Backend, An RDBMS Backed Object Development Framework

K. Raghu Prasad
Prangya Technologies Pvt. Ltd.

15th August 2002
Revised 29th March 2003

Contents

1 Introduction

BackEnd is an RDBMS backed object development framework written entirely in *Python*. The main goal of development of this framework is to free the developer from the task of writing code for database connection and querying while still using it effectively for storage and retrieval of the attributes of the objects. It provides ways to connect to the database seamlessly and manage database connection-pools. While writing applications, the developer needs to concentrate only on the business logic of the objects being created and used.

If use of the database is more complicated than just storage and retrieval of attribute values, *backend* framework provides easy access to concerned databases from within the object. Developer can obtain a connection object from any member method using which he/she can talk to the database using SQL.

This framework is tested with *MySQL* and *PostgreSQL* databases. It should work[1] with other RDBMS' for which *Python* based client modules are available. The client module must confirm to Database API version 2 (DB SIG 2).

2 License

3 The Architecture

The *backend* framework uses an object oriented design for achieving its goals. Figure 1 shows the composition of its classes. There are four major classes involved; namely *DBMap*, *BackDB*, *DBConnection* and *BackEnd*.

Class *DBMap* handles interpretation of configuration file. Present implementation takes configuration from a disk based file. But by extending this class one can obtain it from a remote host over LAN or WAN. As shown in figure 1, all interfaces of *DBMap* are well contained in *BackEnd* and so they are hidden from the user.

Class *BackDB* handles database connections and connection-pool management. Class *BackEnd* maintains one instance of this class per database being used. Initialization and maintenance of multiple connections for respective databases are done transparently to the user of this framework.

Class *DBConnection* provides a thin wrapper over standard database connection object. Instances of this class are created on request by instances of class *BackDB*.

These classes along with additional utility modules provide basic building blocks for *backend* framework.

[1] As per DB SIG 2 there is no standard order of appearance defined for the arguments for method *connect()*. To overcome certain generalization problems, a hack(*xforms.dbconnect()*) is implemented which might require changes depending on the database module being used.

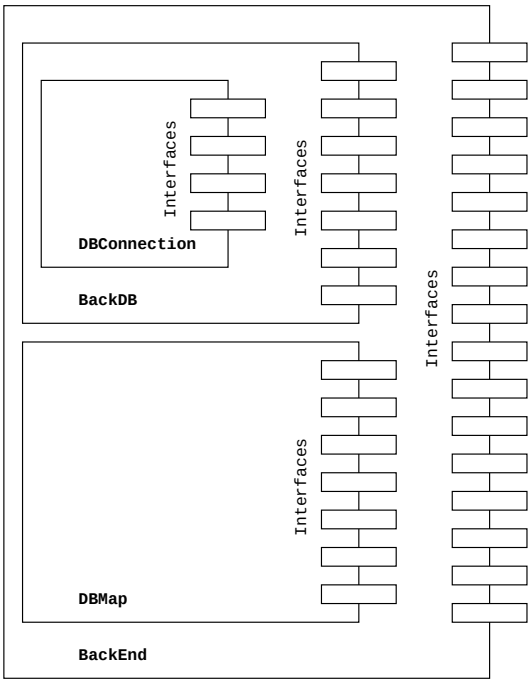

Figure 1: Composition of classes in *backend* framework

4 Support Modules

BackEnd framework uses various standard *Python* modules. It also uses some special python modules available in *prangya* extension series. Read standard *Python* documentation for details of standard modules. Besides them, following additional modules are used in *backend* framework.

4.1 *prangya.excgeneric*

This module is part of *prangya* extension series. It contains a set of generic exception classes required in normal programming practice. Refer respective documentation for further details.

4.2 *prangya.dburlparse*

This module is part of *prangya* extension series. It contains function to parse and tokenize a database URL (called DBURL) used for storing all the details needed for accessing an RDBMS. Refer respective documentation for further details.

4.3 *prangya.randutils*

This module is part of *prangya* extension series. It contains some helper functions for handling pseudo random number related requirements. Refer

respective documentation for further details.

4.4 *prangya.utils*

This module is part of *prangya* extension series. It contains some utility functions. Refer respective documentation for further details.

4.5 *backend*

This module contains a collection of classes to abstract the database backend related activities. It provides interfaces to seamlessly connect to multiple databases, database connection-pool management and object creation and storage from data retrieved from back-end tables *etc.* Here follows the list of the classes defined in this modules along with brief descriptions.

1. **DBConnection:** It is a wrapper class for database connection object supplied by underlying database module. It defines all the interfaces specified in python database API version 2(DB SIG 2). The tasks involved in these interfaces are handled by underlying database module. The only exception being the implementation of method DBConnection.close(). Instead of closing the database connection, it returns the actual Connection object back to the db-pool held by class BackDB described below.

2. **BackDB:** This class manages a pool of database connection objects generated from the underlying database connection module. It provides a framework to the application to obtain a database connection and reserve it for transaction purposes. It is done with the help of identification tags supplied while requesting a database connection. These tags must be unique across an application for unique connection objects. The maximum number of database connection permissible can be set through the interface provided by this class. Generally one instance of this class per database is needed. This class is thread-safe.

3. **DBMap:** This class manages the constraints involved in mapping attributes of various classes to corresponding backend database entities. A configuration file with a structure similar to *Windows*' *.ini* files are used to represent these mappings. It provides interfaces for reading such configuration file and writing existing live configuration back to a file on disk. This class is not thread safe. In threaded applications you should avoid changing attributes of underlying DBMap object after initializing it through a configuration file.

4. **BackEnd:** It is the main engine of *backend* framework. It handles creation of objects from an RDBMS backend. It also updates the same backend, whenever changes are made to the attributes of the objects created by it. To implement backend framework, normally the user needs to study and understand the interfaces of class *BackEnd* only.

   All user defined classes must inherit from this class to use the backend framework. No writes should be made directly on any tables used in any

backend compliant class. In short it means, you must not write directly on any table which is used in the configuration (DBMap object) as this would cause data inconsistency between the database and the objects currently under use. Though class *BackEnd* itself is thread-safe, you must ensure thread-safety of all classes inherited from it. It can be done by implementing proper object locking mechanisms[2] on them.

5 Classes

The details of the classes involved in *BackEnd*framework is described in this section. The details include module names, class and instance attributes, methods of the class and their return values along with possible exceptions. All attributes and methods with prefix of double underscores (__) are private for all types of accesses. Attributes with prefix of single underscore (_) are read-only attributes by convention and hence must not be changed. Attribute *id* of the classes should not be changed either.

5.1 *DBConnection*

5.1.1 Module

Its a member of module *backend*.

5.1.2 Members

Name	Var. Type	Data Type	Description
id	Instance	String	The identification string of the instance.
_paramstyle	Instance	String	Specifies the *paramstyle* attribute of underlying db module. Its value should be one of the following (as defined in module *prangya.utils)*.: DB_FMT_QMARK DB_FMT_NUMERIC DB_FMT_NAMED DB_FMT_FORMAT DB_FMT_PYFORMAT
__closed	Instance	Integer	The flag indicating object's closure.
__dbCon	Instance	Object	The database connection object obtained from the underlying database module.
__lock	Instance	Object	The re-entrant lock object.

[2]Refer documentation of method *Backend.setLock()* for setting locks on user defined objects inherited from class *BackEnd*.

5.1.3 Methods

1. **__init__()**: Its the constructor for this class. The two parameters supplied to it are the identification string of the instance and a database connection object. The later is optional.

Parameter	Data Type	Default	Description
id	String		The identification string of this object.
dbCon	Mixed	0	The database connection object supplied by underlying database module or 0.

Returns: Nothing.

Raises:
ParamException: Invalid parameters supplied.

2. **isAlive()**: Method to identify whether the object is alive or closed for database transactions. It does not take any parameter.

Parameters: Nothing

Returns: True(1) if object is usable for database related activities, else false(0).

Raises: Nothing.

3. **setDBCon()**: Method to associate a database connection object with the instance of this class. If a connection object is not supplied in the constructor, it can be supplied through this function. If a connection object is already supplied, this function call fails.

Parameter	Data Type	Default	Description
dbCon	Object		The database connection object obtained from the underlying database module.

Returns: Nothing.

Raises:
ClosedException: The database connection object is closed.
ParamException: The parameter dbCon is not an object.
ExistsException: A database connection object is already associated with this instance.

