FDB Documentation

Release 2.0

Pavel Cisar

Apr 27, 2018

Contents

FDB is a Python library package that implements Python Database API 2.0-compliant support for the open source relational database Firebird ®. In addition to the minimal feature set of the standard Python DB API, FDB also exposes the entire native (old-style) client API of the database engine and number of additional extensions and enhancements for convenient use of Firebird.

FDB is developed under the Firebird Project, and is used internally as key component for Firebird QA.

FDB is free – covered by a permissive BSD-style license that both commercial and noncommercial users should find agreeable.

This documentation set is not a tutorial on Python, SQL, or Firebird; rather, it is a topical presentation of FDB's feature set, with example code to demonstrate basic usage patterns. For detailed information about Firebird features, see the Firebird documentation, and especially the excellent The Firebird Book written by Helen Borrie and published by IBPhoenix.

CHAPTER 1

Content

1.1 Getting Started

1.1.1 Installation

FDB is written as pure-Python module (requires Python 2.7 or 3.4+) on top of Firebird client library (fbclient.so/dll) using ctypes, so **make sure you have Firebird client properly installed before you try to install FDB**, otherwise the installation will fail. FDB supports Firebird version 2.0 and higher.

FDB is distributed as setuptools package and the preferred installation method is via pip tool.

Installation from PYPI

Run pip:

```
$ pip install fdb
```

Installation from source

Download the source tarball, uncompress it, then run the install command:

```
$ tar -xzvf fdb-2.0.tar.gz
$ cd fdb-2.0
$ python setup.py install
```

1.1.2 Quick-start Guide

This brief tutorial aims to get the reader started by demonstrating elementary usage of FDB. It is not a comprehensive Python Database API tutorial, nor is it comprehensive in its coverage of anything else.

The numerous advanced features of FDB are covered in another section of this documentation, which is not in a tutorial format, though it is replete with examples.

Connecting to a Database

Example 1

A database connection is typically established with code such as this:

```
import fdb

# The server is named 'bison'; the database file is at '/temp/test.db'.
con = fdb.connect(dsn='bison:/temp/test.db', user='sysdba', password='pass')

# Or, equivalently:
con = fdb.connect(
    host='bison', database='/temp/test.db',
    user='sysdba', password='pass'
  )
```

Example 2

Suppose we want to connect to the database in SQL Dialect 1 and specifying UTF-8 as the character set of the connection:

```
import fdb

con = fdb.connect(
    dsn='bison:/temp/test.db',
    user='sysdba', password='pass',
    dialect=1, # necessary for all dialect 1 databases
    charset='UTF8' # specify a character set for the connection
  )
```

Executing SQL Statements

For this section, suppose we have a table defined and populated by the following SQL code:

```
create table languages
(
  name               varchar(20),
  year_released      integer
);

insert into languages (name, year_released) values ('C',        1972);
insert into languages (name, year_released) values ('Python',   1991);
```

Example 1

This example shows the *simplest* way to print the entire contents of the *languages* table:

```
import fdb

con = fdb.connect(dsn='/temp/test.db', user='sysdba', password='masterkey')

# Create a Cursor object that operates in the context of Connection con:
```

(continues on next page)

(continued from previous page)

```
cur = con.cursor()

# Execute the SELECT statement:
cur.execute("select * from languages order by year_released")

# Retrieve all rows as a sequence and print that sequence:
print cur.fetchall()
```

Sample output:

```
[('C', 1972), ('Python', 1991)]
```

Example 2

Here's another trivial example that demonstrates various ways of fetching a single row at a time from a *SELECT*-cursor:

```
import fdb

con = fdb.connect(dsn='/temp/test.db', user='sysdba', password='masterkey')

cur = con.cursor()
SELECT = "select name, year_released from languages order by year_released"

# 1. Iterate over the rows available from the cursor, unpacking the
# resulting sequences to yield their elements (name, year_released):
cur.execute(SELECT)
for (name, year_released) in cur:
    print '%s has been publicly available since %d.' % (name, year_released)

# 2. Equivalently:
cur.execute(SELECT)
for row in cur:
    print '%s has been publicly available since %d.' % (row[0], row[1])

# 3. Using mapping-iteration rather than sequence-iteration:
cur.execute(SELECT)
for row in cur.itermap():
    print '%(name)s has been publicly available since %(year_released)d.' % row
```

Sample output:

```
C has been publicly available since 1972.
Python has been publicly available since 1991.
C has been publicly available since 1972.
Python has been publicly available since 1991.
C has been publicly available since 1972.
Python has been publicly available since 1991.
```

Example 3

The following program is a simplistic table printer (applied in this example to *languages*):

```
import fdb

TABLE_NAME = 'languages'
SELECT = 'select * from %s order by year_released' % TABLE_NAME
```

(continues on next page)

(continued from previous page)

```
con = fdb.connect(dsn='/temp/test.db', user='sysdba', password='masterkey')

cur = con.cursor()
cur.execute(SELECT)

# Print a header.
for fieldDesc in cur.description:
    print fieldDesc[fdb.DESCRIPTION_NAME].ljust(fieldDesc[fdb.DESCRIPTION_DISPLAY_
↪SIZE]) ,
print # Finish the header with a newline.
print '-' * 78

# For each row, print the value of each field left-justified within
# the maximum possible width of that field.
fieldIndices = range(len(cur.description))
for row in cur:
    for fieldIndex in fieldIndices:
        fieldValue = str(row[fieldIndex])
        fieldMaxWidth = cur.description[fieldIndex][fdb.DESCRIPTION_DISPLAY_SIZE]

        print fieldValue.ljust(fieldMaxWidth) ,

    print # Finish the row with a newline.
```

Sample output:

```
NAME                 YEAR_RELEASED
------------------------------------------------------------------------------
C                    1972
Python               1991
```

Example 4

Let's insert more languages:

```
import fdb

con = fdb.connect(dsn='/temp/test.db', user='sysdba', password='masterkey')

cur = con.cursor()

newLanguages = [
    ('Lisp',  1958),
    ('Dylan', 1995),
  ]

cur.executemany("insert into languages (name, year_released) values (?, ?)",
    newLanguages
  )

# The changes will not be saved unless the transaction is committed explicitly:
con.commit()
```

Note the use of a *parameterized* SQL statement above. When dealing with repetitive statements, this is much faster and less error-prone than assembling each SQL statement manually. (You can read more about parameterized SQL statements in the section on *Prepared Statements*.)

After running Example 4, the table printer from Example 3 would print:

```
NAME                 YEAR_RELEASED
-------------------------------------------------------------------------------
Lisp                 1958
C                    1972
Python               1991
Dylan                1995
```

Calling Stored Procedures

Firebird supports stored procedures written in a proprietary procedural SQL language. Firebird stored procedures can have *input* parameters and/or *output* parameters. Some databases support *input/output* parameters, where the same parameter is used for both input and output; Firebird does not support this.

It is important to distinguish between procedures that *return a result set* and procedures that *populate and return their output parameters exactly once*. Conceptually, the latter "return their output parameters" like a Python function, whereas the former "yield result rows" like a Python generator.

Firebird's *server-side* procedural SQL syntax makes no such distinction, but *client-side* SQL code (and C API code) must. A result set is retrieved from a stored procedure by *SELECT'ing from the procedure, whereas output parameters are retrieved with an 'EXECUTE PROCEDURE* statement.

To *retrieve a result set* from a stored procedure with FDB, use code such as this:

```
cur.execute("select output1, output2 from the_proc(?, ?)", (input1, input2))

# Ordinary fetch code here, such as:
for row in cur:
    ... # process row

con.commit() # If the procedure had any side effects, commit them.
```

To *execute* a stored procedure and *access its output parameters*, use code such as this:

```
cur.callproc("the_proc", (input1, input2))

# If there are output parameters, retrieve them as though they were the
# first row of a result set.  For example:
outputParams = cur.fetchone()

con.commit() # If the procedure had any side effects, commit them.
```

This latter is not very elegant; it would be preferable to access the procedure's output parameters as the return value of *Cursor.callproc()*. The Python DB API specification requires the current behavior, however.

1.2 Usage Guide

1.2.1 Driver structure

Source code is currently divided into next submodules:

- `ibase` - Python ctypes interface to Firebird client library.
- `fbcore` - Main driver source code.

- `services` - Driver code to work with Firebird Services.
- `schema` - Driver code to work with Firebird database schema (metadata).
- `monitor` - Driver code to work with Firebird monitoring tables.
- `trace` - Code for Firebird Trace & Audit processing.
- `gstat` - Code for Firebird gstat output processing.
- `log` - Code for Firebird server log processing.
- `utils` - Various classes and functions used by driver that are generally useful.
- `blr` - Firebird BLR-related definitions.

