Skytools: PgQ

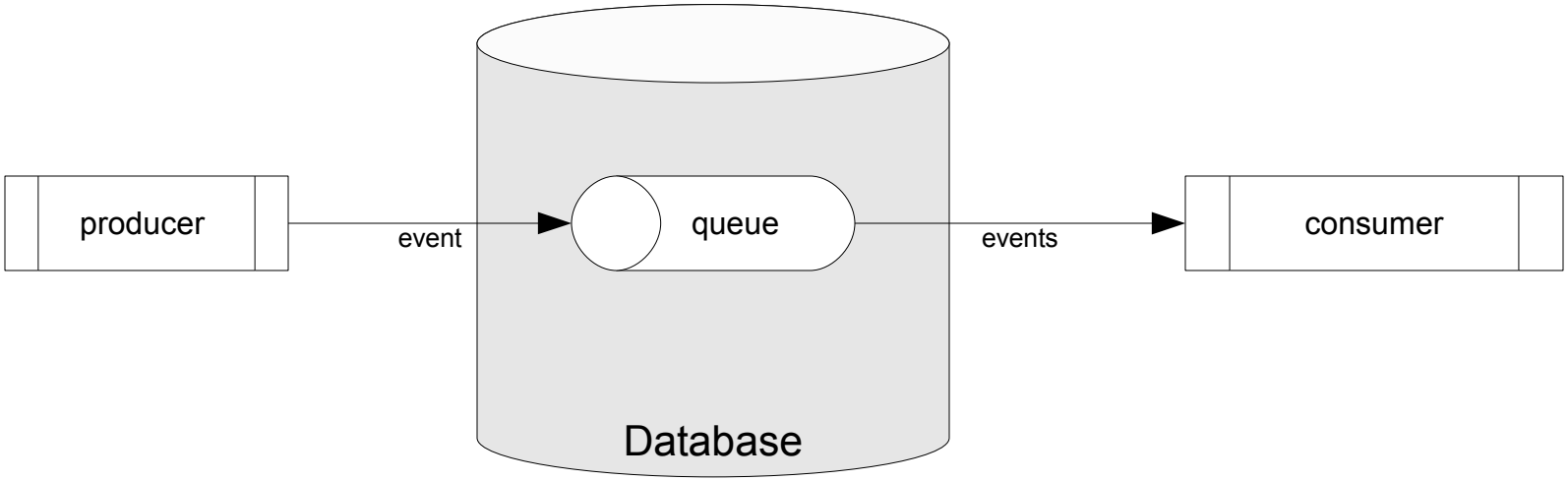
Queues and applications

Agenda

- PgQ basics
  - Queues, producers and consumers
  - New features in 3.0
- Skytools – toolset and scripting framework
  - Custom consumers
  - Replication toolset
- Managing the Skytools environment
  - Installing
  - Migrations and upgrades
  - Monitoring

What is PgQ?

- A **queue** implementation on PostgreSQL with a stored procedure interface.
- Events are queued by producers to be subsequently processed by consumers.



PgQ: what it's good for?

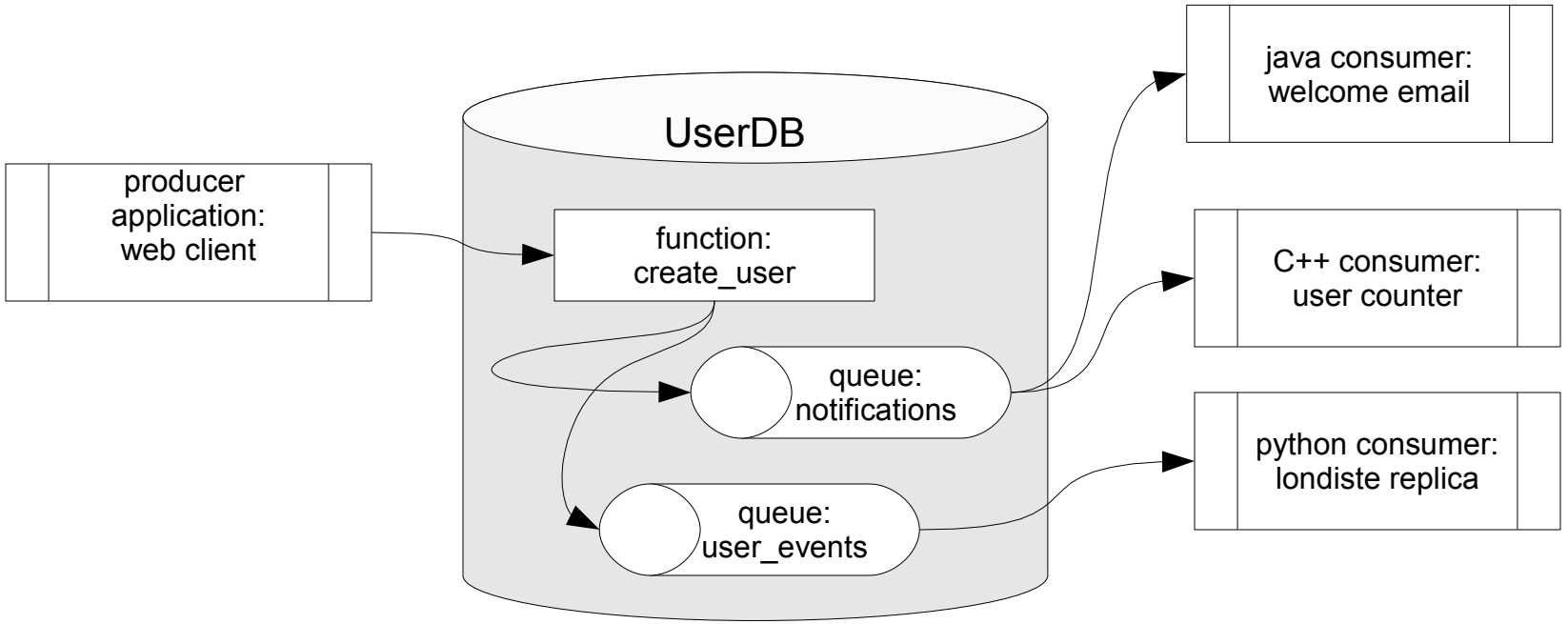
- Asynchronous messaging
- Batch processing
- Replication
- Distributed transactions

PgQ: Features

- **Transactional**. Events are created transactionally, can be coupled with surrounding business logic.
- **Efficient**. Events are processed in batches which gives low per event overhead.
- **Flexible.** No limits on the number of producers or consumers. Custom event formats.
- **Reliable.** Events are stored in PostgreSQL database – this adds the benefit of write ahead logging and crash recovery.
- **Easy to use**. Simple SQL interface, API-s for several languages.
- **Open Source.** No licensing fees, but occasionally you'll have to get your hands dirty.

