Newt DB Documentation

Release 0.1

Jim Fulton

Jun 29, 2017

Contents

Contents:

CHAPTER 1

Newt DB, the amphibious database

- In Python: enjoy the ease of working with your data as ordinary objects in memory.

  Data are moved in and out of memory as needed, so databases can be as large as needed.
- In Postgres: index and search your data using PostgreSQL, from within your application and externally.
- Within your application, search results may be returned as application objects, or as data, depending on your needs.
- Transactional and built on mature technology for reliability.

Learn more:

- *Getting started*
- *How it works*
- *Fine print*
- *Topics*
- *Reference*

CHAPTER 2

Getting started with Newt DB

Contents

You'll need a Postgres Database server. You can install one yourself or you can use a hosted Postgres server. You'll need Postgres 9.5 or later.

Next, install newt.db:

```
pip install newt.db
```

You'll eventually want to create a dedicated database and database user for Newt's use, but if you've installed Postgres locally, you can just use the default database.

From Python, to get started:

```
>>> import newt.db
>>> connection = newt.db.connection('')
```

In this example, we've asked newt to connect to the default Postgres database. You can also supply a *connection string*.

The connection has a root object:

```
>>> connection.root
<root: >
```

This is the starting point for adding objects to the database.

To add data, we simply add objects to the root, directly:

```
>>> connection.root.first = newt.db.Object(name='My first object')
```

Or indirectly, as a subobject:

```
>>> connection.root.first.child = newt.db.Object(name='First child')
```

When we're ready to save our data, we need to tell Newt to commit the changes:

```
>>> connection.commit()
```

Or, if we decide we made a mistake, we can abort any changes made since the last commit:

```
>>> connection.abort()
```

Above, we used the `newt.db.Object` class to create new objects. This class creates objects that behave a little bit like JavaScript objects. They're just generic containers for properties. They're handy for playing, and when you have a little data to store and you don't want to bother making a custom class.

Normally, you'd create application-specific objects by subclassing `Persistent`[1]:

```
class Task(newt.db.Persistent):

    assigned = None

    def __init__(self, title, description):
        self.title = title
        self.description = description

    def assign(self, user):
        self.assigned = user
```

The `Persistent` base class helps track object changes. When we modify an object, by setting an attribute, the object is marked as changed, so that Newt will write it to the database when your application commits changes.

With a class like the one above, we can add tasks to the database:

```
>>> connection.root.task = Task("First task", "Explain collections")
>>> connection.commit()
```

Collections

Having all objects in the root doesn't provide much organization. It's better to create container objects. For example, we can create a task list:

[1] Newt makes `Persistent` available as an attribute, but it's an alias for `persistent.Persistent`. In fact many of the classes provided by Newt are just aliases.

```
class TaskList(newt.db.Persistent):

  def __init__(self):
      self.tasks = newt.db.List()

  def add(self, task):
      self.tasks.append(task)
```

Then when setting up our database, we'd do something like:

```
>>> connection.root.tasks = TaskList()
>>> connection.commit()
```

In the `TaskList` class, we using a `List` object. This is similar to a Python list, except that, like the `Persistent` base class, it tracks changes so they're saved when your application commits changes.

Rather than supporting a single task list, we could create a list container, perhaps organized by list name:

```
class TaskLists(newt.db.Persistent):

    def __init__(self):
        self.lists = newt.db.BTree()

    def add(self, name, list):
        if name in self.lists:
            raise KeyError("There's already a list named", name)
        self.lists[name] = list

    def __getitem__(self, name):
        return self.lists[name]
```

Here, we used a `BTree` as the basis of our container. BTrees are mapping objects that keep data sorted on their keys.

BTrees handle very large collections well, because, when they get large, they spread their data over multiple database records, reducing the amount of data read and written and allowing collections that would be too large to keep in memory at once.

With this, building up the database could look like:

```
>>> connection.root.lists = TaskLists()
>>> connection.root.lists.add('docs', TaskList())
>>> connection.root.lists['docs'].add(
...     Task("First task", "Explain collections"))
>>> connection.commit()
```

Notice that the database is hierarchical. We access different parts of the database by traversing from object to object.

Searching

Newt leverages PostgreSQL's powerful index and search capabilities. The simplest way to search is with a connection's `where` method:

```
>>> tasks = connection.where("""state @> '{"title": "First task"}'""")
```

The search above used a Postgres JSON `@>` operator that tests whether its right side appears in its left side. This sort of search is indexed automatically by newt. You can also use the `search` method:

```
>>> tasks = connection.search("""
...     select * from newt where state @> '{"title": "First task"}'
...     """)
```

When using `search`, you can compose any SQL you wish, but the result must contain columns `zoid` and `ghost_pickle`. When you first use a database with Newt, it creates a number of tables, including `newt`:

```
    Table "public.newt"
    Column     |  Type   | Modifiers
---------------+---------+-----------
 zoid          | bigint  | not null
 class_name    | text    |
 ghost_pickle  | bytea   |
 state         | jsonb   |
Indexes:
    "newt_pkey" PRIMARY KEY, btree (zoid)
    "newt_json_idx" gin (state)
```

The `zoid` column is the database primary key. Every persistent object in Newt has a unique zoid. The `ghost_pickle` pickle contains minimal information to, along with `zoid` create newt objects. The `class_name` column contains object's class name, which can be useful for search. The `state` column contains a JSON representation of object state suitable for searching and access from other applications.

You can use PostgreSQL to define more sophisticated or application-specific indexes, as needed.

Newt has a built-in helper for defining full-text indexes on your data:

```
>>> connection.create_text_index('mytext', ['title', 'description', 'text'])
```

This creates a PL/pgSQL text-extraction function named `mytext` and uses it to create a text index. With the index in place, you can search it like this:

```
>>> tasks = connection.where("mytext(state) @@ 'explain'")
```

The example above finds all of the objects containing the word "explain" in their title, description, or text. We've assumed that these are tasks. If we wanted to make sure, we could add a "class" restriction:

```
>>> tasks = connection.where(
...   "mytext(state) @@ 'explain' and class_name = 'newt.demo.Task'")
```

Rather than creating an index directly, we can ask Newt to just return the PostgreSQL code to create them:

```
>>> sql = connection.create_text_index_sql(
...           'mytext', ['title', 'description', 'text'])
```

You can customize the returned code or just view it to see how it works.