All important data, functions, classes and constants are available directly in fdb namespace, so there is no need to import or use `fbcore` and `ibase` submodules directly. Other submodules (like `fdb.services` submodule that contains functions and classes for work with Firebird Services) contain optional driver functionality that is not exposed directly through main module namespace. Because `services` submodule contains names also used by main driver (`connect()`, `Connection`), it's advised to use fully qualified names when refering to them instead importing them via *from fdb.services import ...*.

1.2.2 Databases

Access to the database is made available through `Connection` objects. FDB provides two constructors for these:

- `connect()` - Returns *Connection* to database that already exists.
- `create_database()` - Returns *Connection* to newly created database.

Using *connect*

This constructor has number of keyword parameters that could be divided into several groups:

- Database specification (parameters *dsn*, *host*, *database* and *port*)
- User specification (parameters *user*, *password* and *role*)
- Connection options (parameters *sql_dialect*, *charset*, *isolation_level*, *buffers*, *force_writes*, *no_reserve* and *db_key_scope*)

To establish a connection to database, you always must specify the database, either as *connection string* parameter *dsn*, or as required combination of parameters *host*, *database* and *port*.

Important: Current driver version ignores the value of *port* parameter. If you need to specify the port number, you have to use *dsn* parameter instead.

Although specification of *user* and *password* parameters is optional (if environment variables *ISC_USER* and *ISC_PASSWORD* are set, their values are used if these parameters are ommited), it's recommended practice to use them. Parameter *role* is needed only when you use Firebird roles.

Connection options are optional (see Firebird Documentation for details). However you may often want to specify *charset*, as it directs automatic conversions of string data between client and server, and automatic conversions from/to unicode performed by FDB driver (see *Data handling and conversions* for details).

Examples:

```
# Connecting via 'dsn'
#
# Local database (local protocol, if supported)
con = fdb.connect(dsn='/path/database.fdb', user='sysdba', password='pass')
# Local database (TCP/IP)
con = fdb.connect(dsn='localhost:/path/database.fdb', user='sysdba', password='pass')
# Local database (TCP/IP with port specification)
con = fdb.connect(dsn='localhost/3050:/path/database.fdb', user='sysdba', password=
↪'pass')
# Remote database
con = fdb.connect(dsn='host:/path/database.db', user='sysdba', password='pass')
# Remote database with port specification
con = fdb.connect(dsn='host/3050:/path/database.db', user='sysdba', password='pass')
#
# Connecting via 'database', 'host' and 'port'
#
# Local database (local protocol, if supported)
con = fdb.connect(database='/path/database.db', user='sysdba', password='pass')
# Local database (TCP/IP)
con = fdb.connect(host='localhost', database='/path/database.db', user='sysdba',␣
↪password='pass')
# Local database (TCP/IP with port specification)
con = fdb.connect(host='localhost', port=3050, database='/path/database.db', user=
↪'sysdba', password='pass')
# Remote database
con = fdb.connect(host='myhost', database='/path/database.db', user='sysdba',␣
↪password='pass')
```

Since version 1.2 FDB supports additional *Connection* class(es) that extend `Connection` functionality in optional (opt-in) way. For example `ConnectionWithSchema` extends Connection interface with methods and attributes provided by `Schema`. New *connection_class* parameter was introduced to *connect* and *create_database* to connect to/create database using different class than descends from *Connection*.

Example:

```
# Connecting through ConnectionWithSchema
#
con = fdb.connect(dsn='/path/database.fdb', user='sysdba', password='pass',
                  connection_class=fdb.ConnectionWithSchema)
```

Using *create_database*

The Firebird engine supports dynamic database creation via the SQL statement *CREATE DATABASE*. FDB wraps it into `create_database()`, that returns *Connection* instance attached to newly created database.

Example:

```
con = fdb.create_database("create database 'host:/temp/db.db' user 'sysdba' password
↪'pass'")
```

Note: Since version 1.2 FDB supports additional method for database creation. Instead *CREATE DATABASE* SQL statement you can use number of optional keyword parameters introduced to `create_database()`.

Example:

```
con = fdb.create_database(dsn='/temp/db.fdb',user='sysdba',password='pass',page_
↪size=8192)
```

Deleting databases

The Firebird engine also supports dropping (deleting) databases dynamically, but dropping is a more complicated operation than creating, for several reasons: an existing database may be in use by users other than the one who requests the deletion, it may have supporting objects such as temporary sort files, and it may even have dependent shadow databases. Although the database engine recognizes a *DROP DATABASE* SQL statement, support for that statement is limited to the *isql* command-line administration utility. However, the engine supports the deletion of databases via an API call, which FDB exposes as *drop_database()* method in *Connection* class. So, to drop a database you need to connect to it first.

Examples:

```
import fdb

con = fdb.create_database("create database '/temp/db.db' user 'sysdba' password 'pass'
↪")
con.drop_database()

con = fdb.connect(dsn='/path/database.fdb', user='sysdba', password='pass')
con.drop_database()
```

Connection object

Connection object represents a direct link to database, and works as gateway for next operations with it:

- *Executing SQL Statements*: methods *execute_immediate()* and *cursor()*.
- Dropping database: method *drop_database()*.
- *Transanction management*: methods *begin()*, *commit()*, *rollback()*, *savepoint()*, *trans()*, *trans_info()* and *transaction_info()*, and attributes *main_transaction*, *transactions*, *default_tpb* and *group*.
- Work with *Database Events*: method *event_conduit()*.
- *Getting information about Firebird version*: attributes *server_version*, *firebird_version*, *version* and *engine_version*.
- *Getting information about database*: methods *db_info()* and *database_info()*.
- *Getting information about database metadata*: attribute *schema* and *ods*.

Getting information about Firebird version

Because functionality and some features depends on actual Firebird version, it could be important for FDB users to check it. This (otherwise) simple task could be confusing for new Firebird users, because Firebird uses two different version lineages. This abomination was introduced to Firebird thanks to its InterBase legacy (Firebird 1.0 is a fork of InterBase 6.0), as applications designed to work with InterBase can often work with Firebird without problems (and vice versa). However, legacy applications designed to work with InterBase may stop working properly if they would detect unexpectedly low server version, so default version number returned by Firebird (and FDB) is based on InterBase version number. For example this version for Firebird 2.5.2 is 6.3.2, so condition for legacy applications that require at least IB 6.0 is met.

FDB provides these version strings as two *Connection* properties:

- *server_version* - Legacy InterBase-friendly version string.
- *firebird_version* - Firebird's own version string.

However, this version string contains more information than version number. For example for Linux Firebird 2.5.2 it's 'LI-V2.5.2.26540 Firebird 2.5'. So FDB provides two more properties for your convenience:

- *version* - Only Firebird version number. It's a string with format: major.minor.subrelease.build
- *engine_version* - Engine (major.minor) version as (float) number.

FDB also provides convenient constants for supported engine versions: *ODS_FB_20*,'ODS_FB_21' and *ODS_FB_25*.

Database On-Disk Structure

Particular Firebird features may also depend on specific support in database (for example monitoring tables introduced in Firebird 2.1). These required structures are present automatically when database is created by particular engine verison that needs them, but Firebird engine may typically work with databases created by older versions and thus with older structure, so it could be necessary to consult also On-Disk Structure (ODS for short) version. FDB provides this number as *Connection.ods* (float) property.

Example:

```
con = fdb.connect(dsn='/path/database.fdb', user='sysdba', password='pass')
print 'Firebird version:',con.version
print 'ODS version:',con.ods
```

```
Firebird version: 2.5.2.26540
ODS version: 11.1
```

In abover example although connected Firebird engine is version 2.5, connected database has ODS 11.1 which came with Firebird 2.1, and some Firebird 2.5 features will not be available on this database.

Getting information about database

Firebird provides various informations about server and connected database via *database_info* API call. FDB surfaces this API through methods *db_info()* and *database_info()* on Connection object.

Connection.database_info() is a *very thin* wrapper around function *isc_database_info()*. This method does not attempt to interpret its results except with regard to whether they are a string or an integer. For example, requesting *isc_info_user_names* with the call:

```
con.database_info(fdb.isc_info_user_names, 's')
```

will return a binary string containing a raw succession of length-name pairs.

