text, OUT val text)
RETURNS SETOF record AS $$
    SELECT 'connection_lifetime'::text as key, text( 30*60 ) as val;
$$ LANGUAGE sql;
```

plProxy: Configuration Functions

```
CREATE FUNCTION plproxy.get_cluster_partitions(cluster_name text)
RETURNS SETOF text AS $$
begin
    if cluster_name = 'userdb' then
        return next 'port=9000 dbname=userdb_p00 user=proxy';
        return next 'port=9000 dbname=userdb_p01 user=proxy';
        return;
    end if;
    raise exception 'no such cluster: %', cluster_name;
end; $$ LANGUAGE plpgsql SECURITY DEFINER;

CREATE FUNCTION plproxy.get_cluster_partitions(i_cluster_name text)
RETURNS SETOF text AS $$
declare    r record;
begin
    for r in
        select connect_string from plproxy.conf
        where cluster_name = i_cluster_name
        order by part_nr
    loop
        return next r.connect_string;
    end loop;
    if not found then
        raise exception 'no such cluster: %', i_cluster_name;
    end if;
    return;
end; $$ LANGUAGE plpgsql SECURITY DEFINER;
```

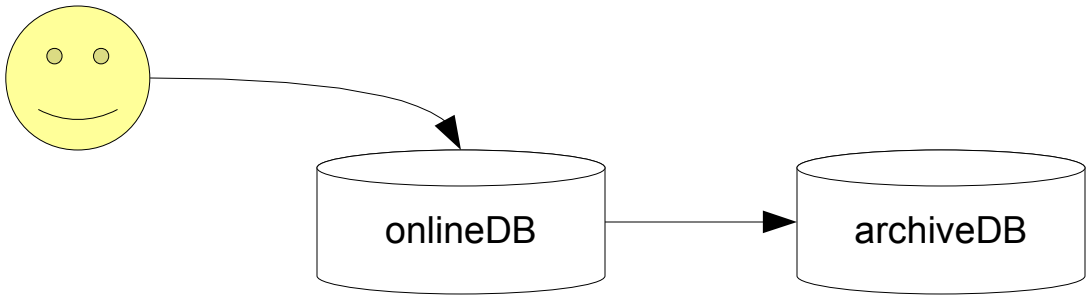
plProxy: Read Only Remote Calls

- We use remote calls mostly for read only queries in cases where it is not reasonable to replicate data needed to calling database.
- For example balance data is changing very often but whenever doing decisions based on balance we must use the latest balance so we use remote call to get user balance.
- plProxy remote calls are not suitable for data modification because there is no guarantee that both transactions will be committed or rolled back (No 2phase commit). Instead of using remote calls we try to use PgQ queues and various consumers on them that provide our 2phase commit.



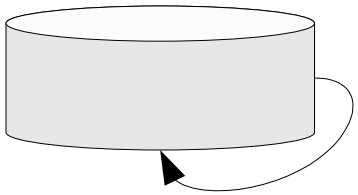
plProxy: Remote Calls to Archive

