

User Guide, Dejavu v. 2.0, alpha (30 May 08)

Introduction.....	1
Basic Structure.....	1
Simple Example.....	2
Design Goals.....	3
Obtaining and Installing.....	3
Compared To Other Database Wrappers.....	4
SQLObject.....	4
Application Designers: Using Dejavu to Construct a Domain Model.....	6
Units.....	6
UnitProperty.....	6
datetime.datetime.....	6
Unit ID's.....	7
Creating and Populating Properties.....	7
Unit Properties are First-Class Objects.....	8
decimal.....	8
binary.....	8
Triggers.....	8
TriggerProperty.....	9
Registration of Unit Classes.....	9
Synchronizing the Model.....	9
Associations between Unit Classes.....	10
"Related units" methods.....	11
Custom Unit Associations.....	12
Managing Schemas.....	12
Conflicts.....	12
error.....	12

warn.....	12
repair.....	13
ignore.....	13
Installation.....	13
Modifying Storage Structures.....	13
Upgrading: Schema Objects.....	14
Versions: The Deployed Version Unit.....	15
Automatic Unit Classes.....	15
Application Developers: Managing Units.....	16
Sandboxes.....	16
Memorizing Units.....	16
Sequencing.....	16
Recalling.....	17
recall().....	17
Recalling multiple classes at once (JOINS).....	17
xrecall().....	18
unit().....	18
"Magic recaller" methods.....	18
Forgetting and Repressing.....	18
Flushing Sandboxes.....	19
Views.....	19
xview().....	20
Transactions.....	20
Querying.....	21
The Expression Class.....	21
Early binding.....	22
External functions within Expressions.....	22
Using filter to Form Expressions.....	23

Using comparison to Form Expressions.....	23
Combining Expressions.....	24
Specifying types for Expression kwargs.....	25
Exporting the logic module.....	25
Unit Engines.....	25
Collections: Lists of Units.....	25
Engines.....	26
Rules.....	26
Engine Functions.....	28
Analysis Tools.....	28
Sorting Units.....	28
Cross-tabulation.....	28
Deployers: Configuring Storage.....	30
Database Storage Managers.....	30
Microsoft SQL Server /Microsoft Access (Jet).....	30
PostgreSQL.....	31
MySQL (MySQLdb).....	31
SQLite (pysqlite/sqlite3).....	31
Firebird (kinterbasdb).....	32
Common Database Configuration Entries.....	32
Other Storage Managers.....	33
RAM.....	33
Memcached.....	33
External Dependency: python-memcached.....	33
JSON.....	33
External Dependency: simplejson.....	33
Shelve.....	34
Folders.....	34

Middleware.....	35
Object Cache.....	35
Aged Cache.....	36
Burned Cache.....	36
Partitioning.....	37
Vertical Partitioner.....	37
SM Comparison Chart.....	37
Advanced Topics.....	41
Subclassing Sandbox.....	41
Store Hacking.....	41
Passing through SQL.....	41
Custom Storage Managers.....	42
Generic Database Wrappers.....	43
Adapters.....	43
Decompiler.....	43
Database/Table/IndexSet.....	44
Legacy Database Wrappers.....	44
Other Serialization Mechanisms.....	45
sockets.....	45

Introduction

Dejavu is a thread-safe Object-Relational Mapper for Python applications. It is designed to provide the "Model" third of an MVC application. When you build an application using Dejavu, you must supply the Controller(s) and View(s) yourself. Dejavu does not provide these, and does its best to not limit your choices regarding them.

If you're familiar with Martin Fowler's work [\[1\]](#), you can think of Dejavu as providing a Data Source layer, plus the tools to write your own Domain layer. For the Presentation layer, you're on your own. ;) It primarily uses a generic Data Mapper architecture (as opposed to the more tightly-coupled Active Record architecture).

Basic Structure

Developers build their Model by creating classes which subclass `dejavu.Unit`; in RDBMS terminology, each Unit subclass corresponds to a table; instances of the class correspond to the rows. Each subclass possesses a set of attributes known as "properties", which you can think of as columns in your database table. These attributes are generally formed from a `UnitProperty` descriptor. Any Unit data which needs to be persisted ought to be contained in a Unit Property. However, Unit classes can also possess arbitrary methods and attributes which aid their use within the application.

Unit classes can be *associated* to other Unit classes. This means that one of the properties of UnitA maps to one of the properties of UnitB. Related objects may then be looked up more easily.

Units are managed in memory by `Sandbox` objects, which function as "Identity Maps" [\[1\]](#): in-memory caches of Units which keep commit conflicts to a minimum. Unit objects can be "memorized" and "recalled" from a `Sandbox`, using pure Python lambda expressions [\[2\]](#) as a query language. The lambda is wrapped in an `Expression` object to make it portable.

Sandboxes persist Unit data by `StorageManager` objects. Each persistence mechanism has its own subclass of the `StorageManager` class; for example, persisting Unit data to a Microsoft SQL Server database requires a `StorageManagerADO_SQLServer` object. When recalling data, Storage Managers receive `Expression` objects; database SM's, for example, will typically examine these Expressions and produce SQL statements from them, which they then use to retrieve data. Storage Managers also handle the creation of new Units, and their destruction.

Finally, Dejavu provides a core `Arena` class which you should be able to leverage for any sort of application you are building. The Arena object functions as a top-level "Application" object, collecting the global settings for an application into one place. It doles out Sandboxes, maintains a registry of Units and their associations, and manages startup and shutdown operations.

Simple Example

Since a block of code is often worth a thousand words, here's a minimal example of a Dejavu application:

zookeeper.py

```
import dejavu

class Zoo(dejavu.Unit):
    Name = dejavu.UnitProperty()
    Size = dejavu.UnitProperty(int)

    def total_legs(self):
        return sum([x.Legs for x in self.Animal()])

class Animal(dejavu.Unit):
    Legs = dejavu.UnitProperty(int, default=4)

Animal.set_properties({"Name": unicode,
                      "ZooID": int,
                      })
Animal.many_to_one('ZooID', Zoo, 'ID')

# Set up a global Arena object.
arena = dejavu.Arena()
conf = {u'Connect': r"PROVIDER=MICROSOFT.JET.OLEDB.4.0;DATA
SOURCE=C:\zookeeper.mdb;"}
arena.add_store("main", "access", conf)
arena.register_all(globals())
```

The above creates the model for the zookeeper application. There are three basic things happening:

1. The `Zoo` and `Animal` classes, which subclass `dejavu.Unit`. These will correspond to the `Zoo` and `Animal` tables within the database. Notice the two different methods of declaring Unit properties. Each class also inherits an 'ID' property (an int) from `dejavu.Unit`.
2. The association between the `Animal` class and the `Zoo` class (many-to-one).
3. The setup of a `dejavu` Arena object, including a Storage Manager which uses a Microsoft Access (Jet) database.

Here's a simple interactive session which uses the above (assume that tables have been created and populated elsewhere):

```
>>> import zookeeper
>>> box = zookeeper.arena.new_sandbox()
>>> box.recall(zookeeper.Animal)
[<zookeeper.Animal object at 0x013281F0>, <zookeeper.Animal object at 0x01328150>,
 <zookeeper.Animal object at 0x01328130>, <zookeeper.Animal object at 0x01328230>]
>>> box.recall(zookeeper.Zoo)
[]
>>> zoo = zookeeper.Zoo(Name='San Diego Zoo', Size='38')
>>> box.memorize(zoo)
>>> zoo.ID
1
```

```
>>> box.unit(zookeeper.Zoo, ID=1) is zoo
True
>>> for creature in box.recall(zookeeper.Animal):
    zoo.add(creature)
>>> len(zoo.Animal())
4
```

Design Goals

Dejavu is designed to function in environments with complex integration needs, and tends to separate concerns as much as possible. In particular, Dejavu tries to avoid making decisions in the framework which are better left to developers. Some of those decisions are:

- User interface. Dejavu works well in all sorts of applications, whether desktop, thin-client or web.
- Application package architecture. You can place your application within a single Python module, develop complete packages, or use Dejavu inside a larger framework.
- Which types to use for `Unit` properties. Builtin types are fully supported out of the box, including `datetime` and `decimal`. Tim Peters' excellent `fixedpoint` module is also available. New types are easily added.
- Which keys to use when associating `Unit` classes.
- What to name identifiers.

In the same way, Dejavu tries to avoid having developers make decisions which are better left to deployers. Some of those decisions are:

- Where (and how) to log error messages.
- Which storage mechanism (database or ...?) to use. In particular, deployers are allowed to mix and match stores, including how and when to cache objects in memory. Dejavu tries to make it easy for *deployers* to tune applications to their particular environment.

Unlike most generic storage wrappers, Dejavu does not *require* you to have complete control of your back end. For example, consider Mission Control, the first application built on Dejavu. Mission Control required an ORM which transparently supported two very different backends. Half of the data was to be stored in an MS Access database, over which the application developers had full control. But half of the data was stored in a third-party application, "The Raiser's Edge" (RE) from Blackbaud. RE provides read-only database access; all writes must go through their object-oriented API. Further, reading via that API was found to be too slow. Therefore, a custom Storage Manager (about 2500 lines of code) was developed, which searches for and loads objects via SQL, but writes `Unit` data via the REAPI. Dejavu allows the application logic to be completely ignorant of this complex mass of storage details. If Blackbaud closed its doors tomorrow, the solution could be quickly migrated to another data store; business downtime is reduced in the face of inevitable change.

Obtaining and Installing

You can obtain Dejavu from its Subversion repository at <http://projects.amor.org/dejavu/svn/trunk>. Dejavu is designed to be installed in `site-packages/dejavu` or some other root python path.

Dejavu was built using Python 2.3.2. You should probably use at least 2.3; Dejavu depends upon the `datetime` module. Although Dejavu *supports* additional modules like `fixedpoint` and `decimal`, it does not *require* them.

Dejavu uses bytecode hacks, and therefore requires CPython [\[2\]](#).

Compared To Other Database Wrappers

SQLObject

No matter what project I start on, odds are I'll discover that Ian Bicking has already done the same thing, usually better.

See <http://blog.ianbicking.org/another-less-sleepy-alternative-to-hibernate.html>

Which was a reply to Ruby's ActiveRecord: <http://www.loudthinking.com/arc/000297.html>

Which was a reply to Java's Hibernate:

<http://informit.com/guides/content.asp?g=java&seqNum=127&f1=rss>

Using dejavu, the application developer supplies the following code to define the Units and their relationships:

```
from dejavu import *
import fixedpoint # or decimal, for Python 2.4+
import datetime

class Book(Unit):
    # The ID field is already set to 'int' for all Unit subclasses.
    title = UnitProperty(str)
    price = UnitProperty(fixedpoint.Fixedpoint)
    publishDate = UnitProperty(datetime.datetime)
    publisher = UnitProperty(int)

    def addAuthor(self, author):
        a = Authorship(authorID=author.ID, bookID=self.ID)
        self.sandbox.memorize(a)

    def author_names(self):
        names = []
        for authorship in self.Authorship():
            author = authorship.Author()
            if author:
                names.append(author.name)
        return u', '.join(names)

class Publisher(Unit):
    name = UnitProperty(str)

class Author(Unit):
    name = UnitProperty(str)

class Authorship(Unit):
    authorID = UnitProperty(int)
    bookID = UnitProperty(int)

Book.many_to_one('publisher', Publisher, 'ID')
Authorship.many_to_one('bookID', Book, 'ID')
Authorship.many_to_one('authorID', Author, 'ID')

arena = Arena()
arena.register_all(globals())
```

The deployer would write in a .conf file:

```
[Books]
Class: dejavu.storage.storepygresql.StorageManagerPgSQL
Connect: host=localhost dbname=bookstore user=postgres password=****
```

[User Guide, Dejavu v. 2.0, alpha \(30 May 08\)](#)

To create the tables:

```
for cls in (Author, Publisher, Book):
    arena.create_storage(cls)
```

The app developer's runtime code reads as follows:

```
box = arena.new_sandbox()
ppython = Book(title='Programming Python', price=20,
               publishDate=datetime.datetime(2001, 3, 1))
# This next line is redundant; all properties default to None.
# But explicitness is rarely a bad thing.
ppython.publisher = None
box.memorize(ppython)

print ppython.title # output: 'Programming Python'

mlutz = Author(name = 'Mark Lutz')
box.memorize(mlutz) # give mlutz an ID
ppython.addAuthor(mlutz)

print len(ppython.Authorship()) # output: 1
print ppython.author_names() # output: 'Mark Lutz'

oreilly = Publisher(name="O'Reilly")
box.memorize(oreilly) # give oreilly an ID

ppython.publisher = oreilly.ID
print ppython.Publisher().name # output: "O'Reilly"

print len(oreilly.Book()) # output: 1

print 'Hi,', oreilly.Book().author_names() # output: "Hi, Mark Lutz"
```

[1] Fowler, [Patterns of Enterprise Application Architecture](#).

