The F2 DBMS in Python

Thibault Estier

september 2004 - june 2014

Ecole des HEC - University of Lausanne
Email: thibault.estier@unil.ch

Abstract

This document is a brief introduction to F2-Python, the Python version of the F2 Database Management System. F2 is an evolution oriented DBMS. New users of F2 may use this text to discover how to build, manage and transform a database (or more appropriately an *object base*) in Python. This introduction is more a tutorial than a full reference manual, and is intended to give an overview of F2 features and ease of use from Python.

Contents

1 Introduction

F2 is an object-oriented Database Management System (DBMS) specially built to support *schema evolution*. Its main purpose is to support persistent data for which the initial schema is very prone to changes. Its underlying design choices are based on *reflexivity* (no separation between meta-levels and objects, no DDL) and *transposed storage* (grouping values of the same attribute for different objects, instead of grouping all attributes values of an object). It was developed in the database research laboratory leaded by prof. Michel Léonard at University of Geneva, between 1989 and 1991. Further research and applications of F2 (benchmarking, support of additional concepts, building of ontologies) continued up to 1997. The initial version of F2 was written in Ada by Thibault Estier. For several years, F2 development has been continued in Ada and F2 is still available on every platform supporting the Gnu-Ada compiler (called Gnat).

In october 2003, Thibault started to bootstrap a new version of F2 in Python, using the ZODB[1] persistence engine. The motivation for a Python version is twofold:

- to offer a native python access to F2, allowing usage from within web application servers like Zope,
- to give an interpretative environment to F2 by substituting Python to the original F2 manipulation language (called FarTalk). This avoids the traditional syntactic gap[2] between a programming language and a DB manipulation/query language (like embedding SQL in a C^{++} program).

The F2Python DBMS may run in two different modes: directly on local data files (called FileStorage in ZODB), or on a RPC-controlled server (called ZEO Storage in ZODB). The first mode is used when F2 runs within a Zope server: F2 classes collections are managed in the current Zope storage. The second mode is used by python clients programs sharing a common F2 server, without the use of a Zope application server.

2 Installing F2 for Python

2.1 Downloading ZODB before installing F2

F2 requires ZODB so you have to install it before F2. If you intent to use F2 uniquely from within a Zope application server, ZODB is already there, so you don't need to worry about this point. If you want to use F2 from other python programs, or if you want to test your python scripts without Zope, then you need to install it:

1. get the latest (stable) version of ZODB + ZEO from PyPi.
   see http://www.zodb.org,
2. depending on your installation environment, one of the following command should do it:

   ```
   $ pip install ZODB3
   ```

   or

   ```
   $ easy_install ZODB3
   ```

The installation of ZODB3 combines automatically for you four installations into a single one: ZODB itself, ZEO, persistent, and Btrees. These four packages are required by F2. When the ZODB installation procedure starts, it will try to compile a few C source code files. If it doesn't find a compiler onboard your machine, the installation script

[1] ZODB is the storage mechanism developed for and used by Zope application servers, see http://www.zope.org
[2] also called impedance mismatch in database literature

will end with errors. So you will need a C compiler installed. This is most certainly already the case for a Debian-like Linux install. This may also be the case on a Mac if you have the MacOS Developper tools installed. If not, get them from the Apple App Store. On Windows, you may install VCforPython27 , a small package containing the Microsoft C++ compiler plus some python addons and development libraries.[3] Once you have your compiler installed, run the ZODB installation script again: `easy_install ZODB3 .`

2.2 Installing F2 itself

Whatever the way you got F2 for python on your machine, you probably received a zip file with a name like 'F2-1.3.1.zip' or 'F2-1.3.1.tar.gz', depending on the platform you are using it. Open this archive file with the appropriate tool [4] to obtain somewhere on your disc a 'F2-1.3.1' directory.

To install this software, open a command shell (or terminal) and type the following command:

```
$ cd ./F2-1.3.1
$ python setup.py install
```

You will see a lot of messages, indicating essentially the move of files into appropriate places in your installation. To be successful, the installation must be run with administration rights. On several platforms (MacOSX or Linux) you may have to prefix your command with the **sudo** command:

```
$ sudo python setup.py install
```

This will usually require your password before actually executing install instructions.

2.3 Checking your installation

Type the following command:

```
$ python -c 'import F2'
```

If this command returns without any effect or message, then your installation is a success. If it displays any form of error message containing "ImportError" then something failed during the preceding steps.

2.4 Testing F2 without server (DB in a simple file)

Type the following command:

```
$ python bootf2.py file:root.db
```

You should normally see a succession of lines commenting the ongoing bootstrap of a F2 kernel. The resulting database will be in the current directory under name 'root.db'.

[3] download this package from (https://www.microsoft.com/en-us/search/result.aspx?q=vcforpython27).

