PostgreSQL Notification Enhancements

Presenter: Andrew Dunstan

Principal Consultant, Dunslane Consulting LLC

adunstan@postgresql.org

Where we are today

- A listener subscribes or unsubscribes to notifications with LISTEN and UNLISTEN
- A notifier creates events with NOTIFY
- Both must be clients connected to the same database
- PostgresSQL handles the mechanics

What is it good for?

- Many things!
  - e.g. Job scheduling/coordinating
- Lots easier and more efficient than other methods
  - Especially for one to many notifications
- Can be called by Rules and Triggers

Current implementation

- pg_listener table:

```
    Column     |   Type    | Modifiers
---------------+-----------+-----------
 relname       | name      | not null
 listenerpid   | integer   | not null
 notification  | integer   | not null
```

- *relname = event name (for historical reasons)*

Mechanics – Listening / Unlistening

- LISTEN ⇒ new row (eventname, mypid, 0)
- UNLISTEN ⇒ delete row

Mechanics - Notifying

- NOTIFY ⇒ update pg_listener
  set notifier = mypid
  where relname = eventname
- NOTIFY ⇒ signal relevant backends
  - If I am listening for this event, don't do this but forward event to my frontend directly

Mechanics - Collection

- For each row where mypid = listener, forward event to my frontend and set notifier back to 0.

Mechanics – Transactions

- NOTIFY / LISTEN / UNLISTEN actions only applied on commit
  - held in a backend local queue until then
- Collection happens in its own transaction (from users POV between transactions)

Limitations

- Events can be lost!
  - If the same event occurs between two calls on collection by a backend, it will only see one of them
  - Because pg_listener has one row per (event, listener) pair.
- No provision for accompanying message

Payloads

- A message to accompany an event
  - e.g. Event = “Batch Finished”, message = batch_id
- Already provision in V3 protocol for it
- Will make system design easier
- Reduce number of events listened for

And it looks like this

- NOTIFY stage1 'batch 57';
  - Omitting the message is equivalent to an empty message
  - No breaking existing applications

New implementation scheme

- Based on existing shared cache invalidation scheme
- Keep an event queue in shared memory
- Every event will be in the queue
  - Once! (NOT once per listener)
- No listener registration needed
- Each listener has its own queue pointer

What do we need in shared memory?

- Global queue head and tail pointers
- One queue tail pointer per backend
- Queue buffer – size configurable
  - Entries contain database oid + length + event name + payload + alignment padding
  - Conceptually circular

How much buffer space

- We hope enough not to block
- Average entry size ×
  Maximum event burst rate ×
  Maximum time waiting for collection
  - Listeners should not run long running transactions, although notifiers can

Example

- Average entry size = 150
- Maximum event burst rate = 1event per second
- Maximum transaction time by listener = 1 hour
  - Buffer needed = 540,000 bytes

What should be the default?

- Those rates are probably a bit extreme
  - 1 event per second is high
  - 1 hour wait by a listener is very high
- PostgreSQL tends to be conservative, especially about shared memory
- I am thinking of having a default around 100kB.

Adding an entry

- If there is room between head and tail, just add it and adjust head
- If not, move tail forward to least of listener tails, and if there is now enough room add it and adjust head
- If not, signal listeners and sleep for a short period before retrying

Collecting entries

- Check regularly – call from CHECK_FOR_INTERRUPTS()
- For each entry from our tail to head, if db oid matches our db and event name is in our event list, collect entry
- Set our tail pointer to head

Locking

- Need 2 locks - “head” lock and “tail” lock.
  - Adding entry requires exclusive “head” lock
  - Adjusting tail requires exclusive “tail” lock
  - Collecting entries requires “shared” tail lock.
    - Because collecting entry doesn't change global tail pointer
- Notifiers block each other, sometimes block listeners. Listeners don't block each other.

Other functionality

- Since there is no pg_listener any more, we need a function to tell us what events we're listening on:

  pg_listened_events(out event name)
  returns setof record

- We can't have a function that tell us the events every listener is listing for, as there is no longer a central list of those.

Summary: Benefits + Risks

- Guaranteed delivery of all events, in order
- Payload messages
- Efficiency gain – should be much faster
- Potential downside: blocked notifiers if buffer is too small or listeners are too slow