Example program:

```
import fdb

con = fdb.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass')

# Retrieving an integer info item is quite simple.
bytesInUse = con.database_info(fdb.isc_info_current_memory, 'i')

print 'The server is currently using %d bytes of memory.' % bytesInUse
```

(continues on next page)

(continued from previous page)

```
# Retrieving a string info item is somewhat more involved, because the
# information is returned in a raw binary buffer that must be parsed
# according to the rules defined in the Interbase® 6 API Guide section
# entitled "Requesting buffer items and result buffer values" (page 51).
#
# Often, the buffer contains a succession of length-string pairs
# (one byte telling the length of s, followed by s itself).
# Function fdb.ibase.ord2 is provided to convert a raw
# byte to a Python integer (see examples below).
buf = con.database_info(fdb.isc_info_db_id, 's')

# Parse the filename from the buffer.
beginningOfFilename = 2
# The second byte in the buffer contains the size of the database filename
# in bytes.
lengthOfFilename = fdb.ibase.ord2(buf[1])
filename = buf[beginningOfFilename:beginningOfFilename + lengthOfFilename]

# Parse the host name from the buffer.
beginningOfHostName = (beginningOfFilename + lengthOfFilename) + 1
# The first byte after the end of the database filename contains the size
# of the host name in bytes.
lengthOfHostName = fdb.ibase.ord2(buf[beginningOfHostName - 1])
host = buf[beginningOfHostName:beginningOfHostName + lengthOfHostName]

print 'We are connected to the database at %s on host %s.' % (filename, host)
```

Sample output:

```
The server is currently using 8931328 bytes of memory.
We are connected to the database at C:\TEMP\TEST.DB on host WEASEL.
```

A more convenient way to access the same functionality is via the *db_info()* method, which is high-level convenience wrapper around the *database_info()* method that parses the output of database_info into Python-friendly objects instead of returning raw binary buffers in the case of complex result types. For example, requesting *isc_info_user_names* with the call:

```
con.db_info(fdb.isc_info_user_names)
```

returns a dictionary that maps (username -> number of open connections). If SYSDBA has one open connection to the database to which *con* is connected, and TEST_USER_1 has three open connections to that same database, the return value would be:

```
{'SYSDBA': 1, 'TEST_USER_1': 3}
```

Example program:

```
import fdb
import os.path

##########################################################################
# Querying an isc_info_* item that has a complex result:
##########################################################################
# Establish three connections to the test database as TEST_USER_1, and one
# connection as SYSDBA.  Then use the Connection.db_info method to query the
```

(continues on next page)

(continued from previous page)

```
# number of attachments by each user to the test database.
testUserCons = []
for i in range(3):
  tcon = fdb.connect(dsn='localhost:/temp/test.db', user='TEST_USER_1', password='pass
↪')
  testUserCons.append(tcon)

con = fdb.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass')

print 'Open connections to this database:'
print con.db_info(fdb.isc_info_user_names)

##############################################################################
# Querying multiple isc_info_* items at once:
##############################################################################
# Request multiple db_info items at once, specifically the page size of the
# database and the number of pages currently allocated.  Compare the size
# computed by that method with the size reported by the file system.
# The advantages of using db_info instead of the file system to compute
# database size are:
#   - db_info works seamlessly on connections to remote databases that reside
#     in file systems to which the client program lacks access.
#   - If the database is split across multiple files, db_info includes all of
#     them.
res = con.db_info([fdb.isc_info_page_size, fdb.isc_info_allocation])
pagesAllocated = res[fdb.isc_info_allocation]
pageSize = res[fdb.isc_info_page_size]
print '\ndb_info indicates database size is', pageSize * pagesAllocated, 'bytes'
print 'os.path.getsize indicates size is ', os.path.getsize(DB_FILENAME), 'bytes'
```

Sample output:

```
Open connections to this database:
{'SYSDBA': 1, 'TEST_USER_1': 3}

db_info indicates database size is 20684800 bytes
os.path.getsize indicates size is  20684800 bytes
```

1.2.3 Executing SQL Statements

FDB implements two ways for execution of SQL commands against connected database:

- *execute_immediate()* - for execution of SQL commands that don't return any result.
- *Cursor* objects that offer rich interface for execution of SQL commands and fetching their results.

Cursor object

Because *Cursor* objects always operate in context of single *Connection* (and *Transaction*), *Cursor* instances are not created directly, but by constructor method. Python DB API 2.0 assume that if database engine supports transactions, it supports only one transaction per connection, hence it defines constructor method *cursor()* (and other transaction-related methods) as part of *Connection* interface. However, Firebird supports multiple independent transactions per connection. To conform to Python DB API, FDB uses concept of internal *main_transaction* and secondary *transactions*. Cursor constructor is primarily defined by *Transaction*, and Cursor constructor on *Connection* is therefore a shortcut for *main_transaction.cursor()*.

Cursor objects are used for next operations:

- Execution of SQL Statemets: methods `execute()`, `executemany()` and `callproc()`.
- Creating `PreparedStatement` objects for efficient repeated execution of SQL statements, and to obtain additional information about SQL statements (like execution `plan`): method `prep()`.
- Fetching results: methods `fetchone()`, `fetchmany()`, `fetchall()`, `fetchonemap()`, `fetchmanymap()`, `fetchallmap()`, `iter()`, `itermap()` and `next()`.

SQL Execution Basics

There are three methods how to execute SQL commands:

- `Connection.execute_immediate()` or `Transaction.execute_immediate()` for SQL commands that don't return any result, and are not executed frequently. This method also **doesn't** support either *parametrized statements* or *prepared statements*.

  Tip: This method is efficient for *administrative* and DDL SQL commands, like *DROP*, *CREATE* or *ALTER* commands, *SET STATISTICS* etc.

- `Cursor.execute()` or `Cursor.executemany()` for commands that return result sets, i.e. sequence of *rows* of the same structure, and sequence has unknown number of *rows* (including zero).

  Tip: This method is preferred for all *SELECT* and other DML statements, or any statement that is executed frequently, either *as is* or in *parametrized* form.

- `Cursor.callproc()` for execution of *Stored procedures* that always return exactly one set of values.

  Note: This method of SP invocation is equivalent to *"EXECUTE PROCEDURE . . . "* SQL statement.

Parametrized statements

When SQL command you want to execute contains data *values*, you can either:

- Embed them *directly* or via *string formatting* into command *string*, e.g.:

```
cur.execute("insert into the_table (a,b,c) values ('aardvark', 1, 0.1)")
# or
cur.execute("select * from the_table where col == 'aardvark'")
# or
cur.execute("insert into the_table (a,b,c) values ('%s', %i, %f)" % ('aardvark',1,
↪0.1))
# or
cur.execute("select * from the_table where col == '%s'" % 'aardvark')
```

- Use parameter marker (*?*) in command *string* in the slots where values are expected, then supply those values as Python list or tuple:

```
cur.execute("insert into the_table (a,b,c) values (?,?,?)", ('aardvark', 1, 0.1))
# or
cur.execute("select * from the_table where col == ?",('aardvark',))
```

While both methods have the same results, the second one (called *parametrized*) has several important advantages:

- You don't need to handle conversions from Python data types to strings.
- FDB will handle all data type conversions (if necessary) from Python data types to Firebird ones, including *None/NULL* conversion and conversion from *unicode* to *byte strings* in encoding expected by server.
- You may pass BLOB values as open *file-like* objects, and FDB will handle the transfer of BLOB value.

Parametrized statemets also have some limitations. Currently:

- *DATE*, *TIME* and *DATETIME* values must be relevant `datetime` objects.
- *NUMERIC* and *DECIMAL* values must be `decimal` objects.

Fetching data from server

Result of SQL statement execution consists from sequence of zero to unknown number of *rows*, where each *row* is a set of exactly the same number of values. `Cursor` object offer number of different methods for fetching these *rows*, that should satisfy all your specific needs:

- `fetchone()` - Returns the next row of a query result set, or *None* when no more data is available.

  Tip: Cursor supports the iterator protocol, yielding tuples of values like `fetchone()`.

- `fetchmany()` - Returns the next set of rows of a query result, returning a sequence of sequences (e.g. a list of tuples). An empty sequence is returned when no more rows are available.

  The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's `arraysize` determines the number of rows to be fetched. The method does try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

  Note: The default value of `arraysize` is *1*, so without paremeter it's equivalent to `fetchone()`, but returns list of *rows*, instead actual *row* directly.

- `fetchall()` - Returns all (remaining) rows of a query result as list of tuples, where each tuple is one row of returned values.

  Tip: This method can potentially return huge amount of data, that may exhaust available memory. If you need just *iteration* over potentially big result set, use loops with `fetchone()`, Cursor's built-in support for iterator protocol or call to `iter()` instead this method.

- `fetchonemap()` - Returns the next row like `fetchone()`, but returns a mapping of *field name* to *field value*, rather than a tuple.
- `fetchmanymap()` - Returns the next set of rows of a query result like `fetchmany()`, but returns a list of mapping of *field name* to *field value*, rather than a tuple.
- `fetchallmap()` - Returns all (remaining) rows of a query result like `fetchall()`, returns a list of mappings of *field name* to *field value*, rather than a tuple.

  Tip: This method can potentially return huge amount of data, that may exhaust available memory. If you need just *iteration* over potentially big result set with mapping support, use `itermap()` instead this method.

- *iter()* - Equivalent to the *fetchall()*, except that it returns iterator rather than materialized list.
- *itermap()* - Equivalent to the *fetchallmap()*, except that it returns iterator rather than materialized list.
- Call to *execute()* returns self (Cursor instance) that itself supports the iterator protocol, yielding tuples of values like *fetchone()*.