[2] Dejavu relies upon bytecode hacking to achieve its clean lambda syntax for data queries. Therefore, it is CPython-specific. In addition, the bytecode of Python may change from one version of Python to another; if you find your version of Python does not work with Dejavu's `codewalk` and `logic` modules, please let me know.

Application Designers: Using Dejavu to Construct a Domain Model

Units

When constructing a Domain Model for your application, you will want to distinguish between objects that will be persisted and objects that will not. By registering a subclass of `dejavu.Unit`, you allow instances of that subclass to be persisted.

Before you can register your Unit class, you must create it:

```
import dejavu
class Printer(dejavu.Unit): pass
```

This is all you need for a fully-functioning Unit class. There are no methods or attributes that you are required to override; simply subclass from `Unit`. However, this is a fairly uninteresting class. It automatically has an ID property, but doesn't provide any functionality other than what `Unit` already provides. The first thing we will probably want to add to our new class is persistent data.

UnitProperty

Once you have defined a persistent class (by subclassing `Unit`), you need to make another decision. Rather than persist the entire object `__dict__`, you specify a subset of persistent attributes by using `UnitProperty`, a data descriptor. If you've used Python's builtin `property()` construct, you've used descriptors before.

We might enhance our `Printer` example thusly:

```
from dejavu import Unit, UnitProperty
class Printer(Unit):
    Manufacturer = UnitProperty(unicode)
    ColorCopies = UnitProperty(bool)
    PPM = UnitProperty(float)
```

This adds three persistent attributes to our `Printer` objects, each with a different datatype. In addition, every subclass of `Unit` inherits an 'ID' property, an `int`.

When you get and set `UnitProperty` attributes, they behave just like any other attributes:

```
>>> p = Printer()
>>> p.PPM = 25
>>> p.PPM
25.0
```

However, you will notice right away that the `int` value we provided has been coerced to a float behind the scenes. This is because we specified the PPM attribute as a 'float' type when we created it. The value of a Unit Property is restricted to the type which you specify. The only other valid value for a Unit Property is `None`; any Property may be `None` at any time, and in fact, all Properties are `None` until you assign values to them:

```
>>> p.ColorCopies is None
True
```

datetime.datetime

If you use `datetime.datetime` for the type of a `UnitProperty`, most `StorageManagers` will throw away the microseconds. This is an unfortunate oversight that should be corrected sometime in the future.

[User Guide, Dejavu v. 2.0, alpha \(30 May 08\)](#)

Unit ID's

All Units possess an `identifiers` attribute, a tuple of their `UnitProperty` names which define the uniqueness of a Unit. The `Unit` base class possesses a single `UnitProperty`, an `int` named 'ID', and its `identifiers` attribute is therefore ('ID',). That's a string in the tuple; older versions of Dejavu used a reference to the actual `UnitProperty` class instead. If you wish to use identifiers of a different number, types, or names, simply replace the `identifiers` attribute in your subclass:

```
class Printer(Unit):
    # Set ID to None to remove the ID property from this subclass.
    ID = None

    Model = UnitProperty(unicode)
    UnitNumber = UnitProperty(int)
    identifiers = ('Model', 'UnitNumber')
```

Every Unit should possess at least one identifier. This ensures that each Unit within the system is unique. You should consider any `UnitProperty` which is one of the identifiers to be read-only after a Unit has been memorized. Extremely rare applications (like write-only log tables) are allowed to use an empty `identifiers` tuple, but in most OLTP/OLAP scenarios, all your Units should have at least one identifier property.

Creating and Populating Properties

In addition to defining `Unit Properties` within your class body, you can define them after the class body has been executed via the classmethod `Unit.set_property()`. For example, the following two classes are equivalent:

```
class Book(Unit):
    Content = UnitProperty(unicode)

class Book(Unit): pass
Book.set_property('Content', unicode)
```

Declarations outside of the class body allow more dynamic setting of `Unit properties`. You can define multiple properties at once via the `Unit.set_properties()` classmethod:

```
class Book(Unit): pass
Book.set_properties({'Content': unicode,
                    'Publisher': unicode,
                    'Year': int,
                    })
```

You also have options when populating `Unit Properties`. The standard way is simply to reference them as normal Python instance attributes. However, you may also use the `adjust()` method to modify multiple properties at once; pass in keyword arguments which match the properties you wish to modify. Keyword arguments also work when instantiating the object. For example, the following three code snippets are equivalent:

```
pub = Book()
pub.Publisher = 'Walter J. Black'
pub.Year = 1928

pub = Book()
pub.adjust(Publisher='Walter J. Black', Year=1928)

pub = Book(Publisher='Walter J. Black', Year=1928)
```

Unit Properties are First-Class Objects

Like many descriptors, Unit Properties behave differently when you access them from the class, rather than from an instance as above. When calling them from the class, you receive the `UnitProperty` object itself, rather than its value for a given instance. That is,

```
>>> c = Printer.ColorCopies
>>> c
<dejavu.UnitProperty object at 0x01112970>
```

This is significant, because it allows us to store metadata about the property itself:

```
>>> c.type, c.index, c.hints, c.key
(<type 'bool'>, False, {}, 'ColorCopies')
```

When you define a `UnitProperty` instance, you can pass in these extra attributes. Its signature is `UnitProperty(type=unicode, index=False, hints={}, key=None, default=None)`. Supply any, all, or none of them as needed. The `key` attribute is merely the property's canonical name, and is usually set for you. The `index` value tells database Storage Managers whether or not to index the column (if they do any indexing). The `type` attribute limits property values to instances of that type (or `None`). Finally, the `hints` dictionary provides hints to Storage Managers to help optimize storage. If you write a custom Storage Manager, you may define and use your own hints. Here are the ones that most builtin SM's understand:

Key	Values	Description
bytes	$b \geq 0$	Inform SMs that a particular property will never exceed b bytes. This applies to <code>long</code> and <code>int</code> properties, as well as <code>str</code> and <code>unicode</code> . A value of 0 implies no limit. If not specified, a default maximum will be used. Many database backends default to 255; non-database backends often have no limit. Check your Storage Manager.
scale	$s \geq 0$	Scale is the number of digits to the right of the decimal point in a NUMERIC (fixedpoint or decimal) field. This hint informs SMs that would usually store such data at a default scale (usually 2), that the property should use a different scale.
precision	$p \geq 0$	Precision is the total number of decimal digits in a NUMERIC (fixedpoint or decimal) field, or the total number of binary digits in a <code>float</code> field. This hint informs SMs that the property will never exceed p digits. If missing, the StorageManager will supply a default maximum precision. For example, PostgreSQL can handle 1000 decimal digits. If explicitly set to 0 (zero), the StorageManager will allow unlimited precision, if possible. Note that the <code>fixedpoint</code> module uses the word "precision" where we use the word "scale"; it actually has unlimited precision (as we use the word). The <code>decimal</code> module, in contrast, uses limited precision but no scale.

Triggers

Triggers are behaviors which fire when the value of a Unit Property is changed. You can override a `UnitProperty`'s `__set__` method to achieve this in Dejavu. For example:

```
class DatedProperty(UnitProperty):
    def __set__(self, unit, value):
        UnitProperty.__set__(self, unit, value)
        unit.Date = datetime.date.today()
        parent = unit.Forum()
        if parent:
            parent.Date = unit.Date
```

```

class Topic(Unit):
    Date = UnitProperty(datetime.date)
    Content = DatedProperty()
    ForumID = UnitProperty(int)

class Forum(Unit):
    Date = UnitProperty(datetime.date)

Topic.many_to_one('ForumID', Forum, 'ID')

```

In this example, whenever `topic.Content` is set, the `__set__` method will be called and the object's `Date` attribute will be modified. Then, the `Topic`'s parent `Forum` is looked up and *its* `Date` is modified.

As with any trigger system, you need to be careful not to have triggers called out of order. For example, if a user changes both the `ForumID` and `Content` properties in a single operation (like a web page submit), the old `Forum` will be incorrectly modified if the `Content` property is applied first. I don't have any cool tools built into Dejavu to help you with this, but I'm open to suggestions.

TriggerProperty

There is also a `TriggerProperty` class, which overrides `__set__` for you. If the value in question changes (and the unit has a `sandbox`), then the `on_set(unit, oldvalue)` method will be called. Override it in your subclass like this:

```

class NoteContentProperty(TriggerProperty):

    def on_set(self, unit, oldvalue):
        unit.LastModified = datetime.date.today()

```

Note that, if you need to know what the *new* value is, it's already been set on the unit.

Registration of Unit Classes

In addition to defining your `Unit` class, you must also register that class with your application's `StorageManager`. Each class which you want Dejavu to manage must be passed to `store.register(cls)`. If you create a module with multiple classes, you can register them all at once with `store.register_all(globals())`. It will grab any `Unit` subclasses out of your module's `globals()` (or any other mapping you pass to `register_all`) and register them. It then returns a list of the classes it found.

The `register` and `register_all` methods also register any `Associations` you have defined between `Units`.

If you're using multiple `StorageManagers` in a network, you must register classes for each of them. You can inspect which classes have been registered to a given store via `store.classes`, a set. You shouldn't manipulate this structure on your own; use `register` or `register_all` instead.

Each `StorageManager` object also manages the associations between `Unit` classes in its `associations` attribute, which is a simple, unweighted, undirected graph. Whenever you register a `Unit` class, the `SM` will add its associations to this graph. The only other common operation is to call `associations.shortest_path(start, end)`, to retrieve the chain of associations between two `Unit` classes.

Synchronizing the Model

Any database code in a general-purpose programming language will eventually have to come to terms with the gap between native code types and native storage types. In most cases for us, this means

[User Guide, Dejavu v. 2.0, alpha \(30 May 08\)](#)

matching Python types (like `int` and `datetime`) to database types (like `INT8` and `TEXT`). Dejavu provides this layer for databases by using a mapping layer between your model code (Unit classes) and the underlying tables and columns. The implementation of that is unimportant (and possibly storage-dependent), but Dejavu needs to know the database types in effect in order to translate data safely.

When you start your application, you need to call `store.map_all(conflicts='error')` *after* you have registered all of your Unit classes (but before you attempt to execute commands on them).

If your application has created all of its own tables using Dejavu, then there is generally nothing to worry about in terms of the "type gap"; Dejavu will default to creating columns of the types it knows best, and you may be able to set the store's `auto_discover` attribute to `False` and reduce application start-up time (Dejavu will use a mock mapping layer in this case, based on your model). But if you are building a Dejavu interface onto an existing database, or if you customize/optimize your database by hand, then you should leave it set to `True` (the default) for safety's sake.

Associations between Unit Classes

Once you've put together some Unit classes, chances are you're going to want to associate them. Generally, this is accomplished by creating a property in the `Unit_B` class which stores IDs of `Unit_A` objects (which might be called *foreign keys* in a database context).

```
class Archaeologist(Unit):
    Height = UnitProperty(float)

class Biography(Unit):
    ArchID = UnitProperty(int)
    PubDate = UnitProperty(datetime.date)
```

In this example, each `Biography` object will have an `ArchID` attribute, which will equal the `ID` of some `Archaeologist`. In Dejavu terms, we say that there is a *near class* (with a *near key*) and a *far class* (with a *far key*). Associations in Dejavu are not one-way, so it doesn't matter which class you choose for the "near" one and which for the "far" one.

You could stop at this point in your design, and simply remember what these keys are and how they relate, and manipulate them accordingly. But Dejavu allows you to explicitly declare these associations:

```
Archaeologist.one_to_many('ID', Biography, 'ArchID')
```

You pass in the the near key, the far class, and the far key. There are similar methods for `one_to_one` and `many_to_one`. In addition, there is a `Unit.associate` method which allows you to use your own relationship objects.

What does an explicit association buy for you? First, you can [join](#) Units without having to remember which keys are related. Second, `StorageManagers` discover associations and fill the `store.associations` registry, so that smart consumer code (like [Unit Engine Rules](#)) can automatically follow association paths for you. Third, each Unit class has a private `_associations` attribute, a `dict`. Each Unit involved in in the association gains an entry in that `dict`: the key is the far class name, and the value is a `UnitAssociation` instance, a non-data (method) descriptor, with additional `nearClass`, `nearKey`, `farClass`, `farKey`, and `to_many` attributes.