4. **setCallbackOnClose():** Method to associate a callback function to this instance. The assigned function will be called when this instance is closed.

Parameter	Data Type	Default	Description
method	Function		The method which is to be called when this object is closed.

Returns: Nothing.

Raises:
ClosedException: The database connection object is closed.

ParamException: The parameter supplied is not a method or function object.

5. **commit()**: Wrapper method to commit transactions over a database connection. It calls the commit() method of underlying database connection object.

   Parameters: Nothing.

   Returns: Nothing.

   Raises:
   ClosedException: The database connection object is closed.
   CommitException: Failure in committing pending transactions.

6. **rollback()**: Wrapper method to rollback to the start of any pending transactions over a database connection. It calls the rollback() method of underlying database connection object.

   Parameters: Nothing.

   Returns: Nothing.

   Raises:
   ClosedException: The database connection object is closed.
   Exception: Failure in rollback operation.

7. **cursor()**: Wrapper method to create a cursor object from the underlying database connection.

   Parameters: Nothing.

   Returns: An instance of the Cursor object.

   Raises:
   ClosedException: The database connection object is closed.
   CursorException: Cursor creation failure.

8. **close()**: Method to return the actual connection object associated with this instance to the creator, *viz.* to the instance of BackDB from where this object was initialized. It is done with the help of a callback function which is registered to this instance of DBConnection. It is invoked from within this method with id of the object as parameter. It effectively takes away the database connection object from this instance of DBConnection. The actual connection object is in fact returned back to the pool of free db connections maintained by the creator *viz.* instance of BackDB. If the attribute holding the callback method is not found, no error is raised and the instance of DBConnection class is silently flagged off as closed.

   Parameters: Nothing.

   Returns: Nothing.

   Raises: Nothing.

5.2 *BackDB*

5.2.1 Module

Its a member of module *backend*.

5.2.2 Members

Name	Var. Type	Data Type	Description
id	Instance	String	The identification string of the database being represented by the instance of this class.
__freeDBQ	Instance	Object	A queue(instance of Queue) of unlimited size to hold the free database connections.
__dbMaps	Instance	Dictionary	A dictionary to map connection-id to database connection objects currently under usage.
__dbConObjs	Instance	Dictionary	A dictionary to map connection-id to list of DBConnection objects created by factory method BackDB.connect().
__initSQLs	Instance	List	List of SQL statements to be executed as soon as a new database connection is made.
__conMax	Instance	Integer	Maximum number of database connections permitted for this database.
__conSema	Instance	Object	A threading.Semaphore object used to control the creation of number of database connections.
__lock	Instance	Object	A threading.RLock object used for various thread-safe operations.
__dbScheme	Instance	String	Access-scheme for this database.
__dbDriver	Instance	String	Name of the database driver-module.
__dbUser	Instance	String	Database user-name.
__dbPassword	Instance	String	Password for the database user.
__dbHost	Instance	String	Name or IP of the host where database server is running.
__dbPort	Instance	String	The port number on which the database server process listens for TCP connections.
__dbSock	Instance	String	The full path to the disk based socket file[3] used to communicate with the database server process.
_dbModule	Instance	Module	The loaded python module specified by __dbDriver.

5.2.3 Methods

1. **__init__()**: Its the constructor for class *BackDB*. It takes two compulsory parameters, first being dbId (database identification string) and the second one dbURL(DBURL string). It does groundwork for managing pools of database-handles and other relevant functionalities.

Parameter	Data Type	Default	Description
dbId	String		The database identification string for the database represented by the instance of this class.
dbURL	String		The DBURL string containing information about the underlying database and the ways to access it.

Returns: Nothing.

Raises:
ParamException: Invalid parameters supplied.
ModuleException: Failure in loading of database driver module.
ConnectionException: Failure in database connection.

2. **connect()**: A factory method to return an instance of class DBConnection containing a database connection object. The id supplied, known as connection-identifier is used to identify and track the connection object. The various steps required for using this framework are as follows:

   (a) Create BackDB instance and set various tuning parameters.
   (b) Call to connect() with an id and obtain a DBConnection object.
   (c) Use that object for various database related operations(SQL).
   (d) Close that DBConnection object by calling method close() of it.
   (e) Repeat step 2b to obtain a new DBConnection object.
   (f) Repeat steps 2c and 2d in that order.

   In the case described above there is no guaranty that the underlying database connection object wrapped up in object DBConnection on step 2e is the same as the one obtained in step 2b. In short, it means that you can't possess exclusive rights on a database connection for an extended period of time in the scheme described above.

   But this method can also be used to reserve a particular database connection exclusively for a series of operations in multiple batches. For this, pass on the same id each time while calling the method connect(). Besides that, do not call the method close() on any of the DBConnection object till you want that connection to be exclusively available to you. The steps required are as follows:

   (a) Create BackDB instance and set various tuning parameters.
   (b) Call to connect() with an id and obtain a DBConnection object.
   (c) Use that object for various database related operations(SQL).

(d) Call connect() with the same id and receive another instance of DBConnection wrapping the same old database connection object within it.

(e) Use this object for various database related operations(SQL).

(f) Repeat steps 2d and 2e as many times as you want.

(g) Call method close() on any one of the DBConnection object obtained in above steps(2b, 2d or subsequent ones).

The calling framework described above is helpful to use transaction facility of underlying database. You can carry out commits and rollbacks over an extended period of time in multiple function calls without worrying about the usage of the same connection by other threads in the same application.

Remember that calling method close() on any of the DBConnection object with certain id will close all the objects created by factory method connect() with that id as parameter.

Parameter	Data Type	Default	Description
id	String		The connection identification string.

Returns: An instance of *DBConnection* (see section 5.1).

Raises:
ParamException: Parameter *id* is not of string type.
ConnectionException: Failure in database connection.
LimitException: Max-limit of database connection reached.

3. **__connect()**: A private method to open a connection to the database server.

   Parameters: Nothing.

   Returns: Nothing.

   Raises:
   ConnectionException: Failure in database connection.
   LimitException: Max-limit of database connection reached.

4. **disconnect()**: Method to release a database connection object from the exclusive possession of some entity like *DBConnection* object. It should not be called directly as far as possible. It is normally registered as an on-close callback function to the instance of class DBConnection. It calls it when its close() method is invoked. Once this method is invoked with a valid id, any DBConnection object with that id, created through the factory method connect() of BackDB instance, looses its existence. None of them could be used further for any database related activity. Direct call to this method from a program is only advisable under some special cases where the normal usage may turn out to be infeasible.

Parameter	Data Type	Default	Description
id	String		The connection identification string.

   Returns: Nothing.

Raises:
ParamException: No matching *DBConnection* object found for the supplied connection-identification string.

5. **__disconnect()**: Private method to pop one free database connection and close it.

   Parameters: Nothing.

   Returns: Nothing.

   Raises:
   Exception: Disconnection-operation fails.
   LimitException: All database connections are in use.

6. **setInitSQLs()**: Method to register a list of SQL statements with the instance of this class which are to be executed whenever a new database connection is made.

Parameter	Data Type	Default	Description
sqlList	List		The list of SQL statements.

   Returns: Nothing.

   Raises:
   ParamException: Supplied parameter is not a list object.

7. **getInitSQLs()**: Method to retrieve the list of initialization statements (in SQL) currently used by this object.

   Parameters: Nothing.

   Returns: A list containing SQL statements used by the object for initialization of database connection. If none is available, then an empty list is returned.

   Raises: Nothing.

8. **setConMax()**: Method to set the limit on maximum number of connections to the underlying database. If the new limit is greater than the existing one, it will be set successfully and the new limit is returned to the caller. If the new limit is less than the existing one, this method may or may not set it successfully depending on the current usage of database connections. In such cases the limit is reduced as much as possible till the requested value is reached and the actual value set by this method is returned back to the caller.

Parameter	Data Type	Default	Description
maxLimit	Integer		The requested maximum limit on number of database connections.

   Returns: The actual limit set successfully by this method.

   Raises:
   ParamException: The parameter maxLimit is either a non-integer or its zero or negative.

9. **getConMax()**: Method to retrieve current limit on number of database connections.

   Parameters: Nothing.

   Returns: The present limit on number of database connections.

   Raises: Nothing.