Important: FDB makes absolutely no guarantees about the return value of the *fetchone* / *fetchmany* / *fetchall* methods except that it is a sequence indexed by field position. FDB makes absolutely no guarantees about the return value of the *fetchonemap* / *fetchmanymap* / *fetchallmap* methods except that it is a mapping of field name to field value. Therefore, client programmers should not rely on the return value being an instance of a particular class or type.

Examples:

```
import fdb

con = fdb.connect(dsn='/temp/test.db', user='sysdba', password='masterkey')

cur = con.cursor()
SELECT = "select name, year_released from languages order by year_released"

# 1. Using built-in support for iteration protocol to iterate over the rows available␣
↪from the cursor,
# unpacking the resulting sequences to yield their elements (name, year_released):
cur.execute(SELECT)
for (name, year_released) in cur:
    print '%s has been publicly available since %d.' % (name, year_released)
# or alternatively you can take an advantage of cur.execute returning self.
for (name, year_released) in cur.execute(SELECT):
    print '%s has been publicly available since %d.' % (name, year_released)

# 2. Equivalently using fetchall():
# This is potentially dangerous if result set is huge, as the whole result set is␣
↪first materialized
# as list and then used for iteration.
cur.execute(SELECT)
for row in cur.fetchall():
    print '%s has been publicly available since %d.' % (row[0], row[1])

# 3. Using mapping-iteration rather than sequence-iteration:
cur.execute(SELECT)
for row in cur.itermap():
    print '%(name)s has been publicly available since %(year_released)d.' % row
```

Tip: *Cursor.execute()* and *Cursor.executemany()* return self, so you can use calls to them as iterators (see example above).

Prepared Statements

Execution of any SQL statement has three phases:

- *Preparation*: command is analyzed, validated, execution plan is determined by optimizer and all necessary data structures (for example for input and output parameters) are initialized.

- *Execution*: input parameters (if any) are passed to server and previously prepared statement is actually executed by database engine.
- *Fetching*: result of execution and data (if any) are transfered from server to client, and allocated resources are then released.

The preparation phase consumes some amount of server resources (memory and CPU). Although preparation and release of resources typically takes only small amount of CPU time, it builds up as number of executed statements grows. Firebird (like most database engines) allows to spare this time for subsequent execution if particular statement should be executed repeatedly - by reusing once prepared statement for repeated execution. This may save significant amount of server processing time, and result in better overall performance.

FDB builds on this by encapsulating all statement-related code into separate `PreparedStatement` class, and implementing `Cursor` class as a wrapper around it.

> **Warning:** FDB's implementation of Cursor somewhat violates the Python DB API 2.0, which requires that cursor will be unusable after call to *close*; and an Error (or subclass) exception should be raised if any operation is attempted with the cursor.
>
> If you'll take advantage of this *anomaly*, your code would be less portable to other Python DB API 2.0 compliant drivers.

Beside SQL command string, *Cursor* also allows to aquire and use *PreparedStatement* instances explicitly. *PreparedStatement* are aquired by calling `prep()` method could be then passed to `execute()` or `executemany()` instead *command string*.

Example:

```
insertStatement = cur.prep("insert into the_table (a,b,c) values (?,?,?)")

inputRows = [
    ('aardvark', 1, 0.1),
    ('zymurgy', 2147483647, 99999.999),
    ('foobar', 2000, 9.9)
  ]

for row in inputRows:
   cur.execute(insertStatement,row)
#
# or you can use executemany
#
cur.executemany(insertStatement, inputRows)
```

Prepared statements are bound to *Cursor* instance that created them, and can't be used with any other *Cursor* instance. Beside repeated execution they are also useful to get information about statement (like its output `description`, execution `plan` or `statement_type`) before its execution.

Example Program:

The following program demonstrates the explicit use of *PreparedStatements*. It also benchmarks explicit *PreparedStatement* reuse against normal execution that prepares statements on each execution.

```
import time
import fdb

con = fdb.connect(dsn='localhost:employee',
    user='sysdba', password='masterkey'
```

(continues on next page)

(continued from previous page)

```
  )

cur = con.cursor()

# Create supporting database entities:
cur.execute("recreate table t (a int, b varchar(50))")
con.commit()
cur.execute("create unique index unique_t_a on t(a)")
con.commit()

# Explicitly prepare the insert statement:
psIns = cur.prep("insert into t (a,b) values (?,?)")
print 'psIns.sql: "%s"' % psIns.sql
print 'psIns.statement_type == fdb.isc_info_sql_stmt_insert:', (
    psIns.statement_type == fdb.isc_info_sql_stmt_insert
  )
print 'psIns.n_input_params: %d' % psIns.n_input_params
print 'psIns.n_output_params: %d' % psIns.n_output_params
print 'psIns.plan: %s' % psIns.plan

print

N = 50000
iStart = 0

# The client programmer uses a PreparedStatement explicitly:
startTime = time.time()
for i in xrange(iStart, iStart + N):
    cur.execute(psIns, (i, str(i)))
print (
    'With explicit prepared statement, performed'
    '\n  %0.2f insertions per second.' % (N / (time.time() - startTime))
  )
con.commit()

iStart += N

# A new SQL string containing the inputs is submitted every time.  Also, in a
# more complicated scenario where the end user supplied the string input
# values, the program would risk SQL injection attacks:
startTime = time.time()
for i in xrange(iStart, iStart + N):
    cur.execute("insert into t (a,b) values (%d,'%s')" % (i, str(i)))
print (
    'When unable to reuse prepared statement, performed'
    '\n  %0.2f insertions per second.' % (N / (time.time() - startTime))
  )
con.commit()

# Prepare a SELECT statement and examine its properties.  The optimizer's plan
# should use the unique index that we created at the beginning of this program.
print
psSel = cur.prep("select * from t where a = ?")
print 'psSel.sql: "%s"' % psSel.sql
print 'psSel.statement_type == fdb.isc_info_sql_stmt_select:', (
    psSel.statement_type == fdb.isc_info_sql_stmt_select
  )
```

(continues on next page)

(continued from previous page)

```
print 'psSel.n_input_params: %d' % psSel.n_input_params
print 'psSel.n_output_params: %d' % psSel.n_output_params
print 'psSel.plan: %s' % psSel.plan

# The current implementation does not allow PreparedStatements to be prepared
# on one Cursor and executed on another:
print
print 'Note that PreparedStatements are not transferrable from one cursor to another:'
cur2 = con.cursor()
cur2.execute(psSel)
```

Sample output:

```
psIns.sql: "insert into t (a,b) values (?,?)"
psIns.statement_type == fdb.isc_info_sql_stmt_insert: True
psIns.n_input_params: 2
psIns.n_output_params: 0
psIns.plan: None

With explicit prepared statement, performed
  4276.00 insertions per second.
When unable to reuse prepared statement, performed
  2037.70 insertions per second.

psSel.sql: "select * from t where a = ?"
psSel.statement_type == fdb.isc_info_sql_stmt_select: True
psSel.n_input_params: 1
psSel.n_output_params: 2
psSel.plan: PLAN (T INDEX (UNIQUE_T_A))

Note that PreparedStatements are not transferrable from one cursor to another:
Traceback (most recent call last):
  File "pstest.py", line 85, in <module>
    cur2.execute(psSel)
     File "/home/job/python/envs/pyfirebird/fdb/fdb/fbcore.py", line 2623, in execute
    raise ValueError("PreparedStatement was created by different Cursor.")
ValueError: PreparedStatement was created by different Cursor.
```

As you can see, the version that prevents the reuse of prepared statements is about two times slower – *for a trivial statement*. In a real application, SQL statements are likely to be far more complicated, so the speed advantage of using prepared statements would only increase.

Named Cursors

To allow the Python programmer to perform scrolling *UPDATE* or *DELETE* via the *"SELECT ... FOR UPDATE"* syntax, FDB provides the read/write property `Cursor.name`.

Example Program:

```
import fdb

con = fdb.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass')
curScroll = con.cursor()
curUpdate = con.cursor()
```

(continues on next page)

(continued from previous page)

```
curScroll.execute("select city from addresses for update")
curScroll.name = 'city_scroller'
update = "update addresses set city=? where current of " + curScroll.name

for (city,) in curScroll:
    city = ... # make some changes to city
    curUpdate.execute( update, (city,) )

con.commit()
```

Working with stored procedures

Firebird stored procedures can have *input* parameters and/or *output* parameters. Some databases support *input/output* parameters, where the same parameter is used for both input and output; Firebird does not support this.

It is important to distinguish between procedures that *return a result set* and procedures that *populate and return their output parameters* exactly once. Conceptually, the latter "return their output parameters" like a Python function, whereas the former "yield result rows" like a Python generator.

Firebird's *server-side* procedural SQL syntax makes no such distinction, but client-side SQL code (and C API code) must. A result set is retrieved from a stored procedure by *SELECT'ing from the procedure, whereas output parameters are retrieved with an 'EXECUTE PROCEDURE' statement*.