`Unit.add()`

Once two classes have been associated, you attach Unit *instances* to each other by equating their associated properties. That was a mouthful. Here's an example:

```
>>> evbio = Biography()
>>> evbio.ArchID = Eversley.ID
```

The two unit *instances* (evbio and Eversley) are now associated (only their *classes* were before). Keep in mind that many Unit instances need to be memorized in order to obtain an ID.

Rather than forcing you to remember all of the related classes and keys, Dejavu Units all have an add method, which does the same thing:

```
>>> evbio = Biography()
>>> evbio.add(Eversley)
```

The add method works in either direction, so you could just as well write:

```
>>> evbio = Biography()
>>> Eversley.add(evbio)
```

The add method will take any number of unit instances as arguments, and add each one in turn. That is:

```
>>> evbio1 = Biography()
>>> evbio2 = Biography()
>>> evbio3 = Biography()
>>> Eversley.add(evbio1, evbio2, evbio3)
```

"Related units" methods

To make querying easier, each of the two Unit classes involved in an association will gain a new "related units" method which simplifies looking up related instances of the other class. The new method for Unit_B will have the name of Unit_A, and vice-versa. In our example:

```
>>> Archaeologist.Biography
<unbound method Archaeologist.related_units>

>>> Eversley = Archaeologist(Height=6.417)
>>> Eversley.Biography
<bound method Archaeologist.related_units of <__main__.Archaeologist
object at 0x011A1930>>

>>> bios = Eversley.Biography()
>>> bios
[<arch.Biography object at 0x01158E10>,
 <arch.Biography object at 0x0118B350>,
 <arch.Biography object at 0x0118B170>]
>>> evbio1.Archaeologist()
<__main__.Archaeologist object at 0x011A1930>
```

We've only created three Biographies at this point, so we can print the list easily. At the other extreme (when you have hundreds of Biographies to filter), you can pass an optional `Expression` object or keyword arguments to the "related units" method, just like you can with [recall](#). When you do, the list of associated Units will be filtered accordingly.

Notice that, because our relationship is one-to-many, **the two "related units" methods behave differently**. The "one" (Archaeologist) which is retrieving the "many" (Biography) retrieves a list. The "many" retrieving the "one" retrieves a single Unit. When retrieving "to-one", the result will always be a single Unit (or None if there is no matching Unit). When retrieving "to-many", the result will always be a list, (it will be empty if there are no matches).

Because the "related units" method names are formed automatically, you need to **take care not to use the names of Unit classes for your Unit properties**. In our example, we used "ArchID" for the name of our "foreign key". If we had used "Archaeologist" instead, we would have had problems; when we associated the classes, the *property* named "Archaeologist" would have collided with the *related units method* named "Archaeologist". Be careful when naming your properties, and plan for the future. The best approach is probably to end your property name with "ID" every time.

Unlike some other ORM's, Dejavu doesn't cache far Units within the near Unit. Each time you call the "related units" method, the data is recalled from your Sandbox. It is quite probable that those far Units are still sitting in memory in the Sandbox, but they're not going to persist in the near Unit itself in any way.

Finally, some of you may want to override the default "related units" methods. Feel free; `Unit.associate` takes two optional arguments, which should be subclasses of the `UnitAssociation` descriptor. See the source code for more information.

Custom Unit Associations

Sometimes you need an association between two classes that is more complicated. For example, you might have an `Archaeologist` object and want to retrieve just their *last* Biography. Here's an example of how to do this:

```
class LastBiographyAssociation(dejavu.UnitAssociation):
    """Unit Association to relate an Archaeologist to their last Biography."""

    to_many = False
    register = False

    def related(self, unit, expr=None, **kwargs):
        bios = unit.Biography(expr, order=["PubDate DESC"], **kwargs)
        try:
            return bios.next()
        except StopIteration:
            return None

descriptor = LastBiographyAssociation(u'ID', Biography, u'ID')
descriptor.nearClass = Archaeologist
Archaeologist._associations["Last Biography"] = descriptor
```

There are a couple of things to note, here. We are basically doing by hand what the `associate` method does for you automatically, but that method makes *two* associations (one in each direction), and we're only making one. The `related(unit, expr, **kw)` method is overridden to do the actual lookup of far units. Because the `to_many` attribute is `False`, `related` returns a single Unit, or `None`. Finally, the `register` attribute, when `False`, keeps the store from registering this association in its graph (see [Registration](#), above).

Managing Schemas

Conflicts

Dejavu helps you make a *model* (in Python code) that matches some *reality* (like an RDBMS, file, or cache) elsewhere. Because both the model and reality can change independently, you'll find *conflicts* between them from time to time. The most common occurrence of such conflicts is during a call to `map_all`, since it tries to match up your entire model to reality. Similar conflicts arise whenever you ask Dejavu to make changes to reality: add an index, drop storage, or rename a property.

When conflicts may occur, Dejavu adds a `conflicts` argument to the method arguments. The value you supply for this argument tells Dejavu what to do if a conflict arises:

- **error**: This is the default value. `MappingError` is raised for the first conflict and the call is aborted.
- **warn**: `StorageWarning` is raised (instead of an error) for each issue, and the call is not aborted. This allows you to see all errors at once, without having to stop and fix each one and then execute the call again.

- **repair**: Each issue will be resolved by changing the database to match the model. Not all calls support this mode for all errors; any which do not support this mode will error instead.
- **ignore**: Any model conflicts are silently ignored. Use of this mode causes mandelbugs. You have been warned.

Installation

Since this procedure typically happens once per deployed application, Dejavu doesn't try to over-engineer it. But the deployer will still have to go through an installation step at some point. Dejavu offers minimal library calls on top of which you can then build installation tools (and upgrade, and uninstall tools).

For example, a simple install process could look like this:

```
elif cmd == "install":
    store.log = getlogger(os.path.join(os.getcwd(), localDir, "install.log"))
    store.logflags = logflags.ERROR + logflags.SQL + logflags.SANDBOX

    print "Creating databases...",
    store.create_database()
    print "ok"

    print "Creating tables...",
    store.map_all(conflicts='repair')
    print "ok"

    sys.exit(0)
```

In addition to `create_database(conflicts='error')`, all Storage Managers also have a `drop_database(conflicts='error')` method.

Modifying Storage Structures

The `StorageManager` class has some methods to help you make changes to keep storage structures in sync with changes to your Unit classes. For example, let's say that we deploy our Archaeology-Biography application at various libraries around the world. After a year, one of the developers wishes to implement a new reporting feature; however, it would be easiest to build if the Unit Property names could be exposed to the users. Unfortunately, our "ArchID" property on the Biography class isn't very informative. It would be better if we could rename that to "ArchaeologistID":

```
store.rename_property(Biography, "ArchID", "ArchaeologistID")
```

Assuming we've already made the change to our model, the above example renames the property in the persistence layer (the database) using the `rename_property(cls, oldname, newname, conflicts='error')` method. Additional `StorageManager` methods:

Unit classes (tables):

- `create_storage(cls, conflicts='error')`
- `has_storage(cls)`
- `drop_storage(cls, conflicts='error')`

Unit properties (columns):

- `add_property(cls, name, conflicts='error')`
- `has_property(cls, name)`
- `drop_property(cls, name, conflicts='error')`

Unit property (column) indices:

- `add_index(cls, name, conflicts='error')`
- `has_index(cls, name)`
- `drop_index(cls, name, conflicts='error')`

Upgrading: Schema Objects

The Schema class helps you manage changes to your Dejavu model throughout its lifetime. Taking our `rename_property` example from above, we can rewrite it in a Schema objects like this:

```
class ArchBioSchema(dejavu.Schema):

    guid = 'da39a3ee5e6b4b0d3255bfef95601890afd80709'
    latest = 2

    def upgrade_to_2(self):
        self.store.rename_property(Biography, "ArchID", "ArchaeologistID")

abs = ArchBioSchema(store)
abs.upgrade()
```

The example declares this change to be "version 2" of our schema. If you examine the base Schema class, you will see that it already has an `upgrade_to_0` method. The "zeroth" upgrade makes no schema changes; it merely marks all deployed databases with "version 0". I skipped version 1 in the example, just in case I need some setup code in the future ;).

If you call `schema.upgrade(version)` with a version argument, then your deployment will be upgraded to that version. If no argument is given, the installation will be upgraded to `schema.latest`. You can even skip steps (i.e. remove methods for broken steps) if it comes to that.

Each Schema also has a `stage` attribute. While an upgrade is in process, this value will be an int, the same number as that of the upgrade method. That is, while `upgrade_to_2` is running, `stage` will be 2. If no upgrade method is running, `stage` will be None.

After you run `upgrade`, you can call the `assert_storage` method of the Schema object to tell Dejavu to create storage (tables in your database) for all the Unit classes registered in your store. If storage already exists for a given class, it is skipped.

Please note: the installed version defaults to "latest". This allows new installs to skip all the upgrade steps, and just use the latest class definitions when they call `assert_storage`. However, it means that if you deploy your apps for a while without a Schema, and then introduce one later, you must manually decrement `DeployedVersion` from "latest" to the actual deployed version **before** running your app for the first time (or things will break due to the difference between the latest and deployed schema).

Versions: The DeployedVersion Unit

The Schema class uses a magic table in the database to keep track of each deployment's schema version. The Unit class is called "DeployedVersion", and it has ID and Version attributes.

The ID attribute will be set to whatever your Schema.guid is. It's a simple way to isolate multiple installed Dejavu applications. A given application should use the same guid throughout its lifetime. I used `sha.new().hexdigest()` to generate the example. Feel free to use `sha.new`, a guid generator, a descriptive name, or whatever you like.

Automatic Unit Classes

When you create your first Dejavu model, you might be forming it to match some existing database schema. If so, Dejavu has a `Modeler` tool to help you inside `dejavu.storage.db`.

The `make_class(tablename, newclassname=None)` method finds an existing Table by name and returns a subclass of Unit which models that table. By default, the new class will have the same name as the database table; supply the 'newclassname' argument to use a different name (for example, to capitalize the class name).

```
>>> sm = storage.resolve('mysql',
                        {"host": "localhost", "db": "existing_db",
                         "user": "root", "passwd": "xxxx",
                        })
>>> from dejavu.storage import db
>>> modeler = db.Modeler(s.db)
>>> Zoo = modeler.make_class("zoo", "Zoo")
>>> Zoo
<class 'dejavu.storage.db.Zoo'>
>>> Zoo.properties
['id', 'name', 'admission', 'founded', 'lastescape', 'opens']
```

The `make_source(tablename, newclassname=None)` method does the same thing as `make_class`, but returns a string that contains valid Python code to generate the requested Unit class:

```
>>> print modeler.make_source("exhibit", "Exhibit")
class Exhibit(Unit):
    pettingallowed = UnitProperty(bool)
    animals = UnitProperty(str)
    name = UnitProperty(str)
    zooid = UnitProperty(int)
    acreage = UnitProperty(decimal.Decimal)
    creators = UnitProperty(str)
    # Remove the default 'ID' property.
    ID = None
    identifiers = ('name', 'zooid')
    sequencer = UnitSequencer()
```

Finally, you can perform this sort of modeling on *all* tables in a database at once with the `all_classes()` and `all_source()` methods of the `Modeler`. These simply iterate over all of the known tables and return the results in a list instead of a single value.

Application Developers: Managing Units

Sandboxes

During the life of a client connection, your application should create and use a `Sandbox` to manage the set of "live" Units. A `Sandbox` manages the in-memory lifecycle of Units: creation, identity, mutation, and destruction. Sandboxes route persistence operations on Units to the correct Storage Manager.

You can create `Sandbox` objects directly. They take a single argument, a `StorageManager` object. All Storage Managers also provide a convenience function, `new_sandbox`, which does this for you. The following lines are equivalent:

```
box = dejavu.Sandbox(store)
```

```
box = store.new_sandbox()
```

You might often choose the latter when you have a reference to the store, and would rather avoid importing `dejavu` yet again just to obtain the `Sandbox` class.