10. **closeFree()**: Method to close down the free database connections. The number of connections to be closed is supplied as an argument to this method. If no parameter or 0 is given, all free database connections are closed down. It returns the number of successful closure of connections. It is not necessary that the value returned would match with the value requested.

Parameter	Data Type	Default	Description
num	Integer	0	Number of free (unused) database connections to be closed. The default is all unused connections.

   Returns: The number of successful closure of connections due to this function call.

   Raises:
   ParamException: The parameter num is not a positive integer.

11. **getDBDriverName()**: Method to retrieve the name of the python database client module being used for the underlying connection object.

    Parameters: Nothing.

    Returns: A string containing the name of the database driver module.

    Raises: Nothing.

12. **getInfo()**: Method to retrieve status of internal data structures of this instance. Generally this method is used for debugging only. It should never be used to retrieve data for any other purpose as there is no thread-locks in place during data-retrieval.

    Parameters: Nothing.

    Returns: String-representation of a dictionary containing name-value pairs of public and private data of this class. Remember to not to use this data for any purpose other than debugging this class. That too should be done in a single threaded environment.

    Raises: Nothing.

5.3 *DBMap*

5.3.1 Module

This class is defined in module *backend*.

5.3.2 Members

Name	Var. Type	Data Type	Description
configFile	Instance	String	Configuration file path.
__catcMaps	Instance	Dictionary	Storage for class-attribute-tablecolumn mappings.
__uncTabCols	Instance	Dictionary	Storage for class-table-column(of un-comparable datatype) maps.
_ctcaMaps	Instance	Dictionary	Storage for class-tablecolumn-attribute mappings.
_ctcaRels	Instance	Dictionary	Storage for class-table-column-attribute relationships.
__tableRels	Instance	Dictionary	Storage for table-to-table relationship mappings.
__classDBMaps	Instance	Dictionary	Storage for class-dbname mappings.
__dbURLs	Instance	Dictionary	Storage for database-DBURL mappings.
__dbCons	Instance	Dictionary	Storage for database to number of dbconnection mappings.
__idInsertVals	Instance	Dictionary	Class name to id-insertion value mappings.
__initSQLs	Instance	Dictionary	dbId to initialization SQL maps.
__lastIdSQLs	Instance	Dictionary	dbId to id-retrieval SQL maps.
__delTables	Instance	Dictionary	Class name to deletion table maps.
_noInsTables	Instance	Dictionary	Mapping of class names to table list for no insertion.
_noUpdTables	Instance	Dictionary	Mapping of class names to table list for no update.

5.3.3 Methods

1. **__init__**(): Constructor to initialize class to database mappings either from a configuration file or with empty parameters. Its not necessary to have a configuration file in advance. You can create an empty DBMap object and then set respective values into it. Then the whole configuration can be written to the file whose name and path is supplied in the constructor.

Parameter	Data Type	Default	Description
configFile	String		Full path to the configuration file.

   Returns: Nothing.

   Raises: *ParamException*: The parameter configFile is not of string type or required permissions not available on that file or the directory containing it.

2. **read**(): Method to populate the internal data structures of this class with the data obtained from the configuration file. If configuration file is not

supplied as a parameter, the one supplied during the construction of the object (*self.configFile*) is used for this purpose.

Parameter	Data Type	Default	Description
confFile	String	None	Full path to the configuration file.

Returns: Nothing.

Raises:
ParamException: The parameter confFile is not of string type.
FileException: Error in accessing configuration file.
ConfigException: Inconsistency or error in configuration file.
ConfigParser.DuplicateSectionError: Duplicate sections found in the configuration file.
ConfigParser.MissingSectionHeaderError: Config file contains no section headers.
ConfigParser.ParsingError: Failure in configuration file parsing.

3. **write()**: Method to write current configuration into a disk file. If the full file path is not provided as a parameter, the one supplied during the construction of the object (self.configFile) is used instead.

Parameter	Data Type	Default	Description
confFile	String	None	The full path of the file into which the configuration is to be written.

Returns: Nothing.

Raises:
ParamException: The parameter confFile is not of string type.
IOError: Failure in opening/writing configuration file.

4. **getUncomparables()**: Method to retrieve the list of columns having uncomparable data types like float or double.

Parameter	Data Type	Default	Description
className	String		The name of the class for which the data is being retrieved.
tableName	String	None	The name of the table associated with given class for which the un-comparable columns are being retrieved.

Returns: A list or dictionary containing details of un-comparable columns of tables involved for given class. If table name is provided, then a list of un-comparable columns in that table are returned. Else a dictionary with table-name as key and a list of such columns as value is returned.

Raises:
ParamException: The parameter *className* or *tableName* is not of string type.
NotExistsException: The parameter *className* is not found in the configuration.

5. **setUncomparables**(): Method to set or remove un-comparable column types for a class and table.

Parameter	Data Type	Default	Description
className	String		Name of the class which must not be None or empty string.
tableName	String	None	Name of the table which can be None or empty string if all existing un-comparable column details for given class are to be removed from existing configuration. If it is empty or *None* then parameter *columnList* is not used or tested.
columnList	List	None	A list containing the columns of given table. If it is *None* or empty, then all existing (uncomparable) columns for the given table are removed from the configuration. Else supplied ones are set after removing existing columns for the given table.

Returns: Nothing.

Raises:
ParamException: The parameters *className* or *tableName* are not string objects or *columnList* is not list object.
NotExistsException: The parameter *className* is not found in the configuration.

6. **columnIsUncomparable**(): Method to test whether supplied column of given table associated with given class is un-comparable or not.

Parameter	Data Type	Default	Description
className	String		Name of the class which must not be None or empty string.
tableName	String		Name of the table. Invalid table name does not raise any exception. In such case the return value is 0.
columnName	String		The name of the column which is to be tested. Invalid column name does not raise any exception. In such case the return value is 0.

Returns: 1 if column is un-comparable. In all the other cases including invalid table or column names, 0.

Raises:
ParamException: Any of the parameters *className*, *tableName* or *columnName* are either empty or not string objects.

NotExistsException: The parameter *className* is not found in the configuration.

7. **getNoUpdateOnTables**(): Method to retrieve the list of tables whose fields should not be updated while updating certain attribute(s) of a object of given class.

Parameter	Data Type	Default	Description
className	String		Name of the class.

Returns: A list containing the table names which should not be updated while updating the attributes of object of given class.

Raises:
ParamException: The parameter *className* is not of string type.
NotExistsException: The parameter *className* is not found in the configuration.

8. **setNoUpdateOnTables**(): Method to set the list of table names for a particular class which should not be touched while updating certain attribute(s) of the object of supplied class. Make sure that the table-names supplied are the correct ones for the class involved. No test is being carried out to check its validity. Wrong table names may produce unpredictable results including data corruption. Though empty list is allowed which will reset the dictionary self.noInsTables to empty.

Parameter	Data Type	Default	Description
className	String		The name of the class for which table list is to be set.
tabList	List		The list of tables which should not be touched while updating attribute(s) of the object of the given class. It will be added to the existing table-names in the configuration. If tabList is an empty list, all the existing table-names will be removed from the configuration.

Returns: Nothing.

Raises:
ParamException: The parameters *className* is not of string type or the parameter *tabList* is not of list type.
NotExistsException: The parameter *className* is not found in the configuration.

9. **getNoInsertOnTables**(): Method to retrieve the list of tables into which the records should not be inserted while creating a new object of given class.

Parameter	Data Type	Default	Description
className	String		Name of the class.

Returns: A list containing the table names which should not be touched while inserting the attributes of object of given class.

Raises:
ParamException: The parameter *className* is not of string type.
NotExistsException: The parameter *className* is not found in the configuration.

10. **setNoInsertOnTables()**: Method to set the list of table names for a particular class into which records will not be entered while creating a new object of supplied class. Make sure that the table-names passed on are the correct ones for the class involved. No test is being carried out to check it. Wrong table names may produce unpredictable results including data corruption. Though empty list is allowed which will reset the dictionary self.noInsTables to empty.

Parameter	Data Type	Default	Description
className	String		The name of the class for which table list is to be set.
tabList	List		The list of tables which should not be touched while inserting new data for a newly created object. It will be added to the existing table-names in the configuration. If tabList is an empty list, all the existing table-names will be removed from the configuration.

Raises:
ParamException: The parameters *className* is not of string type or the parameter *tabList* is not of list type.
NotExistsException: The parameter *className* is not found in the configuration.