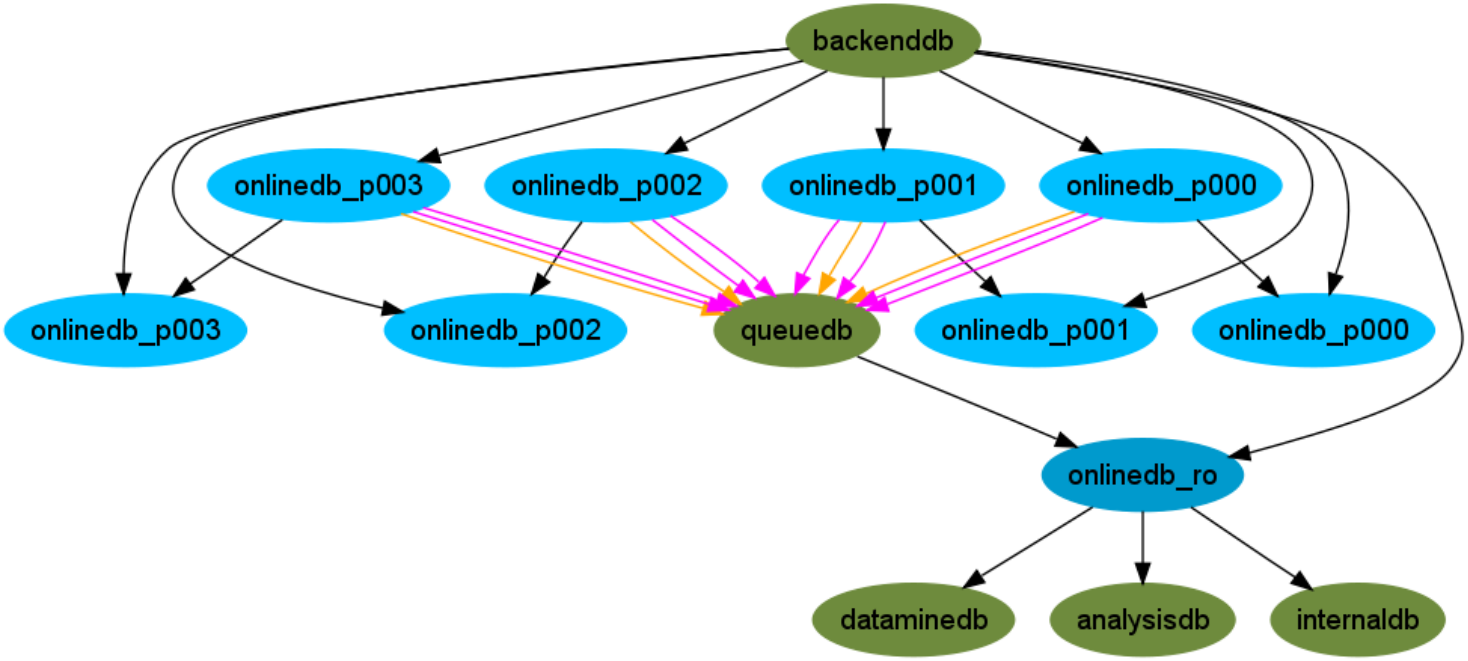
PgQ: example

- Database for registering user accounts.
- Events are generated by a pl/pgsql stored procedure.
- Consumers also talk to the queue through stored procedure interface.



PgQ: at Skype

- Used everywhere where asynchronous data processing is needed.
- Hundreds of queues and consumers.
- Centrally monitored.



PgQ: brief history

- Started on 2006
- Inspired by ideas from Slony
- First application was Londiste replication
- Open source since 2007 as part of Skytools framework
- Version 3.0 in progress, alpha version out.

PgQ: glossary

- **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.
- **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: Queue

- Essentially a set of tables in a PostgreSQL database.
- Default is to have 3 tables per queue, these are rotated to efficiently purge discarded events.
- Event is discarded when all the consumers have processed it.
- Queues are accessible through stored procedure API. Tools also available.
- There can be multiple queues in one database.
- Any number of producers and consumers to the queue.

```
userdb=# select * from pgq.queue;
-[ RECORD 1 ]------------+---
queue_id                 | 1
queue_name               | q1
queue_ntables            | 3
queue_cur_table          | 2
queue_data_pfx           | pgq.event_1
...
userdb=# \dt pgq.event_1*
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+---------
 pgq    | event_1   | table | martinp
 pgq    | event_1_0 | table | martinp
 pgq    | event_1_1 | table | martinp
 pgq    | event_1_2 | table | martinp
(4 rows)
```

PgQ: Queue API

- Creating and dropping queues
  - `pgq.create_queue(qname)`
  - `pgq.drop_queue(qname)`
- Queue information functions
  - `pgq.get_queue_info(qname)`
  - `pgq.get_consumer_info(qname)`
  - `pgq.current_event_table(qname)`
- Managing consumers
  - `pgq.register_consumer(qname, cname)`
  - `pgq.unregister_consumer(qname, cname)`

PgQ: Event

- A record in the queue table.
- Internal fields used for event processing.
- Payload data, with user defined content.
- The content format is agreed between producer and consumer.
- Field names hint at their intended usage.

```
userdb=# \d pgq.event_1
  Column   |           Type
-----------+--------------------------
 ev_id     | bigint
 ev_time   | timestamp with time zone
 ev_txid   | bigint
 ev_owner  | integer
 ev_retry  | integer
 ev_type   | text
 ev_data   | text
 ev_extra1 | text
 ev_extra2 | text
 ev_extra3 | text
 ev_extra4 | text
Inherits: pgq.event_template
```

PgQ: Batch

- Events are grouped into batches for efficient processing.
- Consumers obtain events in batches.
- Batch size can be tuned to suit the application or network topology. For example, we might want to use a larger batch size for processing over wide area networks.
- Small batches have higher processing overhead, however too big batches have their own disadvantages.
- Batches are prepared by separate process called **the ticker**.

PgQ: Ticker

- Is a daemon that periodically creates **ticks** on the queues. The tick is essentially a position in the event stream.
- A batch is formed of events that are enqueued between two ticks.
- Without ticker there are no batches, without batches events cannot be processed.
- Pausing the ticker for extended period will produce a huge batch, consumers might not be able to cope with it.
- Ticker is also involved in miscellaneous housekeeping task, such as vacuuming pgq tables, scheduling retry events and rotating queue tables.
- Keep the ticker running!

PgQ: Consumer

- Subscribes to queue.
- Obtains events from queue by asking for a batch.
- Sees only events that have been produced **after** the subscription.
- Events are seen **at least once** – events don't get lost.
- Must have some sort of event tracking to process **only once** – skytools has several implementations.
- If the event cannot be immediately processed it can be postponed for retry processing (eg. some resource temporarily unavailable).

PgQ: Event processing

- Ask for next batch id:
  `pgq.next_batch(queue, consumer)`
- Nothing to do if **NULL** returned – sleep and try again.
- Ask the set of events to be returned:
  `pgq.get_batch_events(batch_id)`
- Process the events. Note that the batch can be empty if there were no events for the period.
- Schedule the event for retry processing if necessary:
  `pgq.event_retry(batch_id, ev_id, sec)`
- Finalize the batch:
  `pgq.finish_batch(batch_id)`

Event structure

```
  Column   |  Type
-----------+------------
 ev_id     | bigint
 ev_time   | timestamptz
 ev_txid   | bigint
 ev_retry  | integer
 ev_type   | text
 ev_data   | text
 ev_extra1 | text
 ev_extra2 | text
 ev_extra3 | text
 ev_extra4 | text
```

PgQ: Consumer status

- Can be obtained by calling `pgq.get_consumer_info()`
- Reports queue, consumer name, lag and last seen for all of the consumers. Parameterized versions also available.
- **lag** is the age of the last finished batch.
- **last seen** is the elapsed time since consumer processed a batch.

```
userdb=# select * from pgq.get_consumer_info();
-[ RECORD 1 ]-+----------------
queue_name    | notifications
consumer_name | welcome_emailer
lag           | 02:32:34.440654
last_seen     | 00:00:00.019398
last_tick     | 4785
current_batch | 4754
next_tick     | 4786
```

PgQ: Event tracking

- PgQ guarantees that the consumer sees the event **at least once**. But this could mean that the consumer could see the event **more than once**.
- This happens if the consumer crashes in the middle of processing a batch.
- Not a problem if the events are processed in the same database that they are produced – just keep the processing in the same transaction.
- We need to address the case where events are processed outside a database or in a remote database.
- Consumer needs to be able to keep track of which events are already processed.