Memorizing Units

When you create a Unit instance, it exists in isolation. There is no connection between that Unit and storage; your Unit will not be persisted, because `Dejavu` doesn't yet possess a reference to your Unit. To provide that link, you *memorize* your Unit (or rather, you tell your `Sandbox` to memorize it):

```
class Publisher(Unit):
    City = UnitProperty(unicode)

p = Publisher(ID='Walter J. Black')
box.memorize(p)
```

Memorization does several things. First, it places your new Unit into your store. That Unit instance will now be persisted by the appropriate Storage Manager. It can be recalled from storage when needed, using the built-in Expression syntax. It may have been given an ID (see [Sequencing](#), below). Memorization also makes your Unit *concrete*; that is, your Unit will now possess a `sandbox` attribute. Units whose `sandbox` attribute is not set (is `None`) have no relationships, and their Unit Property triggers (if any) will not fire.

You may define special methods on your Units to provide start-of-life behaviors. If a Unit possesses an `on_memorize` method, it will be called after the Unit has been 'reserved' in storage and placed in the `Sandbox` cache.

Sequencing

Every Unit has one or more identifiers. The default ID property is of type `int`; however, you can override that to whatever type you like. As long as you provide your own identifier values for Units, nothing will break--you can memorize and recall Units without problems. However, if you memorize a Unit with an ID of `None`, the store may attempt to provide an ID for it.

The `Unit` base class possesses a `sequencer` attribute to help Sandboxes generate new IDs. The default value is an instance of `UnitSequencerInteger`, which examines all existing Units, finds the maximum integer ID, adds 1, and uses that value for the new ID. [StorageManagers are free to optimize any sequencer, whether builtin or custom, in order to take advantage of ID-generation tools in the database or other storage provider. All builtin database StorageManagers optimize `UnitSequencerInteger` in this way.]

The other useful Sequencer is the base class `UnitSequencer`, which simply raises an error when asked to generate an ID. If you set `ClassA.ID` to a string or unicode type, you'll probably want to set `ClassA.sequencer = dejavu.UnitSequencer()`, and form ID values in your own code.

Recalling

Once you have memorized a Unit or two, you will probably want to recall them at some point. Sandboxes possess four member functions to accomplish this.

recall()

First, the appropriately named `recall(cls, expr=None, order=None, limit=None, offset=None)` function. This is the full-blown query method. As a first argument, you pass it the class (**not** the name of the class, but the actual class) of which you expect to retrieve instances. The second argument should be a lambda or an instance of `dejavu.logic.Expression` (an object which encapsulates your specific query, see [Querying](#)). Alternately, you may supply a dict, which will then be combined into an Expression for you (using [logic.filter](#)). The following three examples are equivalent and all result in the same output:

```
>>> units = box.recall(Book, dict(Year=1928))
>>> units = box.recall(Book, lambda x: x.Year == 1928)
>>> units = box.recall(Book, logic.Expression(lambda x: x.Year == 1928))
>>> [x.Title for x in units]
[u'The Giant Horse of Oz', u'Kai Lung Unrolls His Mat',
 u'Tarzan, The Lord of the Jungle']
```

If you do not supply an expression, all Units of the given Unit class will be retrieved in a list.

If your Unit class defines an `on_recall()` method, it will be called when each Unit has been loaded from storage (at the end of the recall process). Once the unit is loaded into a Sandbox, however, `on_recall` will not be called again; it's only called at the Sandbox/SM boundary. If `on_recall` raises `UnrecallableError`, the unit will not be yielded back to the caller, nor placed in the Sandbox cache.

Recalling multiple classes at once (JOINS)

In addition to providing a single class to `recall`, you have the option of providing a tree of classes, a nested set of `UnitJoin(class1, class2, leftbiased=None)` instances.

The "leftbiased" argument specifies how the results will be joined:

leftbiased	Join Type	Description	Operator
None	Inner Join	All related pairs of both classes will be returned.	& OR +
True	Left Join	All related pairs of both classes will be returned. In addition, if any Unit in class1 has no match in class2, we return a single row with Unit1 and a "null Unit" (a Unit, all of whose properties are None).	<<
False	Right Join	All related pairs of both classes will be returned. In addition, if any Unit in class2 has no match in class1, we return a single row with a "null Unit" (a Unit, all of whose properties are None) and Unit2.	>>

Look hard? Fear not. There's a **much** easier way to join units than writing a big tree of UnitJoins. Use the `&`, `<<`, and `>>` operators directly with Unit classes:

```
tree = (Book << Publisher) & Author
```

This example will automatically produce a UnitJoin tree for you, with Book 'left joined' to Publisher, and then 'inner joined' to Author.

When you provide multiple classes, the `recall` method returns a list of rows. Each row will be a list of units, one per class in the `classes` arg. The `expr` arg should be a lambda or `logic.Expression` which can evaluate all of the units in any given row at once (you cannot use a dict expr with multiple classes).

```
for pub, book in box.recall(Publisher & Book, lambda p, b: p.ID == 4)
```

This example will retrieve a series of [Publisher, Book] pairs. Note that all three constructs (the UnitJoins, the lambda arguments, and the resulting rows) have the same classes listed in order from left to right.

In database terminology, this technique performs a series of joins between each pair of classes in your UnitJoin tree. However, repeated units in the results will reference the same object; in the example above, each "pub" unit will be the same object, since we limited that expression to a single Publisher. So we might retrieve multiple (pub, book) pairs, but the first unit in each pair will be the same unit instance.

The relationships (joins) between each class are specified by [Unit Associations](#).

xrecall()

Just like `recall`, but returns an iterator instead of a list. Use `xrecall` to load Units in a more lazy fashion.

unit()

The `recall` method can be verbose. When you want a one-liner and only expect a single Unit, use the `unit(cls, **kw)` method of Sandboxes. Again, you pass the class of Units you wish to retrieve as the first argument. Then, supply keyword arguments of the form "property_name=value". The method will form an equivalent Expression for you from the keyword args. For example:

```
>>> book = box.unit(Book, ID=1)
>>> if book:
...     print book.Title
u'Ladies in Hades'
```

If no Unit can be found that matches the criteria, `None` is returned. If multiple Units match the criteria, only the first one is returned (although the rest may be loaded into memory).

The `unit` method is heavily optimized (in both the sandbox and all stores) for retrieving a single Unit by its identifiers. When using key-value stores like [memcached](#) in your storage manager network, calling `unit` may be much faster than `recall`, even for multiple units.

"Magic recaller" methods

For each class you have registered with your store, the Sandbox will have a "magic recaller" method of the same name, to make single-unit lookups easier. Instead of the above example for `box.unit()`, we might just as well have written:

```
>>> book = box.Book(1)
```

Note that for the magic methods, unlike for the `unit` method, you may pass identifiers as positional arguments. If the class has multiple identifiers, you should probably stick to keyword arguments; otherwise, you must remember the order of the class' identifiers tuple.

Forgetting and Repressing

To *forget* a Unit is to destroy it forever. You have two options for forgetting Units: you can call `sandbox.forget(unit)` or the simpler version, `unit.forget()`. Either of these will clear the Unit from the Sandbox' cache, and the Sandbox will tell the appropriate Storage Manager to destroy the stored Unit data. If a Unit has not yet been memorized, you do not need to forget it.

[User Guide, Dejavu v. 2.0, alpha \(30 May 08\)](#)

In some circumstances, you may wish to only clear the Unit from the Sandbox without destroying it. You can do this by calling either `sandbox.repress(unit)` or the simpler version, `unit.repress()`.

You may define special methods on your Units to provide end-of-life behaviors. If a Unit possesses an `on_forget` method, it will be called after the Unit has been destroyed. If a Unit possesses an `on_repress` method, it will be called *before* the Unit has been repressed. I'm sure there was a good reason for this disparity, but I've forgotten (or perhaps repressed) it.

Be aware that many of the things you put in an `on_repress` handler might also need to go into `on_forget`. The one doesn't call the other automatically, because sometimes you *don't* want the same behavior.

Flushing Sandboxes

When the client connection has closed, you should *flush* the Sandbox caches. In general, a single call to `sandbox.flush_all()` will do the trick. Notice that `flush_all()` calls any `on_repress()` handler for each Unit in the Sandbox.

Warning: You should **NOT** call `flush_all()` indiscriminately. You will rapidly get into concurrency trouble. You can stay out of trouble following an easy rule: call `flush_all` only at the end of your client's connection.

If you want the "hard" rule, here it is. If you flush any Unit class, then any instances of that class hanging around need to be re-recalled. If you don't re-recall them, then any changes you make to the old instance won't be saved on the next flush, since flushing only iterates through units in the sandbox. For example, if you do this:

```
box = store.new_sandbox()
thing = box.unit(Thing)
box.flush_all()
thing.Size += 12
box.flush_all()
```

...then the change you make to "Size" won't be persisted, since the Thing object is no longer in the sandbox--it's been flushed out. You have to recall it somehow to get it stuck in the sandbox again. You could go through all kinds of gyrations to save the old units directly to storage, but don't bother. Just get new references to them and save yourself a lot of headache.

Views

Sandboxes provide a `view(query, distinct=False)` function. This works like `recall`, but returns values, rather than Units. The 'query' argument should be an instance of `dejavu.Query`, or more commonly, a 3-tuple of (cls, attrs, expr). Put simply, it yields all values for the given attribute(s) of the Unit class provided; each unit will yield a tuple of its values in the same order as the `attrs` sequence you provide. Providing an `expr` argument (a lambda, an `Expression` object, or a dict, see below), will filter the set of Units before obtaining the value tuples.

```
>>> v = sandbox.view((zoo.Animal, ['Name', 'Lifespan']))
>>> [row for row in v] # or list(v), or iterate over v...
[('Leopard', 73.5),
 ('Slug', .75),
 ('Tiger', None),
 ('Lion', None),
 ('Bear', None),
 ('Ostrich', 103.2),
 ('Centipede', None),
 ('Emperor Penguin', None),
 ('Adelie Penguin', None),
 ('Millipede', None)
]
```

In this example (pulled from the "zoo" test suite), we grab the name and Lifespan for each Animal. The `attrs` argument must always be an iterable.

If the 'distinct' argument is True, `view` returns distinct tuples rather than all tuples.

The `view` function can also be used as a count function by passing `attrs = cls.identifiers` and setting 'distinct' to True. Sandboxes provide a `count(cls, expr=None)` method which does just this.

There are two additional sandbox methods for aggregates: `range` and `sum`. The `range(cls, attr, expr=None)` method takes a single attribute and returns the closed interval `[min(attr), ..., max(attr)]`. The `sum(cls, attr, expr=None)` method also takes a single attribute and returns the sum of all non-None values for the given `cls.attr`.

xview()

Just like `view`, but returns an iterator instead of a list. Use `xview` to load Unit values in a more lazy fashion.

Transactions

Dejavu supports distributed transactions at all levels (however, it does not yet use distributed two-phase commit! That's planned for later). Most often, your code will call transaction methods on the current sandbox object. When you call `sandbox.start(isolation=None)`, you are telling Dejavu to begin a transaction on all known stores. Note that this will start a transaction on all stores regardless of which classes are registered for each store; Dejavu has no way of knowing beforehand which classes or stores your next statements will affect.

The `isolation` argument to the `start` method is very important. It determines the "isolation level" of the transaction; that is, the degree to which the current transaction can see changes made to a concurrent transaction. The ANSI/ISO SQL92 standard defines four isolation levels, based on three phenomena:

		Level			
		Read Uncommitted	Read Committed	Repeatable Read	Serializable
Phenomena	Dirty Read	Possible	Not possible	Not possible	Not possible
	Fuzzy Read	Possible	Possible	Not possible	Not possible
	Phantom	Possible	Possible	Possible	Not possible

A "dirty read" occurs when TX 1 writes a value and TX 2 is able to read the change before TX 1 commits.

A "fuzzy read" (or "nonrepeatable read" occurs when TX 1 reads a value, TX 2 changes that value and commits, and TX 1 obtains the new value when it re-reads.

A "phantom" occurs when TX 1 reads a set, TX 2 adds to the set, and TX 1 obtains the new rows when it re-reads.

Dejavu supports a variety of stores, and not every store supports every isolation level. See the [comparison chart](#) for details. Some stores, like shelve and RAM, don't support transactions at all. In addition, different stores "prevent" the above phenomena in different ways. In some cases, the phenomena is simply not allowed to exist. In other cases, the phenomena raises an error immediately. In still other cases, the phenomena is prevented by waiting (up to a timeout) until one of the offending transactions completes.

Once you have finished executing statements, you should call `flush_all`, which will call `commit()` for you. Alternately, you may call `rollback()` if you need to ignore your changes.

If you're using a store that supports implicit transactions (also sometimes called "autocommit"), you can skip calling `start` by setting the Database attribute `connections.implicit_trans` to True (it's False by default). This can be done in code or config. See the [comparison chart](#) for details.