Query errors

If you enter an invalid query and then retry, you may get an error like: "InternalError: current transaction is aborted, commands ignored until end of transaction block". If this happens, you'll need to abort the current transaction:

```
>>> connection.abort()
```

After that, you should be able to query again.

You can only search committed data

If you change objects, you won't see the changes in search results until changes are committed, because data aren't written to Postgres until the transaction is committed.

Raw queries

You can query for raw data, rather than objects using the `query_data` method. For example, to get a count of the various classes in your database, you could use:

```
>>> counts = connection.query_data("""
...     select class_name, count(*)
...     from newt
...     group by class_name
...     order by class_name
...     """)
```

Learning more

To learn more about Newt, see the Newt topics and the Newt *topics* and *reference*.

CHAPTER 3

Newt DB Architecture

Newt builds on ZODB and Postgresql. Both of these are mature open-source projects with years of production experience.

ZODB is an object-oriented database for Python. It provides transparent object persistence. ZODB has a pluggable storage layer and Newt leverages RelStorage to store data in Postgres.

Newt adds conversion of data from the native serialization used by ZODB to JSON, stored in a Postgres JSONB column. The JSON data supplements the native data to support indexing, search, and access from non-Python application. Because the JSON format is lossy, compared to the native format, the native format is still used for loading objects from the database. For this reason, the JSON data are read-only.

Newt adds a search API for searching the Postgres JSON data and returning persistent objects. It also provides a convenience API for raw data searches.

Finally, Newt adds additional convenience APIs to more directly support it's intended audience. These are intended to augment but not hide ZODB and RelStorage. Some of these are just aliases. It will be possible to integrate Newt with existing ZODB applications.

CHAPTER 4

Fine print – things you should know

Up to this point, we've emphasized how Newt DB leverages ZODB and Postgres to give you the best of both worlds. We've given some examples showing how easy working with an object-oriented database can be, and how Postgres can allow powerful queries to be easily expressed. Like anything, however, any database has some topics that have to be mastered to get full advantage and avoid pitfalls.

Contents

Highly, but not completely transparent object persistence

Newt and ZODB try to make accessing and updating objects as simple and natural as working with objects in memory. This is done in two ways:

1. When an object is accessed or modified, data are loaded automatically and saved if a transaction is committed.

   The database keeps track of objects that have been marked as changed. If a transaction is committed, changed objects are saved to Postgres. If a transaction is aborted, then changed objects' states are discarded and will be reloaded with current state when they're accessed next.

2. Object accesses and changes are detected by observing attribute access. This works very well for accesses, but can miss updates. For example, consider this class:

```
class Tasks(newt.db.Persistent):
```

```
    def __init__(self):
        self._data = set()

    def add(self, task):
        self._data.add(task)
```

In this example, the `add` method updates the object by updating a subobject. It doesn't set an attribute, and the change isn't detected automatically. There are a number of ways we can fix this, for example by explicitly marking the object as changed:

```
def add(self, task):
    self._data.add(task)
    self._p_changed = True
```

To learn more about writing persistent objects, see:

http://www.zodb.org/en/latest/guide/writing-persistent-objects.html

Learn about indexing and querying PostgreSQL

By default, Newt creates a JSON index on your data. Read about support for querying and indexing JSON data here:

https://www.postgresql.org/docs/current/static/datatype-json.html

Postgres can index expressions, not just column values. This can provide a lot of power. For example, Newt provides helper functions for setting up full-text indexes. These helpers generate text extraction functions and then define indexes on them. For example, if we ask for SQL statements to index title fields:

```
>>> import newt.db.search
>>> print(newt.db.search.create_text_index_sql('title_text', 'title'))
create or replace function title_text(state jsonb) returns tsvector as $$
declare
  text text;
  result tsvector;
begin
  if state is null then return null; end if;

  text = coalesce(state ->> 'title', '');
  result := to_tsvector(text);

  return result;
end
$$ language plpgsql immutable;

create index newt_title_text_idx on newt using gin (title_text(state));
```

A PL/pgSQL function is generated that extracts the title from the JSON. Then an index is created using the function. To learn more about full-text search in Postgres, see:

```
https://www.postgresql.org/docs/current/static/textsearch.html
```

To search the index generated in the example above, you use the function as well:

```
select * from newt where title_text(state) @@ 'green'
```

In this query, the function, `title_text(state)` isn't evaluated but is instead used to match the search term against the index[1].

Indexing expressions allows a lot of power, especially when working with JSON data.

When designing queries for your application, you'll want to experiment and learn how to use the Postgres EXPLAIN command.

Postgres is not (really) object oriented

Using Newt DB, search and indexing use Postgres. The data to be indexed have to be in the object state. You can't call object methods to get data to be indexed. You can write database functions to extract data and these functions can branch based on object class.

Transactions

Transactions are a core feature of Newt, ZODB and Postgres. Transactions are extremely important for implementing reliable applications. At a high-level, transactions provide:

Atomicity Data modified by a transaction is saved in its entirety or not at all. This makes error handling much easier. If an error occurs in your application, the transaction is rolled back and no changes are saved. Without atomicity, if there was an error, you the programmer would be responsible for rolling back the changes, which is difficult and likely to produce inconsistent data.

Isolation Transactions provide isolation between concurrently running programs. You as a programmer don't need to worry about concurrency control yourself.

In the examples in *Getting started*, a simple form of transaction interaction was used, which is appropriate for interactive sessions. For programs, there are a number of transaction-execution forms that can be used. See:

> http://www.zodb.org/en/latest/guide/transactions-and-threading.html

for more information.

[1] In a more complex query, Postgres might evaluate the expression. It depends on what other indexes might be in play.