PgQ: Event tracking

- Event and batch tracking support is included in **pgq_ext** schema – this is part of Skytools, but must be installed separately to target database.

- Use batch tracking when the whole batch is processed transactionally
```
pgq_ext.is_batch_done(consumer, batch_id)
pgq_ext.set_batch_done(consumer, batch_id)
```

- Per-event tracking is used to keep track of single events:
```
pgq_ext.is_event_done(consumer, batch_id, ev_id)
pgq_ext.set_event_done(consumer, batch_id, ev_id)
```

- Event tracking can be used to implement distributed transactions.

PgQ: distributed transactions

- Event tracking can be used to implement asynchronous distributed transactions.
- We'll use batch tracking as an example.
- Skip the batch if it is already processed on target.
- Otherwise process the batch and mark as processed.
- Commit on target.
- Finish batch and commit on source.
- Skytools framework handles this automatically!

PgQ: Producer

- Anything that places events into queues.
- Event payload format agreed between producer and consumer.
- Basic usage via SQL API.
  - `pgq.insert_event(queue, ev_type, ev_data)`
- Replication uses triggers for providing events.
  - `pgq.sqltriga(queue, options)`
  - `pgq.logutriga(queue, options)`
- Bulk load is also possible.

PgQ: sqltriga and logutriga

- Log triggers are used for enqueuing table change log entries. Typically used for replication, but have other uses as well.
- Table structure is detected automatically - no messing around with column definition lists.
- Arguments:
  - `SKIP` – enqueue only, skip the actual DML operation. Used in before-triggers.
  - `ignore=cols` - The listed columns will be omitted from payload.
  - `pkey=cols` - Defines a primary key for the table.
  - `backup` – Add a copy of the original row to payload.
- Event format:
  - `pgq.sqltriga` – partial SQL format used by Londiste replication.
  - `pgq.logutriga` – URL encoded format.

PgQ: logutriga payload

- logutriga uses database specific URL encoding for payload data:
  - `column1=value1&column2=value2&`**`column3`**`&...`
  - column names and data values are URL encoded.
  - NULL values are specified by omitting value and equal sign.
- Payload:
  - `ev_type` – operation type: I/U/D plus primary key columns.
  - `ev_data` – URL encoded row data
  - `ev_extra1` – Table name
  - `ev_extra2` – URL encoded row backup

PgQ: logutriga example

- We'll add a trigger to users table that enqueues notification events for new users.
- Add **after-insert** trigger that executes **logutriga** with some columns ignored.
- We could use **before-insert** triggers with **SKIP** option to implement queue only tables.

```
create table users (
user_id       serial primary key,
username      text   unique,
password      text   not null,
email         text,
date_created  timestamp default now()
);

create trigger welcome_user_trg
after insert on users
for each row execute procedure
pgq.logutriga(
    'notifications',
    'ignore=password');
```

PgQ: logutriga in action

Insert statement

```
insert into users (
    username,
    password,
    email
) values (
    'bob',
    'secret',
    'bob@foo.bar'
);
```

Event data

```
ev_id     | 26
ev_time   | 2009-05-14 11:07:54.231954+02
ev_txid   | 263834
ev_owner  |
ev_retry  |
ev_type   | I:user_id
ev_data   | user_id=1&username=bob&email=bob%40foo.bar
ev_extra1 | public.users
ev_extra2 |
ev_extra3 |
ev_extra4 |
```

Skytools 3: new features

- **Cooperative consumer** – distributing the load of a single consumer between cooperating sub consumers.
- **Cascading support** – identical queue is maintained across nodes, consumers can easily move between nodes.
- Per-database tickers replaced with a single **pgqd** daemon.
- **qadmin** utility for managing queues and consumers.

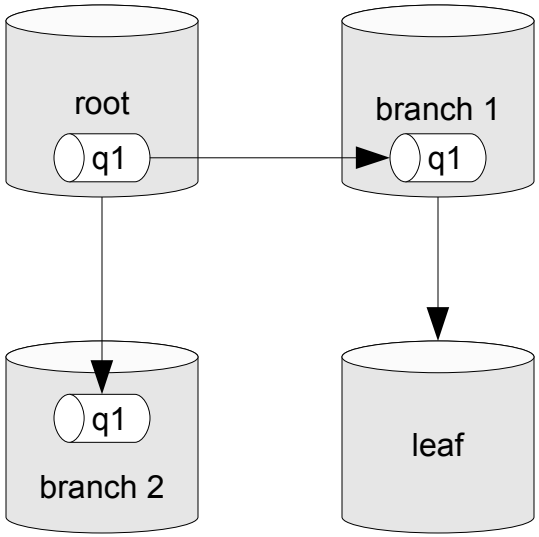
PgQ: Cooperative consumer

- Sometimes single consumer cannot keep up with the volume of incoming events.
- Parallel consumers would help, but we need to somehow divide the workload – avoid processing the same event twice.
- We need the consumers to work in cooperation!
- Skytools 3 introduces sub consumers for the purpose. These share the workload of the master consumer in a cooperative manner.
- There are some differences in registration and batch handling, but they look a lot like regular consumers:

```
pgq_coop.register_subconsumer(qname,cname,scname)
pgq_coop.unregister_subconsumer(qname,cname,scname,mode)
pgq_coop.next_batch(qname,cname,scname)
pgq_coop.finish_batch(batch_id)
```

PgQ: Cascading

- The cascade is a set of database nodes and a queue that is distributed between the nodes. Event and batch numbers are kept identical!
- The cascade can be depicted as a tree, where events created in the root are propagated down the cascade to other nodes.
- There can be only one root node, but any number of branches or leaves.
- Leaf nodes are specific to replication. They don't have a copy of the queue and don't participate in event propagation.