[4] with a zip file, use the command `$unzip F2-1.3.1.zip` , with a tar.gz file, use the command `$tar zxvf F2-1.3.1.tar.gz`

2.5 Testing F2 with a client/server configuration

1. check content of file 'f2_zeoserver.conf', verify that port 8081 is not already used on your machine, this is the port on which your F2 server will listen and serve clients. If this is the case, replace '8081' by some other appropriate port number in the 'f2_zeoserver.conf' file (this is a simple text file that you may modify with the help of any standard editor).
2. create a local subdirectory called 'var' in the current directory. The server will store F2 data files, logs and other control files in it.
3. launch your server with the following command:

   ```
   $ ./start_f2_server
   ```

4. bootstrap the server database with the following command:

   ```
   $ python bootf2.py rpc:127.0.0.1:8081
   ```

   You will of course replace '8081' by the port number you choosed in point 1) if you changed it. The '127.0.0.1' address is the standard IP address designing your own machine. On many platforms, it may also be replaced by the equivalent 'localhost' string, so you may use the `rpc:localhost:8081` form instead. A client program running on another machine would use instead the effective address of your machine. This implies you opened the port 8081 to other machines on the network. If this is the case, BEWARE because anyone can use your F2 database while your server is running. When in doubt, ask your local network administrator.
5. to stop your F2 server use:

   ```
   $ ./stop_f2_server
   ```

2.6 Restricting access to a F2 server

You can restrict access to your F2 server using a simple user/password scheme, relying completely on the authentication mechanism offerd by ZODB/ZEO. To do this, modify the file 'f2_zeoserver.conf' by removing comments on lines defining authentication parameters. The file which contains (encrypted) usernames and passwords is in the './var/useraccess' file. You update this file using the **zeopasswd.py** utility, which you can find in your ZODB distribution (in directory 'ZEO' of your local installation). Usage of this utility is described by the program itself when you launch it without any parameter. By default, your F2 distribution is configured with only one user called *admin* with password *admin*. Change it immediately if you decide to activate authentification !

3 Using F2: a short introduction

Before beginning to execute F2 queries and manipulations, ensure you are working on a non empty database: an F2 database has to contain a minimal set of objects called the *F2 kernel*. This set of objects is actually a self describing collection of F2 classes. The F2 kernel is described in details in section 4, page 11. The sections 2.4 and 2.5 describe how to create the F2 kernel in your DB.

3.1 Opening access to F2

Using the F2 DBMS from a python script or interactively from a python shell is not very difficult. Basically, you open a connection to a F2 database and keep this connection in a python variable. Then this variable is like a namespace giving you direct access to F2 classes: if you know the name of your classes, you will have access to all your objects, and you will execute methods of these objects and classes to manipulate your database. The following example illustrates this step by step on an interactive python session.

```
$ python
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import F2
>>> f2 = F2.connect('rpc:127.0.0.1:8081')
```

Now variable *f2* can be used as a namespace to give access to already defined F2 classes.

```
>>> f2.String, f2.Int, f2.CLASS, f2.Attribute
(<CLASS.String>, <CLASS.Int>, <CLASS.CLASS>, <CLASS.Attribute>)
```

3.2 Creating a new class, creating objects

Our *f2* variable can also be used to create a new class, using method `class_create`. This method is actually a shortcut to create new objects in the class *CLASS*. We will come back to this later on.

```
>>> D = f2.class_create(
...              className='Department', database='test',
...              schema={'name': f2.String})
>>> D
<TupleClass.Department>
>>> market = D.create(name = 'Marketing')
>>> market
<Department.0>
>>> market.name
'Marketing'
```

This shows the simplest way for creating a new class *Department*, to attach this new class to the database named 'test' and then immediately a department called *Marketing*. Now, lets create a class to describe *Person*'s and a subclass of *Employee*'s who work in departments and have managers.

```
>>> P = f2.class_create(
...           className = 'Person', database='test',
...           schema={'firstname': f2.String,
...                   'lastname' : f2.String})
>>> P
<TupleClass.Person>
>>> E = f2.class_create(
...           className = 'Employee', database='test',
...           superClass = P,
...           schema={'work_in' : (f2.Department, 1, '*'),
...                   'manager' :  f2.Person })
>>> E
<TupleClass.Employee>
```

This example shows how you can declare cardinality constraints for your attributes: an employee (who is a person, of course) works in at least one department, and possibly several ('*' means the maximum is not bounded). These three new classes (*Person*, *Employee*, and *Department*) now belong to the current namespace *f2*. In order to create new objects in these classes both following notations are equivalent :

```
>>> joe = P.create(firstname='Joe', lastname='Cool')
>>> larry = f2.Person.create(firstname='Larry', lastname='Ellison')
```