CHAPTER 5

Newt DB Topics

Connection strings

Postgres connection strings are documented in section 32.1.1 of the Postgres documentation. They take 2 forms:

1. URL syntax:

```
postgresql://[user[:password]@][netloc][:port][/dbname][?param1=value1&...]
```

2. Keyword/Value syntax:

```
host=localhost port=5432 dbname=mydb user=sally password=123456
```

All of the parameters have defaults and may be excluded. An empty string is just an application of the keyword/value syntax with no parameters specified.

To avoid including passwords in connection strings, you can use a Postgres password file.

Text Configuration

Newt DB provides a Python API for creating database connections. You can also use ZODB's text configuration API. Text configuration usually provides the easiest way to configure a database, especially if you need to provide non-default options. Configuration strings can be included in configuration files by themselves or as parts of larger configurations.

Here's an example text configuration for Newt DB:

```
%import newt.db

<newtdb>
  <zodb>
    <relstorage>
      keep-history false
```

```
      <newt>
        <postgresql>
          dsn dbname=''
        </postgresql>
      </newt>
    </relstorage>
  </zodb>
</newtdb>
```

The syntax used is based on the syntax used by web servers such as Apache. Elements, in angle brackets identify configuration objects with name-value pairs inside elements to specify options. Optional indentation indicates containment relationships and element start and end tags must appear on their own lines.

Newt DB provides two configuration elements: `newtdb` and `newt`. These elements augment existing elements to provide extra behavior.

newt Wraps a RelStorage `postgresql` element to provide a Newt Postgres database adapter that stores JSON data in addition to normal database data.

An optional `transformation` option may be provided to provide a data-transformation function. See the *Data transformation topic*.

newtdb Wraps a `zodb` element to provide a Newt database rather than a normal ZODB database. The Newt database provides extra APIs for searching and transaction management.

Some things to note:

- An `%import` directive is used to load the configuration schema for Newt DB.
- A `keep-history` option is used to request a history-free storage. History-free storages only keep current data and discard transaction meta data. History-preserving storages keep past data records until they are packed away and allow "time-travel" to view data in the past. History-preserving storages are much slower and require more maintenance. Newt DB works with either history-preserving or history-free storages, but history-free storages are recommended and are the default for the Python API.

The RelStorage documentation provides information on the options for the relstorage element and for the postgresql element.

The ZODB documentation provides information on the options for the zodb element.

Database Administration

Because Newt stores it's data in PostgreSQL, you'll want to become familiar with PostgreSQL database administration. You can forego much of this if you use a a hosted Postgres server, especially for production deployments.

If you decide to install PostgreSQL, consider one of the binary distributions which can make installation and simple operation very easy. Another option is to use Docker images, which have self-contained PostgreSQL installations. Simple binary installations are a good choice for development environments.

Packing

In addition to administration of the underlying Postgres database, Newt DB databases need to be packed periodically. Packing performs 2 functions:

- For history-preserving[1] databases, packing removes non-current records that were written prior to the pack time.

[1] History-preserving databases allow time travel to times and undo of transactions written before the pack time. History-preserving databases are slower and require more frequent packing than history-free databases. This is why Newt DB makes history-free databases its default configuration.

- Packing detects and removes "garbage" object records. Garbage objects are objects that are no-longer reachable from the database root object. When you remove an object from a container, making it unreachable, it isn't deleted right away, but is removed the next time the database is garbage collected.

When you install Newt DB with pip[2]:

```
pip install newt.db
```

A zodbpack script is also installed. This is a command-line script used to pack a database, typically through some sort of scheduled process such as a cron job. The basic usage is:

```
zodbpack -d 1 CONFIG_FILE
```

The `-d` option specified the number of days in the past to pack to. The default is to pack 0 days in the past[3]. `CONFIG_FILE` is that path to a *Newt configuration file*. For more information, see the zodbpack documentation.

Database Schemas

Some databases claim to have no schemas. There is always a schema, even if it's only in programmers' heads.

In Newt DB, there isn't a server-side schema. Newt DB is object-oriented on the client and the schema is expressed semi-formally by Python classes and their data expectations.

Newt Schemas are highly dynamic. It's easy to add and remove data elements. See the ZODB documentation for tips on schema migration.

Information for ZODB Users

Newt DB builds on RelStorage and Postgres, adding JSON conversion and search support.

Newt provides some significant enhancements to ZODB applications:

- Access to data outside of Python. Data are stored in a JSON representation that can be accessed by non-Python tools.
- Fast, powerful search, via Postgres SQL and indexes.
- Because indexes are maintained in Postgres rather than in the app, far fewer objects are stored in the database.
- Database writes may be much smaller, again because indexing data structures don't have to be updated at the database level, and the likelihood of conflict errors is reduced.

It's easy to migrate existing applications to Newt DB. The standard RelStorage `zodbconvert` works with Newt DB.

The next version of Newt will provide a options for batch-computation of JSON data, which will allow the conversion of existing Postgres RelStorage databases in place.

Updating an existing PostgreSQL RelStorage ZODB application to use Newt DB

There are two ways to add Newt DB to an existing PostgreSQL RelStorage ZODB application.

[2] If you use another tool, you may have to make sure the scripts from the RelStorage package are installed. For example, Buildout won't install RelStorage scripts unless RelStorage is explicitly listed.

[3] You may want to pack to a day in the past, rather than 0 days to guard against an application bug in which an object is removed from a container in one transaction and added to another container in a separate transaction. In this case, the object is temporarily garbage and, if you're unlucky, it could be garbage collection while temporarily garbage.

1. Update your *database text configuration* to include a `newt` tag and optionally a `newtdb` tag. After all of your database clients have been updated (and restarted), then new database records will be written to the `newt` table. You'll need to run the *newt updater* with the `--compute-missing` option to write `newt` records for your older data:

   ```
   newt-updater --compute-missing CONNECTION_STRING
   ```

   Note that this option requires PostgreSQL 9.5 or later.

2. Use the *Newt DB updater* to maintain Newt data asynchronously. This requires no change to your database setup, but requires managing a separate process. Because updates are asynchronous, Newt JSON data may be slightly out of date at times.

JSON Serialization

This topic explains some details about how data are serialized to JSON, beyond the behavior of the standard Python json module.

