Choosing a Logical Replication System

David Christensen
david@endpoint.com
Introduction

❖ Physical Replication
❖ Logical Replication
❖ Slony vs Bucardo
Is there one right choice?

- No!
- If you were expecting an absolute answer to this, there’s still time to go to another talk.
- A matter of understanding the constraints and applying engineering decisions
  - Aw, man!
Why Replication?

- Replication used for many reasons:
  - Disaster recovery
  - High Availability / Failover
  - Read/write scaling
  - Backups
Logical vs Physical Replication

❖ The big question: Why do we need logical replication systems when PostgreSQL includes native replication?
Logical vs Physical Replication

- supports differing hardware/PostgreSQL versions
- replicate only specific databases or tables
- need writable clusters on the slaves
- not everyone running latest/greatest
Physical Replication
Physical Replication: Overview

- **Warm / Hot Standby**
- Refers to the physical files on-disk
- PostgreSQL's core replication facilities are built atop its recovery system
- As WAL records are processed on the standby it will make the same changes to the disk files as was made on the master
Physical Replication: History

- Warm Standby recovery - better than nothing but management very primitive
- 16MB WAL files, unless you were super-busy need Standby to stay up-to-date and not lose changes, things like `archive_timeout`
- Huge upgrades in Postgres 9.0: Hot Standby/Streaming Replication
- Now can query standbys, streaming means we’re never too far behind
- Cascading replication
- Postgres 9.4 brought big changes -> introduced logical decoding
  - We’ll cover this later
Physical Replication: Benefits

- Easy to setup/use
  - `pg_basebackup`
- Ideal for High Availability/Read Scaling
- Supports synchronous replication
- All changes (DDL, DML) automatically propagated
Physical Replication: Limitations

- Requires same PostgreSQL versions, hardware architecture, etc
- Standby servers are strictly read-only, no local changes at all
- Every change in the entire cluster replicated; can't replicate only a subset of tables or databases
Logical Replication in 9.4
Logical Replication: Pg 9.4

- Big changes in Pg 9.4 for LR
- Logical log decoding
  - Works via parsing the WAL stream and extracting information about modified tuples
- New `postgresql.conf` settings:
  - `wal_level = logical`
  - `max_replication_slots`
Replication Slots

- In order to ensure WAL stream changes are consumed once, 9.4 introduced the concept of replication slots
- Replication slots keep track of a WAL location
- Changes are available in order as committed
- Slots have a unique identifier across the database cluster
- Enable consumers to read WAL events
Replication Slots

- Slots are created/managed via the replication protocol
- Can more easily interface with `pg_recvlogical`
- Also SQL functions
Replication Slots: `pg_recvlogical`

- **Create slots:**
  
  ```
  pg_recvlogical --slot=foo -d <database> --create-slot
  ```

- **Drop slots:**
  
  ```
  pg_recvlogical --slot=foo -d <database> --drop-slot
  ```

- **Start reading events:**
  
  ```
  pg_recvlogical --slot=foo -d <database> --start
  ```
Replication Slots: SQL functions

- Create with:
  `pg_create_logical_replication_slot()`

- Drop with:
  `pg_drop_replication_slot()`

- Consume changes:
  `pg_logical_slot_get_changes()`

- Peek at changes without consuming them:
  `pg_logical_slot.peek_changes()`
Output Plugin

- WAL has to be decoded; this is the job of an Output Plugin
- The Output is selected when you create a logical replication slot
- In `pg_recvlogical`, use the `--plugin` option
- Can be used to write your own methods for translating the data
The future?