11. **getDeleteOnTables()**: Method to retrieve the list of tables from which the entries for a particular object of given class is to be deleted.

Parameter	Data Type	Default	Description
className	String		Name of the class.

Returns: A list containing the table names from which record should be deleted while deleting the object of given class.

Raises:
ParamException: The parameter *className* is not of string type.
NotExistsException: The parameter *className* is not found in the configuration.

12. **setDeleteOnTables()**: Method to set the list of table names for a particular class from which the entries should be deleted when removing that class' object from the backend database. Please make sure that the table-names passed on are the correct one for the class involved. No test is being carried out to check it. Wrong table names may produce unpredictable results including data corruption. Though empty list is allowed. In such cases there would be no delete operations possible on instances of corresponding class.

Parameter	Data Type	Default	Description
className	String		The name of the class for which table list is to be set.
tabList	List		The list of tables involved in object-deletion.

Returns: Nothing.

Raises:
ParamException: The parameters *className* is not of string type or the parameter *tabList* is not of list type.
NotExistsException: The parameter *className* is not found in the configuration.

13. **getLastIdSQL**(): Method to retrieve the SQL statement required to obtain the last (object)id inserted into the database table.

Parameter	Data Type	Default	Description
dbId	String		Database identification string of the database for which the id-retrieval statement(SQL) is to be obtained.

Returns: An SQL statement or and empty string if none available.

Raises:
ParamException: The parameter dbId is not of string type.
NotExistsException: The parameter dbId is not found in the configuration.

14. **setLastIdSQL**(): Method to set the SQL statement needed to retrieve last inserted object id from a table.

Parameter	Data Type	Default	Description
dbId	String		Database identification string of the database for which the id-retrieval statement(SQL) is to be set.
sql	String		SQL statement.

Returns: Nothing.

Raises:
ParamException: The parameters dbId or sql are not of string types.
NotExistsException: The parameter dbId is not found in the configuration.

15. **getInitSQLs()**: Method to retrieve the list of SQL statements for initializing connections for given database.

Parameter	Data Type	Default	Description
dbId	String		The database identification string for which the SQLs are to be retrieved.

Returns: A list containing SQL statements.

Raises:
ParamException: The parameter dbId is not of list type.
NotExistsException: If parameter dbId is not found in the configuration.

16. **setInitSQLs()**: Method to set the SQL statements needed to initialize a database as soon as a new connection is made.

Parameter	Data Type	Default	Description
dbId	String		Database identification string of the database for which the initialization statements(SQLs) are to be set.
sqlList	List		List of SQL statements.

Returns: Nothing.

Raises:
ParamException: The parameter dbId or sqlList are not of string type of list type respectively.
NotExistsException: The parameter dbId is not found in the configuration.

17. **getIdInsertValue()**: Method to retrieve the value to be used in SQL (for insert query) for attribute id of given class.

Parameter	Data Type	Default	Description
className	String		Name of the class.

Returns: A string to be used in insert query as value for attribute id of given class.

Raises:
ParamException: The parameter className is not of string type.
NotExistsException: The parameter className is not found in the configuration.

18. **setIdInsertValue()**:Method to set id-value for insert query for attribute id of given class.

Parameter	Data Type	Default	Description
className	String		Name of the class for which the id value is being set.
insertVal	String		A string to be used as id-value in insert query for given class.

Returns: Nothing.

Raises:
ParamException: The parameters insertVal or className are not of string types.

19. **getRelations()**: Method to retrieve the list of relationships amongst table-columns for given class.

Parameter	Data Type	Default	Description
className	String		Name of the class.

Returns: List of strings specifying table-column relationships for given class.

Raises:
ParamException: The parameter className is not of string type.
KeyError: If parameter className is not found in the configuration.

20. **setRelations()**: Method to set table-column relationships for a particular class.

Parameter	Data Type	Default	Description
className	String		Name of the class for which table-column relationship is being set.
relations	List		A list containing all table-column relationships meant for a class.

Returns: Nothing.

Raises:
ParamException: The parameters className or relations are not of string type or list type respectively.

21. **getNumConnection()**: Method to retrieve the maximum number of connection permitted for a database identified by dbId.

Parameter	Data Type	Default	Description
dbId	String		Database identification string.

Returns: An integer specifying the maximum number of database connections permitted for given database.

Raises:
ParamException: The parameter dbId is not of string type.
KeyError: The parameter dbId is not found in the configuration.

22. **setNumConnection()**: Method to set limit on maximum number of connections for given database.

Parameter	Data Type	Default	Description
dbId	String		The database identification string for which the number of connection is being set.
numCon	Integer		Integer specifying the number of connection.

Returns: Nothing.

Raises:
ParamException: The parameters dbId or numCon are not of string type or integer type respectively.

23. **getDBURL()**: Method to retrieve the DBURL for a database identified by dbId.

Parameter	Data Type	Default	Description
dbId	String		The database identification string.

Returns: DBURL.

Raises:
ParamException: The parameter dbId is not of string type.
KeyError: If parameter dbId is not found in the configuration.

24. **setDBURL()**: Method to associate a database with a DBURL used to access it.

Parameter	Data Type	Default	Description
dbId	String		The database identification string for which the DBURL is being set.
dbURL	String		The DBURL.

Returns: Nothing.

Raises:
ParamException: The parameter dbId or dbURL is not of string type.

25. **getDBId()**: Method to retrieve the database identification string associated with a class.

Parameter	Data Type	Default	Description
className	String		The name of the class(class section name in config file).

Returns: Database identification string.

Raises:
ParamException: The parameter className is not of string type.
KeyError: If parameter className is not found in the configuration.

26. **setDBId()**: Method to associate a class with a database identification string. The description-entry for that class must exist in the configuration already, either from configuration file or through the method __setitem__().

Parameter	Data Type	Default	Description
className	String		The name of the class which is to be mapped to given database identification string.
idString	String		The database identification string.

Returns: Nothing.

Raises:
ParamException: The parameter className or idString is not of string type.

27. **__getitem__()**: Method to let the instance of this class mimic the behavior of a dictionary. Its used to retrieve data relevant to object-attributes to table-column mappings of the database backend.

Parameter	Data Type	Default	Description
item	String		The name of the class(class section name in config file).

Returns: A dictionary containing the attribute maps of that class, provided it exists in the config file. Else raises exception.

Raises:
ParamException: The parameter item is not of string type.
KeyError: If class item is not specified in the config file.

28. **__setitem__()**: Method to (implicitly) add a class name to the configuration and initialize it with a given value.

Parameter	Data Type	Default	Description
item	String		The name of the class to be added into the configuration.
value	Dictionary		The initialization value. It must be a dictionary.

Returns: Nothing.

Raises:
ParamException: The parameter item is not of string type or value is not of dictionary type.

29. **getAttributeName()**: Method to obtain the name of the attribute of certain object of class className used to store the data of given column columnName of table tableName.

Parameter	Data Type	Default	Description
className	String		The name of the class whose instance is under consideration.
tableName	String		The Name of the database table whose column is given.
columnName	String		The Name of the column of given table.

Returns: The name of the attribute related to given table and column names.

Raises:
ParamException: The parameters className, tableName or columnName are not string objects.
ConfigException: Failure to retrieve the attribute-name due to inconsistency in the configuration.

30. **classes()**: Method to retrieve the list of classes defined in the configuration file.

Parameters: Nothing.

Returns: A list containing the names of the classes defined in the config file.

Raises: Nothing.

31. **attributes()**: Method to retrieve the list of attributes defined in the configuration file for the given class.

Parameter	Data Type	Default	Description
className	String		The name of the class whose attribute-names are to be retrieved.

Returns: A list containing the attributes of the given class.

Raises:
ParamException: The parameter className is not of string type.
KeyError: The parameter className is not found in configuration.

32. **dbURLs()**: Method to retrieve all pairs of database identification string and corresponding DBURL.

    Parameters: Nothing.

    Returns: A list of pairs of database identification string and respective DBURL(in tuple form).

    Raises: Nothing.

33. **getInfo()**: Method to retrieve status of internal data structures of this instance. Generally this method is used for debugging only. It should never be used to retrieve data for any other purpose.

    Parameters: Nothing.

    Returns: String-representation of a dictionary containing name-value pairs of public and private data of this class. Remember to not to use this data for any purpose other than to debug this class.

    Raises: Nothing.