The first thing to note is that JSON serialization is lossy. This is why Newt saves data in both pickle and JSON format.

The serialization nevertheless preserves class information.

The serialization, like pickle, supports cyclic data structures using a combination of persistent references and intra-record references.

Non-persistent instances

Non-persistent instances are converted to JSON objects with `::` properties giving their dotted class names. In the common case of objects with their instance dictionaries used as their pickled state, the object attributes become properties.

So, for example, given a class `MyClass` in module `mymodule`:

```
class MyClass:

    def __init__(self, a, b):
        self.a = a
        self.b = b
```

The JSON serialization would look like:

```
{"::": "mymodule.MyClass", "a": 1, "b": 2}
```

Non-dictionary state

For instances with pickled state that's not a dictionary, a JSON object is created with a `state` property containing the serialized state and a `::` property with the dotted class name.

New arguments

Objects that take arguments to their `__new__` method will have the arguments serialized in the `::()` property.

Intra-object reference ids

If a record has cycles and an object in the record is referenced more than once, then the object will have an `::id` property who's value is an internal reference id.

For objects like lists and sets, which aren't normally serialized as objects, when an object is referenced more than once, it's wrapped in a "shared" object with an `::id` property and a `value` property.

Intra-record cycles

Cyclic data structures are allowed within persistent object records, although they are **extremely rare**. When there's a cycle, then objects that are referenced more than once:

- have `::id` properties that assign them intra-record ids.

  Objects like lists, who's state are not dictionary are wrapped in a "shared" objects.

- Are replaced with reference objects in all bit one of the references. Reference objects have a single property, `::->` giving the intra-record id of the object being referenced.

Here's an example:

```
>>> from newt.db.tests.testjsonpickle import I
>>> i = I(a=1)
>>> d = dict(b=1)
>>> l = [i, i, d, d]
>>> l.append(l)
```

The serialization of the list, `l` would be equivalent to:

```
{
  "::": "shared",
  "::id": 0,
  "value": [
    {
      "::": "newt.db.tests.testjsonpickle.I",
      "::id": 2,
      "a": 1
    },
    {"::->": 2},
    {
      "::id": 5,
      "b": 1
    },
    {"::->": 5},
    {"::->": 0}
  ]
}
```

Intra-record references like these are difficult to work with, which is a good reason to avoid intra-record cycles.

Persistent object

Persistent objects are stored in 4 columns of the `newt` table:

Column	Type
zoid	bigint
class_name	text
ghost_pickle	bytea
state	jsonb

The class name and state are separated and the state doesn't have a `::` property containing the dotted class name.

The `ghost_pickle` field contains the class name and `__new__` arguments if necessary. It's used to create new objects when searching.

Persistent references

When one persistent object references another persistent object, the reference is serialized with a reference object, having a property `::=>` whose value is the object id of the referenced object[1]. For example, serialization of a sub-task object containing a reference to a parent task would be equivalent to:

```
{
  "title": "Do something",
  "parent": {"::=>": 42}
}
```

Note that cycles among persistent objects are common and don't present any problems for serialization because persistent objects are serialized separately.

Dates and times

`datetime.date` objects and `datetime.datetime` instances without time zones are converted strings using their `isoformat` methods.

`datetime.datetime` instances with time zones are serialized as objects with a `::` property of `datetime`, a `value` property with their ISO formatted value, and a `tz` property containing a JSON serialization of their time zones.

Data transformation

You can have more control over how is JSON is generated from your data by supplying a transform function. A transform function accepts two positional arguments:

class_name The full dotted name of a persistent object's class.

state The object's state as a bytes string in JSON format.

The transform function must return a new state string or `None`. If `None` is returned, then the original state is used.

A transform function may return an empty string to indicate that a record should be skipped and not written to the `newt` table.

For example, `persistent.mapping.PersistentMapping` objects store their data in a `data` attribute, so their JSON representation is more complex than we might want. Here's a transform that replaces the JSON representation of a `PersistentMapping` with its data:

[1] This is a change from versions of Newt before 0.4.0. Earlier versions represented persistent references as objects with a `::` property with the value `persistent` and an `id` property who's value is an integer object id or a list containing an integer object id and a dotted class name. The attributes will be retained until Newt DB version 1, at which point they will no longer be included.

```
import json

def flatten_persistent_mapping(class_name, state):
    if class_name == 'persistent.mapping.PersistentMapping':
       state = json.loads(state)
       state = state['data']
       return json.dumps(state)
```

We can supply a transform function to the Python constructor using the `transform` keyword argument:

```
import newt.db

conn = newt.db.connection('', transform=flatten_persistent_mapping)
```

To specify a transform in text configuration, use a `transform` option to supply the dotted name of your transform function in the `newt` configuration element:

```
%import newt.db

<newtdb>
  <zodb>
    <relstorage>
      keep-history false
      <newt>
        transform myproject.flatten_persistent_mapping
        <postgresql>
          dsn dbname=''
        </postgresql>
      </newt>
    </relstorage>
  </zodb>
</newtdb>
```

Asynchronous JSON conversion

Normally, newt converts data to JSON as it's saved in the database. In turn, any indexes defined on the JSON data are updated at the same time. With the default JSON index, informal tests show Newt DB write performance to be around 10 percent slower than RelStorage. Adding a text index brought the performance down to the point where writes took twice as long, but were still fairly fast, several hundred per second on a laptop.

If you have a lot of indexes to update or write performance is critical, you may want to leverage Newt DB's ability to update the JSON data asynchronously. Doing so, allows the primary transactions to execute more quickly.

Updating indexes asynchronously will usually be more efficient, because Newt DB's asynchronous updater batches updates. When indexes are updated for many objects in the same transaction, less data has to be written per transaction.

If you want to try New DB, with an existing RelStorage/PostgreSQL database, you can use the updater to populate Newt DB without changing your application and introduce the use of Newt DB's search API gradually.

There are some caveats however:

- Because updates are asynchronous, search results may not always reflect the current data.
- Packing requires some special care, as will be discussed below.
- You'll need to run a separate daemon, `newt-updater` in addition to your database server.