- Closed records from large online tables (invoices, payments, call detail records, etc) are moved to archive database on monthly basis after month is closed and data is locked against changes.
- Normally users work only on online table but in some cases they need to see data from all the history.
- plProxy can used to also query data from archive database when user requests data for longer period than online database holds.



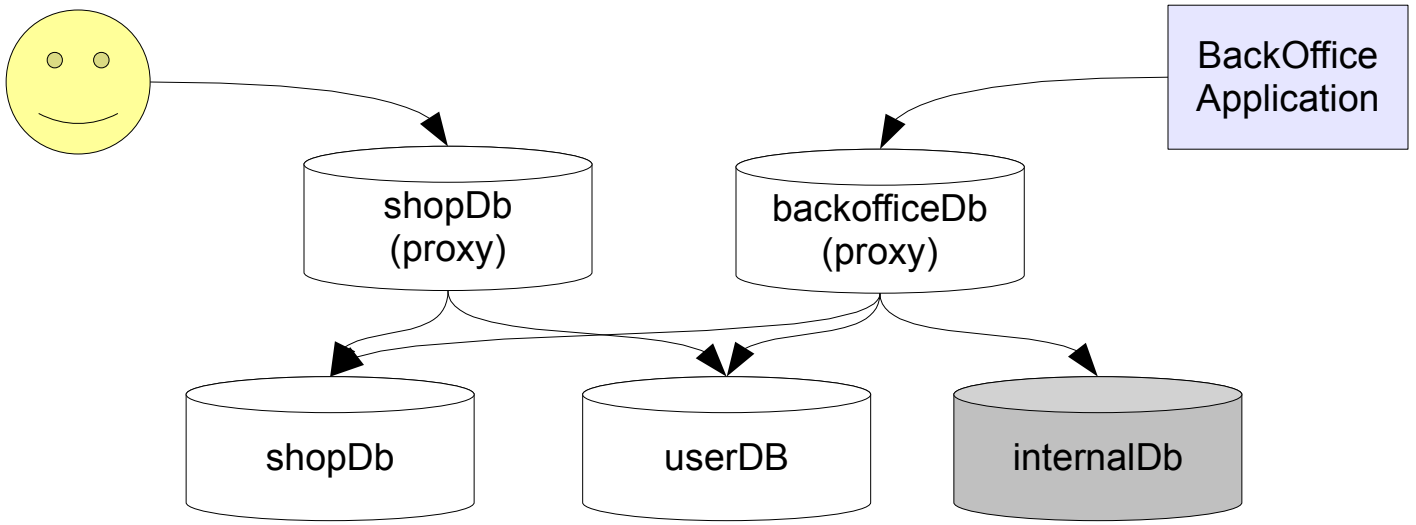
plProxy: Remote Calls for Autonomous Transactions

- PostgreSQL does not have autonomous transactions but plProxy calls can be used for mimicking them.
- Autonomous transactions can be useful for logging failed procedure calls.
- If used extensively then usage of pgBouncer should be considered to reduce number of additional connections needed and reducing connection creation overhead.



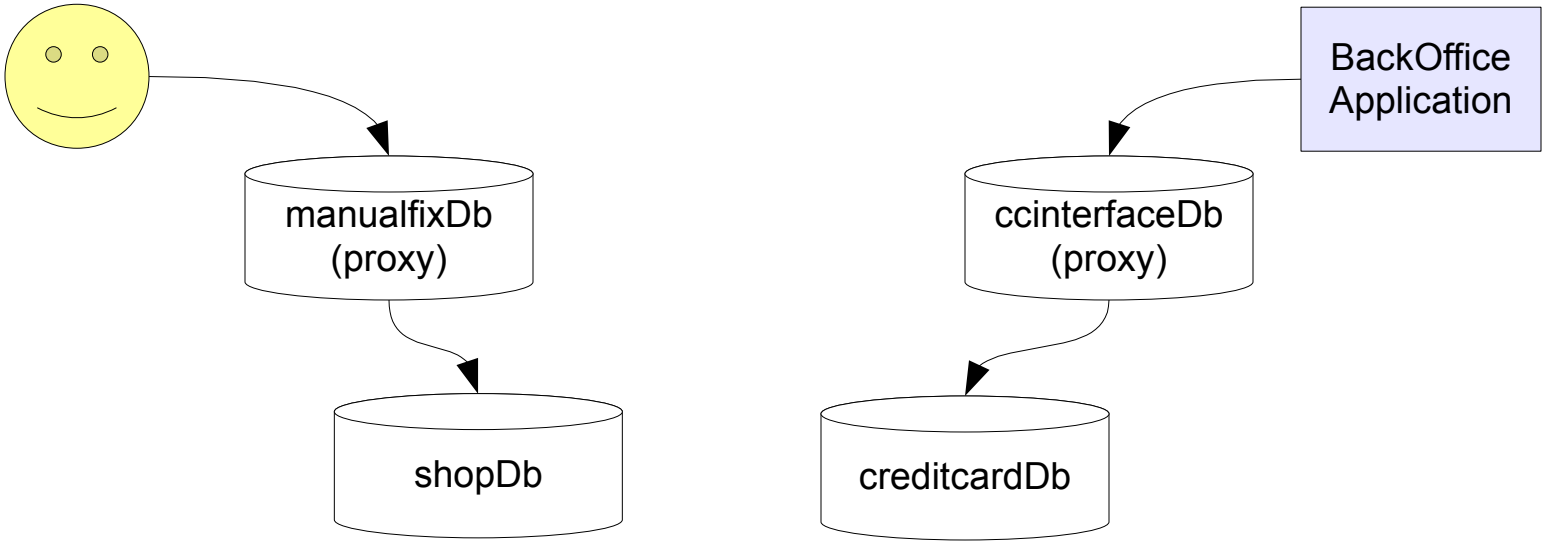
plProxy: Proxy Databases as Interface

- Additional layer between application and databases.
- Keep applications database connectivity simpler giving DBA's and developer's more flexibility for moving data and functionality around.
- It gets really useful when number of databases gets bigger then connectivity management outside database layer gets too complicated :)



plProxy: Proxy Databases for Security

- Security layer. By giving access to proxy database DBA's can be sure that user has no way of accessing tables by accident or by any other means as only functions published in proxy database are visible to user.
- Such proxy databases may be located in servers visible from outside network while databases containing data are usually not.

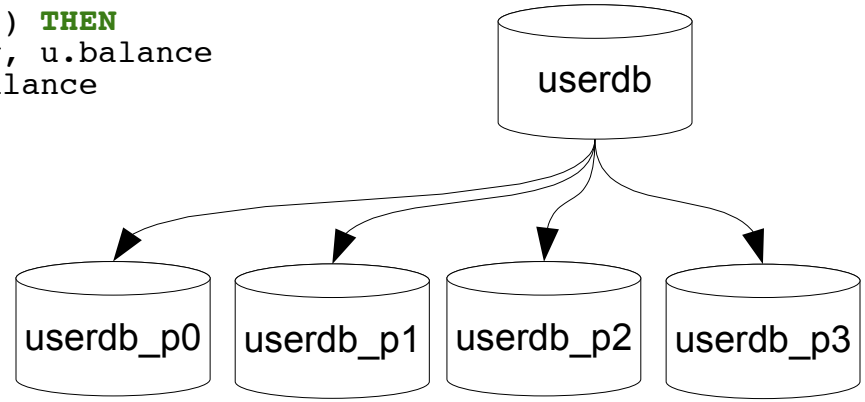


plProxy: Run On All for Data

- Also usable when exact partition where data resides is not known. Then function may be run on all partitions and only the one that has data does something.