3.3 Accessing object values, assigning new values

To evaluate objects attributes or to assign new values to them, we simply use the natural python dotted notation:

```
>>> (larry.firstname, larry.lastname)
('Larry', 'Ellison')
>>> larry.firstname = 'Lawrence'
>>> linus = f2.Employee.create(firstname = 'Linus', lastname = 'Torvald',
...                            manager = larry, work_in = market)
>>> linus.work_in.name
'Marketing'
>>> linus.manager.firstname
'Lawrence'
>>> linus.manager.work_in
AttributeError: F2 attribute work_in not applicable to this object (<Person.1>)
```

Oops, that's right, Larry is a person, but not an employee (so it doesn't mean anything to ask for the department he works in). Let's hire him: make the object *larry* enter into (sub-)class *Employee*. This is legal because *Employee* is a subclass of *Person*.

```
>>> larry = f2.Employee.enter(larry)
>>> larry.work_in = market
>>> larry.work_in.name
'Marketing'
>>> linus.manager.work_in == linus.work_in
True
```

The dotted notation of the Python language also implies that you can use the builtins functions `getattr()`, `hasattr()`, and `setattr()` to access and assign object values when attribute names are variables. The fol-

lowing example shows simply how to print all attribute values of an object, whatever the class schema is at the time being:

```
for attr in f2.Employee.all_attributes():
    an = attr.attributeName
    print an, '=', getattr(larry, an)
```

3.4 Selecting objects

To retrieve objects using their attribute values (the same way you *select* tuples from a table in SQL), you apply selectors to a class. If you do not specify any selector, this will return all objects of the class. A selector is a pair giving the name of an attribute and a matching value: an object belongs to the selected result if the attribute applies to it (because its an inherited attribute for example) and if it takes exactly this value.

```
>>> f2.Person( work_in = market )
[<Person.1>, <Person.2>]
>>> f2.Employee( work_in = market )
[<Employee.1>, <Employee.2>]
>>> john = f2.Person(firstname = 'John', lastname='Smith')
>>> john_vehicles = f2.Vehicle(owner = john)
```

The result of a selection is a (python-)list, even if the result contains only one object. More precisely, the result is a *F2_object_list* on which you can apply further attributes evaluations, for example:

```
>>> f2.Employee(work_in = market).firstname
['Lawrence', 'Linus']
```

The use of selectors may be used very efficiently by F2 to select/retrieve objects, specially when the corresponding attribute has been configured with its *reversed* function. Selectors allow only very simple expressions to be used. To select objects using arbitrary complex predicates, you can use the special *_where* selector which takes an arbitrary lambda expression as value. This lets you build any selection predicates using python syntax. The following example shows how to select employees who work in the same department as *larry* does and whose firstname is not 'John' :

```
>>> f2.Employee(_where = lambda e:
...                 e.work_in == larry.work_in and e.firstname != 'John' )
[<Employee.1>, <Employee.2>]
```

You may also combine freely simple selectors and *_where* predicates in the same selection. Selectors are first applied (efficiently when possible), and then the predicate is filtering the resulting object set.

```
>>> johns_vehicles = f2.Vehicle(owner = john,
...                             model = 'BMW',
...                            _where = lambda v : v.year_built >= 1999
...                               and v.color in ('blue', 'black', 'grey') )
```

Actually, the *_where* parameter is not necessarily a lambda expression. It might as well be a function or a method as long as it takes exactly one parameter (the selected object) and returns a boolean value (`True` or `False`). So you can define separately your predicate, and reuse it in several selections.

3.5 Using keys in classes

Consider the relation built over all the attributes of class *Person* (list of attributes that you can obtain by `f2.Person.all_attributes()`). You may want to declare that the pair (*firstname*, *lastname*) acts as a key on this relation. In other words, there is at most one person $p \in f2.Person()$ with a given pair of strings (fn, ln) with: `(fn == p.firstname and ln == p.lastname)`. To obtain support from F2 in order to maintain this rule (this functional dependency), you would declare that class *Person* has the key $(firstname, lastname)$ with the following instruction:

```
newK = f2.add_key( f2.Person, 'firstname', 'lastname')
```

This has the following effect: you will be prevented to create object in class *Person* with an already used pair $(firstname, lastname)$ by a previously existing person, or you will not be able to update the name of a person in any way if it leads to give the same combination of values for two of them.

F2 will also consider from now that this pair of attributes forms an identification value combination. This means that these attributes can be used to enhance the display of an object of class Person: replacing `<Person.3112>` by `<Person.Joe_Cool>`. In the case of the class *Department*, if you declare the attribute *name* as a key with:

```
newK = f2.add_key( f2.Department, 'name')
```

then you will see results displaying things like `<Department.Marketing>` instead of an ugly `<Department.15223>`, because F2 considers now the name of a department as an identifying value. The queries shown above would now give the following result:

```
 >>> market
<Department.Marketing>
>>> f2.Person( work_in = market )
[<Person.Lawrence_Ellison>, <Person.Linus_Torvald>]
>>> f2.Employee( work_in = market )
[<Employee.Lawrence_Ellison>, <Employee.Linus_Torvald>]
>>> f2.Employee( work_in = market ).firstname
['Lawrence', 'Linus']
```

3.6 Deleting objects

In F2, you delete objects by removing them from any (sub-)class they belong to. This is done by applying the `leave()` primitive. It has the exact opposite effect of the `enter()` primitive: applying `Employee.leave(joe)` will remove object *joe* from the subclass *Employee*, but it remains an object of class *Person*. When an object leaves the topmost class of a hierarchy, it disappears completely from the base: so `Person.leave(joe)` is equivalent to a delete.