Contents

Using Newt's Asynchronous Updater

To use Newt's asynchronous updater:

- Omit `newt` tag from your database configuration, as in:

```
%import newt.db

<newtdb foo>
  <zodb>
    <relstorage>
      keep-history false
      <postgresql>
        dsn postgresql://localhost/mydb
      </postgresql>
    </relstorage>
  </zodb>
</newtdb>
```

- Run the `newt-updater` script:

```
newt-updater postgresql://localhost/mydb
```

  You'll want to run this using a daemonizer like supervisord or ZDaemon.

`newt-updater` has a number of options:

-l, --logging-configuration Logging configuration.

This can be a log level, like `INFO` (the default), or the path to a ZConfig logging configuration file.

-g, --gc-only Collect garbage and exit.

This removes Newt DB records that don't have corresponding database records. This is done by executing:

```
delete from newt n where not exists (
  select from object_state s where n.zoid = s.zoid)
```

Note that garbage collection is normally performed on startup unless the -G option is used.

-G, --no-gc Don't perform garbage collection on startup.

--nagios Check the status of the updater.

The status is checked by checking the updater lag, which is the difference between the last transaction committed to the database, and the last transaction processed

by the updater. The option takes 2 numbers, separated by commas. The first number is the lag, in seconds, for the updater to be considered to be OK. The second number is the maximum lag for which the updater isn't considered to be in error. For example, 1,99 indicates OK if 1 or less, WARNING if more than 1 and less than or equal to 99 and ERROR of more than 99 seconds.

-t, --poll-timeout Specify a poll timeout, in seconds.

Normally, the updater is notified to poll for changes. If it doesn't get notified in poll-timeout seconds, it will poll anyway. This is a backstop to PostgreSQL's notification. The default timeout is 300 seconds.

-m, --transaction-size-limit The target transaction batch size. This limits (loosely) the number of records processed in a batch. Larger batches incur less overhead, but long-lasting transactions can cause interfere with other processing. The default is 100 thousand records.

This option only comes into play when a large number of records have to be processed, typically when first running the updater or using the `--compute-missing` option.

-T, --remove-delete-trigger Remove the Newt DB delete trigger, if it exists.

The Newt DB delete trigger is incompatible with the updater. It can cause deadlock errors is packed while the updater is running. This option is needed if you set up Newt DB normally, and then decided that you wanted update Newt DB asynchronously.

-d, --driver Provide an explicit Postgres driver name (psycopg2 or psycopg2cffi). By default, the appropriate driver will be selected automatically.

--compute-missing Compute missing newt records.

Rather than processing new records, process records written up through the current time and stop. Only missing records are updated. **This option requires PostgreSQL 9.5 or later.**

This is used to compute newt records after adding Newt DB to an existing PostgreSQL RelStorage application.

Garbage collection

The asynchronous updater tracks new database inserts and updates. When a database is *packed*, records are removed without generating updates. Those deletes won't be reflected in the Newt DB. You can tell the updater to clean up Newt DB records for which there are no-longer database records by either restarting it, or running it with the `-g` option:

```
newt-updater -g postgresql://localhost/mydb
```

This tells the updater to just collect garbage. You'll probably want to run this right after running zodbpack.

Monitoring

When running an external updater, like `newt-updater`, you'll want to have some way to monitor that it's working correctly. The `--nagios` option `newt-updater` script can be used to provide a Nagios Plugin:

```
newt-updater postgresql://localhost/mydb --nagios 3,99
```

The argument to the `--nagios` option is a pair of numbers giving limits for OK and warning alerts. They're based on how far behind the updater is. For example, with the example above, the monitor considers the updater to be OK if it is 3 seconds behind or less, in error if it is more than 99 seconds behind and of concern otherwise.

Any monitoring system compatible with the Nagios plugin API can be used.

The monitor output includes the lag, how far behind the updater is, in seconds as a performance metric.

Follow changes in ZODB RelStorage PostgreSQL databases

The *newt.db.follow* module provides an API for subscribing to database changes .

It's used by newt.db for asynchronous updates, but it can be used for other applications, such as:

- creating data indexes, such as Elasticsearch indexes.
- creating alternate representations, such as relational models to support other applications.
- monitoring or analytics of data changes.

You can get an iterator of changes by calling the *updates()* function:

```
>>> import newt.db.follow
>>> import pickle
>>> for batch in newt.db.follow.updates(dsn):
...     for tid, zoid, data in batch:
...         print_(zoid, pickle.loads(data).__name__)
0 PersistentMapping
```

The updates iterator returns batches to facilitate batch processing of data while processing data as soon as possible. Batches are as large as possible, limited loosly by a target batch size with the constraint that transactions aren't split between batches. Batches are themselves iterators.

The interator can run over a range of transactions, or can run indefinately, returning new batches as new data are committed.

See the *updates reference* for detailed documentation. One of the parameters is `end_tid`, an end transaction id for the iteration. If no `end_tid` parameter is provided, the iterator will iterate forever, blocking when necessary to wait for new data to be committed.

The data returned by the follower is a pickle, which probably isn't very useful. You can convert it to JSON using Newt's JSON conversion. We can update the example above:

```
>>> import newt.db.follow
>>> import newt.db.jsonpickle

>>> jsonifier = newt.db.jsonpickle.Jsonifier()
>>> for batch in newt.db.follow.updates(dsn):
...     for tid, zoid, data in batch:
...         class_name, _, data = jsonifier((zoid, tid), data)
...         if data is not None:
...             print_(zoid, class_name, data)
0 persistent.mapping.PersistentMapping {"data": {"x": 1}}
```

Jsonifiers take a label (used for logging errors) and data and return a class_name, a ghost pickle, and object state as JSON data. The ghost pickle is generally only useful to Newt itself, so we ignore it here. If the JSON data returned is `None`, we skip processing the data. The return value may be `None` if:

- the raw data was an empty string, in which case the database record deleted the object,

- the object class was one the JSON conversion skipped, or
- There was an error converting the data.