```
CREATE FUNCTION balance.get_balances(
    i_users text[],
    OUT username text,
    OUT currency text,
    OUT balance numeric)
RETURNS SETOF record AS $$
BEGIN
    FOR i IN COALESCE(array_lower(i_users,1),0) ..
             COALESCE(array_upper(i_users,1),-1)
    LOOP
        IF partconf.valid_hash(i_users[i]) THEN
            SELECT i_users[i], u.currency, u.balance
              INTO username, currency, balance
              FROM balances
             WHERE username = i_users[i];
            RETURN NEXT;
        END IF;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

```
CREATE FUNCTION get_balances(
    i_users text[],
    OUT user text,
    OUT currency text,
    OUT balance numeric)
RETURNS SETOF record AS $$
    CLUSTER 'userdb'; RUN ON ALL;
$$ LANGUAGE plproxy;
```



plProxy: Run On All for for Stats

- Gather statistics from all the partitions and return summaries to the caller

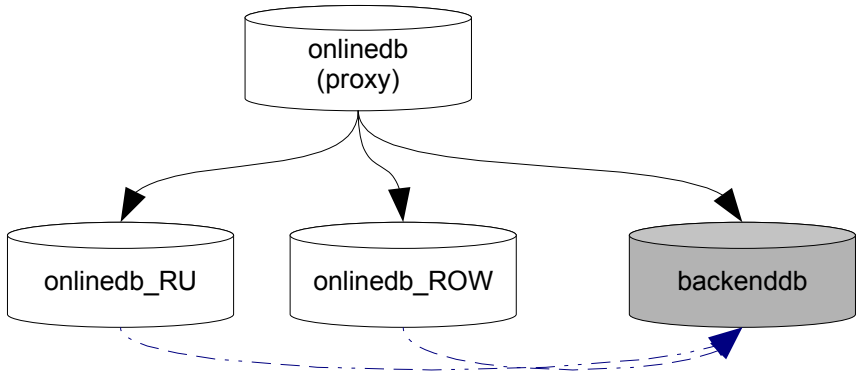
```
CREATE FUNCTION stats._get_stats(
    OUT stat_name text,
    OUT stat_value bigint)
RETURNS SETOF record AS $$
    cluster 'userdb';
    run on all;
$$ LANGUAGE plproxy;

CREATE FUNCTION stats.get_stats(
    OUT stat_name text,
    OUT stat_value bigint)
RETURNS SETOF record AS $$
    select stat_name,
           (sum(stat_value))::bigint
    from stats._get_stats()
    group by stat_name
    order by stat_name;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION stats._get_stats(
    OUT stat_name text,
    OUT stat_value bigint)
RETURNS SETOF record AS $$
declare
    seq record;
begin
    for seq in
        select c.relname,
               'select last_value from stats.'
               || c.relname as sql
          from pg_class c, pg_namespace n
         where n.nspname = 'stats'
           and c.relnamespace = n.oid
           and c.relkind = 'S'
         order by 1
    loop
        stat_name := seq.relname;
        execute seq.sql into stat_value;
        return next;
    end loop;
    return;
end;
$$ LANGUAGE plpgsql;
```

plProxy: Geographical Partitioning

- plProxy can be used to split database into partitions based on country code. Example database is split into 'us' and 'row' (rest of the world)
- Each function call caused by online users has country code as one of the parameters
- All data is replicated into backend database for use by backoffice applications and batch jobs. That also reduces number of indexes needed in online databases.

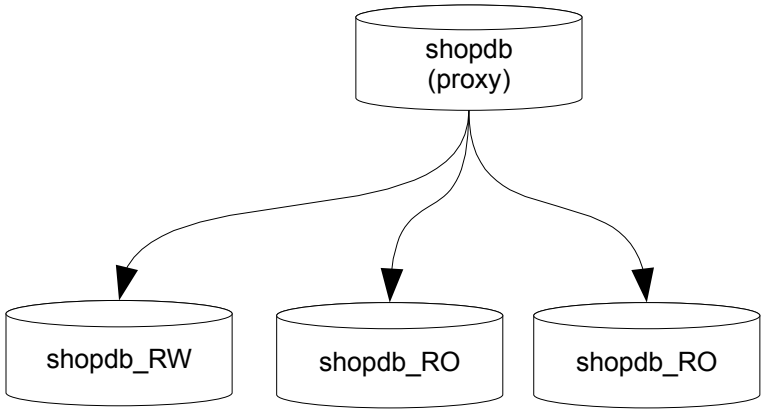
```
CREATE FUNCTION get_cluster(
    i_key_cc text)
RETURNS text AS $$
BEGIN
    IF i_key_cc = 'ru' THEN
        RETURN 'oltp_ru';
    ELSE
        RETURN 'oltp_row';
    END IF;
END;
$$ LANGUAGE plpgsql;
```



plProxy: Application Driven Partitioning

- Sometimes same function can be run on read only replica and on online database but the right partition is known only on runtime depending on context.
- In that situation get_cluster can be used to allow calling application to choose where this function will be executed.

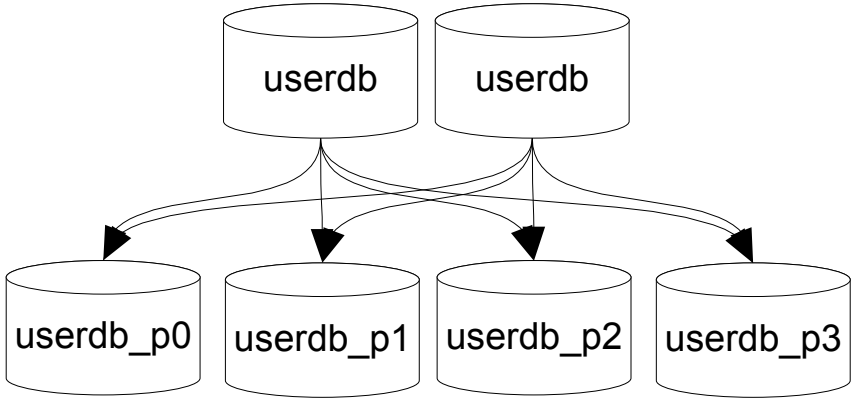
```
CREATE FUNCTION get_cluster(i_refresh boolean)
RETURNS text AS $$
BEGIN
    IF i_refresh THEN
        return 'shopdb';
    ELSE
        return 'shopdb_ro';
    END IF;
END; $$ LANGUAGE plpgsql;
```



plProxy: Horizontal Partitioning

- We have partitioned most of our database by username using PostgreSQL hashtext function to get equal distribution between partitions.
- As proxy databases are stateless we can have several exact duplicates for load balancing and high availability.

```
CREATE FUNCTION public.get_user_email(text)
RETURNS text AS $$
    cluster 'userdb';
    run on get_hash($1);
$$ LANGUAGE plproxy;

CREATE FUNCTION public.get_hash(
    i_key_user text)
RETURNS integer AS $$
BEGIN
    return hashtext(lower(i_key_user));
END;
$$ LANGUAGE plpgsql;
```