The `C.leave(o)` primitive respects the schema consistency of your database: this implies essentially two things:

1. by leaving a class *C*, your object automatically leaves also all subclasses of *C* [5],
2. any object *x* referring directly to *o* via an attribute *a* will be affected, depending on the attribute's minimum cardinality: if *minCard* is defined, the deletion may be cascaded to *x*, if *minCard* is undefined (= '?') then *x.a* simply becomes undefined (= '?').

[5] the symetric is true for primitive `SC.enter(o)`, which automatically forces object *o* to enter into any superclass of *SC* where it was not yet defined.

The following example illustrates this on a class of orders, each order containing several order lines.

```
>>> Ord = f2.class_create(className = 'Order',
...         schema={'doc_number' : f2.Int,
...                 'established': f2.Time,
...                 'order_by' : f2.Client }
...         )
>>> OL = f2.class_create(className='OrderLine',
...         schema={'in_order' : (Ord, 1, 1),
...                 'prod' : f2.Product,
...                 'Q' : f2.Int,
...                 'unit_price' : f2.Real }
...          )
```

Order lines do only exists inside an order and belong to exactly one order, this is expressed by the cardinalities (1,1) of attribute *in_order*. With this schema, the execution of statement:

```
>>> Ord.leave(my_order)
```

will also provoque the deletion of all order lines ol for which $ol.in\_order = my\_order$. This is the traditional cascade effect. In F2, this effect is controlled by the schema definition (*minCard* of attributes) instead of options of a delete primitive, or options of constraints declarations (like in SQL).

4 The F2 kernel

4.1 Classes to define classes, attributes, etc.

The initial state of a F2 database, called the F2 kernel, contains the minimal set of classes and objects necessary to describe F2 schemas. This is equivalent to the *catalog* or *dictionary* part of other database management systems. It contains essentially the class of all classes, the class of all attributes, etc. The figure 1 gives an UML schema of the F2 kernel. A class (an object of class *CLASS*) is either an *AtomClass* or a *TupleClass*.

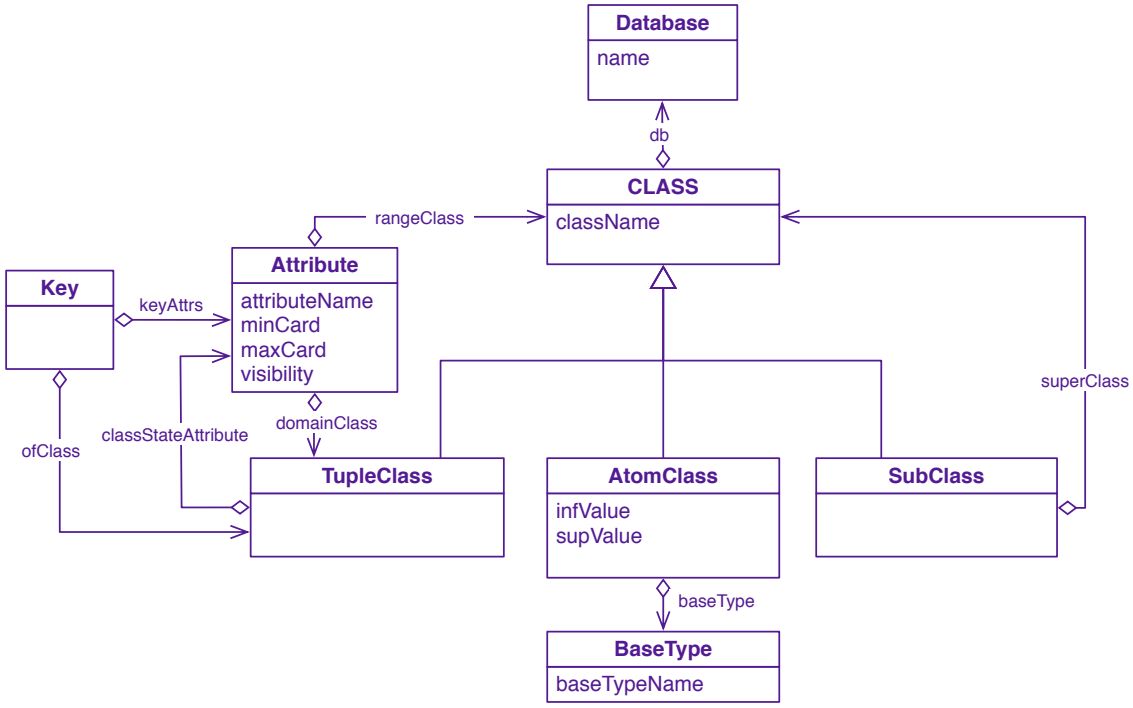

Figure 1: UML schema of the F2 kernel

AtomClasses describe atomic objects (a string, an integer, or a real number are atomic objects). We know the value domain *baseType* of the class, possibly restricted by bounds (*infValue* and *supValue*. Atomic objects are their own identifiers.