- That’s great for clients on 9.4 and future versions
- Clients still need things done yesterday
- Don’t want/can’t upgrade
- Even on 9.4 this isn’t an entire solution
Logical Replication for Now()
Logical Replication: Overview

- Refers to capturing/replicating the "logical" changes made to the database
- Propagate any changes (inserts, updates, deletes) to replica databases
- Generally work via triggers on the tables to record changes and daemons to propagate changes to the intended nodes
- Inherently asynchronous in nature
Logical Replication: Overview

- Traditionally, this has been done using trigger-based systems
- Slony, Bucardo, Londiste, etc
Logical Replication: Benefits

- Can run on different architectures, PostgreSQL versions
- Can replicate only a subset of database changes; specific tables, sequences, etc
- Database clusters are independently writable
Logical Replication: Limitations

- Need to specify replication explicitly
  - itemize all replicated tables
  - new tables not automatically added
  - requires more planning (aw, man...)
- Require special handling for DDL/structural changes
- Requires daemons/external processes to monitor/manage
- More moving parts = more things that might break
Specifics
Specific LR systems

- We will look at:
  - Slony
  - Bucardo
  - (briefly) BDR
# Feature Summary

<table>
<thead>
<tr>
<th>Feature/Topologies</th>
<th>Slony</th>
<th>Bucardo</th>
<th>BDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master/Slave</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Multimaster</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Custom conflict resolution</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Custom data transforms/aggregations</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Multiple replication groups</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Non-Pg target DB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cascaded/custom topologies</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Slony

- System designed explicitly for single master, multiple slaves
Anatomy of a Slony System

Node

Set
Table  Table

Set
Table  Table

Node

Set
Table  Table

Set
Table  Table
Anatomy of a Slony System

Diagram:

- **Slonik**: Central node
- **Slon**: Nodes connected to Slonik
- **Node**: Components of the system
  - **Set**: Collections of data
  - **Table**: Data tables

Diagram shows the relationships and connections between the Slon nodes and the Slonik central node, illustrating the anatomy of the Slony system.
Slony: Nomenclature

- Node
- Table
- Set
- Path
- Subscribers
- Origins
Slony Fundamentals

- At its core, Slony is a distributed, serialized event system.
- Slony's internals stored in a `_slony` schema in each node's database.
- This schema contains all support functions, triggers, etc needed for the database-level part of Slony.
- `slon` daemons run for each node, listening for and processing events.
- Interface to `slon` usually done via `slonik`, a tool with its own DSL for creating the necessary events and distributing across the cluster.
Slony: Architecture

- Each involved database node has a slon daemon
- Custom schema in the database to hold metadata/track events that are logged
- Replicated tables are defined/added to sets
- Database triggers used to log changes (on origin) or prevent access (on replica node)
- Sets are the base unit of what is replicated
Each node has a **slon** daemon, which listens for events and handles them.

- Can run anywhere, but generally runs on the same machine as the database.
A Slony node corresponds to a specific logical database instance.

Each node in the cluster is an independent source of events.

Each node has its own unique ID and queue of events, stored in the `sl_node` and `sl_event` table.

Different types of events corresponding to different actions on the cluster.

Any node additions, changes, etc are each their own event type.
Slony: Paths

- Stores connection information for any two nodes in the cluster
- Connections only made between nodes if there is a defined listener
- Generally best to define paths for all combinations of nodes
- Stored in `sl_path`
- `slonik: CREATE PATH`
Slony: Sets

- Slony sets are groups of tables which are replicated together
- Each set has an "origin node" which is the node id "master" for the set
  - only node on which you can make changes
- Sets of tables replicated together are contained in the `sl_set` table
- Replicated tables are contained in the `sl_table` table
- Each tracked table is in only 1 set
- slonik: `CREATE SET, SET ADD TABLE`
Slony: Subscriptions

- Subscriptions are basically a mapping of sets to nodes.
- If a node is subscribed to a set, it will receive the data changes related to this set.
- Must be a listener to be a subscriber.
With more complex topologies, a non-origin node can still be a provider for other nodes.

This is considered a "cascaded" subscription.