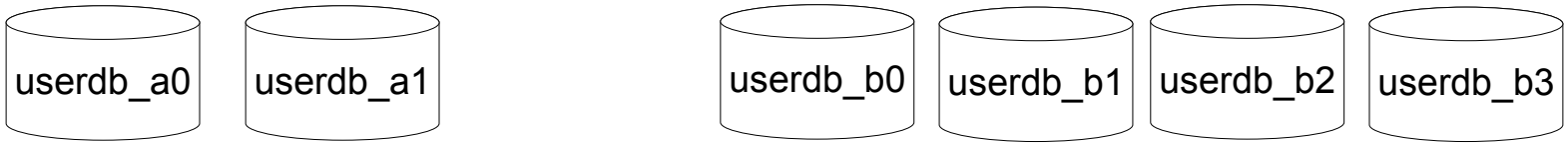
plProxy: Partitioning Partition Functions

- check_hash used to validate incoming calls
- partconf functionality to handle sequences
- validate_hash function in triggers to exclude data during split

```
CREATE FUNCTION public.get_user_email( i_username text)
RETURNS text AS $$
DECLARE
    retval text;
BEGIN
    PERFORM partconf.check_hash(i_username);

    SELECT email FROM users
    WHERE username = lower(i_username) INTO retval;

    RETURN retval;
END;
$$ LANGUAGE plpgsql;
```

plProxy: Horizontal Split

- When splitting databases we usually prepare new partitions in other servers and then switch all traffic at once to keep our life simple.
- Steps how to increase partition count by 2x:
  - Create 2 partitions for each old one.
  - Replicate partition data to corresponding new partitions
  - Delete unnecessary data from new partitions and cluster and analyze tables
  - Prepare new plProxy configuration
  - Switch over to them
    - Take down ip
    - Up plproxy version
    - Wait replica to catch up
    - Set ip up
- You can have several partitions in one server. Moving them out later will be easy.



plProxy: Partconf

- Partconf is useful when you have split database into several partitions and there are also failover replicas of same databases.
  - Some mechanism for handling sequences id's is needed
  - Some mechanism for protecting from misconfiguration makes life safer
- CREATE FUNCTION partconf.set_conf(
  i_part_nr integer ,
  i_max_part integer ,
  i_db_code integer)

  Used when partconf is installed to initialize it. db_code is supposed to be unique over all the databases.
- CREATE FUNCTION partconf.check_hash(key_user text) RETURNS boolean

  used internally from other functions to validate that function call was directed to correct partition. Raises error if not right partition for that username.
- CREATE FUNCTION global_id() RETURNS bigint

  used to get globally unique id's

plProxy: Summary

- plProxy adds very little overhead when used together with pgBouncer.
- On the other hand plProxy adds complexity to development and maintenace so it must be used with care but that is true for most everything.

pgBouncer

pgBouncer

- pgBouncer is lightweight and robust connection pooler for Postgres.
- Low memory requirements (2k per connection by default). This is due to the fact that PgBouncer does not need to see full packet at once.
- It is not tied to one backend server, the destination databases can reside on different hosts.
- Supports online reconfiguration for most of the settings.
- Supports online restart/upgrade without dropping client connections.
- Supports protocol V3 only, so backend version must be >= 7.4.
- Does not parse SQL so is very fast and uses little CPU time.

pgBouncer Pooling Modes

- **Session pooling** - Most polite method. When client connects, a server connection will be assigned to it for the whole duration it stays connected. When client disconnects, the server connection will be put back into pool. Should be used with legacy applications that won't work with more efficient pooling modes.
- **Transaction pooling** - Server connection is assigned to client only during a transaction. When PgBouncer notices that transaction is over, the server will be put back into pool. This is a hack as it breaks application expectations of backend connection. You can use it only when application cooperates with such usage by not using features that can break.
- **Statement pooling** - Most aggressive method. This is transaction pooling with a twist - multi-statement transactions are disallowed. This is meant to enforce "autocommit" mode on client, mostly targeted for PL/Proxy.

pgBouncer configuration

```
[databases]
forcedb = host=127.0.0.1 port=300 user=baz password=foo
bardb = host=127.0.0.1 dbname=bazdb
foodb =

[pgbouncer]
logfile = pgbouncer.log
pidfile = pgbouncer.pid
listen_addr = 127.0.0.1
listen_port = 6432
auth_type = any / trust / crypt / md5
auth_file = etc/userlist.txt
pool_mode = session / transaction / statement
max_client_conn = 100
default_pool_size = 20
admin_users = admin, postgres
stats_users = stats
```

pgBouncer console

- Connect to database 'pgbouncer':
  ```
  $ psql -d pgbouncer -U admin -p 6432
  ```
- SHOW HELP;
- SHOW CONFIG: SHOW DATABASES;
- SET default_pool_size = 50;
- RELOAD;
- SHOW POOLS; SHOW STATS;
- PAUSE; PAUSE <db>;
- RESUME; RESUME <db>;

pgBouncer Use-cases

- Decreasing the number of connections hitting real databases
- Taking the logging-in load from real databases
- Central redirection point for database connections. Function s like ServiceIP. Allows pausing, redirecting of users.
- Backend pooler for PL/Proxy
- Database aliases. You can have different external names connected to one DSN. Makes it possible to pause them separately and merge databases.

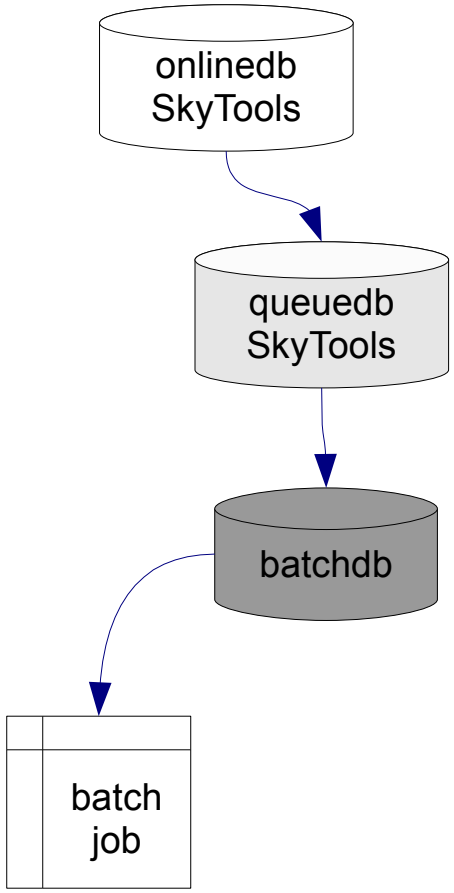
SkyTools/PgQ

SkyTools and PgQ

- We keep SkyTools like pgBouncer and plProxy in our standard database server image so each new server has it installed from the start.
- Framework provides database connectivity, logging, stats management, encoding, decoding etc for batch jobs. Developers need only to take care of business logic in batch jobs all the rest is handled by batch jobs.
- SkyTools contains many reusable generic scripts.