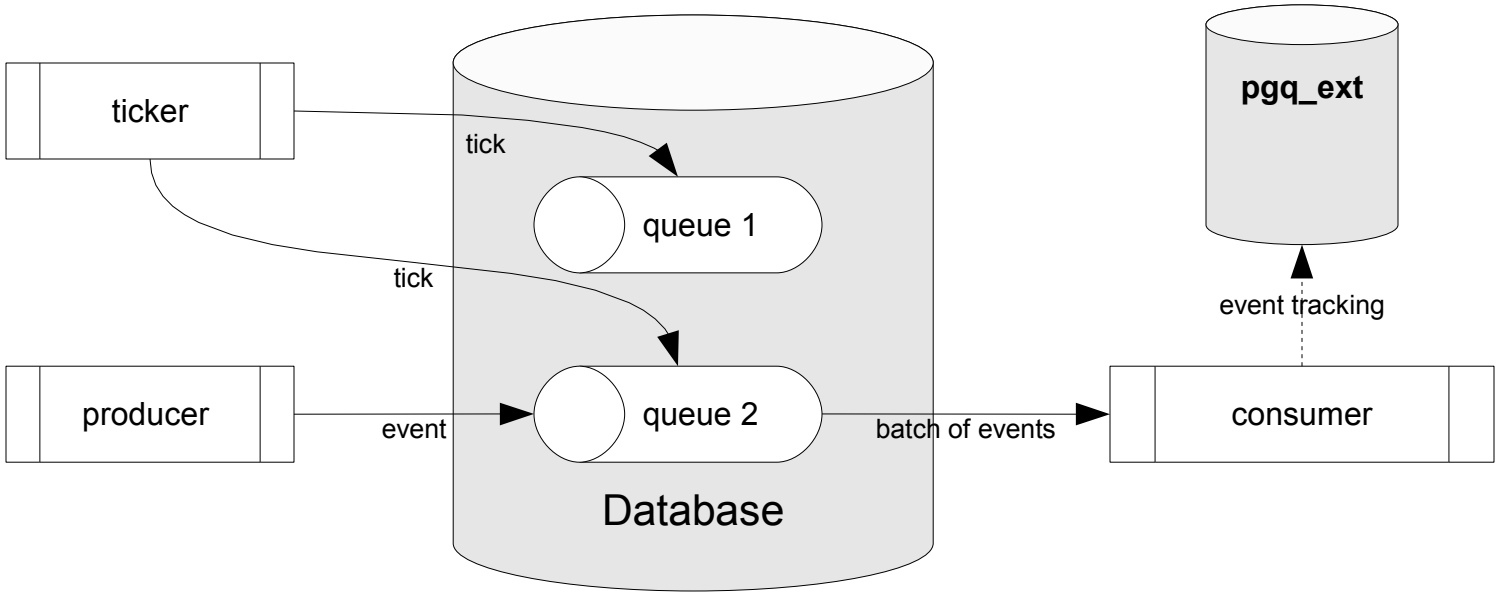


PgQ: Cascading

- Typical replication cascade would consist of a primary database -- the root, replicated to standby database – a branch.
- We can easily switch the replication roles of root and branch nodes. Consumers will continue as if nothing happened.
- On branch node failure we move its consumers to some surviving node. Business as usual.
- On root node failure we promote some surviving branch to root and reattach the consumers. In this scenario we have to deal with data loss.

PgQ wrapup

- **Producers** produce **events** into **queues**.
- **Ticker** groups events into **batches**.
- Batches are served to **consumers** in FIFO order.
- Consumers can track processed events with **pgq_ext**.



Skytools

Toolset and scripting framework

Skytools: introduction

- Set of applications for replication, batch processing and queue management.
- Includes a Python scripting framework that greatly simplifies the implementation of custom consumers.
- Written mostly in Python with some bits in C.
- PgQ is distributed as part of Skytools.
- Get it from http://pgfoundry.org/projects/skytools/
- We'll start by configuring the ticker.

The Ticker

- The ticker prepares batches for the consumers.
- Also performs queue maintenance: table rotation, vacuuming, re-queuing retry events.
- We'll need to configure it for each database that has queues.
- Skytools 3 includes a super-ticker that handles all the databases in a PostgreSQL cluster.

pgqadm.py: configuration file

- **db** – the name of the database where queues are located.
- **maint_delay_min** – interval, at which maintenance commands are run.
- **loop_delay** – interval, at which the ticker wakes up to check for work.

```
[pgqadm]
job_name = pgqadm_userdb
db = dbname=userdb

# how often to run maintenance [minutes]
maint_delay_min = 5

# how often to check for activity [secs]
loop_delay = 0.5

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

pgqadm.py: installing and starting

- Install pgq schema
- Start the ticker in background
- Repeat the process for all databases with queues.

```
$ pgqadm.py pgqadm_userdb.ini install
2009-05-13 12:37:17,913 19376 INFO plpgsql is installed
2009-05-13 12:37:17,936 19376 INFO txid_current_snapshot is installed
2009-05-13 12:37:17,936 19376 INFO Installing pgq
2009-05-13 12:37:17,984 19376 INFO   Reading from /usr/local/share/skytools/pgq.sql

$ pgqadm.py pgqadm_userdb.ini ticker -d
...
$ tail log/pgqadm_userdb.log
2009-05-13 12:45:42,572 17184 INFO {ticks: 1}
2009-05-13 12:45:42,586 17184 INFO {maint_duration: 0.0229749679565}
2009-05-13 12:50:42,639 17184 INFO {maint_duration: 0.0530850887299}
2009-05-13 12:50:42,719 17184 INFO {ticks: 9}
```

pgqadm.py: command line

- pgqadm.py provides additional functionality besides the ticker:
  - creating and configuring queues
  - managing consumers
  - querying status
- Convenient front end for the PgQ SQL API.

```
Usage: pgqadm.py [options] INI CMD [subcmd args]

commands:
  ticker                    start ticking & maintenance process

  status                    show overview of queue health

  install                   install code into db
  create QNAME              create queue
  drop QNAME                drop queue
  register QNAME CONS       install code into db
  unregister QNAME CONS     install code into db
  config QNAME [VAR=VAL]    show or change queue config
```

pgqadm.py: creating queues and consumers

- We'll create a queue called *notifications* and subscribe a consumer *welcome_consumer* to it.

```
$ pgqadm.py pgqadm_userdb.ini create notifications
2009-05-12 17:53:34,726 12610 INFO Creating queue: notifications

$ pgqadm.py pgqadm_userdb.ini register notifications welcome_consumer
2009-05-12 17:53:47,808 12632 INFO Registering consumer welcome_consumer on queue
notifications

$ pgqadm.py pgqadm_userdb.ini status
Postgres version: 8.3.7   PgQ version: 2.1.8

Event queue                                      Rotation        Ticker   TLag
------------------------------------------------------------------------------
notifications                                     3/7200s    500/3s/60s     1s
------------------------------------------------------------------------------
Consumer                                                        Lag  LastSeen
------------------------------------------------------------------------------
notifications:
  welcome_consumer                                             116s      103s
------------------------------------------------------------------------------
```

pgqadm.py: setting queue parameters

- Per-queue tuning options:
  - `ticker_max_lag` – Max time between ticks.
  - `ticker_idle_period` – Tick at interval, if no events.
  - `ticker_max_count` – Tick, if number of events exceeds this. Can be used to tune batch sizes.
  - `rotation_period` – How often to rotate queue tables, balance between disk space and event history.