When subscribing a non-origin node as a provider, must be configured to forward log data.

e.g.: `SUBSCRIBE SET (set id = 3, origin = 1, forward = yes)`
Slony: Triggers

- When a table is added to a replication set, Slony installs the necessary log/deny triggers to the table.
- When a node is subscribed to a set, Slony will truncate the table on the receiver side and copy the data as part of its event processing.
- This ensures the data is guaranteed identical at each step.
Slony: Tracking Changes

- Slony takes "ownership" of tracked tables:
  - on an "origin" node:
    - adds log triggers
    - log triggers are triggers which fire after any DML change on tracked tables.
    - stores information about the row affected (PK), the tuple data, and the current snapshot.
  - on a "subscriber" node:
    - deny access triggers
    - prevent any change to the table data, accidental or otherwise
    - ensures slony is the only process able to change the data
Slony: Applying Changes

- The node's `slon` daemon listens for `SYNC` event notifications
- Gathers event data from the provider node since last applied `SYNC` event
- Applies the data from the `sl_log_*` tables
- Confirms the event on the remote node
- Process is inherently serial
Bucardo

- A stand-alone replication system
- Push changes from Postgres to other databases
- Trigger-based, asynchronous
- Both Master/Slave and Multi-master
Anatomy of a Bucardo System
Anatomy of a Bucardo System
Bucardo: History

- Started at backcountry.com in 2002, using Postgres 7.2
- Released publicly in 2007
- Bucardo 5 introduced true multi-master, lots of improvements over previous versions
Bucardo: Strengths

- Low requirements (Pg 8.3, plperlu, DBD::Pg)
- No changes to Postgres or its configuration
- Only 1 daemon, can be run anywhere as long as it can connect to all DBs
- Fast, handles poor network connectivity
- Good command-line monitoring
- Easy install/setup
Bucardo: Strengths

- Targets (slaves) are not locked
- Configuration in the database
- Customcode (conflict handlers, data transforms)
- Multiple target types (Oracle, MySQL, Mongo, etc)
- In case of multiple modifications to the same table row, replicates only the final state for replicated rows
- Good for heterogenous environments
Bucardo: Limitations

- No automatic handling of DDL
- No built-in failover
- Requires PKs
- Targets (slaves) not locked
  - (hey, wasn’t that just a strength?)
- Replicates only final state of affected rows
  - (hey, wasn’t that just a strength?)
Bucardo: Nomenclature

- Goat - Replicated object (table/sequence)
- Herd - Named group of Goats (= “set” in Slony)
- Sync - Specification for what kind of data is replication and how
Bucardo: Architecture

- Master Control Process (MCP)
  - runs and monitors the overall process
- Controller Processes (CTL)
  - spawned by MCP, responsible for handling/monitoring KID processes
- KID processes
  - handle the actual hard work of replication
- I can’t get enough of goat puns
Bucardo CLI

- The main user interface to Bucardo.
- Performs all control-related actions
- Sends commands to the running "bucardo" daemon
Bucardo CLI, cont

- $ bucardo install
- $ bucardo add ...
- $ bucardo status
- $ bucardo kick
- $ bucardo help
Bucardo Configuration

- Configuration stored in a special database “bucardo”
- Bucardo keeps track of dbs, tables, herds, syncs, settings, etc in its own database.
- Database owned by the "bucardo" superuser.
- created for you via bucardo install.
- The bucardo tool uses the "bucardo" database to store state for the Bucardo daemon and information about the replication at hand.
Bucardo: Tracking Changes

- Whenever a change is made on a table (I,U,D), the PK of the table is logged in a special table in the **bucardo** schema (known as the delta table)
- Depending on the sync settings, this will also trigger a notification that data has changed
- Bucardo will get the notification and process/apply the changes
When Bucardo runs syncs (either automatically or manually kicked) it:

- checks the delta tables for all affected PKs
- then deletes them on the target
- copies current row (if any in case of deletes)
Bucardo: Multi-master

- Bucardo 5 has true multi-master, using round-robin syncing approach
- Each master in the cluster logs changed rows (by PK) into a custom table
Bucardo: Multi-master Conflicts

- When there are conflicts, Bucardo runs custom code handlers to determine how to resolve the conflicts.
- When using master-master, have the potential to have both nodes modify the same row = conflict.
- Conflicts = BAD; we need to choose which row is "right" in this situation.
- Bucardo has some standard conflict resolution methods: source, target, random, or latest.
- Can always write your own custom conflict resolution
BDR