PgQ: PostgreSQL Queuing

- Ideally suited for all kinds of batch processing.
- **Efficient**. Creating events is adds very little overhead and queue has beautiful interface that allows events to be created using standard SQL insert into syntax.
- **Fast**. Events are processed in batches that usually means higher speed when processing.
- **Flexible**. Each queue can have several producers and consumers and any number of event types handled in it.
- **Robust**. PgQ guarantees that each consumers sees event at least once. There several methods how the processed events can be tracked depending on business needs.



PgQ - Setup

Basic PgQ setup and usage can be illustrated by the following steps:

1. create the database
2. edit a PgQ ticker configuration file, say ticker.ini
3. install PgQ internal tables

   ```
   $ pgqadm.py ticker.ini install
   ```
4. launch the PgQ ticker on database machine as daemon

   ```
   $ pgqadm.py -d ticker.ini ticker
   ```
5. create queue

   ```
   $ pgqadm.py ticker.ini create <queue>
   ```
6. register or run consumer to register it automatically

   ```
   $ pgqadm.py ticker.ini register <queue> <consumer>
   ```
7. start producing events

PgQ – Configuration

```
[pgqadm]
job_name = pgqadm_somedb

db = dbname=somedb

# how often to run maintenance
maint_delay = 600

# how often to check for activity
loop_delay = 0.1

logfile = ~/log/%(job_name)s.log
pidfile = ~/pid/%(job_name)s.pid
```

- Ticker reads event id sequence for each queue.
- If new events have appeared, then inserts tick if:
  - Configurable amount of events have appeared **`ticker_max_count (500)`**
  - Configurable amount of time has passed from last tick **`ticker_max_lag (3 sec)`**
- If no events in the queue, creates tick if some time has passed.
  - **`ticker_idle_period (60 sec)`**
- Configuring from command line:
  - **`pgqadm.py ticker.ini config my_queue ticker_max_count=100`**

PgQ event structure

```
CREATE TABLE pgq.event (
      ev_id   int8 NOT NULL,
      ev_txid int8 NOT NULL DEFAULT txid_current(),
      ev_time timestamptz NOT NULL DEFAULT now(),
      -- rest are user fields --
      ev_type text,   -- what to expect from ev_data
      ev_data text,   -- main data, urlenc, xml, json
      ev_extra1 text, -- metadata
      ev_extra2 text, -- metadata
      ev_extra3 text, -- metadata
      ev_extra4 text  -- metadata
);
CREATE INDEX txid_idx ON pgq.event (ev_txid);
```

PgQ: Producer - API Event Insertion

- Single event insertion:
  - `pgq.insert_event(queue, ev_type, ev_data): int8`
- Bulk insertion, in single transaction:
  - `pgq.current_event_table(queue): text`
- Inserting with triggers:
  - `pgq.sqltriga(queue, ...) - partial SQL format`
  - `pgq.logutriga(queue, ...) - urlencoded format`

PgQ API: insert complex event with pure SQL

- Create interface table for queue with logutriga for encoding inserted events
- Type safety, default values, sequences, constraints!
- Several tables can insert into same queue.
- Encoding decoding is totally hidden from developers they just work with tables.

```
CREATE TABLE queue.mailq (
    mk_template bigint NOT NULL,
    username text NOT NULL,
    email text,
    mk_language text,
    payload text
);

CREATE TRIGGER send_mailq_trg
    BEFORE INSERT ON queue.mailq
    FOR EACH ROW
    EXECUTE PROCEDURE pgq.logutriga('mail_events', 'SKIP');

-- send e-mail
INSERT INTO queue.mailq (mk_template, username, email)
VALUES(73, _username, _email);
```

PgQ: Consumer - Reading Events

Registering

```
pgq.register_consumer(queue, consumer)
pgq.unregister_consumer(queue, consumer)
```

or

$ pgqadm.py <ini> register <queue> <consumer>

$ pgqadm.py <ini> unregister <queue> <consumer>

Reading

```
pgq.next_batch(queue, consumer): int8
pgq.get_batch_events(batch_id): SETOF record
pgq.finish_batch(batch_id)
```

PgQ: Tracking Events

- Per-event overhead
- Need to avoid accumulating
- pgq_ext solution
  - `pgq_ext.is_event_done(consumer, batch_id, ev_id)`
  - `pgq_ext.set_event_done(consumer, batch_id, ev_id)`
- If batch changes, deletes old events
- Eg. email sender, plProxy.

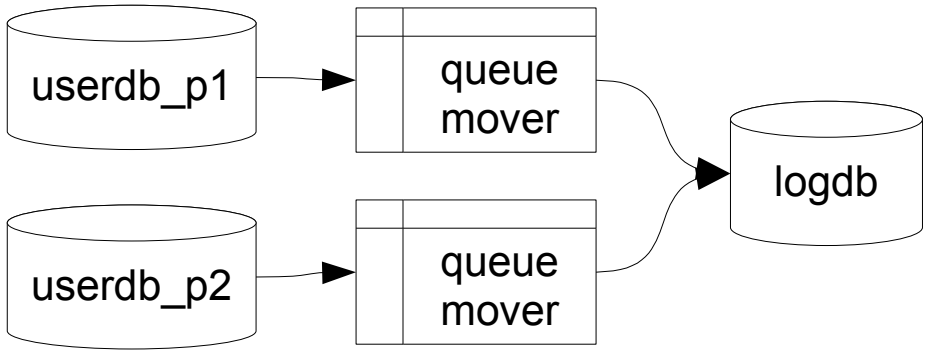
PgQ: Tracking Batches

- Minimal per-event overhead
- Requires that all batch is processed in one TX
  - `pgq_ext.is_batch_done(consumer, batch_id)`
  - `pgq_ext.set_batch_done(consumer, batch_id)`
- Eg. replication, most of the SkyTools partitioning scripts like queue_mover, queue_splitter, table_dispatcher, cube_dispatcher.