Tracking progress

Often the data returned by the `updates` iterator is used to update some other data. Often clients will be stopped and later restarted and need to keep track of where they left off. The *`set_progress_tid()`* method can be used to save progress for a client:

```
>>> newt.db.follow.set_progress_tid(dsn, 'mypackage.mymodule', tid)
```

The first argument is a PostgreSQL database connection. The second argument is a client identifier, typically the dotted name of the client module. The third argument is the last transaction id that was processed.

Later, you can use *`get_progress_tid()`* to retrieve the saved transaction id:

```
>>> start_tid = newt.db.follow.get_progress_tid(dsn, 'mypackage.mymodule')
```

You'd then pass the retrieved transaction identifier as the `start_tid` argument to *`updates()`*.

Garbage collection

One complication in dealing with updating external data is garbage collection. When a Newt DB database is *packed*, records are removed without generating updates. Data that's removed from Newt DB when it's packed should be removed from external representations as well. The easiest way to do this is by splitting packing into 3 steps:

1. Run zodbpack with the `--prepack` option:

   ```
   zodbpack -d 1 --prepack CONFIG_FILE
   ```

   This tells `zeopack` to stop after identifying garbage.

2. Call the `newt.db.follow.garbage()` function to get an iterator of object ids that will be deleted in the second phase of packing:

   ```
   import newt.db.follow
   for zoid in newt.db.follow.garbage(dsn):
       my_remove_external_data_function(zoid)
   ```

3. Run zodbpack with the `--use-prepack-state` option:

   ```
   zodbpack -d 1 --use-prepack-state CONFIG_FILE
   ```

   This tells `zeopack` to remove the garbage identified in the first step.

zodburi URI support

A number of applications, most notably Pyramid, use URI syntax to define ZODB databases they use. Newt DB supports this syntax through the `newt` scheme. For example:

```
newt://mydbserver/my_database?keep_history=true&connection_cache_size=100000
```

Newt URIs have roughly the same form as *PostgreSQL URI connection strings* except that they use the `newt` URI scheme instead of the `postgresql` schema and they support extra query-string parameters:

keep_history Boolean (true, false, yes, no, 1 or 0) indicating whether non-current database records should be kept. This is false, by default.

driver The Postgres driver name (psycopg2 or psycopg2cffi). By default, the driver is determined automatically.

connection_cache_size The target maximum number of objects to keep in the per-connection object cache.

connection_pool_size The target maximum number of ZODB connections to keep in the connection pool.

Limitations

There are a number of limitations to be aware of when using the URI syntax.

- Because the zodburi framework can only be used to set up ordinary ZODB databases, resulting connection objects won't have the extra search or transaction convenience functions provided by Newt DB. When searching, you'll use the *Newt search module* functions, passing your database connections as the first arguments.
- zodburi provides insufficient control over database configuration. You'll end up having to use something else for production deployments.

CHAPTER 6

Reference

Contents

newt.db module-level functions

`newt.db.`**`connection`**(*dsn*, ***kw*)
Create a newt *`newt.db.Connection`*.

Keyword options can be used to provide either ZODB.DB options or RelStorage options.

`newt.db.`**`DB`**(*dsn*, ***kw*)
Create a Newt DB database object.

Keyword options can be used to provide either ZODB.DB options or RelStorage options.

A Newt DB object is a thin wrapper around `ZODB.DB` objects. When it's `open` method is called, it returns *`newt.db.Connection`* objects.

`newt.db.`**`storage`**(*dsn*, *keep_history=False*, *transform=None*, *auxiliary_tables=()*, ***kw*)
Create a RelStorage storage using the newt PostgresQL adapter.

Keyword options can be used to provide either ZODB.DB options or RelStorage options.

`newt.db.`**`pg_connection`**(*dsn*, *driver_name='auto'*)
Create a PostgreSQL (not newt) database connection

This function should be used rather than, for example, calling `psycopg2.connect`, because it can use other Postgres drivers depending on the Python environment and available modules.

class `newt.db.`**`Connection`**(*connection*)
Wrapper for ZODB.Connection.Connection objects

`newt.db.Connection` objects provide extra helper methods for searching and for transaction management.

`abort`()
Abort the current transaction

`commit`()
Commit the current transaction

`create_text_index`(*fname*, *D=None*, *C=None*, *B=None*, *A=None*, *config=None*)
Set up a newt full-text index.

The `create_text_index_sql` method is used to compute SQL, which is then executed to set up the index. (This can take a long time on an existing database with many records.)

The SQL is executed against the database associated with the given connection, but a separate connection is used, so it's execution is independent of the current transaction.

static **`create_text_index_sql`**(*fname*, *D=None*, *C=None*, *B=None*, *A=None*)
Compute and return SQL to set up a newt text index.

The resulting SQL contains a statement to create a PL/pgSQL function and an index-creation function that uses it.

The first argument is the name of the function to be generated. The second argument is a single expression or property name or a sequence of expressions or property names. If expressions are given, they will be evaluated against the newt JSON `state` column. Values consisting of alphanumeric characters (including underscores) are threaded as names, and other values are treated as expressions.

Additional arguments, `C`, `B`, and `A` can be used to supply expressions and/or names for text to be extracted with different weights for ranking. See: https://www.postgresql.org/docs/current/static/textsearch-controls.html#TEXTSEARCH-RANKING

The `config` argument may be used to specify which text search configuration to use. If not specified, the server-configured default configuration is used.

`query_data`(*query*, **args*, ***kw*)
Query the newt Postgres database for raw data.

Query parameters may be provided as either positional arguments or keyword arguments. They are inserted into the query where there are placeholders of the form: `%s` for positional arguments, or `%(NAME)s` for keyword arguments.

A sequence of data tuples is returned.

`search`(*query*, **args*, ***kw*)
Search for newt objects using an SQL query.

Query parameters may be provided as either positional arguments or keyword arguments. They are inserted into the query where there are placeholders of the form `%s` for positional arguments or `%(NAME)s` for keyword arguments.

The query results must contain the columns `zoid` and `ghost_pickle`. It's simplest and costs nothing to simply select all columns (using `*`) from the `newt` table.