```
$ pgqadm.py pgqadm_userdb.ini config notifications
notifications
    ticker_max_lag      =      3
    ticker_idle_period  =     60
    rotation_period     =   7200
    ticker_max_count    =    500

$ pgqadm.py pgqadm_userdb.ini config notifications ticker_max_count=1000
Change queue notifications config to: queue_ticker_max_count='1000'
```

pgqadm.py: summary

- Runs the ticker.
- Performs queue maintenance.
- Provides command line interface for managing the queues and consumers.

Python framework

- The framework handles database connection management, logging, statistics, quoting, daemonization etc.
- Most of the Skytools applications are implemented by extending **DBScript** - a Python class providing the infrastructure needed for typical database batch job.
- Several base classes available for implementing custom consumer applications.

DBScript: configuration

- Most DBScripts have a configuration file that defines the parameters for the script – source database, queue name, log file location etc.
- Python ConfigParser format.
- Common options:
  - `job_name` – Identifies the current script
  - `loop_delay` – how often to check for work, seconds.
  - `pidfile` – pid file for daemons
  - `logfile` – log file name
  - `log_size` – size of individual log files.
  - `log_count` – number of log files kept.
  - `use_skylog` – override logging configuration by skylog.ini

DBScript: command line

- Standard invocation is:
  `script.py` *`configuration.ini`* `[options]`

- Common options:
  - `-h, --help` – Show usage for the particular script.
  - `-v, --verbose` – Make the script more verbose.
  - `-q, --quiet` – Log only errors and warnings.
  - `-d, --daemon` – Run the script in background.
  - `-r, --reload` – Reload a running application,
  - `-s, --stop` – Wait for work loop to finish, then stop.
  - `-k, --kill` – Terminate the application immediately.

Custom consumer

- We'll write a small queue application called *welcome_consumer*.
- The application reads events from a queue and prints out event payload if type matches “welcome”.
- The application base class extends **Consumer** which in turn extends **DBScript**.
- Consumer implements the pgq consumer event loop. We only need to add the bits that do the event handling.
- All of the regular DBScript configuration and command options apply.

Custom consumer: configuration

- **`pgq_queue_name`** – name of the queue the consumer is subscribing to.
- **`pgq_consumer_id`** – name of the consumer, defaults to `job_name` if not present.

```
[welcome_app]
job_name        = welcome_consumer

src_db          = dbname=userdb

pgq_queue_name  = notifications
pgq_consumer_id = %(job_name)s

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

Custom consumer: Python code

```
import sys, pgq, skytools

class WelcomeConsumer(pgq.Consumer):
    def __init__(self, args):
        pgq.Consumer.__init__(self,
           "welcome_app", "src_db", args)

    def process_event(self, src_db, ev):
        if ev.ev_type == 'welcome':
            self.log.info('Welcome %s!' % ev.ev_data)
        ev.tag_done()

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

Event structure

```
  Column   |   Type

-----------+------------
 ev_id     | bigint
 ev_time   | timestamptz
 ev_txid   | bigint
 ev_retry  | integer
 ev_type   | text
 ev_data   | text
 ev_extra1 | text
 ev_extra2 | text
 ev_extra3 | text
 ev_extra4 | text
```

Custom consumer: event processing

- The **`process_event()`** function is called for each event in a batch. If there are no batches the script sleeps **loop_delay** seconds and retries.
- A DB API connection to the queue database is passed in **src_db**, the transaction will be committed after successfully returning from process_event().
- On failure, the transaction will be rolled back, active batch will be reprocessed on next iteration.
- We need to call **tag_done()** for the processed events – otherwise they'll be scheduled for retry processing.

```
...
def process_event(self, src_db, ev):
    if ev.ev_type == 'welcome':
        self.log.info('Welcome %s!' % ev.ev_data)
    ev.tag_done()
...
```

Custom consumer: running it

- The consumer must be subscribed to the queue before events can be processed. In Skytools 2 this happens automatically.
- Skytools 3 requires explicit subscription, provides a `--register` switch for the purpose.
- For each processed batch the script logs the number of events and processing duration.

```
$ python welcome_consumer.py welcome_consumer.ini
2009-05-13 11:50:27,700 4318 INFO {count: 0, duration: 0.0322341918945}
2009-05-13 11:50:27,705 4318 INFO {count: 0, duration: 0.00421690940857}
2009-05-13 11:51:13,720 4318 INFO {count: 0, duration: 0.0121331214905}
```

Custom consumer: event processing

- So far our consumer hasn't seen any events.
- We'll use **pgq.insert_event()** stored procedure to feed some test events into the queue.
- In it's simplest form it takes queue name, event type and payload as arguments.

```
userdb# select pgq.insert_event('notifications', 'welcome', 'Custom Consumer');
userdb# select pgq.insert_event('notifications', 'irrelevant', 'Another Event');
...
2009-05-13 12:19:11,563 6884 INFO {count: 0, duration: 0.00770711898804}
2009-05-13 12:19:14,583 6884 INFO {count: 0, duration: 0.0210809707642}
2009-05-13 12:19:25,591 6884 INFO Welcome Custom Consumer!
2009-05-13 12:19:25,595 6884 INFO {count: 2, duration: 0.012876033783}
2009-05-13 12:19:28,608 6884 INFO {count: 0, duration: 0.0131230354309}
```

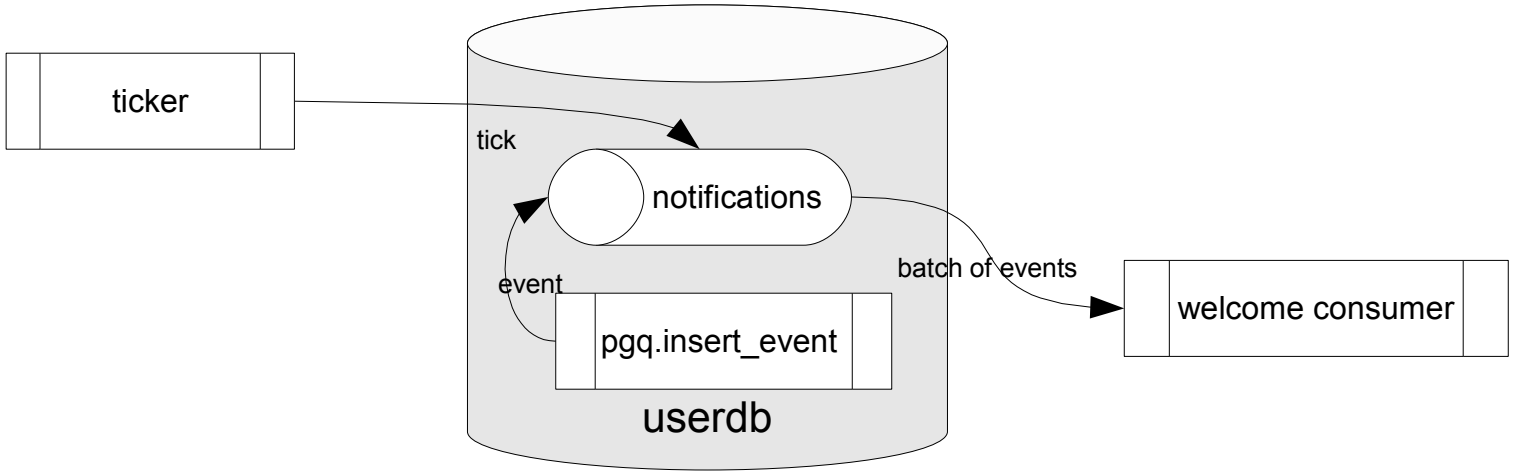
Custom consumer: event tracking

- Extend **RemoteConsumer** to add batch tracking support.
- pgq_ext must be installed on the target database.
- We'll use a simple counter application as an example.
- This actually implements distributed transactions.