PgQ: Queue Mover

- Moves data from source queue in one database to another queue in other database.
- Used to move events from online databases to other databases for processing or just storage.
- Consolidates events if there are several producers as in case of partitioned databases.

```
[queue_mover]
job_name = eventlog_to_target_mover

src_db = dbname=oltpdb
dst_db = dbname=batchdb

pgq_queue_name = batch_events
dst_queue_name = batch_events

pidfile = log/%(job_name)s.pid
logfile = pid/%(job_name)s.log
```

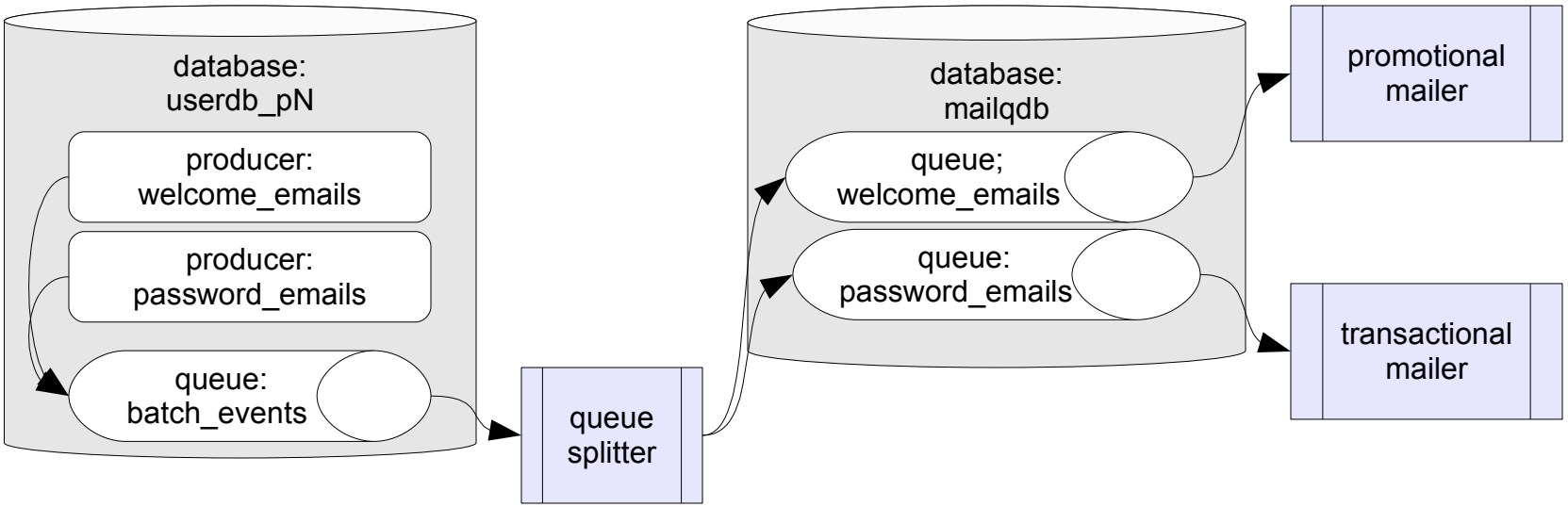


PgQ: Queue Mover Code

```
import pgq
class QueueMover(pgq.RemoteConsumer):
  def process_remote_batch(self, db, batch_id, ev_list, dst_db):
    # prepare data
    rows = []
    for ev in ev_list:
      rows.append([ev.type, ev.data, ev.time])
      ev.tag_done()

    # insert data
    fields = ['ev_type', 'ev_data', 'ev_time']
    curs = dst_db.cursor()
    dst_queue = self.cf.get('dst_queue_name')
    pgq.bulk_insert_events(curs, rows, fields, dst_queue)

script = QueueMover('queue_mover', 'src_db', 'dst_db', sys.argv[1:])
script.start()
```

PgQ: Queue Splitter

- Moves data from source queue in one database to one or more queue's in target database based on producer. That is another version of queue_mover but it has it's own benefits.
- Used to move events from online databases to queue databases.
- Reduces number of dependencies of online databases.



PgQ: Simple Consumer

- Simplest consumer. Basic idea is to read url encoded events from queue and execute SQL using fields from those events as parameters.
- Very convenient when you have events generated in one database and want to get background processing or play them into another database.
- Does not guarantee that each event is processed only once so it's up to the called function to guarantee that duplicate calls will be ignored.

```
[simple_consumer]
job_name        = job_to_be_done
src_db          = dbname=sourcedb
dst_db          = dbname=targetdb
pgq_queue_name = event_queue
dst_query       =
    SELECT * FROM function_to_be_run(%%(username)s, %%(email)s);

logfile         = ~/log/%(job_name)s.log
pidfile         = ~/pid/%(job_name)s.pid
```