To **retrieve a result set** from a stored procedure with FDB, use code such as this:

```
cur.execute("select output1, output2 from the_proc(?, ?)", (input1, input2))

# Ordinary fetch code here, such as:
for row in cur:
   ... # process row

con.commit() # If the procedure had any side effects, commit them.
```

To **execute** a stored procedure and **access its output parameters**, use code such as this:

```
cur.callproc("the_proc", (input1, input2))

# If there are output parameters, retrieve them as though they were the
# first row of a result set.  For example:
outputParams = cur.fetchone()

con.commit() # If the procedure had any side effects, commit them.
```

This latter is not very elegant; it would be preferable to access the procedure's output parameters as the return value of *Cursor.callproc()*. The Python DB API specification requires the current behavior, however.

1.2.4 Data handling and conversions

Implicit Conversion of Input Parameters from Strings

The database engine treats most SQL data types in a weakly typed fashion: the engine may attempt to convert the raw value to a different type, as appropriate for the current context. For instance, the SQL expressions *123* (integer) and *'123'* (string) are treated equivalently when the value is to be inserted into an *integer* field; the same applies when *'123'* and *123* are to be inserted into a *varchar* field.

This weak typing model is quite unlike Python's dynamic yet strong typing. Although weak typing is regarded with suspicion by most experienced Python programmers, the database engine is in certain situations so aggressive about its typing model that FDB must compromise in order to remain an elegant means of programming the database engine.

An example is the handling of "magic values" for date and time fields. The database engine interprets certain string values such as *'yesterday'* and *'now'* as having special meaning in a date/time context. If FDB did not accept strings as the values of parameters destined for storage in date/time fields, the resulting code would be awkward. Consider the difference between the two Python snippets below, which insert a row containing an integer and a timestamp into a table defined with the following DDL statement:

```
create table test_table (i integer, t timestamp)
```

```
i = 1
t = 'now'
sqlWithMagicValues = "insert into test_table (i, t) values (?, '%s')" % t
cur.execute( sqlWithMagicValues, (i,) )
```

```
i = 1
t = 'now'
cur.execute( "insert into test_table (i, t) values (?, ?)", (i, t) )
```

If FDB did not support weak parameter typing, string parameters that the database engine is to interpret as "magic values" would have to be rolled into the SQL statement in a separate operation from the binding of the rest of the parameters, as in the first Python snippet above. Implicit conversion of parameter values from strings allows the consistency evident in the second snippet, which is both more readable and more general.

It should be noted that FDB does not perform the conversion from string itself. Instead, it passes that responsibility to the database engine by changing the parameter metadata structure dynamically at the last moment, then restoring the original state of the metadata structure after the database engine has performed the conversion.

A secondary benefit is that when one uses FDB to import large amounts of data from flat files into the database, the incoming values need not necessarily be converted to their proper Python types before being passed to the database engine. Eliminating this intermediate step may accelerate the import process considerably, although other factors such as the chosen connection protocol and the deactivation of indexes during the import are more consequential. For bulk import tasks, the database engine's external tables also deserve consideration. External tables can be used to suck semi-structured data from flat files directly into the relational database without the intervention of an ad hoc conversion program.

Automatic conversion from/to unicode

In Firebird, every *CHAR*, *VARCHAR* or textual *BLOB* field can (or, better: must) have a *character set* assigned. While it's possible to define single character set for whole database, it's also possible to define different character set for each textual field. This information is used to correctly store the bytes that make up the character string, and together with collation information (that defines the sort ordering and uppercase conversions for a string) is vital for correct data manupulation, including automatic transliteration between character sets when necessary.

Important: Because data also flow between server and client application, it's vital that client will send data encoded only in character set(s) that server expects. While it's possible to leave this responsibility completely on client application, it's better when client and server settle on single character set they would use for communication, especially when database operates with multiple character sets, or uses character set that is not *native* for client application.

Character set for communication is specified using *charset* parameter in `connection` call.

When *connection charset* is defined, all textual data returned from server are encoded in this charset, and client application must ensure that all textual data sent to server are encoded only in this charset as well.

FDB helps with client side of this character set bargain by automatically converting *unicode* strings into *bytes/strings* encoded in connection character set, and vice versa. However, developers are still responsible that *non-unicode* strings passed to server are in correct encoding (because FDB makes no assuption about encoding of non-unicode strings, so it can't recode them to connection charset).

Important: In case that *connection charset* is NOT defined at all, or *NONE* charset is specified, FDB uses `locale.getpreferredencoding()` to determine encoding for conversions from/to *unicode*.

Important: There is one exception to automatic conversion: when character set OCTETS is defined for data column. Values assigned to OCTETS columns are always passed *as is*, because they're basically binary streams. This has specific implications regarding Python version you use. Python 2.x *native strings* are *bytes*, suitable for such binary streams, but Python 3 native strings are *unicode*, and you would probably want to use *bytes* type instead. However, FDB in this case doesn't check the value type at all, so you'll not be warned if you'll make a mistake and pass *unicode* to OCTETS column (unless you'll pass more bytes than column may hold, or you intend to store unicode that way).

Rules for automatic conversion depend on Python version you use:

- Native Python 2.x *strings* are passed to server as is, and developers must explicitly use *unicode* strings to take advantage of automatic conversion. String values coming from server are converted to *unicode* **only**:
  - for data stored in database (i.e. not for string values returned by Firebird Service and *info* calls etc.).
  - when *connection charset* is specified.
- Native Python 3 strings are *unicode*, so conversion is fully automatic in both directions for all textual data, i.e. including for string values returned by Firebird Service and *info* calls etc. When *connection charset* is not specified, FDB uses `locale.getpreferredencoding()` to determine encoding for conversions from/to *unicode*.

Tip: Except for legacy databases that doesn't have *character set* defined, **always** define character set for your databases and specify *connection charset*. It will make your life much easier.

Working with BLOBs

FDB uses two types of BLOB values:

- **Materialized** BLOB values are Python strings. This is the **default** type.
- **Streamed** BLOB values are *file-like* objects.

Materialized BLOBs are easy to work with, but are not suitable for:

- **deferred loading** of BLOBs. They're called *materialized* because they're always fetched from server as part of row fetch. Fetching BLOB value means separate API calls (and network roundtrips), which may slow down you application considerably.
- **large values**, as they are always stored in memory in full size.

These drawbacks are addressed by *stream* BLOBs. Using BLOBs in *stream* mode is easy:

- For **input** values, simply use *parametrized statement* and pass any *file-like* object in place of BLOB parameter. The *file-like* object must implement only the `read()` method, as no other metod is used.
- For **output** values, you have to call *`Cursor.set_stream_blob()`* (or *`PreparedStatement.set_stream_blob()`*) method with specification of column name(s) that should be returned as *file-like*

objects. FDB then returns *BlobReader* instance instead string in place of returned BLOB value for these column(s).

Important: **Before FDB version 1.8** load of materialized blob with multiple segments (i.e. larger than 64K) failed with error (SQLCODE: 101 - segment buffer length shorter than expected). This was an artefact of backward compatibility with KInterbasDB that prevented them to exhaust your memory with very large materialized blobs.

Since FDB version 1.8 this memory exhaustion safeguard was enhanced in more convenient (but backward incompatible) way. New methods *PreparedStatement.set_stream_blob_treshold()* and *Cursor.set_stream_blob_treshold()* were introduced to control the maximum size of materialized blobs. When particular blob value exceeds this threshold, an instance of *BlobReader* is returned instead string value, so your application has to be prepared to handle BLOBs in both incarnations.

Zero value effectively forces all blobs to be returned as stream blobs. Negative value means no size limit for materialized blobs (use at your own risk). **The default treshold value is 64K.**

Blob size treshold has effect only on materialized blob columns, i.e. columns not explicitly requested to be returned as streamed ones using *PreparedStatement.set_stream_blob()* that are always returned as stream blobs.

The *BlobReader* instance is bound to particular BLOB value returned by server, so its life time is limited. The actual BLOB value is not opened initially, so no additonal API calls to server are made if you'll decide to ignore the value completely. You also don't need to open the BLOB value explicitly, as BLOB is opened automatically on first call to *next()*, *read()*, *readline()*, *readlines()* or *seek()*. However, it's good practice to *close()* the reader once you're finished reading, as it's likely that Python's garbage collector would call the *__del__* method too late, when fetch context is already gone, and closing the reader would cause an error.

Warning: If BLOB was NOT CREATED as *stream* BLOB, calling *BlobReader.seek()* method will raise *DatabaseError* exception. **This constraint is set by Firebird.**

Important: When working with BLOB values, always have memory efficiency in mind, especially when you're processing huge quantity of rows with BLOB values at once. Materialized BLOB values may exhaust your memory quickly, but using stream BLOBs may have inpact on performance too, as new *BlobReader* instance is created for each value fetched.