5.4 *BackEnd*

5.4.1 Module

This class is defined in module *backend*.

5.4.2 Members

Name	Var. Type	Data Type	Description
__initialized	Class	Integer	Flag to test class initialization.
__dbMap	Class	Object	DBMap object holding details of classes, their attributes and respective database tables and columns.
__bdbs	Class	Dictionary	Class-name to BackDB object mappings.
__sqls	Class	Dictionary	Class-name to object construction SQL mappings.
__rsOrder	Class	Dictionary	Class-name to order of columns in result-set mappings.
__delQueries	Class	Dictionary	Class-name to deletion-query-list mappings.

5.4.3 Methods

1. **__init__()**: Constructor to initialize class BackEnd. There are two modes of possible initializations. To initialize the class members used for data management tasks, this class is initialized with an instance of DBMap. This is called stand-alone mode of initialization. It must be done only

once in an application. No instantiation of the classes inheriting from this class is permitted unless stand-alone initialization is done. In multi threaded applications it is advised to do stand-alone initialization in main thread before starting any child thread. In single threaded applications this should be done at the beginning of the program.

The second mode of initialization is called super-class initialization. In this case no parameter is passed on to the constructor. It is supposed to happen whenever an object is initialized whose class is inherited from *BackEnd*. Here the constructor is called without arguments. The child class must call this method explicitly at the end of its own initialization work; i.e. it should call *BackEnd.__init__(self)*. Normally SQL related code is not needed to initialize an RDBMS backed object. All is done automatically in the super class initialization.

In multi-threaded applications, you should create a re-entrant lock as an attribute of your object and use it whenever updating the object-attributes by calling method *updateSelf()*. Use method *setLock()* (see 20) for setting a lock on your object.

Parameter	Data Type	Default	Description
dbMap	Object	None	An instance of class DBMap.

Returns: Nothing.

Raises:
ParamException: The parameters dbMap is not initialized appropriately or number of database connections specified in configuration is not a positive non-zero integral number.
InitException: Class not initialized in stand-alone mode.
MulticallException: Constructor is called more than once in stand-alone mode.
ConfigException: Inconsistent configuration supplied through dbMap.
ConnectionException: Database connection failed.
LimitException: The maximum limits on database connection usage reached.
ClosedException: The database connection object no longer exists.
CursorException: Failure to create a Cursor object.
ObjectException: Failure to create the object from backend.

Note: This method may raise database related errors too. To catch them dynamically, use *dbModule.Error* in except clauses where *dbModule* may be any one of the database modules being used by BackEnd. The list of all python db modules[4] are returned by method *getDBModules()* (see method 15).

2. **__checkConfig()**: Private method to test the consistency of parameters supplied through the instance of class DBMap.

   Following constraints are expected on any configuration supplied:

   (a) A database identification string must be present for each of the class whose definition is provided in the DBMap object.

[4] All modules must be compliant to Python Database API version 2.

(b) A non-empty DBURL must be mapped for each database identification string which has a mapping with any class defined in the configuration.

(c) SQL for retrieval of Object-id must be present for each database identification string supplied.

(d) The SQL element(*IdInsertValue*) required to insert object-id must be defined for each class.

Parameter	Data Type	Default	Description
dbMap	Object		An instance of class DBMap.

Returns: Nothing.

Raises:
ConfigException: Inconsistency in configuration supplied through dbMap.

3. **dbConnect**(): A factory method to obtain a named DBConnection object for the specified database.

Parameter	Data Type	Default	Description
dbId	String		The database identification string.
conStr	String		The connection-registration string.

Returns: An instance of DBConnection class.

Raises:
ParamException: Either the parameters dbId or conStr are not valid string objects or dbId is not registered with class BackEnd.
LimitException: The maximum limits on database connection-usage reached.
InitException: Class not initialized in stand-alone mode.
ConnectionException: Database connection failed.

4. **registerId**(): Method to assign an identification number to an object of a class derived from BackEnd. Its usually called from the constructor of the user defined (and derived from BackEnd) class.

Parameter	Data Type	Default	Description
id	String		An integer specifying the object id. Usually represented by the primary key of a table in the database.

Returns: Nothing.

Raises:
ParamException: The parameter id is not of type integer or long.

5. **insertSelf**(): Method to insert a set of data(attributes) into the database backend and retrieve the new id of the object. Attributes for all not-null columns of respective backend tables must be supplied as elements of parameter "attributes". Do not pass on the key *id* in it.

Parameter	Data Type	Default	Description
attributes	Dictionary		A dictionary containing the the attributes of the object which is to be created(inserted into database).

Returns: Nothing.

Raises:
ParamException: Parameter "attributes" is not a dictionary.
InitException: Class not initialized in stand-alone mode.
NotExistsException: Certain attribute in the parameter *attributes* is not found in the configuration for given class.
CursorException: Cursor creation failure.
CommitException: Commit-failure on the database transaction.
ConnectionException: Database connection failed.
LimitException: The maximum limits on database connection-usage.

Note: This method may raise database related errors too. See the doc string of method *BackEnd.__init__()* for ways to catch these errors.

6. **__buildQuery_insert()**: Private method to build INSERT queries required to insert the attributes of an object into the backend table. The details of the tables and columns are passed on as a parameter along with the name of the class whose attributes are being inserted.

Parameter	Data Type	Default	Description
className	String		The name of the class for which the attributes are being inserted.
tabDetails	Dictionary		A dictionary of dictionaries. The keys of the first one are the names of the tables. The second dictionary contains the column-names of the table as keys and corresponding attribute-names of the objects as values. The structure is as follows: {table1:{col11:attr11, col21:attr21, ...}, table2:{col12:attr12, col22:attr22, ...}, ...}

Returns: A list of SQL queries built for the table details provided.

Raises: Nothing.

7. **__executeInsert()**: Private method to execute insert queries. First query must be the one to insert fields into the table which contains column for object id. Once insertion is successfully completed, new id is retrieved and passed on to the attribute dictionary, which in turn can be used for further insertion into other tables.

Parameter	Data Type	Default	Description
queries	List		A list containing query-strings to be executed. The queries are executed in the order in which they are stored in this list.
attributes	Dictionary		A dictionary containing the values to be used in the query in attribute(key) and value pairs.
className	String		The name of the class of this object(used to identify the database identification string).

Returns: Nothing.

Raises:
CursorException: Cursor creation failure.
CommitException: Commit-failure on the database transaction.
ConnectionException: Database connection failed.
LimitException: The maximum limits on database connection-usage.
ObjectException: Failure in retrieving id.

Note: This method may raise database related errors too. See the doc string of method *BackEnd.__init__()* for ways to catch these errors.

8. **updateSelf()**: Method to set attributes of an object along with respective columns in database backend.

   Please note that this method does not take care of thread-safety at object level. You need to serialize this method-call explicitly by implementing thread-locking if it is a threaded application. User *setLock()* (see method 20) for setting locks on classes or objects derived from *BackEnd*. In multi-threaded applications, it is advised to acquire re-entrant lock on the instance of your class (which in turn is inherited from class *BackEnd*) while calling this method. Once this method returns (successfully or unsuccessfully) release that lock.

   The choice of class level or instance level locks depends on the number of objects and frequency of update of its attributes. For N number of in-memory instances of the class, making a re-entrant lock as an instance-attribute will consume N times the memory consumed by a single lock object. But it will make each object independent of the other for update purpose as multiple object-attributes can be updated simultaneously. On the other hand if lock is a class-attribute, then for all the instances of the class only one lock object will be used. But this will serialize update calls at the class level. Thus attributes of two instances of the class can not be updated simultaneously even in multi-threaded applications.

Parameter	Data Type	Default	Description
attributes	Dictionary	None	A dictionary containing key-value pairs of attribute-names and its value. You should not set the value for attributes["id"]. If you do, the value supplied is overwritten by object's own id.

Returns: Nothing.

Raises:
ParamException: Parameters attributes is not of dictionary type.
InitException: Class not initialized in stand-alone mode.
ObjectException: Attribute id is not found in the object.
ConnectionException: Database connection failed.
LimitException: The maximum limits on database connection-usage reached.
NotExistsException: Either an attribute being updated is not defined in the configuration of the class or class itself is not defined in the configuration.
CursorException: Failure to create a Cursor object.
CommitException: Commit-failure on the database transaction.
AttributeError: An attribute as specified in the configuration is missing in the object. Usually caused by faulty constructor.