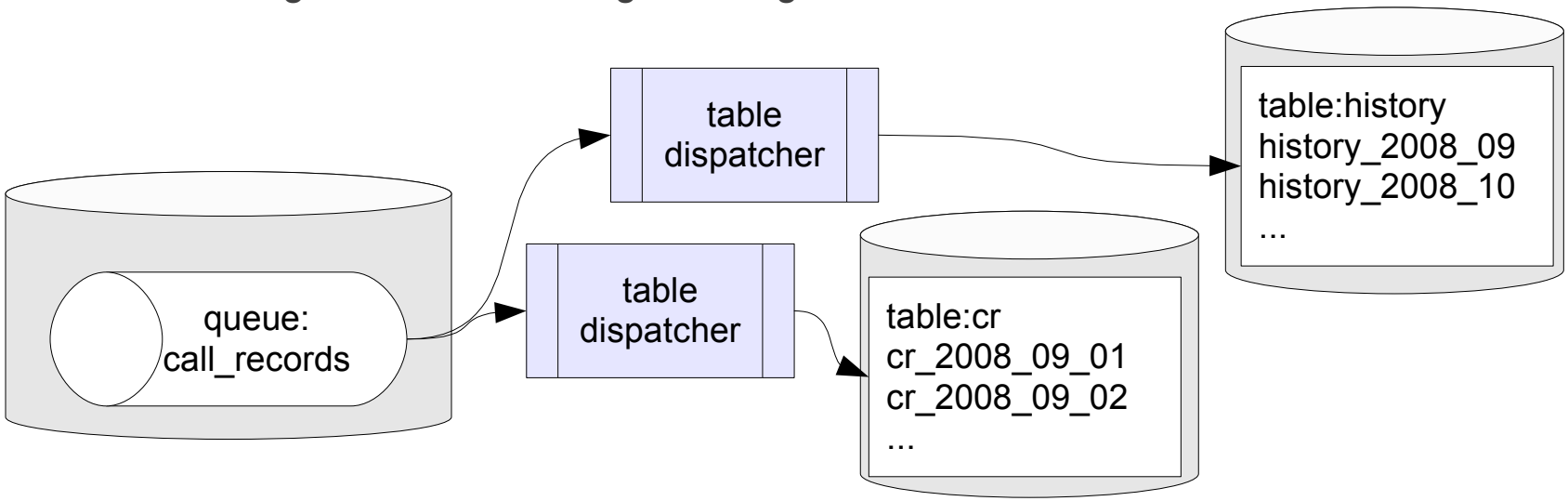
PgQ: Simple Serial Consumer

- Similar to simpler consumer but uses pgq_ext to guarantee that each event will be processed only once.

```
import sys, pgq, skytools

class SimpleSerialConsumer(pgq.SerialConsumer):
    def __init__(self, args):
        pgq.SerialConsumer.__init__(self,"simple_serial_consumer","src_db","dst_db", args)
        self.dst_query = self.cf.get("dst_query")

    def process_remote_batch(self, batch_id, ev_list, dst_db):
        curs = dst_db.cursor()
        for ev in ev_list:
            payload = skytools.db_urldecode(ev.payload)
            curs.execute(self.dst_query, payload)
            res = curs.dictfetchone()
            ev.tag_done()

if __name__ == '__main__':
    script = SimpleSerialConsumer(sys.argv[1:])
    script.start()
```

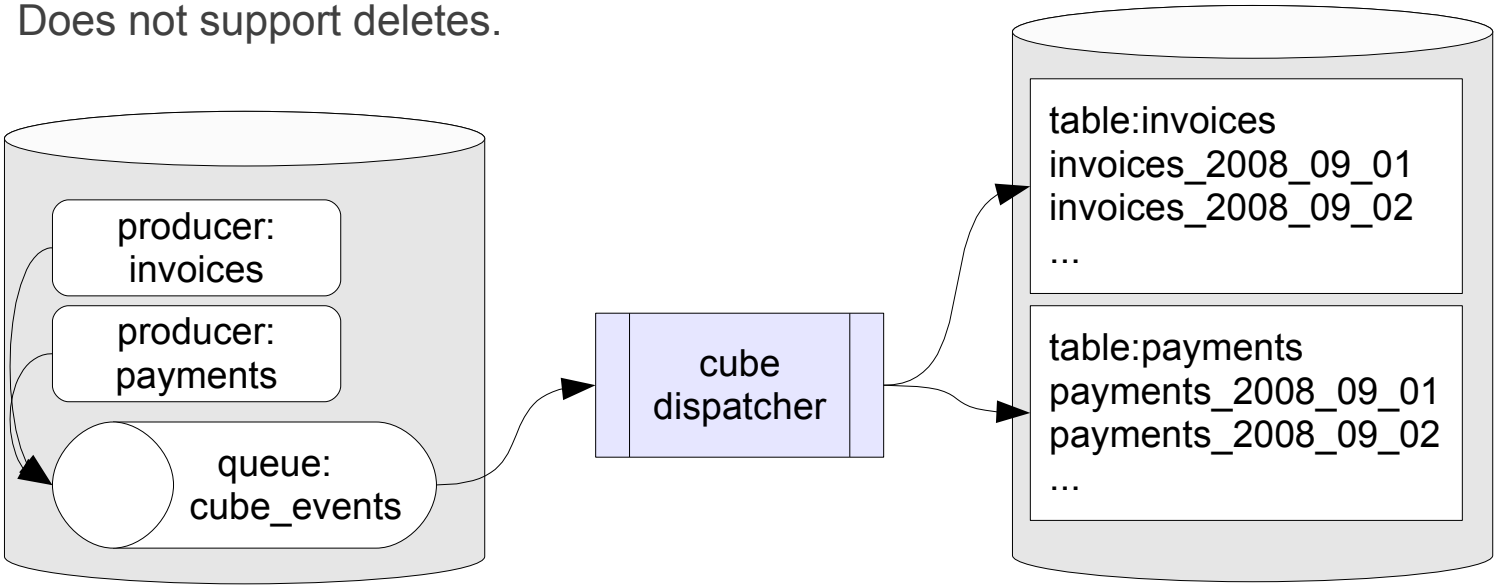
PgQ: Table Dispatcher

- Has url encoded events as data source and writes them into table on target
- database.
- Used to partiton data. For example change log's that need to kept online only shortly
- can be written to daily tables and then dropped as they become irrelevant.
- Also allows to select which columns have to be written into target database
- Creates target tables according to configuration file as needed