TupleClasses describe composite objects (tuples), defined with the help of other classes through attributes. Each attribute has a name, a domain class (*domainClass*) and a range of values (*rangeClass*). An attribute *a* may also be interpreted as a function from $domainClass(a)$ to $2^{rangeClass(a)}$. The size of values may be bounded by *minCard* & *maxCard* [6].

SubClasses describe specialization of other classes (or ISA-links). In the full F2 model, a subclass may be either an AtomClass or a TupleClass. In the present version of F2-Python, only subclasses of TupleClasses (traditional subclasses) are supported.

Classes from the F2 kernel contain their own description, so you can query an initial F2 database (like the dictionary of any DBMS) to retrieve information about it.

[6] more precisely, for an object o having an attribute a, the range of possible values for $a(o)$ (also noted $o.a$) contains all finite subsets s of $rangeClass(a)$, with $minCard \leqslant \|s\| \leqslant maxCard$

```
>>> f2.CLASS().className
['Attribute', 'CLASS', 'AtomClass', 'TupleClass', 'SubClass', 'EntityState',
 'Int', 'Real', 'Time', 'String', 'DicIdent', 'AnyValue', 'BaseType',
 'Department', 'Person', 'Employee']


>>> f2.TupleClass().className
['Attribute', 'CLASS', 'AtomClass', 'TupleClass', 'SubClass', 'BaseType',
 'Department', 'Person', 'Employee']


>>> f2.AtomClass().className
['EntityState', 'Int', 'Real', 'Time', 'String', 'DicIdent', 'AnyValue']

>>> f2.Employee.superClass
<CLASS.Person>

>>> for attr in f2.Attribute(domainClass = f2.Person):
...     print attr.attributeName, '->', attr.rangeClass.className
...
statePerson -> EntityState
lastname -> String
firstname -> String
```

Here you may note that the class *Person* of our example has in fact three attributes: *firstname*, *lastname* and *statePerson*. The latest one has been automatically added by F2 and stores the state of each object of this class. This allows, among other things, for a (tuple-)class to have distinct objects even if it doesn't hold any attribute. This is convenient for some subclasses or for tupleclasses which will acquire attributes later on.

4.2 Schema creation and evolution

Classes in the F2 kernel are handled like any other class: you can create new attributes or classes using the `C.create()` and `C.enter(o)` primitives. This is in fact the only direct way to manipulate a schema: apply primitives or assign values to objects of these classes. The `f2.class_create()` method shown in section 3.2 page 6 is actually only a syntactic short path to ease the creation of an F2 schema. In our example of Persons and Employees, `class_create()` actually calls the following F2 primitives:

```
>>> P = f2.TupleClass.create(className = 'Person', db=f2.test)
>>> fn = f2.Attribute.create(attributeName = 'firstname',
...                          domainClass = P,
...                          rangeClass = f2.String)
>>> ln = f2.Attribute.create(attributeName = 'lastname',
...                          domainClass = P,
...                          rangeClass = f2.String)
```

After this, you can immediately start to create objects in class *Person*. You could even start to create objects before you have defined any attributes. The objects already existing will simply take the null or undefined value, marked by a '?' for any new attribute added later.

Similarly, the `f2.add_key()` which declares *firstname* and *lastname* attributes as a key in class Person, does it simply by executing:

```
>>> k = f2.Key.create(ofClass = P, keyAttrs = [fn, ln])
```

The creation of subclass *Employee* (see example page 6) is hardly longer:

```
>>> E = f2.TupleClass.create(className = 'Employee', db=f2.test)
>>> E = f2.SubClass.enter(E, superClass = P)
>>> wi = f2.Attribute.create(attributeName = 'work_in',
...                          domainClass = E,
...                          rangeClass = f2.Department,
...                          minCard = 1,
...                          maxCard = sys.maxint )
>>> ma = f2.Attribute.create(attributeName = 'manager',
...                          domainClass = E,
...                          rangeClass = P)
```