Note: This method may raise database related errors too. See the doc string of method *BackEnd.__init__()* for ways to catch these errors.

9. **__buildQuery_update()**: Private method to build UPDATE queries required to update the attributes of an object. The details of the tables and columns are passed on as a parameter along with the name of the class whose attributes are being updated.

Parameter	Data Type	Default	Description
className	String		The name of the class whose attributes are being updated.
tabDetails	Dictionary		A dictionary of dictionaries. The keys of the first one are the names of the tables. The second dictionary contains the column-names of the table as keys and corresponding attribute-names of the objects as values. The structure is as follows: {table1:{col11:attr11, col21:attr21, ...}, table2:{col12:attr12, col22:attr22, ...}, ...}

Returns: A list of SQL queries built for the table details provided.

Raises:
ConfigException: No where clause found in update query due to logical error in configuration.

10. **__executeUpdate()**: Private method to execute multiple update queries.

Parameter	Data Type	Default	Description
queries	List		A list containing query-strings to be executed. The queries are executed in the order in which they are stored in this list.
attributes	Dictionary		A dictionary containing the values to be used in the query in attribute(key) and value pairs.
className	String		The name of the class of this object(used to identify the database identification string).

Returns: Nothing.

Raises:
CursorException: Cursor creation failure.
CommitException: Commit-failure on the database transaction.
ConnectionException: Database connection failed.
LimitException: The maximum limits on database connection-usage.

Note: This method may raise database related errors too. See the doc string of method *BackEnd.__init__()* for ways to catch these errors.

11. **deleteSelf()**: Method to delete the object (self) from the database. The object in question(i.e. self) must be compliant to the backend framework. Please note that the object (in-memory) is not destroyed when this method is called. It only removes the contents related to the concerned object from backend database. It is the responsibility of the developer to not to use in-memory object anymore e.g. by setting a destroyed flag just before calling this method. Also follow proper thread-safe programming procedure in a multi threaded environment.

    Parameters: Nothing.

    Returns: Nothing.

    Raises:
    ObjectException: Attribute *id* is not found in the object or it is not an integer or long.
    InitException: Class not initialized in stand-alone mode.
    CursorException: Cursor creation failure.
    CommitException: Commit-failure on the database transaction.
    ConnectionException: Database connection failed.
    LimitException: The maximum limits on database connection-usage.

    Note: This method may raise database related errors too. See the doc string of method *BackEnd.__init__()* for ways to catch these errors.

12. **__buildQuery_delete()**: Private method to build DELETE queries required to remove an existing object of given class from the database backend.

Parameter	Data Type	Default	Description
className	String		The name of the class whose instance is to be deleted.

Returns: A list of SQL queries built for the table details provided.

Raises: Nothing.

13. **__executeDelete()**: It refers to the method *__executeUpdate()* (see 10).

14. **__getTablesColumns()**: Private method to fetch names of tables and their columns (with respective attribute-names mapped) involved to build queries for a given set of attributes.

Parameter	Data Type	Default	Description
className	String		Name of the class.
attributes	Dictionary		Attribute-set stored as a dictionary.

Returns: A dictionary where key is the name of the table and value is another dictionary where key is the column-name(of that table) and value is the corresponding attribute-name.

Raises:
NotExistsException: Certain attribute in the parameter *attributes* is not found in the configuration for given class.

15. **getDBModules()**: Method to retrieve all (or any one of) python database modules being used by backend framework.

Parameter	Data Type	Default	Description
dbId	String	None	The database identification string.

Returns: If no parameter is supplied, a list of all python database modules under use are returned. If parameter "dbId" is supplied, it returns the module being used for it.

Raises:
ParamException: The parameter "dbId" is not a string object.
InitException: Class not initialized in stand-alone mode.
NotExistsException: Supplied database identification string is not registered with backend.

16. **classes()**: Method to retrieve the list of classes registered to backend. All classes defined in the configuration must be defined at the time of calling of this method.

Parameter	Data Type	Default	Description
ns	Dictionary	*globals()*	A dictionary with contents of global namespace in key-value form. If this parameter is 'None' or empty string, a list of available class names is returned. Else a list of class objects retrieved from this namespace is returned.

Returns: Depending on the value of parameter 'ns', a list of backend compliant class objects currently registered with class BackEnd or a list of class names present in the configuration.

Raises:
ParamException: Parameter 'ns' is not a dictionary object.
InitException: Class not initialized in stand-alone mode.
NotExistsException: One of the class specified in the configuration is not defined in global namespace or the one supplied in 'ns'.

17. **searchId()**: Method to search and retrieve the ids of instances of class *classObj* matching the parameters supplied in *attributes*.

Parameter	Data Type	Default	Description
classObj	Object		A class object the ids of whose instances are to be retrieved.
attributes	Dictionary		A dictionary containing the attribute-value pairs on the basis of which search is carried out.

Returns: A list of ids if any matches are found, else empty list.

Raises:
ParamException: Parameter classObj is not a class object or attributes is not a dictionary.
InitException: Class not initialized in stand-alone mode.
NotExistsException: Supplied class object is not registered with Back-End.

18. **__buildQuery_select()**: Private method to build SELECT queries required to search a table for ids of an object based on a set of attributes. The details of the tables and columns are passed on as a parameter along with the name of the class whose ids are being searched.

Parameter	Data Type	Default	Description
className	String		The name of the class whose ids are being selected.
tabDetails	Dictionary		A dictionary of dictionaries. The keys of the first one are the names of the tables. The second dictionary contains the column-names of the table as keys and corresponding attribute-names of the objects as values. The structure is as follows: {table1:{col11:attr11, col21:attr21, ...}, table2:{col12:attr12, col22:attr22, ...}, ...}

Returns: A string containing select query(SQL) built for the table details provided.

Raises: Nothing.

19. **__executeSelect()**: Private method to execute a select query on the database associated with given class.

Parameter	Data Type	Default	Description
query	String		A string containing SQL query to be executed.
attributes	Dictionary		A dictionary containing the values to be used in the query in attribute-name(key) and attribute-value(value) pairs.
className	String		The name of the class of this object(used to identify the database identification string).

Returns: A list of tuples. Each tuple contains the number of records being selected through the query. If nothing is retrieved, An empty list is returned.

Raises:
CursorException: Cursor creation failure.
CommitException: Commit-failure on the database transaction.
ConnectionException: Database connection failed.
LimitException: The maximum limits on database connection-usage.

Note: This method may raise database related errors too. See the doc string of method *BackEnd.__init__()* for ways to catch these errors.

20. **setLock()**: Method to set a lock on the objects derived from *BackEnd*. It should be called in the constructor of derived class if an instance based lock is to be created.

Parameter	Data Type	Default	Description
lockName	String		Name of the attribute holding the lock object.
lockObj	Object		A lock object.

Returns: Nothing.

Raises:
ParamException: Parameter *lockName* is not a string object or attribute *lockObj* is not an instance of a class.
InitException: Class *BackEnd*not initialized in stand-alone mode.

21. **getDBConnection()**: Method to obtain an instance of *DBConnection* class for the database involved with this object. This method is supposed to be called through the objects which are children or grand children of *BackEnd*. It must not be called through an instance of class *BackEnd* itself[5].

Parameter	Data Type	Default	Description
conStr	String		A string used to identify the connection object.

Returns: An instance of DBConnection if operation succeeds.

Raises:
ParamException: Parameter *conStr* is either empty or it is not a string

[5]If you need to obtain database connection directly from an instance of *BackEnd*, use method *dbConnect()* (see function 3).

object.
InitException: Class *BackEnd*not initialized in stand-alone mode.
ConnectionException: Database connection failed.
LimitException: Maximum limits on database connection-usage reached.

22. **getInfo()**: Method to retrieve status of internal data structures of the instance/class. Generally this method is used for debugging only. It should never be used to retrieve data for any other purpose as there is no thread-locks in place during data-retrieval.

    Parameters: Nothing.

    Returns: String-representation of a dictionary containing name-value pairs of public and private data of this class. Remember to not to use this data for any purpose other than to debug this class in a single threaded environment.

    Raises: Nothing.