- The “New Kid on the Block”
- Bi-Directional Replication
- Works via the Logical Log Streaming features of Pg
- Not covering extensively in this talk…but seems very cool and powerful
BDR: Strengths

- Doesn’t use triggers, so no write magnification
- Handles many DDL modifications automatically throughout the cluster
- New tables are automatically replicated by default
BDR: Limitations

- Requires a custom/patched Pg install
  - will likely go away in future versions, trying to get into core
- Some DDL/Pg features are restricted
- Only works on Pg >= 9.4
- Limited control on cluster topologies
Side-by-side Overview

- Comparing how to manage clusters across Slony and Bucardo
Constructing the Cluster: Slony

- cluster defined via **slonik** scripts
- **INIT CLUSTER**
- each node has its own immutable id, defined at node creation time
- **CREATE NODE**
- define pathways between nodes via **CREATE PATH**
- define topology via **CREATE LISTEN**
- cluster configuration is stored in the **_slony** schema in each database
Constructing the Cluster: Bucardo

- cluster defined using `bucardo` tool
- `bucardo install`
- `bucardo add db <dbname> [options]`
- `bucardo add dbgroup [db db]`
Specifying Replication Groups: Slony

- define sets containing tables to be replicated
- `CREATE SET (id = ...)`
- `SET ADD TABLE (...)`
Specifying Replication Groups:

Bucardo

- define "herds" via **bucardo**
- **bucardo add table** `<tablename>` `db=<dbname>`
- **bucardo add herd** `<name>` `[goat goat]`
- **bucardo add sync** `<name>` `source=<herdname>` `type=<synctype>` `target=`
to add new tables to the same set:

- **CREATE SET** with temporary set.
- **SUBSCRIBE SET** to match the subscriptions for original set
- **MERGE SET** to join the sets together
Modifying Replication:

Bucardo

- modify the sync definition using **bucardo**
- validate the sync via **bucardo**
DML logging differences:

Slony

- changes are logged per-transaction
- logs the data for the whole changed tuple as it exists
- multiple DML changes to the same row end up with multiple entries in the log table with the latest data
DML logging differences:
Bucardo

- only PK fields are logged
- doesn't differentiate between type of change (I,U,D)
- row values at time of event irrelevant
- when syncing, deletes the noted PK on the target, then inserts the current value of the row (if it still exists)
- multiple changes to the same record don't matter and aren't logged; final state only
DDL changes:

Slony

- create a SQL script to make the DDL changes
- `EXECUTE SCRIPT` to apply changes across the cluster
- events generated before/after will have the necessary data in the log tables when the other nodes replay the events in order
- ensures that the table will be in the correct state to run the DDL
DDL changes:
Bucardo

- stop replication
- perform DDL changes on all affected databases
- restart replication
- alternately, with simple column additions, you can apply the change first to the slaves then the master without affecting replication
- generally easier to just stop/start the replication
Failover:
Slony

- **MOVE SET** preferred method
- ensures cluster is sane
- if missing node comes back, it will properly hand-off control
- **FAILOVER** forcibly removes node from cluster
- use only when the node is likely to be unrecoverable
Failover:
Bucardo

- no formal failover
- available nodes still collect deltas
- if/when(?!?) other node comes back, deltas are replayed
- can always adjust the syncs to remove the node from the sync definition
Monitoring: Slony

- each node has `sl_status` view
- show status of other nodes confirmation of its generated events.
- displays number of events lagged, time
Monitoring: Bucardo

- **bucardo status**
- shows all syncs, lag, statistics
Sync Behavior:
Slony

- SYNCs happen as quickly as possible
In Bucardo, syncs can be defined with multiple behaviors:

- immediate
- timed
- manual
Sync Behavior:
Bucardo

- Immediate syncs
- the general/expected way replication works
Sync Behavior:
Bucardo

- Timed syncs:
  - useful for data that changes frequently in the source but does not need to be up-to-date
  - can be handy with custom select behavior
Sync Behavior: Bucardo