Querying

When you retrieve Units, you often don't want to load the entire set for a given class. In Dejavu, you filter the set according to the UnitProperty attributes for each object. Naturally, there must be a way to express the filter you intend. Dejavu actually provides three ways, all in the `dejavu.logic` module: `Expression`, `filter`, and `comparison`.

The `Expression` class

Regardless of which technique you use to express your filter, you're going to end up with a `logic.Expression` object. You can build an Expression directly, passing a single lambda as an argument:

```
>>> from dejavu import logic
>>> import datetime
>>> f = lambda x: x.Date >= datetime.date(2004, 3, 1)
>>> e = logic.Expression(f)
>>> e
logic.Expression(lambda x: x.Date >= datetime.date(2004, 3, 1))
```

Neat, eh? I worked hard on that `__repr__`. ;)

It may be obvious, but we'll be explicit, here. The lambda which you pass into an Expression must possess a positional argument, which will always be bound to a Unit instance. In the example above, it's named 'x', but you can use any name you like. Using lambdas as a base means that we can simply call `e(unit)`, and receive a boolean value indicating whether our Unit "passes the test". Attribute lookups on our 'x' object will apply to Unit Properties for that Unit object. That is, `x.Date` becomes `unit.Date`.

You can also do fancier things with Expressions (although the vast majority of the time, you won't need to in order to use Dejavu):

```

>>> logic.Expression(lambda x, y, z: "Dave" in x.Name and y.Age > 65)
logic.Expression(lambda x, y, z: ('Dave' in x.Name) and (y.Age > 65))
>>> logic.Expression(lambda *units, **kw: units and
...                 (units[0].Width > units[0].Height or
...                 units[0].Color in kw['Colors']))
logic.Expression(lambda *units, **kw: (units) and
...                 ((units[0].Width > units[0].Height) or
...                 (units[0].Color in kw['Colors'])))
>>>

```

Early binding

What is not obvious from the above code snippet is perhaps the **most important aspect** of Expressions: any globals or cell references (from closures) in the supplied lambda get **bound early**. Compare the following disassemblies:

```

>>> import dis
>>> dis.dis(f)
 1          0 LOAD_FAST          0 (x)
          3 LOAD_ATTR          1 (Date)
          6 LOAD_GLOBAL        2 (datetime)
          9 LOAD_ATTR          3 (date)
         12 LOAD_CONST        1 (2004)
         15 LOAD_CONST        2 (3)
         18 LOAD_CONST        3 (1)
         21 CALL_FUNCTION    3
         24 COMPARE_OP        5 (>=)
         27 RETURN_VALUE

>>> dis.dis(e.func)
 1          0 LOAD_FAST          0 (x)
          3 LOAD_ATTR          1 (Date)
          6 LOAD_CONST        6 (datetime.date(2004, 3, 1))
          9 COMPARE_OP        5 (>=)
         12 RETURN_VALUE

```

As you can see, the function itself references the global 'datetime' module. Once we wrap it in the Expression, however, it becomes a constant! Thanks to Raymond Hettinger for inspiring this solution [\[1\]](#). Early binding, however, implies two consequences:

First, any globals or cell references must be present in the lambda's scope when it is passed into Expression(). This is the norm and shouldn't require too much thought from you when you write Expressions. In the example above, we simply imported datetime as you would expect.

Second, any globals or cell references must **also** be present in the logic module when the Expression is unpickled. Pickling occurs when Expressions are sent over sockets, and also if Expressions are themselves persisted to storage (for example, see [Unit Engines](#), below). This means your application must register such global references in logic.builtins (a dict). Note that the logic module already tries to import datetime, fixedpoint and decimal.

External functions within Expressions

Dejavu provides additional functions which can be used in Expressions. For example, you can construct an Expression like:

```
logic.Expression(lambda x: x.Size < 3 and x.Date > today())
```

In this example, the today() function breaks convention and is actually **bound late**. That is, if you construct this Expression now and use it six months later, the value of today() will change. Storage Managers "know about" these dejavu functions, and can use them to build more appropriate queries. Here are the functions supplied by Dejavu:

[User Guide, Dejavu v. 2.0, alpha \(30 May 08\)](#)

Function	Late bound?	Description
<code>icontains(a, b)</code>		Case-insensitive test b in a. Note the operand order.
<code>icontainedby(a, b)</code>		Case-insensitive test a in b. Note the operand order.
<code>startswith(a, b)</code>		True if a starts with b (case-insensitive), False otherwise.
<code>iendswith(a, b)</code>		True if a ends with b (case-insensitive), False otherwise.
<code>ieq(a, b)</code>		True if a == b (case-insensitive), False otherwise.
<code>year(value)</code>		The year attribute of a date. If value is None, return None.
<code>month(value)</code>		The month attribute of a date. If value is None, return None.
<code>day(value)</code>		The day attribute of a date. If value is None, return None.
<code>now()</code>	Y	<code>datetime.datetime.now()</code>
<code>utcnow()</code>	Y	<code>datetime.datetime.utcnow()</code>
<code>today()</code>	Y	<code>datetime.date.today()</code>
<code>iscurrentweek(value)</code>	Y	If value is in the current week, return True, else False.

It is possible for you, the application developer, to define your own external functions by injecting them into the globals of the `logic` module. For example, `logic.odd = lambda unit: (unit.num % 2) == 1`. However, because the builtin Storage Managers are unaware of your new functions, they will not be able to optimize their use; instead, they will simply retrieve a larger set of objects from storage, evaluate each one against the function you provide, and return those Units which match your function. This isn't necessarily a bad thing; it provides the same functionality as if you wrote the test inline within your own code. By making that test a logic function, you also allow it to be stored in Engine *rules* (see [Unit Engines](#), below). You may, of course, create your own Storage Manager which understands your external function (and can translate its logic into, say, SQL), and thereby achieve end-to-end functionality.

Using `filter` to form Expressions

The `logic` module also provides convenient methods to create common types of Expression objects via the `filter` and `comparison` factory functions.

The `filter(**kwargs)` function produces an Expression by taking the keyword arguments you supply, and rewriting them in lambda form. The only operator allowed is therefore the equals '==' operator. For example:

```
>>> logic.filter(Type='Cat', Mutation='Atomic')
logic.Expression(lambda x: (x.Type == 'Cat') and (x.Mutation == 'Atomic'))
```

Using `comparison` to form Expressions

The `comparison(attr, cmp_op, criteria)` function allows you to form Expressions with dynamic operators. This can come in handy when you are constructing Expressions on the fly from user input. For example, a search page might prompt users for an attribute name, an operator, and an operand (the criteria).

Borrowing from `opcode.cmp_op`, the allowed values for our `cmp_op` argument are as follows:

Numeric Value (cmp_op)	Operator
0	<
1	<=
2	==
3	!=
4	>
5	>=
6	in
7	not in
Most SM's only support the following with None:	
8	is
9	is not

Here's an example of using `comparison`:

```
>>> logic.comparison('Name', 3, 'Mr. Kamikaze')
logic.Expression(lambda x: x.Name != 'Mr. Kamikaze')
```

Although the comparison function only allows a single comparison at a time, the resulting Expressions can be combined with the `&` and `|` operators (see next) to produce more complex Expressions.

Combining Expressions

Expressions are combinable; by using the `&` operator, the two expressions are combined with an adjoining logical "and". For example:

```
>>> a = logic.Expression(lambda x: x.Size > 3)
>>> b = logic.Expression(lambda x: x.Size <= 15)
>>> c = a & b
>>> c
logic.Expression(lambda x: (x.Size > 3) and (x.Size <= 15))
```

The `+` operator works just like the `&` operator. The `|` operator combines the two Expressions with a logical 'or'.

When you combine two Expressions with dissimilar argument lists, what happens? The Expression class doesn't really care what the argument names are, just their order, so the names might not come out as you might expect; however, the logic is preserved:

```
>>> f = logic.filter(Name='Bruce')
>>> f
logic.Expression(lambda x: x.Name == 'Bruce')
>>> g = logic.Expression(lambda a, b, **kw: a.Name + b.Surname == kw['Full Name'])
>>>
>>> f + g
logic.Expression(lambda x, b, **kw: (x.Name == 'Bruce')
                    and (x.Name + b.Surname == kw['Full Name']))
>>> g + f
logic.Expression(lambda a, b, **kw: (a.Name + b.Surname == kw['Full Name'])
                    and (a.Name == 'Bruce'))
```

Specifying types for Expression kwargs

Up to now, we've constructed Expression objects with a single argument, the function which we're going to wrap. But Expression objects may take a second argument, called "kwtypes". This argument must be a dictionary of {keyword: type} pairs. Dejavu doesn't do anything internally with this information; it's simply a standard place to keep it for use by your own applications. However, the kwtypes attribute will be persisted when pickling and unpickling Expression objects, a very common operation.

Exporting the logic module

The logic module (and codewalk, on which it is built) isn't limited to Dejavu. Feel free to use it in some other framework or script! The only change you may have to make (if you relocate the module outside of the dejavu package) would be to the single line: `from dejavu import codewalk`, to point to the new location.

In particular, `logic.Expression` objects can operate on *any* Python objects, not just `dejavu.Unit` instances. If you wish to provide additional logic functions (as dejavu does), simply add them to `logic.builtins`.

You may also find the underlying `codewalk` module useful for other purposes on its own. The `Visitor` base class can be very convenient for building bytecode hacks.

To make a long story short, Dejavu depends on `logic` throughout, but the reverse is not true.

Unit Engines

Once you've created and associated your Unit classes, you can begin to write "business logic" code (mostly inside those classes, we hope), and "presentation logic" code (mostly outside those classes). In most cases, you will construct Expressions within your own code manually to retrieve Units. Sometimes, however, you need to persist query parameters from your users; in other cases, you might store a list of Units which match a query (regardless of who formed the necessary Expression). Finally, you might wish to manipulate lists of Units as sets: differences, intersections, and unions. The `engines` module addresses all of these needs.

Collections: Lists of Units

The `UnitCollection` class provides a means of storing a list of Units, or rather, a list of Unit identifier values. You use its `Type` property to indicate the class of the indexed Units. That value should be the **name** of the Unit Class, **not** the class object itself (this is different than most other calls in Dejavu). If you need to retrieve the actual Unit class, call `UnitCollection().unit_class()`.

`UnitCollection` itself subclasses `dejavu.Unit`; you can therefore persist Unit Collections via Dejavu Storage Managers (most SM's, anyway; it's recommended that SM's handle Unit Collections, but not required. Check your SM to see if it does).

Each Collection has a thread lock (an `RLock`, actually) which you should `acquire()` before you add an ID to the set, and `release()` afterward. If you use the `add(ID)` method, this locking is done for you.

When you need to retrieve the actual Units which are indexed by the Collection, call the `units(quota=None)` method, which will look up the Units and return them in a list. Since the Collection only stores identifier values, it is possible that one of the indexed Units may have been destroyed since the list was built. The `units` method simply passes over these "phantom" Units. You can inspect the full list of IDs in the Collection (whether they reference existing Units or not) with the `ids()` method.

Collections also provide a convenience function for grouping Units by attribute: `xdict(attr)`. This function will look up each Unit in the Collection, inspect the attribute that you specify, and return a dictionary of the form `{attr_val1: [Unit, Unit, ...]}`. Each distinct attribute value will have its own key, with a list of matching Units as the value.

Engines

You can form Collections by hand, but a more powerful technique is the `UnitEngine`, a factory for Collections. Engines are very simple: they possess a set of *rules* which are executed when you want to take a *snapshot* of Units. The snapshot which is produced is a `UnitCollection` object. Whenever you call `take_snapshot()`, the Engine will maintain an association to the resulting Collection. You can access past snapshots with the `snapshots()` method.

Engines are themselves Units, and can be persisted via Storage Managers. The only properties they possess are: an `ID`, a `Name`, an `Owner`, a `FinalClassName`, and `Created`, the creation date of the Engine.

The `Owner` property should either be a user name, or one of the reserved names: "Public" and "System". By default, the `permit()` method allows a user read-access to the Engine if they are the Owner, or the Owner is "Public" or "System". Write-access is permitted if the user is the Owner, or the Owner is "Public". Feel free to override `permit()` in a subclass to provide different behaviors.

The `FinalClassName` is set for you as you add Rules to the Engine. You can use the value of this property, for example, to tell your users, "Engine #23569 is an 'Armadillo' engine," when it produces Collections of `Armadillo` Units. The only time you might want to set this value manually is when you first create the Engine, before you have added any Rules.