Several attributes of F2 kernel classes have a *minCard* = 1. This is the case for *domainClass* and *rangeClass*. As F2 rules apply uniformly on all classes, they also apply to kernel classes: if you delete a class (for instance `f2.CLASS.leave(f2.Person)`) then all its attributes will disappear at the same time, plus any attribute for which *Person* is a range (a *rangeClass*). Now what happens to class *Employee* ? If you look at the kernel schema of figure 1, you can see that the two classes are related by attribute *superClass* of class *Subclass* (`f2.Employee.superClass == f2.Person`). The *minCard* of *Employee* is also = 1. So the suppression of class *Person* has triggered a `f2.SubClass.leave(f2.Employee)`. As a consequence, *Employee* is not a subclass anymore, but it remains a tuple-class, and does not disappear completely from the DB. Only, it lost its super-class *Person*, and objects from *Employee* have lost their *Person's* attributes (firstname, lastname), but they still have their *Employee's* attributes.

This example shows that a good knowledge of the F2 kernel schema and content is necessary to anticipate effects of F2 primitives applied to schema objects, in other words primitives applied to schema evolution.

5 F2 reference guide

5.1 The F2 python package

The F2 package contains all the necessary classes and functions to access an F2 database from python.

connect (*storage_name = 'file:root.db', username = '', passwd = ''*)
Opens an F2 database connection and returns a connection object. The storage_name may have two forms:

- file:<local pathname to file storage> this opens a unique (unshared) access to an F2 database located in the specified file.
- rpc:<host_address>:<port_number> this opens a shared access to an F2 server at the specified (IP-)address and on the specified (TCP-)port number. If no address is given, the local host (127.0.0.1) is used, if the port number is not precised, port 8080 is assumed.

Parameters *username* and *passwd* are only used if authentication is activated (see section 2.6). The object returned is a F2_Connection instance, which methods are described below. Example:

```
>>> my_f2 = F2.connect('rpc:127.0.0.1:8080')
```

Methods of a F2 connection object

An F2 connection object acts as a namespace for an F2 storage: all current classes existing in the storage (i.e. all *classNames*) may be referenced by their names as attributes of the connection object:

```
>>> my_f2.Person
<CLASS.Person>
>>> my_f2.CLASS
<CLASS.CLASS>
```

Sometimes, several classes have the same name in different databases of the same storage. In order to distinguish them, you can prefix their name by the name of the database.

```
>>> my_f2.TestBase.Person == my_f2.PeopleBase.Person
False
```

If a class is unique in the whole storage, prefixing is not necessary [7]. It is recommended to avoid creating classes with exactly the same name as those from the Kernel. Avoid naming your own class *CLASS*, *Attribute*, *Database*, and so-on. It would quickly introduce some confusion in your code.

class_create (*className, superClass=None, database=None, schema={ }*)
A syntactic shortcut for declaring classes. Example:

```
db = F2.connect('file:my_database_file.db')
f2.class_create(className='Person', database='PeopleBase'
                schema={'firstname': f2.String,
                        'lastname' : f2.String,
                        'work_in'  : f2.Department})
```

schema is a python *dict* interpreted as a list of attributes for the future class. For each attribute, cardinalities (min

[7] in other words, if there is only one class Person in your storage, you can safely use the expression my_f2.Person

& max) may be precised in a tuple with the attribute's range or *rangeClass*. See section 3.2 for further examples. If argument *superClass* is given, then F2 creates and returns a sub-class of the given class is created. *database* is the database schema to which the new class will be attached, it is optional. If *None*, the created class will not be attached to any specific database schema. For convenience, this argument may either be a string giving the name of a schema (it will be created on the fly with that name if it doesn't exist yet), or a *F2_Object* instance of class *Database* (the class of all database schemas).

add_key (*ofClass, attributeName (, attributeName)**)
A syntactic shortcut for adding a key to a class definition, or more simply to add an object in kernel class *Key*. The key is made of tuple of values on each of the given attribute names. Example:

```
newKey = f2.add_key(f2.Person, 'firstname', 'lastname')
```

commit ()
Commits current transaction. This is almost equivalent to using the `get_transaction().commit()` primitive of ZODB (See ZODB documentation for details).

rollback ()
Rollbacks current transaction. This is almost equivalent to using the `get_transaction().abort()` primitive of ZODB (See ZODB documentation for details).

pack ()
Forces the actual storage to pack ressources. If underlying storage does not support this function, the call is without effect.

close (*commit_needed=True*)
Last operation on a F2 connection before application stops. If commit_needed is True (which holds by default), the current transaction is commited before closing connection.

html_repr (*some_value, depth=1, using=None*)
Produces an HTML representation of *some_value*, which may be either a single F2_Object, or a F2_Object_list. Recursively represents included values down to a depth of *depth*. If *using* is specified, then it is a list of attributes to be used to display the values, if *using* is *None* then all attributes applicable to this value (included inherited attributes) will be used.

json_repr (*some_value, depth=1, using=None*)
Produces a JSON representation of *some_value*, which may be either a single F2_Object, or a F2_Object_list. Recursively represents included values down to a depth of *depth*. If *using* is specified, then it is a list of attributes to be used to display the values, if *using* is *None* then all attributes applicable to this value (included inherited attributes) will be used.