- Manual syncs:
  - Gives explicit control for specific syncs
Case Studies

- Choosing the Logical Replication system to use
  - No one right answer
  - Different scenarios cater to different strengths
  - Matter of picking the Right Tool For The Job™
Case Study 1
Minimal Downtime Upgrade
Minimal Downtime Upgrade

We had:

[Diagram showing a master with three slaves connected, labeled " Pg 8.2 Slony "]
Minimal Downtime Upgrade

We wanted:
Minimal Downtime Upgrade

We couldn’t upgrade Slony
The current version didn’t support Pg 9

Plus all those pesky:

app app app app app app app app
Minimal Downtime Upgrade

- Specific client issue:
- Upgrade Postgres 8.2/Slony cluster to Postgres 9 HS/SR
- Wanted migration with no application downtime (who doesn’t?)
- Couldn’t disrupt/upgrade the Slony cluster
- Very simple schema, but large table, lots of changes
Minimal Downtime Upgrade

❖ Even though we couldn’t modify the Slony cluster, we were able to use Bucardo in conjunction to accomplish this.

❖ Using Bucardo, total application downtime was measured in minutes, regardless of the size of the database.
Minimal Downtime Upgrade

- Created the new Pg cluster in the new datacenter.
- Setup HS/SR, verified this was working.
- Configure/test remote access.
Minimal Downtime Upgrade

- Installed Bucardo on the new cluster
- Dumped the schema/structure for the database, users, etc
- `pg_dumpall --global` and `pg_dump --schema-only`
- Defined Bucardo configuration, dbs, herds, sync
- With Bucardo setup, we were now capturing all deltas, so dump/load the data for an initial data load
Minimal Downtime Upgrade

- `pg_dump` took a while, changes ongoing, but still had delta triggers capturing everything
- Once the shiny new cluster had the base dump loaded, kicked Bucardo to start the replication
- Now we only had to replicate the rows that had been modified, a much smaller set
Minimal Downtime Upgrade

Bucardo
Minimal Downtime Upgrade

- Once that caught up, we could stop the application, let it finish, then re-point/test
- Slony cluster on old database still running as a backup, so we could fall back as needed
- Bucardo replicated the same tables/changes to the new HS/SR cluster
- Bucardo was then safely removed
Case Study 2
Data Center Migration
Datacenter Migration

- Realizing/fully utilizing the capabilities of an existing tool
- Client Datacenter Migration
Datacenter Migration

- Client had a large database with a Slony cluster and needed to move datacenters
- Datacenters were linked with very slow VPN
- Needed to migrate multiple nodes at the same time
- Test, potentially rollback
- Of course, 0-downtime
Datacenter Migration

- Data set was large: subscription of a new node took > 14 hours
- VPN was flaky and sometimes dropped connections in the middle of subscription
- Needed to recreate 4 node cluster in the new datacenter
- Everything had to work
Datacenter Migration

- Slony’s cascaded replication and multiple sets to the rescue!
Split sets:

- VPN issues always happened on 1 specific table (the largest one, of course)
- 2 really huge tables; related via FK
- Split these tables off into their own replication set
- Everything else subscribed fine
Datacenter Migration

Datacenter 1
- Master
- Slave
- Slave
- Slave

Datacenter 2
- Slave
- Slave
- Slave

VPN
Datacenter Migration

- Added all the new nodes to the Slony cluster without subscriptions yet
- Targeted the intended new master as the subscriber across the weak VPN link
- After that node subscribed, subscribed the new slave nodes directly from that node
- All while leaving the existing cluster in-place and running
Datacenter Migration

- After testing, used `MOVE SET` to switch the origin node across the VPN
- Dropped the old nodes from the cluster
Questions?
Additional Information

- **Slony:**
  #slony

- **Bucardo:**
  [http://bucardo.org/](http://bucardo.org/)
  #bucardo

- **BDR:**
  [http://bdr-project.org/](http://bdr-project.org/)
Thanks!