Rules

Just like Collections and Engines, `UnitEngineRule` is *also* a subclass of `Unit`, and can be persisted via Storage Managers. All three work together to provide a complete, dynamic, application-level query generator.

Okay, so what are Rules? You might say they're a "little language", with the following primitives, or "operations":

Operation	Operand(s)	Description
Operations on a single set		
CREATE	The classname of the new Type	Creates a new Set of the specified Type. All Units of that Type are included in the new Set.
FILTER	A <code>logic.Expression</code>	Removes Units from the current Set which do not match the Expression.
FUNCTION	The name of a function in the <code>store.engine_functions</code> dict	Calls the function, passing the current Set. The function should modify the Set.
TRANSFORM	The classname of the new Type	Transform the current Set into a Set of associated Units (of another Type). The association must be present in the <code>store.associations</code> graph.
RETURN		Optional. If omitted, the last Set handled is returned as the snapshot. If supplied, the ID of the Set to return.
Operations on two sets		
COPY	The Set ID of the new Set	Copies the current Set to a new Set. The current Set is unchanged.
DIFFERENCE	The ID of the Set to mix in	Removes IDs from the current Set which exist in the second Set.
INTERSECTION	The ID of the Set to mix in	Removes IDs from the current Set which <i>do not</i> exist in the second Set.
UNION	The ID of the Set to mix in	Adds any IDs to the current Set which exist in the second Set.

Each Rule has an `Operation` property (a string, one of the above), a `SetID`, and an `Operand`. Here's an example ruleset:

Sequence	Operation	SetID	Operand
1	CREATE	1	"Invoice"
2	FILTER	1	<code>lambda x: x.Date > dejavu.today()</code>
3	CREATE	2	"Inventory"
4	FILTER	2	<code>logic.Expression(lambda z: z.ID < 10)</code>
5	TRANSFORM	2	"Invoice"
6	DIFFERENCE	1	2
7	RETURN	1	

As you can see, every Rule operates on a *Set* of Units. The first rule is always to CREATE a set, declaring it to contain a certain Type of Units. In most cases, you will then FILTER that set. If you simply created a set and then returned it, it would contain all Units of the declared Type. When you filter a set, however, you remove Units from the whole which do not match the filter's Expression.

In the example above, we CREATE a second Set so that we can eventually obtain the DIFFERENCE between Set 1 and Set 2. The second Set contains Units of a different Type than the first. Once we filter

Set 2, we then TRANSFORM it; for each Inventory Unit, we look up associated Invoice Units. Then, we find the difference between the two Invoice sets and RETURN it.

Rules are executed in order according to their `Sequence` attribute (lowest first). When you use the `Engine.add_rule` method, the next `Sequence` value is retrieved for you. Notice that each Rule belongs to one and only one Engine; they are not shared between Engines. Each Rule has its own `EngineID` attribute.

Engine Functions

The FUNCTION rule deserves special mention. The Operand of a FUNCTION rule is a string, a key in the `store.engine_functions` dictionary. When the rule is executed, that key is used to look up the function, which is then called, passing `(sandbox, set)`. The function should mutate the set directly. Use FUNCTION rules to mutate sets in ways which are more complex than those provided by FILTER and TRANSFORM. For example, you might provide a function which removes all but the first Unit in the Set (according to some ordering algorithm).

Analysis Tools

Dejavu includes various tools to help you manipulate groups of Units.

Sorting Units

When you recall Units, you receive a list. If you didn't ask the `recall` method to order the results, you must sort your list in your Python code. Dejavu provides a `sort(attrs)` function to assist you in sorting Units. It returns a function, which you can then use in Python's `sort` function (which operates in place). Continuing our example:

```
people.sort(dejavu.sort(['Size DESC', 'Name']))
```

The most important issue (and the reason we don't just use 2.4's `attrgetter`), is that any Unit property must allow values of `None`, which tends to raise errors when compared to values of other types. The function which `sort` creates for you treats `None` as "less than" any other value.

Cross-tabulation

Cross-tabs (also called *aggregate tables* or *pivot tables*) display aggregate information about objects by category. For example, rather than show a list of Safari records, one row per trip, you might wish to show a table where each row represents a Destination, and each column shows the count of Safaris to that Destination for each distinct Year. In this example, we say that the Safaris are "grouped by" their Destination values, and that we "pivot" on the Year values.

Dejavu helps you form such a table via the `CrossTab` class. You need to specify the group(s) you wish to use, and the pivot attribute. Finally, you must specify the aggregate function. Here's a code example:

```

>>> data = ["a", "b", "cc", "bddd", "a4", "b6"]
>>> group = lambda x: x.isalpha()
>>> pivot = lambda x: x[0]
>>> ctab = analysis.CrossTab(data, [group], pivot, dejavu.COUNT)
>>> data, columns = ctab.results()
>>> data
{(True,): {"a": 1, "b": 2, "c": 1},
 (False,): {"a": 1, "b": 1}}
>>> columns
["a", "b", "c"]

```

You may notice that we're not using Units in our example; the `CrossTab` class is designed to work with any objects. Here's one way to lay out that data:

Is Alpha	a	b	c
Y	1	2	1
N	1	1	0

The `results` method returns two values. First, the table itself in the form of a dictionary; each key is a tuple of group values, and the corresponding value is a sub-dictionary. Each sub-dict has keys which are the pivot attribute, and values which equal the aggregates. I know, that was confusing; look at the example. The second value to be returned is a list of the pivot column values; you'll notice they're sorted.

The groups and pivot arguments may be either strings or functions. If strings, they must be the names of attributes of the source objects. The final `aggfunc` argument defaults to `COUNT`, but may also be `SUM`. More `aggfuncs` may arrive in the future.

[1] Python Cookbook, [Binding Constants at compile time](#)

Deployers: Configuring Storage

The topmost object in Dejavu is a `StorageManager` object. When building a Dejavu application, you must first create a `StorageManager` instance, and must find a way to persist this object across client connections. This can be achieved in multiple ways; web applications, for example, will typically create a single process to serve all requests. Desktop applications will probably create a single `StorageManager` object for each running instance of the program.

Storage Managers insulate an application developer from the specifics of databases, query languages, and cache mechanisms. As the *deployer* of a Dejavu application, you get to be in control of these specifics. But don't worry; in the vast majority of cases, you will set up a single database with just two lines in a configuration file. Often, the application developer will have already prepared default config files which you can simply "plug and play". But if you *need* more control over your data storage, you have it.

When you deploy an app built with Dejavu, you must specify Storage Managers to use for persisting application objects. This is usually done through an ini-style configuration file, although Dejavu itself doesn't currently provide a parser for that. Here's a short example of configuring a store in Python:

```
from dejavu import storage
opts = {'connections.Connect': "PROVIDER=MICROSOFT.JET.OLEDB.4.0;DATA
SOURCE=D:\data\junct.mdb;"}
root = storage.resolve("access", opts)
```

The `storage.resolve(store, options=None)` call tells Dejavu the *class* of SM we'd like to use. For most applications, you'll decide which class to use based on the database you want to use. Our example declares that we want to persist our application data in an "MS Access" (i.e., Jet) database. You may supply a known short name (like "sqlite") or the full dotted-package name. `storage.managers` is a dict of short names to full classes (or dotted class names). If you're including Dejavu in a larger framework, feel free to add to this registry.

The options dict we pass in our example includes a standard ADO Connect string. The MS Access class requires this entry; other SM's may not.

Database Storage Managers

Microsoft SQL Server / Microsoft Access (Jet)

This module was developed against ADO 2.7 and 2.8, using MSDE, SQL Server 2000/2005, and Access 2000.

Classes:

- "sqlserver" (`dejavu.storage.storeado.StorageManagerADO_SQLServer`)
- "[ms]access" (`dejavu.storage.storeado.StorageManagerADO_MSAccess`)

Options:

- **connections.Connect:** A valid ADO connect string. There are plenty of online references for how to form these; for example, at [Microsoft](#).

PostgreSQL

This class was developed against PostgreSQL 8.0.0 rc-1 (Win2k), PostgreSQL 8.2.4 on i686-pc-mingw32 (Vista), and also tested on PostgreSQL 7.6.6-6 on Debian "sarge", using pyPgSQL-2.5.1 and psycopg2-2.0.6/2.0.5.1

Classes:

- "postgres[ql]" or "pypgsql" (`dejavu.storage.storepypgsql.StorageManagerPgSQL`)
- "psycopg[2]" (`dejavu.storage.storepsycopg.StorageManagerPsycoPg`)

Options:

- **connections.Connect:** A connect string of the form "k=v k=v". For example, "host=localhost dbname=myapp user=postgres password=hilarious". See the [libpq](#) docs for complete information.

MySQL (MySQLdb)

This class was developed against mysql Ver 14.7 Distrib 4.1.8, for Win95/Win98 (i32), and also tested on mysql Ver 12.22 Distrib 4.0.23, for pc-linux-gnu (i386), and 5.0.45.community.nt (Vista)

Classes:

- "mysql" (`dejavu.storage.storemysql.StorageManagerMySQL`)

Options:

- Connection arguments: any of "host", "user", "passwd", "db", "port", "unix_socket", "client_flag". See the [docs](#) for complete info.

SQLite (pysqlite/sqlite3)

This class was developed against sqlite 3.0.8 (pysqlite-1.1.6.win32-py2.3), sqlite 3.3.3 (pysqlite-1.1.7.win32-py2.4), sqlite 2.8.15-3 on Debian "sarge", sqlite 3.3.4 (python 2.5 on win2k), and sqlite 3.4.0 (pysqlite-2.3.5.win32-py2.4) on Vista. If you have Python 2.5 or later, the builtin `_sqlite3` library will be used; otherwise, you need to install pysqlite.

Classes:

- "sqlite" (`dejavu.storage.storesqlite.StorageManagerSQLite`)

Options:

- **Database:** Filename of the database. May be a relative path.
- **Mode:** Optional. DB file mode. Defaults to 0755.

Firebird (kinterbasdb)

This class was developed against:

- KInterbasDB Version: (3, 2, 0, 'alpha', 1) and Server Version: 'WI-V1.5.2.4731 Firebird 1.5' on Win2k,
- KInterbasDB Version: (3, 2, 0, 'final', 0) and Server Version: 'WI-V2.0.3.12981 Firebird 2.0' on Vista.

Classes:

- "firebird" (dejavu.storage.storefirebird.StorageManagerFirebird)

Options:

- **Name:** Filename of the database. Must be an absolute path.
- **Host:** The TCP host name, usually "localhost".
- **User:** The user name (e.g. "sysdba").
- **Password:** The password for the given user name.
- **Encoding:** The charset to be used in each connect() call.

The Firebird Storage Manager is new and not yet fully thread-safe. Patches welcome.

Common Database Configuration Entries

In addition to the above, Storage Managers for databases (probably) accept these additional options:

Key	Example Value	Description
schemaclass.prefix	myapp_	Optional. If specified, all tables in the database will have names starting with this prefix. If not provided, it defaults to "" (empty). This helps if you need to mix Dejavu tables with tables from another application. Leave blank if you want no prefix.
connections.poolsize	10	Optional. Defaults to 10. If nonzero, connections will be pooled (up to a total equal to <i>Pool Size</i>). If zero, no pool will be used; each statement (!) will use a new connection.
connections.implicit_trans	False	Optional. Defaults to False. If True, a new connection will automatically call "START TRANSACTION". It will also be associated with the current thread, and any subsequent calls on the same thread will then return the same connection object.
connections.contention	'commit'	Optional. If 'commit' (the default), schema-modifying commands (e.g. add_property) will autocommit any pending transactions. Change this to 'error' if you'd rather play it safe.
connections.default_isolation	"READ COMMITTED"	Optional. All database SM's already have a value for this, but you can select another if you wish. This value should be a "native value" for your database's particular transaction mechanisms. For example, PostgreSQL uses ANSI/SQL names like "READ COMMITTED", but Firebird uses library constants like kinterbasdb.isc_tpb_read_committed.

Other Storage Managers

RAM

Persists Units in RAM; all Units are lost when the process exits.

Memcached

External Dependency: [python-memcached](#)

Persists Units to a set of [memcached](#) servers. This is an extremely simple implementation; every value that is not of type `str` or `int` is pickled. Querying will be slow-- every Unit is sucked in one-by-one and tested in pure Python. But for many cache applications, you don't need heavyweight query tools.