Example program:

```
import os.path
from cStringIO import StringIO

import fdb

con = fdb.connect(dsn='localhost:employee',user='sysdba', password='masterkey')

cur = con.cursor()

cur.execute("recreate table blob_test (a blob)")
con.commit()

# --- Materialized mode (str objects for both input and output) ---
# Insertion:
cur.execute("insert into blob_test values (?)", ('abcdef',))
cur.execute("insert into blob_test values (?)", ('ghijklmnop',))
```

(continues on next page)

(continued from previous page)

```
# Retrieval:
cur.execute("select * from blob_test")
print 'Materialized retrieval (as str):'
print cur.fetchall()

cur.execute("delete from blob_test")

# --- Streaming mode (file-like objects for input; fdb.BlobReader objects for output)␣
↪---

# Insertion:
cur.execute("insert into blob_test values (?)", (StringIO('abcdef'),))
cur.execute("insert into blob_test values (?)", (StringIO('ghijklmnop'),))

f = file(os.path.abspath(__file__), 'rb')
cur.execute("insert into blob_test values (?)", (f,))
f.close()

# Retrieval using the "file-like" methods of BlobReader:
cur.execute("select * from blob_test")
cur.set_stream_blob('A') # Note the capital letter

readerA = cur.fetchone()[0]

print '\nStreaming retrieval (via fdb.BlobReader):'

# Python "file-like" interface:
print 'readerA.mode:    "%s"' % readerA.mode
print 'readerA.closed:   %s'  % readerA.closed
print 'readerA.tell():   %d'  % readerA.tell()
print 'readerA.read(2): "%s"' % readerA.read(2)
print 'readerA.tell():   %d'  % readerA.tell()
print 'readerA.read():  "%s"' % readerA.read()
print 'readerA.tell():   %d'  % readerA.tell()
print 'readerA.read():  "%s"' % readerA.read()
readerA.close()
print 'readerA.closed:   %s'  % readerA.closed
```

Output:

```
Materialized retrieval (as str):
[('abcdef',), ('ghijklmnop',)]

Streaming retrieval (via fdb.BlobReader):
readerA.mode:    "rb"
readerA.closed:   False
readerA.tell():   0
readerA.read(2): "ab"
readerA.tell():   2
readerA.read():  "cdef"
readerA.tell():   6
readerA.read():  ""
readerA.closed:   True
```

Firebird ARRAY type

FDB supports Firebird ARRAY data type. ARRAY values are represented as Python lists. On input, the Python sequence (list or tuple) must be nested appropriately if the array field is multi-dimensional, and the incoming sequence must not fall short of its maximum possible length (it will not be "padded" implicitly–see below). On output, the lists will be nested if the database array has multiple dimensions.

Note: Database arrays have no place in a purely relational data model, which requires that data values be atomized (that is, every value stored in the database must be reduced to elementary, non-decomposable parts). The Firebird implementation of database arrays, like that of most relational database engines that support this data type, is fraught with limitations.

Database arrays are of fixed size, with a predeclared number of dimensions (max. 16) and number of elements per dimension. Individual array elements cannot be set to NULL / None, so the mapping between Python lists (which have dynamic length and are therefore not normally "padded" with dummy values) and non-trivial database arrays is clumsy.

Stored procedures cannot have array parameters.

Finally, many interface libraries, GUIs, and even the isql command line utility do not support database arrays.

In general, it is preferable to avoid using database arrays unless you have a compelling reason.

Example:

```
>>> import fdb
>>> con = fdb.connect(dsn='localhost:employee',user='sysdba', password='masterkey')
>>> cur = con.cursor()
>>> cur.execute("select LANGUAGE_REQ from job where job_code='Eng' and job_grade=3␣
↪and job_country='Japan'")
>>> cur.fetchone()
(['Japanese\n', 'Mandarin\n', 'English\n', '\n', '\n'],)
```

Example program:

```
import fdb

con = fdb.connect(dsn='localhost:/temp/test.db', user='sysdba', password='pass')
con.execute_immediate("recreate table array_table (a int[3,4])")
con.commit()

cur = con.cursor()

arrayIn = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9,10,11,12]
  ]

print 'arrayIn:  %s' % arrayIn
cur.execute("insert into array_table values (?)", (arrayIn,))

cur.execute("select a from array_table")
arrayOut = cur.fetchone()[0]
print 'arrayOut: %s' % arrayOut

con.commit()
```

Output:

```
arrayIn:  [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
arrayOut: [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

1.2.5 Transanction management

For the sake of simplicity, FDB lets the Python programmer ignore transaction management to the greatest extent allowed by the Python Database API Specification 2.0. The specification says, "if the database supports an auto-commit feature, this must be initially off". At a minimum, therefore, it is necessary to call the commit method of the connection in order to persist any changes made to the database.

Remember that because of ACID, every data manipulation operation in the Firebird database engine takes place in the context of a transaction, including operations that are conceptually "read-only", such as a typical SELECT. The client programmer of FDB establishes a transaction implicitly by using any SQL execution method, such as *execute_immediate()*, *Cursor.execute()*, or *Cursor.callproc()*.

Although FDB allows the programmer to pay little attention to transactions, it also exposes the full complement of the database engine's advanced transaction control features: *transaction parameters*, *retaining transactions*, *savepoints*, and *distributed transactions*.

Basics

When it comes to transactions, Python Database API 2.0 specify that *Connection* object has to respond to the following methods:

Connection.commit()

> Commit any pending transaction to the database. Note that if the database supports an auto-commit feature, this must be initially off. An interface method may be provided to turn it back on. Database modules that do not support transactions should implement this method with void functionality.

Connection.rollback()

> (optional) In case a database does provide transactions this method causes the the database to roll back to the start of any pending transaction. **Closing a connection without committing the changes first will cause an implicit rollback to be performed.**

In addition to the implicit transaction initiation required by Python Database API, FDB allows the programmer to start transactions explicitly via the *Connection.begin()* method. Also *Connection.savepoint()* method was added to provide support for Firebird SAVEPOINTs.

But Python Database API 2.0 was created with assumption that connection can support only one transactions per single connection. However, Firebird can support multiple independent transactions that can run simultaneously within single connection / attachment to the database. This feature is very important, as applications may require multiple transaction openned simultaneously to perform various tasks, which would require to open multiple connections and thus consume more resources than necessary.

FDB surfaces this Firebird feature by separating transaction management out from *Connection* into separate *Transaction* objects. To comply with Python DB API 2.0 requirements, *Connection* object uses one *Transaction* instance as *main transaction*, and delegates *begin()*, *savepoint()*, *commit()*, *rollback()*, *trans_info()* and *transaction_info()* calls to it.

See also:

More about using multiple transactions with the same connection in separate *section*.

Example:

```
import fdb

con = fdb.connect(dsn='localhost:employee',user='sysdba', password='masterkey')

cur = con.cursor()

# Most minimalistic transaction management -> implicit start, only commit() and␣
↪rollback()
#␣
↪==========================================================================================
#
# Transaction is started implicitly
cur.execute('insert into country values ('Oz','Crowns')
con.commit() # commits active transaction
# Again, transaction is started implicitly
cur.execute('insert into country values ('Barsoom','XXX')
con.rollback() # rolls back active transaction
cur.execute('insert into country values ('Pellucidar','Shells')

# This will roll back the transaction
# because Python DB API 2.0 requires that closing connection
# with pending transaction must cause an implicit rollback
con.close()
```

Auto-commit

FDB doesn't support *auto-commit* feature directly, but developers may achieve the similar result using *explicit* transaction start, taking advantage of `default_action` and its default value (*commit*).

Example:

```
import fdb

con = fdb.connect(dsn='localhost:employee',user='sysdba', password='masterkey')

cur = con.cursor()

con.begin()
cur.execute('insert into country values ('Oz','Crowns')
con.begin() # commits active transaction and starts new one
cur.execute('insert into country values ('Barsoom','XXX')
con.begin() # commits active transaction and starts new one
cur.execute('insert into country values ('Pellucidar','Shells')

# However, commit is required before connection is closed,
# because Python DB API 2.0 requires that closing connection
# with pending transaction must cause an implicit rollback
con.commit()
con.close()
```

Transaction parameters

The database engine offers the client programmer an optional facility called *transaction parameter buffers* (TPBs) for tweaking the operating characteristics of the transactions he initiates. These include characteristics such as whether

the transaction has read and write access to tables, or read-only access, and whether or not other simultaneously active transactions can share table access with the transaction.

Connections have a *default_tpb* attribute that can be changed to set the default TPB for all transactions subsequently started on the connection. Alternatively, if the programmer only wants to set the TPB for a single transaction, he can start a transaction explicitly via the *begin()* method and pass a TPB for that single transaction.

For details about TPB construction, see the Firebird API documentation. In particular, the *ibase.h* supplied with Firebird contains all possible TPB elements – single bytes that the C API defines as constants whose names begin with *isc_tpb_*. FDB makes all of those TPB constants available (under the same names) as module-level constants. A transaction parameter buffer is handled in C as a character array; FDB requires that TPBs be constructed as Python *strings* (or *bytes* for Python 3). Since the constants in the *fdb.isc_tpb_** family are numbers, they can't be simply concatenated to create a TPB, but you may use utility function **fdb.bs(byte_array)** that accepts sequence of numbers and returns *string* (P2) or *bytes* (P3).

For example next call returns TPB for typical *READ COMMITED* transaction:

```
from fdb import *

TPB = bs([isc_tpb_version3,
          isc_tpb_write,
          isc_tpb_wait,
          isc_tpb_read_committed,
          isc_tpb_rec_version])
```

Warning: This method requires good knowledge of *tpc_block* structure and proper order of various parameters, as Firebird engine will raise an error when badly structured block would be used. Also definition of *table reservation* parameters is uncomfortable as you'll need to mix binary codes with table names passed as Pascal strings (characters preceded by string length).

FDB provides several predefined TPB's for convenience:

- **ISOLATION_LEVEL_READ_COMMITED** Read/Write READ COMMITED with *record version* and WAIT option. Isolation level with greatest concurrent throughput. This is **Default** TPB.

  Tip: This isolation level is optimal for transactions that write data and doesn't require stable snapshot of database for their operations (i.e. most operations are limited to individual rows).

- **ISOLATION_LEVEL_READ_COMMITED_LEGACY** Read/Write READ COMMITED with *NO record version* and WAIT option.

  Warning: This isolation level emulates RDBMS that use locks instead multiversion control (MVC). It's not recommended to use it at all, except for legacy applications lazily ported from such RDBMS to Firebird.

- **ISOLATION_LEVEL_READ_COMMITED_RO** Like *ISOLATION_LEVEL_READ_COMMITED*, but **Read Only**.

  Tip: Firebird threats these transactions as *pre-committed*, so they are best option **for long running transactions that only read data**.

Internaly FDB uses such transaction to read metadata from connected database. This internal transaction is also available to developers for convenience as *Connection.query_transaction*.

- **ISOLATION_LEVEL_REPEATABLE_READ or ISOLATION_LEVEL_SNAPSHOT** Read/Write SNAPSHOT (concurrency) with WAIT option.

  Tip: This isolation level is necessary for transactions that process data in bulk, like reporting, recalculations etc.

- **ISOLATION_LEVEL_SERIALIZABLE or ISOLATION_LEVEL_SNAPSHOT_TABLE_STABILITY** Read/Write SNAPSHOT TABLE STABILITY (consistency) with WAIT option. Like REPEATABLE_READ/SNAPSHOT, but locks whole tables for writes from other transactions. Isolation level with lowest concurrent throughput.

  Warning: Because tables are locked for *protected write* (i.e. no other transaction can write until lock is released) **at time of first access**, there is a great risk of *deadlock* between transactions.

  Tip: To prevent deadlocks and increase concurrent throughput it's recommended to use custom TPB's with *fine-grained table access reservation*.

Example:

```
import fdb

con = fdb.connect(dsn='localhost:employee',user='sysdba', password='masterkey')

cur = con.cursor()

# Start transaction with default_tpb (ISOLATION_LEVEL_READ_COMMITED)
con.begin()
cur.execute('select * from JOB')
com.commit()

# Start using transactions in REPEATABLE READ (SNAPSHOT) isolation
con.default_tpb = fdb.ISOLATION_LEVEL_REPEATABLE_READ
con.begin()
cur.execute('select * from JOB')
com.commit()

# Start THIS transaction as R/O READ COMMITTED
con.begin(fdb.ISOLATION_LEVEL_READ_COMMITED_RO)
cur.execute('select * from JOB')
com.commit()
```

For cases when predefined transaction parameter blocks are not suitable for your needs, FDB offers utility class *TPB* for convenient and safe construction of custom *tpb blocks*. Simply create instance of this class, set member attributes to required values and use either *rendered* binary tpb block or *TPB* instance itself to set *default_tpb* or as paraneter to *begin()*.

Example:

```
import fdb

con = fdb.connect(dsn='localhost:employee',user='sysdba', password='masterkey')

# Use TPB to construct valid transaction parameter block
# from the fdb.isc_tpb_* family.
customTPB = fdb.TPB()
customTPB.isolation_level = fdb.isc_tpb_consistency # SERIALIZABLE
customTPB.table_reservation["MY_TABLE"] = (fdb.isc_tpb_protected, fdb.isc_tpb_lock_
↪write)

# Explicitly start a transaction with the custom TPB:
con.begin(tpb=customTPB)

# For frequent use, it's better to use already assembled version of TPB
customTPB = fdb.TPB()
customTPB.access_mode = fdb.isc_tpb_read  # read only
customTPB.isolation_level = fdb.isc_tpb_concurrency # SNAPSHOT
customTPB = customTPB.render() # Create valid block according to current values of␣
↪member attributes.

for x in range(1000):
   con.begin(tpb=customTPB)
```

If you want to build only *table reservation* part of *tpb* (for example to add to various custom built parameter blocks), you can use class *TableReservation* instead *TPB*.

Getting information about transaction

Transaction object exposes two methods that return information about currently managed active transaction (the same methods are exposed also by *Connection* object for *main_transaction*):

transaction_info() is a very thin wrapper around function *isc_transaction_info()*. This method does not attempt to interpret its results except with regard to whether they are a string or an integer.

A more convenient way to access the same functionality is via the *trans_info()* method, which is high-level convenience wrapper around the *transaction_info* method that parses the output of *transaction_info* into Python-friendly objects instead of returning raw binary buffers in the case of complex result types.

Example program:

```
import fdb

con = fdb.connect(dsn='localhost:employee',user='sysdba', password='masterkey')

# Start transaction, so we can get information about it
con.begin()

info = con.trans_info([fdb.isc_info_tra_id, fdb.isc_info_tra_oldest_interesting,
                       fdb.isc_info_tra_oldest_snapshot, fdb.isc_info_tra_oldest_
↪active,
                       fdb.isc_info_tra_isolation, fdb.isc_info_tra_access,
                       fdb.isc_info_tra_lock_timeout])

print info
print "TransactionID:", info[fdb.isc_info_tra_id]
print "Oldest Interesting (OIT):",info[fdb.isc_info_tra_oldest_interesting]
```

(continues on next page)

(continued from previous page)

```
print "Oldest Snapshot:",info[fdb.isc_info_tra_oldest_snapshot]
print "Oldest Active (OAT):",info[fdb.isc_info_tra_oldest_active]
print "Isolation Level:",info[fdb.isc_info_tra_isolation]
print "Access Mode:",info[fdb.isc_info_tra_access]
print "Lock Timeout:",info[fdb.isc_info_tra_lock_timeout]
```

Output:

```
{4: 459, 5: 430, 6: 459, 7: 459, 8: (3, 1), 9: 1, 10: -1}
TransactionID: 459
Oldest Interesting (OIT): 430
Oldest Snapshot: 459
Oldest Active (OAT): 459
Isolation Level: (3, 1)
Access Mode: 1
Lock Timeout: -1
```

Note: Isolation level info values are available as FDB constants *isc_info_tra_consistency*, *isc_info_tra_concurrency* and *isc_info_tra_read_committed*. For *read committed*, a tuple of two values is returned instead single value, where the second value is record version flag *isc_info_tra_no_rec_version* or *isc_info_tra_rec_version*.

Access mode values are available as FDB constants *isc_info_tra_readonly* and *isc_info_tra_readwrite*.

Retaining transactions

The `commit()` and `rollback()` methods accept an optional boolean parameter *retaining* (**default False**) to indicate whether to recycle the transactional context of the transaction being resolved by the method call.

If retaining is *True*, the infrastructural support for the transaction active at the time of the method call will be "retained" (efficiently and transparently recycled) after the database server has committed or rolled back the conceptual transaction.

Important: In code that commits or rolls back frequently, "retaining" the transaction yields considerably better performance. However, retaining transactions must be used cautiously because they can interfere with the server's ability to garbage collect old record versions. For details about this issue, read the "Garbage" section of this document by Ann Harrison.

For more information about retaining transactions, see *Firebird documentation*.

Savepoints

Savepoints are named, intermediate control points within an open transaction that can later be rolled back to, without affecting the preceding work. Multiple savepoints can exist within a single unresolved transaction, providing "multi-level undo" functionality.

Although Firebird savepoints are fully supported from SQL alone via the *SAVEPOINT 'name'* and *ROLLBACK TO 'name'* statements, FDB also exposes savepoints at the Python API level for the sake of convenience.

Call to method `savepoint()` establishes a savepoint with the specified *name*. To roll back to a specific savepoint, call the `rollback()` method and provide a value (the name of the savepoint) for the optional *savepoint* parameter. If the savepoint parameter of *rollback()* is not specified, the active transaction is cancelled in its entirety, as required by the Python Database API Specification.

The following program demonstrates savepoint manipulation via the FDB API, rather than raw SQL.