```
class UserCounter(pgq.RemoteConsumer):
    def __init__(self, args):
        pgq.RemoteConsumer.__init__(self, "user_counter", "src_db", "dst_db", args)

    def process_remote_batch(self, db, batch_id, event_list, dst_db):
        for ev in event_list:
            ev.tag_done()
        cur = dst_db.cursor()
        cur.execute("update user_count set n = n + %s" % len(event_list))
```

Custom consumer: wrapup

- We have just implemented some simple PgQ consumers.
- Extend **Consumer** class for simple consumers. Advanced consumer base classes also available.
- **RemoteConsumer** and **SerialConsumer** – provide batch tracking, these are used for processing events in remote databases.
- **CascadedConsumer** adds cascading support (Skytools 3).

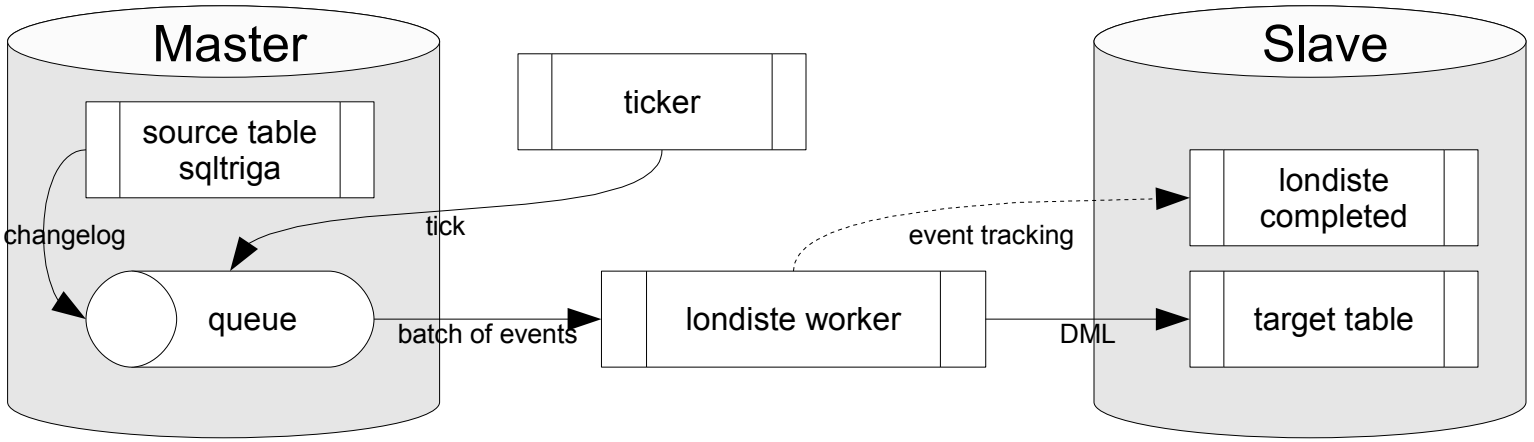


Replication toolset

- Replication tools built on top of PgQ:
  - **londiste** – replication
  - **table dispatcher** – archiving and partitioning
  - **queue mover** – copy events from one queue to another
  - **queue splitter** – split queues into queues
- Changelog triggers are used for capturing table data changes.
- Replication process is just another PgQ consumer.

Londiste

- Master/slave replication system implemented on top of PgQ.
- Uses sqltriga/logtriga to capture table changes on the master.
- PgQ consumer replays the captured events on the slave.
- One master can feed several slaves.
- Slaves can be masters to other slaves.



Londiste: setting up

- Prepare the configuration file – source and target databases, queue name.
- Run londiste **install** commands for provider and subscriber.
- Start the replication process – consume from master and replay on slave.
- The replay process can run anywhere, as long as it can connect to both databases.
- Add tables to replication.
- Initial copy is started, tables are usable on slave after it finishes.

```
[londiste]
job_name = l_u_to_f

provider_db = dbname=userdb
subscriber_db = dbname=foodb

pgq_queue_name = user_events

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

Londiste: demonstration

```
$ londiste.py londiste_userdb_to_foodb.ini provider install
2009-05-14 15:12:42,714 27716 INFO plpgsql is installed
2009-05-14 15:12:42,716 27716 INFO txid_current_snapshot is installed
2009-05-14 15:12:42,716 27716 INFO pgq is installed
2009-05-14 15:12:42,717 27716 INFO Installing londiste
2009-05-14 15:12:42,717 27716 INFO   Reading from /usr/local/share/skytools/londiste.sql
$ londiste.py londiste_userdb_to_foodb.ini subscriber install
2009-05-14 15:12:48,887 27728 INFO plpgsql is installed
2009-05-14 15:12:48,889 27728 INFO Installing londiste
2009-05-14 15:12:48,889 27728 INFO   Reading from /usr/local/share/skytools/londiste.sql
$ londiste.py londiste_userdb_to_foodb.ini replay -d
$ pg_dump -t users -s userdb | psql foodb
$ londiste.py londiste_userdb_to_foodb.ini provider add users
2009-05-14 15:15:19,730 27959 INFO Adding public.users
$ londiste.py londiste_userdb_to_foodb.ini subscriber add users
2009-05-14 15:16:29,845 28082 INFO Checking public.users
2009-05-14 15:16:29,888 28082 INFO Adding public.users
$ tail log/londiste_userdb_to_foodb.log
2009-05-14 15:44:47,293 28122 INFO {count: 0, ignored: 0, duration: 0.0210900306702}
2009-05-14 15:45:47,309 28122 INFO {count: 0, ignored: 0, duration: 0.0170979499817}
```

Table dispatcher

- Archiving and partitioning tool.
- Customizable table structure.
- Automatically creates partitions based on user specified conditions.
- Does not handle updates.

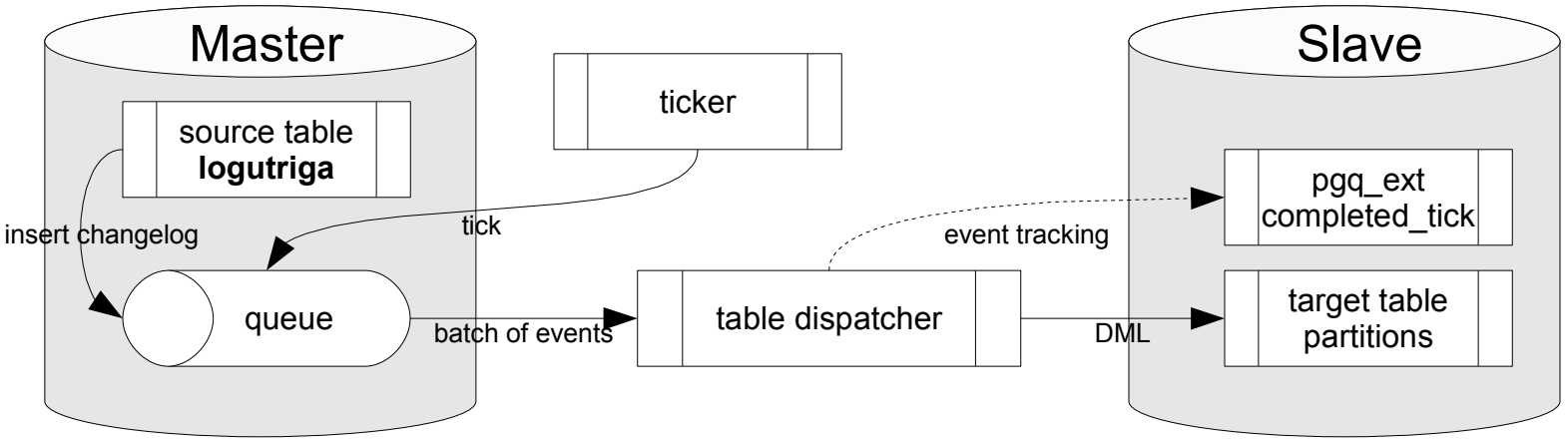


Table dispatcher: setting up

- Add logutriga to the source table, or reuse an existing trigger.
- Create the base table structure on target. Individual table partitions will be inherited from that.
- Prepare the configuration file which specifies the source queue, target table and partitioning options.

```
userdb# \d users
...
Triggers:
welcome_user_trg AFTER INSERT ON users
FOR EACH ROW EXECUTE PROCEDURE
  pgq.logutriga('notifications',
                'ignore=password')

archivedb# \d user_history
    Column    | Type
--------------+----------
 username     | text
 date_created | timestamp
```