5.2 F2.F2_Object and F2.F2_Object_list

The type of objects returned by a class selection or the evaluation of another object's attribute. In other words, these types are containers for results of F2 queries. If the result is an F2 object, the python type is *F2_Object*, if it is a list of F2 objects, the python type is *F2_Object_list*.

F2_Object

An *F2_object* is a python box to hold a reference to an object in the F2 DBMS. It is actually built on a pair (*_klass*, *_rank*), where *_rank* is the internal object id (OID)[8] and *_klass* is the internal class id. You never need to manipulate directly these two OIDS, but it may be interesting to remember that an F2 object is always defined relatively to one of its classes.

[8] called "rank" only for historical reasons.

Attributes of F2 objects are evaluated using standard python dotted notation. According to actual cardinality of attributes *a*, the result may either be a single value or a list of values. The same notation is used to assign new values to object's attributes. If *x* is a F2 Object, and *a,b,c* are attributes in the class schema of x, then all the examples below are syntactically correct:

- `x.a`
- `getattr(x, 'a')`
- `x.a.b.c`
- `x.a = y`
- `setattr(x, 'a', y)`
- `if hasattr(x, 'a'):  ...`

This notation is extended for *flat* evaluation by prepending a '_' (underscore) in front of the name of an attribute. This collapses the possible list of results into a set (each element is distinct). So the flat evaluation of `x.a` is `x._a`. The flat extension cannot be used for values assignment.

If the result of an attribute evaluation is empty, then the result is the python empty list `[]`, if it contains only one element, then the result is this element (rather than a singleton containing this element).

Two F2_Object's can be compared, they are equal if they designate the same F2 object (even if they contain different *_klass* in the same class hierarchy). They can be compared to the null value *F2.F2_NULL*. The truth value of a F2_Object is being different from *F2.F2_NULL* (i.e. being defined). Examples:

```
>>> bob_dept = bob.department
>>> bob_dept == F2.F2_NULL
True
>>> if bob_dept:
...   print bob_dept.name
... else:
...   print "Bob's department undefined."
...
Bob's department undefined.
```

exist_object()
: True if object exists (is defined) in the current class (the *_klass* part of this object). If you obtained this object from an answer to a F2 query, it is most probably already true. If you are modifiying the F2 DBMS state while you are holding this reference, this may become false.

exist_in(*in_class*)
: True if object exists in class *in_class*. This is the standard way to check that an object belongs also to a sub-class, like `John.exist_in(f2.Employee)`.

F2_Object_list

F2_Object_list is a standard python list extended with dotted notation for both normal and flat evaluation of attributes. Each attribute is evaluated on every object of the list. The result is the concatenated result of all these evaluations. This essentially allows the chaining of attributes evaluation to work even in the case of multi-valued attributes. Flat evaluation on *F2_Object_list* implies that attributes evaluation never return lists of lists but flat sets of distinct values instead. Example:

```
>>> Countries.name
['France', 'Deutschland', 'USA', 'Italy']
>>> [c.flag.colors.name for c in Countries]
[['blue', 'white', 'red'],
 ['black', 'red', 'yellow'],
 ['red', 'blue', 'white'],
 ['green', 'white', 'red']]
>>> Countries.flag._colors.name
['red', 'blue', 'yellow', 'green', 'white', 'black']
```

5.3 F2.F2_Class

A python subclass of *F2_Object* used to handle instances of class *CLASS*. F2_Class constructor may take two forms:

- `F2_Class(class_name)` where *class_name* is a string, the result is the class carrying this name in the current namespace (current schema),
- `F2_Class(an_object)` where *an_object* is a F2_Object, the result returns the class of this object.

`__call__`(*_select={}, _where=None, **selectors*)

This methods defines a query for selecting objects from F2 (tuple-)classes. The result is a list of instances of this class matching the current predicate. If no parameters is given, a list of all instances of the class is returned. Example: `f2.Person()` returns all objects of class *Person*.

If one or several selectors of the form `attribut_name = value` are given, only objects matching simultaneously all conditions are returned (forming an implicit **and**). Example:

```
f2.Person(lastname='Smith', department=dept_fin)
```

If a more complex predicate is needed, the parameter *_where* must be used with a filtering python lambda expression or function. The lambda expression given must return *True* or *False* when evaluated against an object of the selected class. These predicates may be arbitrary complex python expressions. Example:

```
f2.Vehicle(_where=lambda v:v.year_built >= 1999
                        or v.color in ('green', 'black'))
```

Note: in the present version, there is a difference of performance between using simple selectors as shown above and using for the same purpose *_where* predicates: simple selectors can be exploited by the F2 system in order to use underlying indexes, while *_where* predicates are applied *as is* on full scans of objects collections [9]. If simpe selectors and *_where* predicates are used together, as in:

```
>>> joes_vehicles = f2.Vehicle(owner = joe,
...                            model = 'BMW',
...                           _where = lambda v : v.year_built >= 1999
...                              and v.color in ('blue', 'black', 'grey') )
```

then, selectors *owner* and *model* are first applied resulting in a intermediate list of matching objects, then the *_where* predicate is evaluated on objects of this list. So best performance is obtained by extracting whenever possible selectors from the *_where* predicate and expressing them as conjunction of `attribute_name=value` parameters.