A sequence of newt objects is returned.

search_batch(*query*, *args*, *batch_start*, *batch_size=None*)
Query for a batch of newt objects.

Query parameters are provided using the `args` argument, which may be a tuple or a dictionary. They are inserted into the query where there are placeholders of the form `%s` for an arguments tuple or `%(NAME)s` for an arguments dict.

The `batch_size` and `batch_size` arguments are used to specify the result batch. An `ORDER BY` clause should be used to order results.

The total result count and sequence of batch result objects are returned.

The query parameters, `args`, may be omitted. (In this case, `batch_size` will be None and the other arguments will be re-arranged appropriately. `batch_size` *is required*.) You might use this feature if you pre-inserted data using a database cursor mogrify method.

where(*query_tail*, **args*, ***kw*)
Query for objects satisfying criteria.

This is a convenience wrapper for the `search` method. The first arument is SQL text for query criteria to be included in an SQL where clause.

This mehod simply appends it's first argument to:

```
select * from newt where
```

and so may also contain code that can be included after a where clause, such as an `ORDER BY` clause.

Query parameters may be provided as either positional arguments or keyword arguments. They are inserted into the query where there are placeholders of the form: `%s` for positional arguments, or `%(NAME)s` for keyword arguments.

A sequence of newt objects is returned.

where_batch(*query_tail*, *args*, *batch_start*, *batch_size=None*)
Query for batch of objects satisfying criteria

Like the `where` method, this is a convenience wrapper for the `search_batch` method.

Query parameters are provided using the second, `args` argument, which may be a tuple or a dictionary. They are inserted into the query where there are placeholders of the form `%s` for an arguments tuple or `%(NAME)s` for an arguments dict.

The `batch_size` and `batch_size` arguments are used to specify the result batch. An `ORDER BY` clause should be used to order results.

The total result count and sequence of batch result objects are returned.

The query parameters, `args`, may be omitted. (In this case, `batch_size` will be None and the other arguments will be re-arranged appropriately. `batch_size` *is required*.) You might use this feature if you pre-inserted data using a database cursor mogrify method.

newt.db.search module-level functions

Search API.

It's assumed that the API is used with an object stored in a RelStorage with a Postgres back end.

`newt.db.search.`**where**(*conn*, *query_tail*, **args*, ***kw*)
Query for objects satisfying criteria.

This is a convenience wrapper for the `search` method. The first arument is SQL text for query criteria to be included in an SQL where clause.

This mehod simply appends it's first argument to:

```
select * from newt where
```

and so may also contain code that can be included after a where clause, such as an `ORDER BY` clause.

Query parameters may be provided as either positional arguments or keyword arguments. They are inserted into the query where there are placeholders of the form: `%s` for positional arguments, or `%(NAME)s` for keyword arguments.

A sequence of newt objects is returned.

`newt.db.search.`**`search`**(*conn*, *query*, **args*, ***kw*)
Search for newt objects using an SQL query.

Query parameters may be provided as either positional arguments or keyword arguments. They are inserted into the query where there are placeholders of the form `%s` for positional arguments or `%(NAME)s` for keyword arguments.

The query results must contain the columns `zoid` and `ghost_pickle`. It's simplest and costs nothing to simply select all columns (using `*`) from the `newt` table.

A sequence of newt objects is returned.

`newt.db.search.`**`where_batch`**(*conn*, *query_tail*, *args*, *batch_start*, *batch_size=None*)
Query for batch of objects satisfying criteria

Like the `where` method, this is a convenience wrapper for the `search_batch` method.

Query parameters are provided using the second, `args` argument, which may be a tuple or a dictionary. They are inserted into the query where there are placeholders of the form `%s` for an arguments tuple or `%(NAME)s` for an arguments dict.

The `batch_size` and `batch_size` arguments are used to specify the result batch. An `ORDER BY` clause should be used to order results.

The total result count and sequence of batch result objects are returned.

The query parameters, `args`, may be omitted. (In this case, `batch_size` will be None and the other arguments will be re-arranged appropriately. `batch_size` *is required*.) You might use this feature if you pre-inserted data using a database cursor mogrify method.

`newt.db.search.`**`search_batch`**(*conn*, *query*, *args*, *batch_start*, *batch_size=None*)
Query for a batch of newt objects.

Query parameters are provided using the `args` argument, which may be a tuple or a dictionary. They are inserted into the query where there are placeholders of the form `%s` for an arguments tuple or `%(NAME)s` for an arguments dict.

The `batch_size` and `batch_size` arguments are used to specify the result batch. An `ORDER BY` clause should be used to order results.

The total result count and sequence of batch result objects are returned.

The query parameters, `args`, may be omitted. (In this case, `batch_size` will be None and the other arguments will be re-arranged appropriately. `batch_size` *is required*.) You might use this feature if you pre-inserted data using a database cursor mogrify method.

`newt.db.search.`**`query_data`**(*conn*, *query*, **args*, ***kw*)
Query the newt Postgres database for raw data.

Query parameters may be provided as either positional arguments or keyword arguments. They are inserted into the query where there are placeholders of the form: `%s` for positional arguments, or `%(NAME)s` for keyword arguments.

A sequence of data tuples is returned.

`newt.db.search.`**`create_text_index_sql`**(*fname*, *D=None*, *C=None*, *B=None*, *A=None*, *config=None*)

Compute and return SQL to set up a newt text index.

The resulting SQL contains a statement to create a PL/pgSQL function and an index-creation function that uses it.

The first argument is the name of the function to be generated. The second argument is a single expression or property name or a sequence of expressions or property names. If expressions are given, they will be evaluated against the newt JSON `state` column. Values consisting of alphanumeric characters (including underscores) are threaded as names, and other values are treated as expressions.

Additional arguments, `C`, `B`, and `A` can be used to supply expressions and/or names for text to be extracted with different weights for ranking. See: https://www.postgresql.org/docs/current/static/textsearch-controls.html#TEXTSEARCH-RANKING

The `config` argument may be used to specify which text search configuration to use. If not specified, the server-configured default configuration is used.

`newt.db.search.`**`create_text_index`**(*conn*, *fname*, *D*, *C=None*, *B=None*, *A=None*, *config=None*)

Set up a newt full-text index.