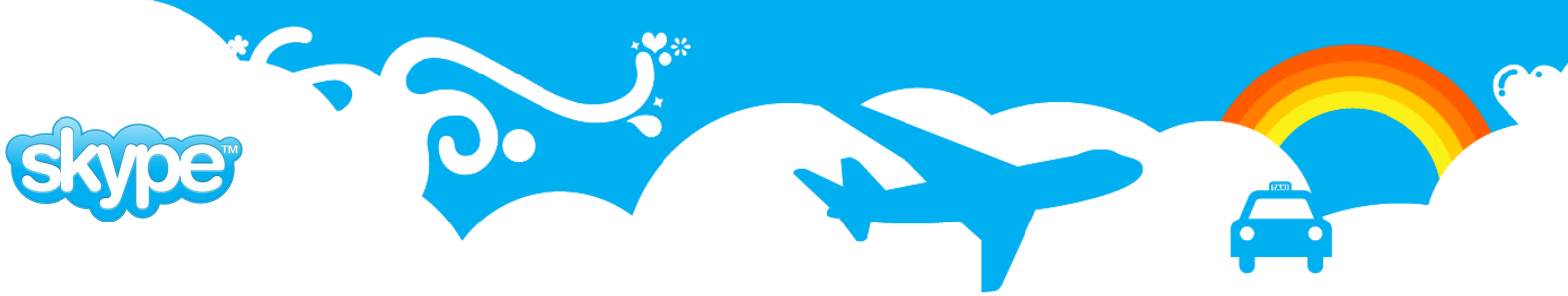
Table dispatcher: configuration

- **dest_table** – Base table for partitions.
- **fields** – select the columns to include, or use * for all.
- **part_field** – the column used for partitioning.
- **part_method** – either daily or monthly.
- **part_template** – SQL template for creating the partitions.

```
[table_dispatcher]
job_name          = user_archiver

src_db            = dbname=userdb
dst_db            = dbname=archivedb

pgq_queue_name    = notifications

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

dest_table = user_history
fields = username, date_created
part_field = date_created
part_method = daily

part_template     =
    create table _DEST_TABLE ()
      inherits (user_history);
    grant select on _DEST_TABLE
      to reporting;
```

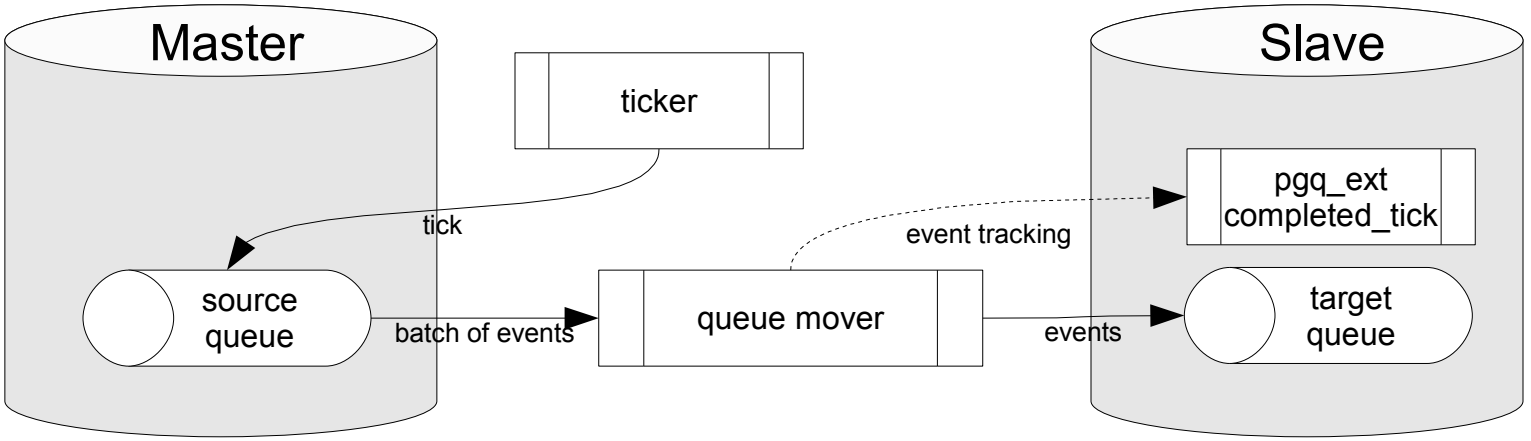
Table dispatcher: demonstration

```
$ table_dispatcher.py td_userdb_to_archivedb.ini
2009-05-18 11:05:14,370 10625 INFO {count: 0, duration: 0.0341429710388}
2009-05-18 11:05:14,379 10625 INFO {count: 1, duration: 0.00861620903015}
2009-05-18 11:05:15,394 10625 INFO {count: 0, duration: 0.0151319503784}
...

$ psql archivedb
archivedb# \dt user_history*
                List of relations
 Schema |          Name           | Type  |  Owner
--------+-------------------------+-------+---------
 public | user_history            | table | martinp
 public | user_history_2009_05_17 | table | martinp
 public | user_history_2009_05_18 | table | martinp
(3 rows)
```