6 Configuration File Format

The configuration file specifying details of classes, attributes and respective database authentication info involved in *backend* framework is defined in a text file. It has a format similar to the .ini files used in *Windows*systems. The file contains multiple sections with section-name enclosed in square braces([]). The option and its value for each section is separated by an equality(=) character. All comments begin with either semi-colon(;) or pound (#) sign. The contents of a sample configuration file with details in comments are given below.

```
; Sample configuration file(dbmap.ini) which can be supplied to the
; constructor of class DBMap. Most of the following discussion
; involves using class DBMap along with class BackEnd. DBMap holds
; just the details required for correct functioning of BackEnd.
; You can consider it just as an interpreter and storage manager
; for configuration data.
[Object-Database Map]
; Class names are mapped to corresponding databases. Here "school"
; and "office" are database identification strings.
Teacher = school
Student = school
Supplier = office
Item = office

[Database-DBURL Map]
; Each database identification string requires corresponding DBURL
; for opening a connection to that database. Here school points to
; a postgres database named schooldb while office points to a mysql
; one named officedb.
school = rdb:pgdb//user1@passwd1:pgserver.example.com:5432/schooldb
office = rdb:MySQLdb//user2@passwd2:localhost:3306/officedb
```

```
[Database-NumConnection Map]
; Maximum number of database connections allowed to be used by the
; application. It should be fixed for each database.
school = 12
office = 8

[Database-InitSQLs Map]
; SQL to be executed for the first time the connection is made.
; Multiple SQL statements can be specified with colon(:) as the
; separator. For switching off auto-commit in mysql, following
; statement is used.
office = SET AUTOCOMMIT=0
; If a particular database does not require any initialization,
; do not mention it in this section as we are not mentioning
; "school" here.

[Database-LastIdSQL Map]
; SQL to be executed for retrieving the last id inserted into the
; table. The macro {TABLE} expands to the correct table name used
; for storing the id of the object of the class for which query is
; being built. The query must be such that only one record of only
; one column is fetched, which is the id itself. Following entries
; show the queries for postgres and mysql respectively. The entry
; for postgres (i.e. "school") works for any SQL-82 compliant RDBMS.
school = SELECT id FROM {TABLE} ORDER BY id DESC LIMIT 1
office = SELECT last_insert_id()

[Class personnel.Teacher]
; Attributes of Teacher object. This class is defined in module
; file personnel.py.
id = teachers.id
name = teachers.name
subject = tsmaps.subjectId
maxQualification = tqmaps.qId

; Since three tables are involved, there must be two relationship-
; definitions separated by commas. They show the relationships
; amongst the columns of these three tables.
{Relations}=tsmaps.teacherId=teachers.id,tqmaps.teacherId=teachers.id

; A sequence in postgres named "teachers_seq" is used to get unique
; ids for each records in table teachers. Following shows the value
; to be used in insertion queries for attribute id.
{IdInsertValue} = nextval('teachers_seq')

; Delete operation on an instance of class Teacher should delete
; records from tables "teachers", "tsmaps" and "tqmaps". So mention
; those table-names here. If some tables are to be ignored, do not
; mention them. Absence of this entry means you do not want to
```

```
; delete the relevant records of the object from the database.
; The order of listing may be important for successful deletion
; of records. The deletion on the table storing object-id will
; always be carried out last irrespective of its position in the
; listing. Also any space character present in the table listing
; will be removed silently.
{DeleteOnTables} = teachers,tsmaps,tqmaps

[Class personnel.Student]
; Attributes of Student object. Attributes of it are stored in
; single table. Hence no relationship is required. Its also defined
; in personnel.py.
id = students.id
age = students.age
minorFlag = students.minor
lastGrade = students.lastgrade
credit = students.netcredit
primarysubject = prisubjects.subject_id
remarks = students.remarks
{IdInsertValue} = nextval('students_seq')
{DeleteOnTables} = students

; Datatype of attribute "credit" is float and it is defined as such
; in the backend database table. Floats and doubles are un-comparable
; data types. They can not be used in the "where" clause of a select
; or update query. You must mention such table-column names
; (separated by commas) here.
{UncompTabCols} = students.netcredit

; If you do not want backend to insert data directly into a table
; even if its column holds certain attribute of the object, you
; can do it with option {NoInsertOnTables}. List those tables
; here separated by commas. Its an optional and rarely used option.
; Same is true for the next option too.
{NoInsertOnTables} = foo, bar, baz

; Similar mechanism is available for updating the backend. If you do
; not wish to update certain tables while updating the attributes of
; the object, list them here separated by commas.
{NoUpdateOnTables} = foo,bar

[Class business.Supplier]
; Attributes of Supplier object. This class is defined in module
; business.py.
id = suppliers.id
name = suppliers.name
address = suppliers.address
remarks = suppliers.remarks
{DeleteOnTables} = suppliers
```

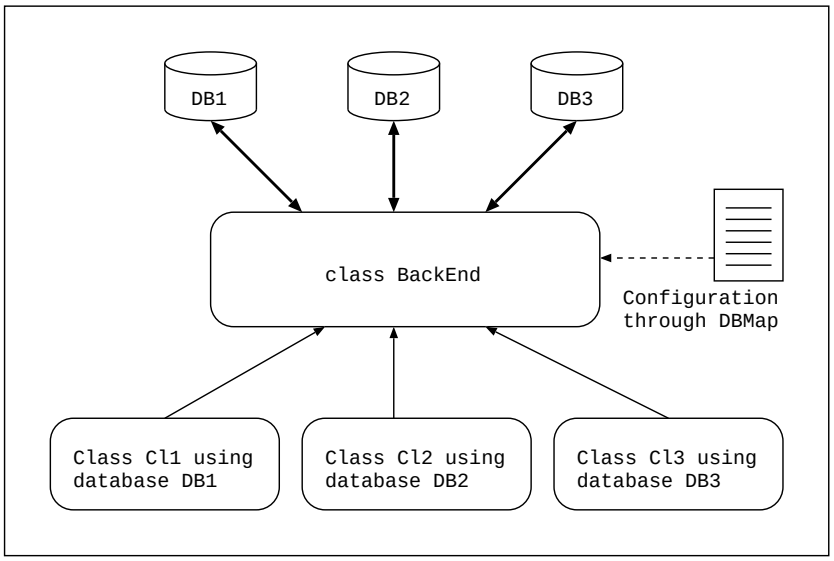

Figure 2: Schematic for using *backend* framework

```
; Since database is mysql, passing empty string in an
; auto-increment field works correctly.
{IdInsertValue} = ''

[Class __main__.Item]
; Attributes of Item object. This class is defined in a file which
; is passed to the python interpreter directly, hence the module
; name "__main__".
id = items.id
name = items.name
batchNo = items.batchno
price = items.price
{IdInsertValue} = ''
{DeleteOnTables} = items
```

7 Using *backend* Framework

The major advantage of *backend* framework over similar system is its ease of use. The user has to create a single configuration file with his/her ideas of the classes, their attributes and corresponding databases involved. Every class compliant to this framework must inherit from class *BackEnd*. Figure 2 shows the general arrangement of user-defined classes in *backend* framework.

7.1 Development Constraints

Following constraints are involved in creating *backend* compliant classes.

1. Every user defined backend compliant class must be inherited from class *BackEnd*.

2. Every class must possess an attribute *id* containing an integer. It stores the unique object-id of instances of that class. Object-id is mapped to a column(primary key) in certain table in the backend database.

3. One class may have attributes from multiple tables. In such cases option {*Relations*} must be set properly. Suppose you have column *name* of table *teachers* and column *subjectId* of table *tsmaps* storing any two attributes of the class *Teacher*. Also primary key of table *teachers* is column *id* and it is mapped(foreign key) to the column *teacherId* of table *tsmaps*. Hence for the records of a particular object of this class you know that *teachers.id* = *tsmaps.teacherId*. This must be specified in option {*Relations*} of class *Teacher* in the configuration file. In this particular case it should be mentioned as:

   {Relations} = teachers.id=tsmaps.teacherId

   Columns mentioned in option {*Relations*} may or may not be part of the attributes of the class under consideration. For further details see section 6 which deals with configuration options.

4. Constructor (*__init__()*) of the class should be able to handle at least two different sets of arguments. The first case involves creating object for the first time where newly supplied data is inserted into the backend database. In this case the attribute *id* (which is created after inserting object's attributes into the database) is not known in advance. Hence a dictionary (say, *dict*) with other available and compulsory attributes should be built with the key being the attribute-name and the the value being the attribute-value. Otherwise the constructor itself can be designed to take up this dictionary as its sole argument. Then call method *self.insertSelf(dict)*.