Classes:

- "memcache[d]" (`dejavu.storage.storememcached.MemcachedStorageManager`)

Options:

- **name:** Required. This string will be used to form namespaced memcached keys.
- **memcached.servers:** Required. A list of strings of the form 'IP-address:port'. These will be passed directly into the `memcache.Client` instance.
- **memcached.indexed:** if True (the default), this store will maintain an index of all stored objects in memcached itself. This is the 'safe' choice, and necessary if your only store is memcached. If you run this store as an `ObjectCache.cache`, however, you should turn this off, allowing `ObjectCache.nextstore` to maintain the indexes--this allows the cache to run orders of magnitude faster.

JSON

External Dependency: [simplejson](#)

Persists Units to a filesystem, one folder per class. Each folder contains files, one per Unit, with the Unit identity as the file name. Each of those unit files contains a JSON dict of Unit property values. For example:

```
root/
  Album/
  |   78952.json
  Song/
    1372.json
    88.json
```

Querying will be slow--every Unit is sucked in one-by-one and tested in pure Python. This is a good choice for test data or system tables--store the data in JSON format for pretty version-control diffs, then migrate it to another store when you run the tests or start the application.

Classes:

- "json" (`dejavu.storage.storejson.StorageManagerJSON`)

Options:

- **root:** Required. The file path (directory) in which to place db files. Each Unit class will get its own subfolder, of the same name as the class.
- **mode:** Optional. The mode arg to pass to `os.mkdir` when creating folders. Defaults to '0777'.
- **idsepchar:** Optional. The character to use for separating unit identities which are multivalent. Defaults to '_' (underscore). For example, a Unit with `identifiers = ('Name', 'DOB')` would get a folder name like 'Fred_20040321'.
- **encoding:** Passed to the `simplejson.Decoder`.
- **skipkeys:** Passed to the `simplejson.Encoder`. Defaults to False.
- **check_circular:** Passed to the `simplejson.Encoder`. Defaults to True.
- **allow_nan:** Passed to the `simplejson.Encoder`. Defaults to False.
- **indent:** Passed to the `simplejson.Encoder`. Defaults to None.

Shelve

Persists Units to shelve-type files. Extremely simple implementation; everything is pickled. Querying will be slow--every Unit is sucked in one-by-one and tested in pure Python using `Expression(unit)`. But for many applications, you don't need heavyweight query tools; for example, an online forum may only need topic content looked up by ID. Or small system tables that only get read at startup might benefit.

Developers note: The shelve implementation in Dejavu does not use "writeback"; that is, changes you make to data are stored only in memory until each shelf has its `close` method called. If `close` is never called, your changes are lost! The easiest way to ensure that your changes are saved is to call `store.shutdown()` when your app is closing. Since one of the design goals of Dejavu is to allow deployers to choose which backend to use, your applications should *always* guarantee that `store.shutdown()` is called on program exit.

Classes:

- "shelve" (`dejavu.storage.storeshelve.StorageManagerShelve`)

Options:

- **Path:** The file path (directory) in which to place db files. Each Unit subclass will get its own file, of the same name as the subclass.

Folders

Persists Units to a filesystem, one folder per class. Each folder contains subfolders, one per Unit, with the Unit identity as the folder name. Each of those unit folders contains one file for each Unit Property. For example:

```

root/
  Album/
    | 78952/
    |   Name.txt
    |   Artist.txt
  Song/
    | 1372/
    |   AlbumID.txt
    |   Data.mp3
    | 88/
    |   AlbumID.txt
    |   Data.mp3

```

This is an extremely simple implementation; every value that is not of type `str` is pickled. Querying will be slow--every Unit is sucked in one-by-one and tested in pure Python. But for many applications, you don't need heavyweight query tools; for example, an upload site may only need files looked up by ID.

Classes:

- "folders" (`dejavu.storage.storeshelve.StorageManagerShelve`)

Options:

- **root:** Required. The file path (directory) in which to place db files. Each Unit class will get its own subfolder, of the same name as the class.
- **mode:** Optional. The mode arg to pass to `os.mkdir` when creating folders. Defaults to '0777'.
- **idsepchar:** Optional. The character to use for separating unit identities which are multivalent. Defaults to '_' (underscore). For example, a Unit with `identifiers = ('Name', 'DOB')` would get a folder name like 'Fred_20040321'.
- **extdefault:** Optional. The default file extension to use for Unit Property files. Defaults to '.txt'.
- **<unit>.<propname>:** Optional. The value should be the file extension for properties of the given propname for the given unit class. For example, `Song.Data = .mp3` (be sure to include the leading 'dot' if you want one).

Middleware

Some Storage Managers act as "middleware", and can be chained together to provide layered functionality. Consider, for example, the `ObjectCache` class; it has another Storage Manager "behind it", which it proxies. It can be used to cache objects between client connections independently from the underlying, database-specific Storage Manager. The beauty of this design is that the decision to use a `ObjectCache` is completely up to the deployer, *not* the application developer. The deployer can separate stores, test response times, and address other integration concerns on their own systems.

Object Cache

Use this class to persist Units in memory between client connections. It must proxy another Storage Manager.

Classes:

- "cache" (`dejavu.storage.caching.ObjectCache`)

Options:

- **Next Store:** Required. The next Storage Manager in the chain.
- **cache:** Optional. The Storage Manager to use for the cache. If not given, it defaults to a RAM store.

Aged Cache

Use this class to persist Units in memory between client connections. It must proxy another Storage Manager.

Classes:

- "aged" (`dejavu.storage.caching.AgedCache`)

Options:

- **Next Store:** Required. The next Storage Manager in the chain.
- **cache:** Optional. The Storage Manager to use for the cache. If not given, it defaults to a RAM store.
- **Lifetime:** Optional. The recurrence string which declares how often to sweep Units out of the in-memory cache. The string you supply should be one of the following types:
 - **By units (intervals):** "3 hours" will run every 3 hours. "7 days" or "1 week" will run once each week.
 - **Daily:** "14:00 each day" will run at 2:00 P.M. every day.
 - **Weekly:** "Mon", "Monday", or "Mondays" will run once each Monday.
 - **Monthly:** "20 each month" will run on the 20th of each month. "0 every month" will run on the *last* day of each month.

See the `recur` module for complete options.

Burned Cache

Use this class to persist Units in memory between client connections. It needs another Storage Manager to proxy. Unlike the ObjectCache above, this Storage Manager recalls all Units at once upon the first request, and won't recall them again from storage. They are "burned" into memory for the lifetime of the application.

Classes:

- "burned" (`dejavu.storage.caching.BurnedCache`)

Options:

- **Next Store:** Required. The next Storage Manager in the chain.
- **cache:** Optional. The Storage Manager to use for the cache. If not given, it defaults to a RAM store.

Partitioning

Vertical Partitioner

This class replaces the old Arena object from Dejavu 1.x. It allows you to aggregate multiple stores into a single interface, partitioned by Unit class. Unlike most other StorageManagers, it takes no options. Instead, you will generally set this as the root of your storage graph and repeatedly call its `add_store(name, store)` method. There's also a corresponding `remove_store(name)` method.

Once you've added stores, the `stores` attribute is a dict from store names to StorageManager instances. However, you shouldn't manipulate this directly--use `add/remove_store` instead. When you call `add_store`, it will also set up the partitioner's `classmap` attribute, which is used to direct queries and other command to the correct store(s) based on the class. DDL methods will generally dispatch to all stores for each class. DML methods will generally dispatch to `classmap[unit.__class__][0]`; those which involve multiple classes (e.g. multirecall), will try to find a single store which handles all classes in the given relation. To override this default search, you can add entries to `classmap` of the form: `{(clsA, clsB, clsC): [store1]}`, which instructs the partitioner to use the given store for any Join with the same order, such as `(clsA << clsB) & clsC`.

Classes:

- `dejavu.storage.partitions.VerticalPartitioner`

Options:

- None

SM Comparison Chart

When selecting a storage implementation, you should be aware of the strengths and limitations of each option. The following chart should help you decide.

First, it shows you which stores do and do not support certain optional features of Dejavu. Your application developer should provide you with a list of any features which they *require*.

Second, it shows you which stores have performance or boundary issues and where. When developing applications, you should avoid these issues either by coding alternative solutions, or by recommending to your deployers that they avoid the problematic stores. Note that some limitations are inherent in the storage mechanism itself, while some are limitations of the current Storage Manager for that mechanism.

- **Y**: The store supports the feature natively.
- **P**: The store does not provide the feature natively, but Dejavu provides a fallback in pure Python (which may be slower). Boundaries and limitations are therefore Python limits.
- **N**: The store does not allow the feature at all.
- **<blank>**: Unknown/not yet documented.

	access	firebird	mysql	postgres	sqlite	sqlserver	ram	memcached	shelve	folders	json
Connection Pool [5]	N (single only)	P	P	P	P	P	N	N	N	N	N
Transactions	Y	Y	Y	Y	Y	Y	N	N	N	N	N
Indexes	Y	Y	Y	Y	Y	Y	N	N	N	N	N
Max identifier length	64	31	64	63	no limit?	128	P	P	P	OS-dependent	OS-dependent
Case-sensitive identifiers	Y	Y	Unix only	Y	Y	Y	Y	Y	Y	Y [3]	Y [3]
Case-sensitive LIKE ("a in b")	P	Y	Y	Y	Y	P	P	P	P	P	P
Case-sensitive string comparison ("a" > "A")	<, <=, ==, !=, >, >=	Y	Y	Y	Y	<, <=, ==, !=, >, >=	P	P	P	P	P
Wildcard literals in LIKE ("a in b")	Y	Y	Y	Y	3.0.8+	Y	P	P	P	P	P
Autoincrement	Y	Y	Y	Y	3.1.0+	Y	P	P	P	P	P
add/drop/rename property	Y	Y	Y	Y	P [2] (add: 3.2.0+)	Y	Y	Y	Y	Y	Y
	access	firebird	mysql	postgres	sqlite	sqlserver	ram	memcached	shelve	folders	json
fixed point/decimal precision (in decimal digits)	12	18	16	1000	0 (always uses TEXT instead)	12	P (pickle)	P (pickle)	P (pickle)	P (pickle)	Python limit?
Max str/unicode bytes	1 GB [6]	32765 (255 for an index)	8000 (row limit)	1 GB?	1 MB (row limit)	8000 [4]	P (pickle)	1 MB (object limit, adjustable)	P (pickle)	P (pickle)	Python limit?

	access	firebird	mysql	postgres	sqlite	sqlserver	ram	memcached	shelve	folders	json
datetime ranges	0100-01-01 to 9999-12-31	1753-01-01 to 9999-12-31	1000-01-01 00:00:00 to 9999-12-31 23:59:59	4713 BC to 5874897 AD	4714-11-24 BC to ???	1753-01-01 00:00:00.0 to 9999-12-31 23:59:59.997	P	P	P	P	P
datetime precision	1 second	1 second	1 second	1 microsecond	1 second	1 second	P	P	P	P	1 second
dejavu.year, month, day functions	Y	P	Y	Y	3.2.3+ [1]	Y	P	P	P	P	P
dejavu.now, today functions	Y	now	Y	Y	3.2.3+ [1]	Y	P	P	P	P	P
startswith, endswith, containedby, dejavu.containedby, dejavu.contains, dejavu.startswith, dejavu.endswith	Y	Y	Y	Y	Y	Y	P	P	P	P	P
builtin function: len	Y	P	Y	Y	Y	Y	P	P	P	P	P
	access	firebird	mysql	postgres	sqlite	sqlserver	ram	memcached	shelve	folders	json
READ UNCOMMITTED	Y	N	Y	N [7]	N	Y					
READ COMMITTED	N	Y	Y	Y	N	Y (timeout)					
REPEATABLE READ	N	N [7]	N [7]	N [7]	N	Y (timeout)					
SERIALIZABLE	N	Y	Y (timeout)	Y	Y [8]	Y (timeout)					
Change isolation inside transaction	N	N	Y	Y	N	Y					

[1] In order to use native date functions in SQLite, you must be storing your date and time values in one of the acceptable formats. See the [SQLite wiki](#) for more information. Once you have verified that you are using such a format, you must then set `AdapterToSQLite.using_perfect_dates` to `True`. This can be done with the configuration entry: `Perfect Dates: True`.

[2] SQLite must copy the entire table to an intermediate table and then to a new, final table in order to alter tables. Beginning in 3.2.0, adding columns may now be performed natively (but not renaming or dropping them).

[3] The Folders store keeps identifier values and property names in folder and file names. Not all filesystems support case-sensitive file/folder names.

[4] Microsoft SQL Server does not allow comparisons on string fields larger than 8000 characters.

[5] Dejavu provides connection pool factories in pure Python, and does not yet make any attempt to use native pooling features.