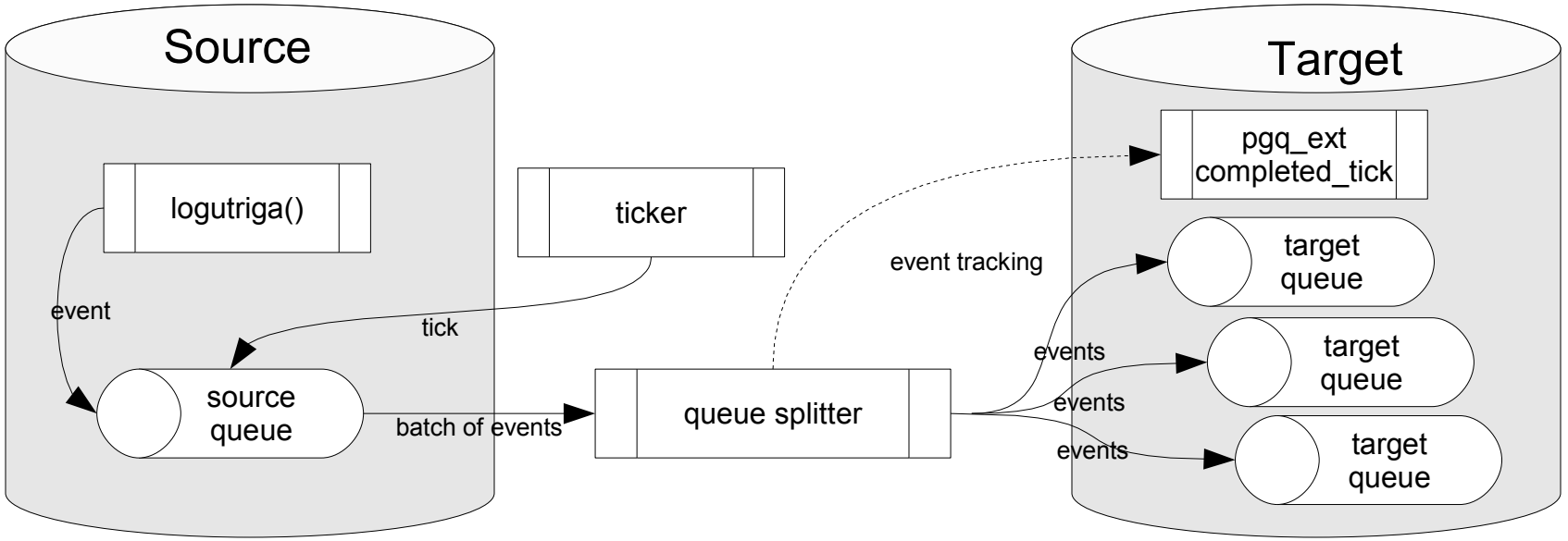
Queue mover

- Transports events from one queue to another.
- Useful for performing queue migrations.
- Consolidating queues from partitioned databases.



Queue splitter

- Transports events from one queue to several target queues.
- **ev_extra1** field is used to determine the target queue.
  logutriga automatically puts table name there.
- Useful for transporting events for batch processing.



Replication tools: wrapup

- Replication tools are ordinary PgQ consumers implemented with Skytools framework.
- On master database changelog events are enqueued through sqltriga/logutriga.
- On slave the DML statements are reconstructed and replayed.
- Event tracking is used to ensure that duplicate batches are not processed.

Skytools

Managing the Skytools environment

Skytools: getting and installing

- Prerequisites:
  - Python
  - psycopg2
- If you are lucky:
  - apt-get install skytools
  - http://yum.pgsqlrpms.org
- Building from source
  - Get it from http://pgfoundry.org/projects/skytools/
  - Needs PostgreSQL development headers and libraries
  - untar, configure, make install
- For the adventurous, Skytools3:
  - http://github.com/markokr/skytools-dev

Skytools2: installing from tarball

- Get the latest tarball from pgfoundry.
- Dependencies:
  - C compiler and make
  - PostgreSQL development headers and libraries
  - Python development package
- Makefile can also generate Debian packages.

```
$ tar zxf skytools-2.1.9.tar.gz
$ cd skytools-2.1.9
$ ./configure –prefix=/usr/local
$ make
$ sudo make install
... or ...
$ make deb83
```

Skytools3: building from Git

- Main repository is on github, clone from there or create your own fork.
- Adds additional dependencies:
  - asciidoc
  - xmlto
  - autoconf

```
$ git clone git://github.com/markokr/skytools-dev.git
$ cd skytools-dev
$ git submodule init
$ git submodule update
$ make boot
$ ./configure –prefix=/usr/local --with-asciidoc
$ make
```

Skytools: migrations and upgrades

- Upgrading a database with PgQ – pretty much straightforward, but has some additional steps.
- Migrating consumers from one database to another – to take some load off the primary server or to prepare for database migrations.
- Migrating whole databases.

Upgrading a PgQ database

1. pg_dump the database, shutdown database, stop tickers and consumers.
2. Run **pg_resetxlog -n** to determine the current epoch (extract from *Latest checkpoint's NextXID*).
3. Upgrade PostgreSQL binaries **AND** skytools modules.
4. Run **pg_resetxlog -e** to increase the epoch value. This is needed to enable pgq to correctly interpret stored txid values.

   Alternatively, if you are using the schema based txid (prior to 8.3), start the cluster and update the epoch in txid schema:

   ```
   UPDATE txid.epoch SET epoch = epoch + 1,
   last_value = (get_current_txid() & 4294967295);
   ```
5. Start the database, import dump file.
6. Start the ticker and consumers.

Skytools2: migrating consumers

1. Set up a **queue mover** to replicate the queue to new database, we will move the consumer subscriptions to the queue replica.
2. Stop the ticker on primary database - no more new batches will be prepared. After processing the pending batches, the consumers will stop at identical positions on the queue.
3. We can now subscribe the consumers to the replicated queue. Note that we need to reset the event tracking for the migrated consumers. Replication tools have `--reset` option for the purpose.
4. Start the ticker. Queue mover will continue queue replication, consumers on the new queue will continue where they left off.
5. If all is good, unregister the consumers from the old master.

Skytools3: migrating consumers

- Cascaded queues are kept identical across nodes - no need to set up explicit queue movers.
- Consumers that extend **CascadedConsumer** can be switched by simply running **change-provider** command of the set admin tool.
- No need to mess around with tickers and configuration files.
- The core features are complete, some development needed.

Skytools2: migrating databases

1. Create the database structure on the new host. Omit pgq, pgq_ext and londiste schemas – better to reinstall those later.
2. Replicate the database to the new host using londiste.
3. Create the queues and changelog triggers on the new database.
4. Pay special attention to applications that use stored procedures to enqueue events - maybe a queue mover is needed?
5. Migrate the consumers to the new database.
6. Make the primary database read-only and wait for londiste replication to catch up.
7. Redirect the applications to the new database.

Skytools3: migrating databases

- Cascading can be used to greatly simplify the migration process.
- The target database should be a branch node of the cascade.
- Migration is then performed by stopping the applications, running a londiste switchover command and redirecting the applications to the new database.
- Switchover will switch the roles of the root and branch, consumers needn't be aware that something changed.

Skytools: monitoring consumers

- We need to ensure that all our consumers are running happily.
- The best indicator for this is the consumer **lag** – if a consumer is lagging, it is not processing events adequately.
- **pgqadm.py status** command or **pgq.get_consumer_info()** SQL function can be used to determine the lag.
- In the example *welcome_consumer* hasn't processed anything in 6 days – probably not running at all.

```
select queue_name, consumer_name, lag, last_seen from pgq.get_consumer_info();

  queue_name   |      consumer_name |  lag      | last_seen
---------------+--------------------+-----------+----------
 notifications | user_counter       | 00:00:43  | 00:00:00
 notifications | welcome_consumer   | 6 days    | 6 days
```

Skytools: logging

- Skytools applications use Python logging module which can be used to forward the log and statistics messages to a central location.
- Just set *use_skylog = 1* in the configuration and configure the log handlers in **skylog.ini**
- Use syslog or write your own log handler. Examples are provided for sending the log over UDP or to a PostgreSQL database via stored procedure calls (see **skylog.py**).
- At Skype, we use the logging facilities to populate a configuration management database and feed error messages to Nagios.

Skytools: links

- PgFoundry project page
  http://pgfoundry.org/projects/skytools
- PgQ tutorial
  http://wiki.postgresql.org/wiki/PGQ_Tutorial
- Tool documentation
  http://skytools.projects.postgresql.org/doc/
- PHP consumer
  http://pgsql.tapoueh.org/pgq/pgq-php/
- Github repository for Skytools3
  http://github.com/markokr/skytools-dev/tree/master

Questions?