PgQ: Cube Dispatcher

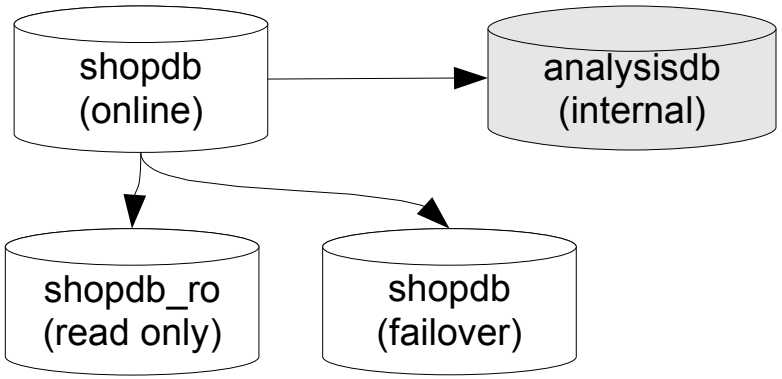
- Has url encoded events as data source and writes them into partitoned tables in target database. Logutriga is used to create events.
- Used to provide batches of data for business intelligence and data cubes.
- Only one instance of each record is stored. For example if record is created and then updated twice only latest version of record stays in that days table.
- Does not support deletes.



Londiste

Londiste: Replication

- We use replication
  - to transfer online data into internal databases
  - to create failover databases for online databases.
  - to switch between PostgreSQL versions
  - to distribute internally produced data into online servers
  - to create read only replicas for load balancing
- Londiste is implemented using PgQ as transport layer.
- It has DBA friendly command line interface.



Londiste: Setup

1. create the subscriber database, with tables to replicate
2. edit a londiste configuration file, say conf.ini, and a PgQ ticker configuration file, say ticker.ini
3. install londiste on the provider and subscriber nodes. This step requires admin privileges on both provider and subscriber sides, and both install commands can be run remotely:

   $ londiste.py conf.ini provider install

   $ londiste.py conf.ini subscriber install
4. launch the PgQ ticker on the provider machine:

   $ pgqadm.py -d ticker.ini ticker
5. launch the londiste replay process:

   $ londiste.py -d conf.ini replay
6. add tables to replicate from the provider database:

   $ londiste.py conf.ini provider add table1 table2 ...
7. add tables to replicate to the subscriber database:

   $ londiste.py conf.ini subscriber add table1 table2 ...

Londiste: Configuration

```
[londiste]
job_name = londiste_providerdb_to_subscriberdb

provider_db = dbname=provider port=5432 host=127.0.0.1
subscriber_db = dbname=subscriber port=6432 host=127.0.0.1

# it will be used as sql ident so no dots/spaces
pgq_queue_name = londiste.replika

logfile = /tmp/%(job_name)s.log
pidfile = /tmp/%(job_name)s.pid
```

Londiste: Usage

- Tables can be added one-by-one into set.
- Initial COPY for one table does not block event replay for other tables.
- Can compare tables on both sides.
- Supports sequences.
- There can be several subscribers listening to one provider queue.
- Each subscriber can listen to any number of tables provided by provider all the other events are ignored.

That was not all

SkyTools: WalMgr

- Used to creat standby servers inside same colocation.
- Also provides point in time recovery.
- Requires more network bandwidth than londiste but has very little management overhead.
- Most common usage is when servers starts showing signs of braking down then we move database into gresh server until other one gets fixed.
- Also upgrade on better hardware is quite common.

SODI Framework

- Rapid application development
- Cheap changes
- Most of the design is in metadata.
- Application design stays in sync with application over time.
- Minimizes number of database roundtrips.
- Can send multiple recordsets to one function call.
- Can receive several recordsets from function call.

Application layer
- java / php ...
- UI logic
. mostly generated
- SODI framework

AppServer layer:
- java / php(...
- user authentication
- roles rights
- no business logic

Database layer
- business logic
- plPython
- PostgreSQL
- SkyTools

Multiple Recordsets in Function Resultset

```
CREATE FUNCTION meta.get_iotype(i_context text, i_params text)
RETURNS SETOF public.ret_multiset AS $$
    import dbservice2
    dbs = dbservice2.ServiceContext( i_context, GD )
    params = dbs.get_record( i_params )

    sql = """
        select id_iotype, mod_code, schema || '.' || iotype_name as iotype
             , comments ,version
        from meta.iotype
        where id_iotype = {id_iotype}
    """
    dbs.return_next_sql( sql, params, 'iotype' )

    sql = """
        select id_iotype_attr, key_iotype ,attr_name, mk_datatype
             , comments, label, version, order_by, br_codes
        from meta.iotype_attr
        where key_iotype = {id_iotype}
        order by order_by
    """
    dbs.return_next_sql( sql, params, 'ioattr')

    return dbs.retval()
$$ LANGUAGE plpythonu;
```

Multiple Recordsets as Function Parameters

```
CREATE FUNCTION dba.set_role_grants(
    i_context text,
    i_params text,
    i_roles text,
    i_grants text)
RETURNS SETOF public.ret_multiset AS $$
    import dbservice2
    dbs = dbservice2.ServiceContext( i_context, GD )
    params = dbs.get_record( i_params )
    t_roles = dbs.get_record_list( i_roles )
    t_grants = dbs.get_record_list( i_grants )
...
    for r in t_roles:
        key_role = roles.get(r.role_name)
        r.id_host = params.id_host
        r.key_role = key_role

        if key_role is None:
            dbs.run_query( """
                insert into dba.role (
                    key_host, role_name, system_privs, role_options, password_hash
                  , connection_limit, can_login, granted_roles
                ) values (
                  {id_host},{role_name},{system_privs},{role_options},{password_hash}
                 , {connection_limit},{can_login},string_to_array({granted_roles},','))
            """, r)
            dbs.tell_user(dbs.INFO, 'dbs1213',  'Role added: %s' % r.role_name)
...
```

That Was All

- :)