   In the second case the object is created from the existing data in the backend database. Here attribute *id* is known in advance. So the constructor gets only one parameter, *i.e.* object-id. Whenever only *id* is supplied, first method to be called is *self.registerId(id)*. Then super class' constructor *BackEnd.__init__(self)* should be called. This will populate attributes of the object from values stored in the database.

5. Method *__setattr__()* should be overridden in the class being constructed. Suppose all attributes of the object are stored in the backend RDBMS and all columns and corresponding attributes are mentioned in configuration (see section 6. In this case just a single line definition is enough:

   ```
   def __setattr__(self, item, val):
       self.updateSelf({item:val})
   ```

   If all attributes are not just mappings of their database records, then selective updates are possible too. For all RDBMS backed attributes call method *self.updateSelf({item:val})* where *item* is the attribute-name and *val* is its value to be updated in the database.

7.2 Sample Program

We will build a small program to show the usage of class *BackEnd* in an object oriented way. It is a single class *Client* whose attributes are mapped to the columns of table *clients* in database *dirdb*. Every instance of *Client* represents one record in the table *clients*.

7.2.1 Database

We will use *MySQL-Max* engine for storing this data in the backend. The table type is *innodb*. Following sql script is used to create it.

```
; Mysql script clients.sql.
;
create database dirdb;
use dirdb;
create table clients (
    id integer(11) auto_increment,
    name varchar(80) not null,
    addr varchar(120) not null,
    location varchar(64) default null,
    city  varchar(32) default null,
    state  varchar(32) default null,
    pincode varchar(8) default null,
    phone  varchar(64) default null,
    fax  varchar(64) default null,
    email  varchar(64) default null,
    contact varchar(80),
    primary key (id)
) TYPE=InnoDB;
```

7.2.2 Configuration File

Following is the contents of our configuration file *sample.ini* used to initialize *DBMap*.

```
[Object-Database Map]
; Class names are mapped to corresponding databases.
clients.Client = directory

[Database-DBURL Map]
directory = rdb:MySQLdb//user@pass:mysql.example.com:3306/dirdb

[Database-NumConnection Map]
; Maximum number of database connections allowed.
directory = 3

[Database-InitSQLs Map]
; SQL to be executed for the first time the connection is made.
; Multiple SQL statements can be specified with colon(:) as the
```

```
; separator.
directory = SET AUTOCOMMIT=0

[Database-LastIdSQL Map]
; SQL to be executed for retrieving the last id inserted into
; the table.
directory = SELECT last_insert_id()

[Class clients.Client]
; Attributes of Client object defined in clients.py module.
id = clients.id
name = clients.name
address = clients.addr
location = clients.location
city = clients.city
state = clients.state
pincode = clients.pincode
phone = clients.phone
fax = clients.fax
email = clients.email
person = clients.contact
{IdInsertValue} = ''
{DeleteOnTables} = clients
```

7.2.3 Code for class *Client*

Class *Client* is defined in module *clients.py*. The contents of it are given below.

```
import types, threading
import backend
from prangya.excgeneric import *

class Client(backend.BackEnd):
    """
    Class Client stores and processes information of client companies.
    Extends: BackEnd.
    """
    def __init__(self, id=None, attrs=None):
        """
        Constructor to initialize a Client object from backend database
        or create a new one and insert relevant data into the database.

        Parameters:
            id: If pertinent data is available in the database and
                integer or long specifying the object-id.
            attrs: A dictionary containing attribute name as key and
                attribute-value as its value. It is supplied only if
                a new object is to be created.
```

```
        Raises:
            ParamException: Invalid parameters supplied.
            InitException, NotExistsException, CursorException,
            CommitException, ConnectionException and
            LimitException: Raised from "BackEnd". See its
                documentation for details.
        """
        if id:
            if not (type(id) is types.IntType or type(id) \
                                        is types.LongType):
                raise ParamException('clients.Client.__init__(): \
                                       Invalid data type for id.')
            self.registerId(id)
            backend.BackEnd.__init__(self)
        elif attrs:
            if type(attrs) is not types.DictType:
                raise ParamException('clients.Client.__init__(): \
                                   Invalid data type for attrs.')
            self.insertSelf(attrs)
        else:
            raise ParamException('clients.Client.__init__(): \
                   Neither "id" nor "attrs" supplied. One of \
                   these parameters must be present in the \
                   constructor.')
        self.setLock('lock', threading.RLock()) # Instance lock.

    def __setattr__(self, attribute, value):
        """
        Auto updation of database is done by overloading method
        __setattr__().

        Parameters:
            attribute: A string containing attribute-name of the
                object.
            value: Value of the attribute "attribute".
        Raises:
            ParamException: Invalid parameter "attribute".
            InitException, ObjectException, ConnectionException,
            LimitException, NotExistsException, CursorException,
            CommitException, AttributeError: Raised from
                "BackEnd". See its documentation for details.
        """
        if not attribute:
            raise ParamException('clients.Client.__setattr__(): \
                Empty value supplied for parameter "attribute".')
        self.lock.acquire()
        try:
            self.updateSelf({attribute:value})
        except:
            self.lock.release()
```

```
            raise
        self.lock.release()
```

7.2.4 Running the sample code

The sample code can be run either in interactive mode or through a script. We chose the path of script. The contents of the script *test_client.py* is given below.

```
from backend import *
import prangya.objstore
import clients

# Create an instance of DBMap containing details of
# object's attributes.
dbm = DBMap('sample.ini')

# This object is used to initialize backend framework.
be = BackEnd(dbm)

# Initialize ObjectStore framework. You don't need to
# preserve object "ost".
ost = prangya.objstore.ObjectStore(be)

# Now create a client object through ObjectStore's
# interface create. Note that the first parameter
# of Client's constructor is None as we still don't
# have the id of object.
cl1 = ost.create(clients.Client, (None,
{'name':'Tata Iron & Steel Co. Ltd.',
'address':'Station Road',
'location':'Bistupur',
'city':'Jamshedpur',
'state':'Jharkhand, INDIA',
'phone':'0751 234 1234'}))

# Check your database tables using SQL console. You will
# notice table clients containing one record with above data.

# Print the newly acquired id.
print cl1.id

# Print attribute fax. Which is None, and hence empty.
print cl1.fax

# Now set the attribute fax.
cl1.fax = '0751 234 1245'

# Check again the record in table clients through SQL console.
# You will see the record for fax in it.
```

```
# Now let us retrieve a new Client object with same id.
myId = cl1.id
cl2 = ost.retrieve(clients.Client, (myId, None))

# Print the id of it. It will match with that of cl1.
print cl2.id

# Print attribute city. You will get 'Jamshedpur'.
print cl2.city

# Check object-ids of cl1 and cl2. Both will be SAME!
print 'id of cl1 is' + str(cl1)
print 'id of cl2 is' + str(cl2)
# It means you are dealing with same object. It is the case
# even if you create cl1 and cl2 in two different threads
# in an application.

# Now you don't have to carry around "ost". Just create an
# instance of ObjectStore when you need it. Notice empty
# constructor.
ostNew = prangya.objstore.ObjectStore()

# Retrieve third object from database with the same id(myId).
cl3 = ostNew.retrieve(clients.Client, (myId, None))

# Print object-id of it. It will be the same as that of cl1
# and cl2. So you are using the same (in-memory) object.
print 'id of cl3 is' + str(cl3)

# Now search and retrieve objects having 'Bistupur' as its
# attribute location.
result = ostNew.search(clients.Client, {'location':'Bistupur'})

# You will get a list with one element. Assign this element
# to cl4.
cl4 = result[0]

# Print its attribute id and object-id.
print cl4.id
print 'id of cl4 is' + str(cl4)
# Again it is the same object which you worked with earlier,
# i.e. cl1, cl2 and cl3.

# To remove this record from the database call destroy().
ostNew.destroy(cl3)

# Check the table clients through SQL console. The table is empty.
# Now its your responsibility to not to use cl1, cl2, cl3 or cl4
# further as such object is non-existent in backend database.
```

8 Conclusion

BackEnd framework is supposed to be useful for a wide range of python applications. It can be used as an enterprise level component to build medium to large scale python based applications. The present version is going through initial beta testing. Active support from the community can make it an invaluable tool in application developers' tool-chest.

****