The *_select* parameter, if defined, is a python dictionary possibly containing *attr_name:value* pairs. It is used when the name of selectors (i.e. the name of selecting attributes) are not constant strings of your program but stored into variables. Example:

[9] Predicates analysis for query optimization is left for future versions

```
if search_by_firstname:
    first_selector = 'firstname'
else:
    first_selector = 'lastname'
val_1 = 'Sean'
second_selector, val_2 = 'department' , dept_fin
result = f2.Person(_select={first_selector : val_1,
                            second_selector: val_2})
```

create(***attr_values*)
Builds a new object for this class, returns the corresponding *F2_Object*. Values for object's attributes can be immediately assigned. Example:

```
P = f2.Person
joe = P.create(firstname='Joe', lastname='Cool')
```

enter(*an_object, **attr_values*)
Takes an existing object and enter it in this subclass, returns the same *F2_Object*, seen now as an object of this subclass. The object must share the same class hierarchy (must have the same class root) than the class to enter. Values for new object's attributes can be immediately assigned. Example:

```
joe = f2.Employee.enter(joe, department='Accounting')
```

If necessary, the object also enters all super-classes of this class where it was not yet an instance. This means that the operation preserves the *IS-A consistency* of the F2 model.

leave(*an_object*)
Takes an existing object and forces it to leave this class. To preserve *IS-A consistency*, the object also leaves all subclasses of this class. If the class is a root, the object definitely disappears from the database [10]. Example:

```
f2.Employee.leave(joe)
```

The *leave* class method also preserves referential integrity. For each attribute *a* referring to the class to leave, the following rules apply:

- if *a.minCard* is undefined (= '?'), then for each object *x* such that *x.a* = *an_object*, x.a becomes '?'
- if *a.minCard* is defined, then for each *x* where *x.a* would become smaller than *minCard*, *x* is also deleted (forced to leave its class).

root()
Returns the specialization root of this class, returns self if the class is a root.

all_subclasses()
Returns the list of all subclasses (either direct or indirect) of this class. The class itself is not included in the list.

all_attributes()
Returns the list of all attributes of this class, either inherited or not.

direct_attributes()
Returns the list of direct attributes of this class. `C.direct_attributes()` is equivalent to the query: `f2.Attribute(domainClass=C)`.

Using the above methods, it is easy to obtain the complete class tree of a given object *x*:

[10] This is equivalent to a delete, there is no explicit **delete** primitive in F2

```
rootC = F2_Class(x).root()
class_tree = [rootC] + [c for c in rootC.all_subclasses() if x.exist_in(c)]
all_x_attribs = [c.direct_attributes() for c in class_tree]
```

is_SubClass()
: True if is an instance of SubClass.

is_AtomClass()
: True if is an instance of AtomClass.

is_TupleClass()
: True if is an instance of TupleClass.

The three methods above are only here for convenience, they are equivalent to simple F2 queries. `C.is_SubClass()` can be tested with the following expression: `C.exist_in(f2.SubClass)`

5.4 F2.F2_Attribute

A python subclass of *F2_Object* used to handle instances of class *Attribute*. F2_Attribute constructor takes the following form: `F2_Attribute(attribute_name, of_class)` where *attribute_name* is a string, and *of_class* is either the name of a class (a string), or an instance of *F2_Class*. Examples:

```
P = f2.Person
attr_name_of_persons = F2_Attribute('name', of_class=P)
attr_birth_date = F2_Attribute('birth_date', of_class=P)
F2_Attribute('owner', of_class='Vehicle')  # python expression equivalent...
f2.Attribute(attributeName='owner', domainClass=f2.Vehicle) # ...to this F2 query
```

set_reverse_function()
: Forces internal storage to manage also the reverse function mapping of this attribute. Normally each attribute is implemented by one B-Tree mapping oids of objects to the attribute values. The reverse function implements a second B-Tree mapping attribute values to oids. This accelerates considerably the browsing of an attribute in the direction (values to objects), at the cost of some overhead in updates (two B-Trees to update at each object value assignment).

5.5 Extending F2

This section describes methods and interface available to extend F2 (meta-)classes and F2 behaviour. A good understanding of the current F2 model and kernel behaviour is necessary to obtain a successful extension of F2.

[to be written...]

Index