The `create_text_index_sql` method is used to compute SQL, which is then executed to set up the index. (This can take a long time on an existing database with many records.)

The SQL is executed against the database associated with the given connection, but a separate connection is used, so it's execution is independent of the current transaction.

`newt.db.search.`**`read_only_cursor`**(*conn*)

Get a database cursor for reading.

The returned cursor can be used to make PostgreSQL queries and to perform safe SQL generation using the cursor's mogrify method.

The caller must close the returned cursor after use.

newt.db.follow module-level functions

`newt.db.follow.`**`updates`**(*conn*, *start_tid=-1*, *end_tid=None*, *batch_limit=100000*, *internal_batch_size=100*, *poll_timeout=300*)

Create a data-update iterator

The iterator returns an iterator of batchs, where each batch is an iterator of records. Each record is a triple consisting of an integer transaction id, integer object id and data. A sample use:

```
>>> import newt.db
>>> import newt.db.follow
>>> connection = newt.db.pg_connection('')
>>> for batch in newt.db.follow.updates(connection):
...     for tid, zoid, data in batch:
...         print(tid, zoid, len(data))
```

If no `end_tid` is provided, the iterator will iterate until interrupted.

Parameters:

conn A Postgres database connection.

start_tid Start tid, expressed as an integer. The iterator starts at the first transaction **after** this tid.

end_tid End tid, expressed as an integer. The iterator stops at this, or at the end of data, whichever is less. If the end tid is None, the iterator will run indefinately, returning new data as they are committed.

batch_limit A soft batch size limit. When a batch reaches this limit, it will end at the next transaction boundary. The purpose of this limit is to limit read-transaction size.

internal_batch_size The size of the internal Postgres iterator. Data aren't loaded from Postgres all at once. Server-side cursors are used and data are loaded from the server in `internal_batch_size` batches.

poll_timeout When no `end_tid` is specified, this specifies how often to poll for changes. Note that a trigger is created and used to notify the iterator of changes, so changes ne detected quickly. The poll timeout is just a backstop.

`newt.db.follow.`**`get_progress_tid`**(*connection*, *id*)
Get the current progress for a follow client.

Return the last saved integer transaction id for the client, or -1, if one hasn't been saved before.

A follow client often updates some other data based on the data returned from `updates`. It may stop and restart later. To do this, it will call `set_progress_tid` to save its progress and later call `get_progress_tid` to find where it left off. It can then pass the returned tid as `start_tid` to `updates`.

The `connection` argument must be a PostgreSQL connection string or connection.

The `id` parameters is used to identify which progress is wanted. This should uniquely identify the client and generally a dotted name (`__name__`) of the client module is used. This allows multiple clients to have their progress tracked.

`newt.db.follow.`**`set_progress_tid`**(*connection*, *id*, *tid*)
Set the current progress for a follow client.

See `get_progress_tid`.

The `connection` argument must be a PostgreSQL connection string or connection.

The `id` argument is a string identifying a client. It should generally be a dotted name (usually `__name__`) of the client module. It must uniquely identify the client.

The `tid` argument is the most recently processed transaction id as an int.

`newt.db.follow.`**`listen`**(*dsn*, *timeout_on_start=False*, *poll_timeout=300*)
Listen for newt database updates.

Returns an iterator that returns integer transaction ids or None values.

The purpose of this method is to determine if there are updates. If transactions are committed very quickly, then not all of them will be returned by the iterator.

None values indicate that `poll_interval` seconds have passed since the last update.

Parameters:

dsn A Postgres connection string

timeout_on_start Force None to be returned immediately after listening for notifications.

This is useful in some special cases to avoid having to time out waiting for changes that happened before the iterator began listening.

poll_timeout A timeout after which None is returned if there are no changes. (This is a backstop to PostgreSQL's notification system.)

newt.db.jsonpickle module-level functions

Convert pickles to JSON

The goal of the conversion is to produce JSON that is useful for indexing, querying and reporting in external systems like Postgres and Elasticsearch.

class `newt.db.jsonpickle.`**`Jsonifier`**(*skip_class=None*, *transform=None*)

`__call__`(*id*, *data*)
Convert data from a ZODB data record to data used by newt.

The data returned is a class name, ghost pickle, and state triple. The state is a JSON-formatted string. The ghost pickle is a binary string that can be used to create a ZODB ghost object.

If there is an error converting data, if the data is empty, or if the skip_class function returns a true value, then `(None, None, None)` is returned.

Parameters:

id A data identifier (e.g. an object id) used when logging errors.

data Pickle data to be converted.

`__init__`(*skip_class=None*, *transform=None*)
Create a callable for converting database data to Newt JSON

Parameters:

skip_class A callable that will be called with the class name extracted from the data. If the callable returns a true value, then data won't be converted to JSON and `(None, None, None)` are returned. The default, which is available as the `skip_class` attribute of the `Jsonifier` class, skips objects from the `BTrees` package and blobs.

transform A function that transforms a record's state JSON.

If provided, it should accept a class name and a state string in JSON format.

If the transform function should return a new state string or None. If None is returned, the original state is used.

If the function returns an empty string, then the Jsonifier will return `(None, None, None)`. In other words, providing a transform that returns an empty string is equivalent to providing a `skip_class` function that returns True.

Returning anything other than None or a string is an error and behavior is undefined.

class `newt.db.jsonpickle.`**`JsonUnpickler`**(*pickle*)
Unpickler that returns JSON

Usage:

```
>>> apickle = pickle.dumps([1,2])
>>> unpickler = JsonUnpickler(apickle)
>>> json_string = unpickler.load()
>>> unpickler.pos == len(apickle)
True
```

CHAPTER 7

Newt DB Community Resources

Mailing list https://groups.google.com/forum/#!forum/newtdb

Github https://github.com/newtdb/db

Documentation: http://www.newtdb.org/

Report bugs: https://github.com/newtdb/db/issues

Wiki https://github.com/newtdb/db/wiki

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

A

C

D

G

J

L

N

P

Q

R

S

U

W