[6] Microsoft Access "MEMO" fields have a 1 GB limit, but so does the entire database. Memo fields also cannot be used as join keys; set `hints['bytes'] = 255` or less to use `VARCHAR` instead.

[7] Some databases over-protect at various isolation levels. For example, "REPEATABLE READ" should prevent fuzzy reads but allow phantoms, but MySQL's and Firebird's REPEATABLE READ prevent both. PostgreSQL only uses two isolation levels internally, so that selecting "READ UNCOMMITTED" behaves like "READ COMMITTED" and "REPEATABLE READ" behaves like "SERIALIZABLE".

[8] SQLite `:memory:` databases cannot use multiple connections, so a single connection is used for all threads. However, this means that transactions are generally not allowed for `:memory:` databases when using multiple threads (because multiple transactions would overlap on the same connection and not be isolated at all!).

Advanced Topics

As with all frameworks, Dejavu can't cover every need out-of-the-box. However, Dejavu has been specifically designed to be hackable. In particular, the creation of new Storage Managers is a well-defined process. Read on if there's a feature you need that you might consider building yourself.

Subclassing Sandbox

Okay, I lied. There's not much I can think of that you'd want to do with Sandboxes. Most things I *can* think of would be better implemented as Storage Manager middleware. But if you think of any, let me know.

Store Hacking

The most common modification to a `StorageManager` object is to use it as a dumping-ground for other application data. Since the store should persist for the lifetime of the application's process, it can serve as a decent top-level Application namespace. Since the only mandatory argument to `Sandbox.__init__` is a store, you can pass Sandboxes around in your code and always have access to the root store. If you use another application framework for your front-end, just stick a reference to it in your store and vice-versa. Python dynamic attributes to the rescue again!

Passing through SQL

If you *want* to keep writing SQL, there's nothing stopping you from doing so. If nothing else, it can be a handy way to prototype or migrate an application, and then replace the SQL with Dejavu API calls later on. You'll need to make your deployers aware that you're using SQL directly (and which DB's your SQL runs on), so they don't try to deploy your application with an unsupported store.

To avoid stale data, you should probably flush any sandboxes before running your query, especially if it updates data. You should also run the following snippet to flush `CachingProxy` or `BurnedProxy` SM's:

```
store.sweep_all()
```

Then, use the extension methods built into `StorageManager` classes' "db" attribute to get data:

```
>>> rows, cols = s.db.fetch("SELECT djvFields.Value, Count(djvCity.ID) AS NumCities
"
                                "FROM djvFields LEFT JOIN djvCity ON djvFields.ID = "
                                "djvCity.Field GROUP BY djvFields.Value")
>>> cols      # [(name, type), (name, type), ...]
[(u'Value', 202), (u'NumCities', 3)]
>>> rows
[(u'Baja California', 3), (u'Ciudad Juarez', 1), (u'Puerto Penasco', 0), (u'Yucatan
Peninsula', 1)]
```

...or update:

```
>>> s.db.execute("UPDATE djvFields SET ShortCode = Left(Value, 1) WHERE ShortCode
Is Null;")
```

There are a **lot** of other things you can do with the builtin Database, Schema, Table, Column, and SQLDecompiler classes. Feel free to open them up in an interactive session and explore. All of the RDBMS Storage Managers are built on top of an independent SQL layer (called [Geniusql](#)) that knows nothing about units or Storage Managers (but it does understand Expressions).

Custom Storage Managers

In most cases, you will add new functionality to Dejavu itself by creating a custom Storage Manager, whether for a new backend, or a custom middleware component. Storage Managers must conform to a simple interface for creating, destroying, and recalling Units. They are free to implement that functionality however they like.

As you can see in the code, the `storage.StorageManager` base class requires you to override most of its methods.

- `__init__(self, allOptions={})`: Optional. Place any startup code here, as each SM should only be instantiated once (at app startup). Any additional arguments should be passed in the `allOptions` dictionary (rather than modifying the signature). You should expect the keys and values of `allOptions` to be strings.
- `reserve(self, unit)`: Required. Take the supplied Unit instance and "make a space" for it in storage. The Unit does not need to be fully populated. If the Unit has an ID when passed to `reserve`, use it. If not, supply it using the class' `UnitSequencer`. If your database provides equivalent sequencing to dejavu Sequencers, feel free to use it. If not, grab all existing distinct ID's (which you are storing), and pass them to `unit.sequencer.assign(unit, ids)`, which should assign the next ID in the sequence to the Unit. Remember that when we say "ID" we mean a tuple of identifier (`UnitProperty` attribute) values. You should probably lock this whole method in a `threading.Lock`.
- `save(self, unit)`: If not `unit.dirty()`, you can exit. Otherwise, iterate through the Unit's properties and persist each value, using an Adapter to coerce each value to the type expected by your database. If you are able to persist all values, call `unit.cleanse()` to mark the Unit as no longer dirty. You are not required to persist any Unit attributes other than `UnitProperties`.
- `destroy(self, unit)`: Required. Remove the Unit's data from storage permanently. For databases, this means `"DELETE FROM %s WHERE %s;" % (table.qname, table.id_clause(**unit.properties))`.
- `recall(self, unitClass, expr=None)`: Required. This method must return an iterable which yields fully-populated Unit objects. The Units must be of the supplied `unitClass`, and must match the Expression, if supplied. If an Expression is not supplied, all stored Units of the specified class must be returned.
- `create_storage(self, unitClass)`: Optional. If you do not override this method, it simply passes. If your Storage Manager needs to set up tables or other structures per `unitClass` (and almost all do at install time), use this method to do so.
- `shutdown(self)`: Optional. If you do not override this method, it simply passes. If your Storage Manager needs to be explicitly closed when the application shuts down, add code to do that here.
- `distinct(self, cls, fields, expr=None)`: Recommended. This method must return an iterable of (tuples of) distinct `UnitProperty` values for the given field(s). The Units from which the values are drawn must be of the supplied class (`cls`), and must match the Expression (`expr`), if supplied. If an Expression is not supplied, all stored Units of the specified class must be examined.
- `multirecall(self, classes, expr)`: Recommended. The 'classes' argument will be a `UnitJoin` and its children. This method must return an iterable of lists; each item in each list will be a Unit. The Units must be of the supplied classes, in order (see the `UnitJoin.classes` method), and must all match `expr(*resultset)` together.

Generic Database Wrappers

Writing a Storage Manager for a database is relatively straightforward, mostly because Dejavu doesn't have complicated storage interfaces or demands. If you find your application depends heavily upon using advanced features of a particular database, or upon hand-crafted SQL, then Dejavu is not for you or your application. A Dejavu SM module for a database usually includes:

1. Adapters, which coerce values from Python types to database types and back again. Base classes for DB Adapters can be found in `dejavu.storage.db`.
2. An SQLDecompiler, which converts `dejavu Expression` objects (essentially, Python lambdas) into SQL.
3. A subclass of `geniusql.Database`, which handles requests to SELECT data using the above two components, as well as ALTER TABLE, etc.

Adapters

Generally, you will end up with three kinds of Adapters (subclasses of `storage.Adapter`): one for converting Dejavu types to your database types, another for the reverse (`storage.db.AdapterFromDB`), and probably a third to insert Dejavu values (with proper quoting, etc.) into SQL statements for your database (`storage.db.AdapterToSQL`). The Adapter class provides a single public method, `coerce(self, value, dbtype, pytype)`, which takes any value and attempts to return a new value.

`adapter.coerce()` handles a request by calling a sibling method (that is, a method of the same subclass). Therefore, you need to add methods to your Adapter for each Python type you wish to support. For example, if you wish to coerce Python ints to INTEGER, you need to add the following method to your Adapter subclass:

```
def coerce_int_to_any(self, value):
    return str(value)
```

Methods are named `coerce_type1_to_type2`, where 'type1' and 'type2' are type names, one of them a Python type and the other a database type. If your type name has dots in it, they will be converted to underscores. If either of the type names is 'any', that method will be used if no more-specific coercion method exists. Again, you can most likely use methods in the base Adapter classes provided.

Your coercion method should receive a single value and return that value, coerced to a type. An outbound adapter coerces from Python types to database types. You supply a Dejavu UnitProperty value to `coerce`, and the appropriate coercion method will be selected based upon the `type()` of that value. An inbound adapter, on the other hand, coerces from DB types to Python types. Call `coerce` with your database value *and* the `valuetype` argument, which is then used to call the appropriate coercion method. That method returns the value, coerced to `type(valuetype)`, which the UnitProperty expects.

If `coerce` cannot find a method for the appropriate Python type, it errors, and rightly so. Don't let these errors pass silently! An earlier version of Dejavu had a "default" coercion method, which was a Bad Idea. Don't replicate it.

Decompiler

The SQLDecompiler is the tricky bit of any Storage Manager. You must receive a Unit class and an Expression, and produce valid SQL for your database from both. For example, given:

```
unitClass = Things
expr = logic.Expression(lambda x: x.Group == 3)
```

...your decompiler should produce something like:

```
"SELECT * FROM [divThings] WHERE [djvThings].[Group] = 3"
```

The above example may seem trivial to you, but add in proper quoting, diverse datatypes (like dates and decimals), complex operators (like 'in', 'Like', and 'ieq'), logic functions (like today() and iscurrentweek()), null queries, and just-in-time keyword args, and it becomes complex very quickly. You are, in effect, writing a mini-ORM.

But, don't despair. Dejavu provides you with tools to make this task easier:

1. The most important tool is `geniusql.select.SQLDecompiler`, a complete base class. You should be able to tweak it for most databases with a couple of SQL syntax changes.
2. `SQLDecompiler` is built on a simple Visitor-style base class, `codewalk.LambdaDecompiler`. More complicated extensions are easily added to this base class; each bytecode in the Expression (Python lambda) gets its own method call.
3. You don't have to handle globals or cell references within the lambda--when the lambda gets wrapped in an Expression, all free variables are converted to constants.
4. You aren't *forced* to handle every possible operator, function, or term in SQL. The base `SQLDecompiler` doesn't; when it encounters a function it can't handle, for example, it punts by flagging the SQL as *imperfect*. This signals the Storage Manager to run each Unit through the lambda (in pure Python) before yielding it back to the caller. In fact, you can start writing your Storage Manager without a decompiler at all! Just return all stored Units of the given class and use the Expression to filter whole Units. Then, when your SM works, add a decompiler.

Database/Table/IndexSet

You'll need a subclass of `geniusql.Database`. Override the container methods (like `__setitem__` and `__delitem__`). For most popular databases, these are pretty straightforward. Some notes:

- **select/where:** If no Expression is supplied, return all Units. Otherwise, use a decompiler to produce SQL which you can then use to grab Unit data from storage. Use each row to populate a Unit (use an Adapter for type coercion), and yield each Unit back to the caller. In general, it's faster to slurp all the data in at once than to make a separate call for each row.
- **__get_tables/__columns/__indices:** use your database's schema-inspection tools to tell Dejavu the names, datatypes, and other metadata that actually exists in each deployed database.

Database SM's should also define the methods `create_database()` and `drop_database()`, if possible.

Use `dejavu/test/zoo_fixture.py` to test your new Storage Manager. Copy one of the (very short) `test_store*` modules for the other SM's, and make the necessary changes for your SM. All of the heavy lifting of the tests is done in `zoo_fixture`.

Legacy Database Wrappers

Sometimes you do not have complete control over the database you want to reference. In that case, you should probably still write a custom Storage Manager, Adapters, and a Decompiler. Often, you can get away with providing a simple column-to-Unit mapping to use as you decompile. I've built one, for example, to wrap [The Raiser's Edge](#) (third-party fundraising software). My Dejavu model manages directory records and income without regard for the underlying database; a custom Storage Manager maps between that ideal model and the Raiser's Edge API. This allows me to integrate data from RE with our custom inventory, invoice, and scheduling software.

One of the more important parts of wrapping existing tables is getting your pretty Python names mapped to ugly database names. Do this by making a custom Database: override the `__column_name` and `table_name` methods to do the mapping.

Other Serialization Mechanisms

sockets

There's a `sockets` module in the `storage` package. It does simple serialization of Units across a socket, so you can run Dejavu in its own process, separate from your front end. I had to do this with a third-party database, which couldn't handle web-traffic threading models. Here's a snippet of how to use it (from that app):

```
def query(self, cmd, unitType='', data=None):
    if isinstance(data, dejavu.Unit):
        data = stream(data)
    elif data is None:
        data = ''
    else:
        data = pickle.dumps(data)
    response = self.socket.query(":".join((cmd, unitType, data)))
    return response
```