```
import fdb

con = fdb.connect(dsn='employee', user='sysdba', password='pass')
cur = con.cursor()

cur.execute("recreate table test_savepoints (a integer)")
con.commit()

print 'Before the first savepoint, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

cur.execute("insert into test_savepoints values (?)", [1])
con.savepoint('A')
print 'After savepoint A, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

cur.execute("insert into test_savepoints values (?)", [2])
con.savepoint('B')
print 'After savepoint B, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

cur.execute("insert into test_savepoints values (?)", [3])
con.savepoint('C')
print 'After savepoint C, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

con.rollback(savepoint='A')
print 'After rolling back to savepoint A, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()

con.rollback()
print 'After rolling back entirely, the contents of the table are:'
cur.execute("select * from test_savepoints")
print ' ', cur.fetchall()
```

The output of the example program is shown below:

```
Before the first savepoint, the contents of the table are:
  []
After savepoint A, the contents of the table are:
  [(1,)]
After savepoint B, the contents of the table are:
  [(1,), (2,)]
After savepoint C, the contents of the table are:
  [(1,), (2,), (3,)]
After rolling back to savepoint A, the contents of the table are:
  [(1,)]
After rolling back entirely, the contents of the table are:
  []
```

Using multiple transactions with the same connection

To use additional transactions that could run simultaneously with *main transaction* managed by *Connection*, create new *Transaction* object calling *Connection.trans()* method. If you don't specify the optional *default_tpb* parameter, this new *Transaction* inherits the *default_tpb* from *Connection*. Physical transaction is not started when *Transaction* instance is created, but implicitly when first SQL statement is executed, or explicitly via *Transaction.begin()* call.

To execute statements in context of this additional transaction you have to use *cursors* obtained directly from this *Transaction* instance calling its *cursor()* method, or call *Transaction.execute_immediate()* method.

Example:

```
import fdb

con = fdb.connect(dsn='employee', user='sysdba', password='pass')
# Cursor for main_transaction context
cur = con.cursor()

# Create new READ ONLY READ COMMITTED transaction
ro_transaction = con.trans(fdb.ISOLATION_LEVEL_READ_COMMITED_RO)
# and cursor
ro_cur = ro_transaction.cursor()

cur.execute('insert into country values ('Oz','Crowns')
con.commit() # commits main transaction

# Read data created by main transaction from second one
ro_cur.execute("select * from COUNTRY where COUNTRY = `Oz`")
print ro_cur.fetchall()

# Insert more data, but don't commit
cur.execute('insert into country values ('Barsoom','XXX')

# Read data created by main transaction from second one
ro_cur.execute("select * from COUNTRY where COUNTRY = `Barsoom`")
print ro_cur.fetchall()
```

Distributed Transactions

Distributed transactions are transactions that span multiple databases. FDB provides this Firebird feature through *ConnectionGroup* class. Instances of this class act as managers for *Transaction* object that is bound to multiple connections, and to *cursors* bound to it and connections participated in group. That's it, distributed transaction is fully independent from all other transactions, main or secondary, of member connections.

To assemble a group of connections, you can either pass the sequence of *Connection* instances to *ConnectionGroup* constructor, or add connections latter calling *ConnectionGroup.add()* method.

Any *Connection* could be a member of only one group, and attempt to add it to another one would raise an exception. Also, *Connection* participating in group cannot be *closed* before it's *removed* or whole group is *disbanded*.

Warning: Never add more than one connection to the same database to the same *ConnectionGroup*!

Similarly to *Transaction*, distributed transactions are managed through *ConnectionGroup.begin()*, *ConnectionGroup.savepoint()*. *ConnectionGroup.commit()* and *ConnectionGroup.*

rollback() methods. Additionally, *ConnectionGroup* exposes method *prepare()* that explicitly initiates the first phase of *Two-Phase Commit Protocol*. Transaction parameters are defined similarly to *Transaction* using *ConnectionGroup.default_tpb* or as optional parameter to *begin()* call.

SQL statements that should belong to context of distributed transaction are executed via *Cursor* instances aquired through *ConnectionGroup.cursor()* method, or calling *ConnectionGroup.execute_immediate()* method.

Note: Because *Cursor* instances can belong to only one *Connection*, the *cursor()* method has manadory parameter *connection*, to specify to which member connection cursor should belong.

Example program:

```
import fdb

# First database
con1 = fdb.create_database("CREATE DATABASE 'testdb-1.fdb' USER 'SYSDBA' PASSWORD
↪'masterkey'")
con1.execute_immediate("recreate table T (PK integer, C1 integer)")
con1.commit()

# Second database
con2 = fdb.create_database("CREATE DATABASE 'testdb-2,fdb' USER 'SYSDBA' PASSWORD
↪'masterkey'")
con2.execute_immediate("recreate table T (PK integer, C1 integer)")
con2.commit()

# Create connection group
cg = fdb.ConnectionGroup((con1,con2))

# Prepare Group cursors for each connection
gc1 = cg.cursor(con1)
gc2 = cg.cursor(con2)

# Connection cursors to check content of databases
q = 'select * from T order by pk'

cc1 = con1.cursor()
p1 = cc1.prep(q)

cc2 = con2.cursor()
p2 = cc2.prep(q)

print "Distributed transaction: COMMIT"
#      ==============================
gc1.execute('insert into t (pk) values (1)')
gc2.execute('insert into t (pk) values (1)')
cg.commit()

# check it
con1.commit()
cc1.execute(p1)
print 'db1:',cc1.fetchall()
con2.commit()
cc2.execute(p2)
print 'db2:',cc2.fetchall()
```

(continues on next page)

(continued from previous page)

```
print "Distributed transaction: PREPARE + COMMIT"
#      ========================================
gc1.execute('insert into t (pk) values (2)')
gc2.execute('insert into t (pk) values (2)')
cg.prepare()
cg.commit()

# check it
con1.commit()
cc1.execute(p1)
print 'db1:',cc1.fetchall()
con2.commit()
cc2.execute(p2)
print 'db2:',cc2.fetchall()

print "Distributed transaction: SAVEPOINT + ROLLBACK to it"
#      ================================================
gc1.execute('insert into t (pk) values (3)')
cg.savepoint('CG_SAVEPOINT')
gc2.execute('insert into t (pk) values (3)')
cg.rollback(savepoint='CG_SAVEPOINT')

# check it - via group cursors, as transaction is still active
gc1.execute(q)
print 'db1:',gc1.fetchall()
gc2.execute(q)
print 'db2:',gc2.fetchall()

print "Distributed transaction: ROLLBACK"
#      ================================
cg.rollback()

# check it
con1.commit()
cc1.execute(p1)
print 'db1:',cc1.fetchall()
con2.commit()
cc2.execute(p2)
print 'db2:',cc2.fetchall()

print "Distributed transaction: EXECUTE_IMMEDIATE"
#      ========================================
cg.execute_immediate('insert into t (pk) values (3)')
cg.commit()

# check it
con1.commit()
cc1.execute(p1)
print 'db1:',cc1.fetchall()
con2.commit()
cc2.execute(p2)
print 'db2:',cc2.fetchall()

# Finalize
con1.drop_database()
con1.close()
```

(continues on next page)

(continued from previous page)

```
con2.drop_database()
con2.close()
```

Output:

```
Distributed transaction: COMMIT
db1: [(1, None)]
db2: [(1, None)]
Distributed transaction: PREPARE + COMMIT
db1: [(1, None), (2, None)]
db2: [(1, None), (2, None)]
Distributed transaction: SAVEPOINT + ROLLBACK to it
db1: [(1, None), (2, None), (3, None)]
db2: [(1, None), (2, None)]
Distributed transaction: ROLLBACK
db1: [(1, None), (2, None)]
db2: [(1, None), (2, None)]
Distributed transaction: EXECUTE_IMMEDIATE
db1: [(1, None), (2, None), (3, None)]
db2: [(1, None), (2, None), (3, None)]
```

Transaction Context Manager

FDB provides context manager *TransactionContext* that allows automatic transaction management using The with statement. It can work with any object that supports *begin()*, *commit()* and *rollback()* methods, i.e. *Connection*, *ConnectionGroup* or *Transaction*.

It starts transaction when WITH block is entered and commits it if no exception occurst within it, or calls *rollback()* otherwise. Exceptions raised in WITH block are never suppressed.

Examples:

```
con = fdb.connect(dsn='employee',user='sysdba',password='masterkey')

# Uses default main transaction
with TransactionContext(con):
   cur = con.cursor()
   cur.execute("insert into T (PK,C1) values (1,'TXT')")

# Uses separate transaction
with TransactionContext(con.trans()) as tr:
   cur = tr.cursor()
   cur.execute("insert into T (PK,C1) values (2,'AAA')")

# Uses connection group (distributed transaction)
con2 = fdb.connect(dsn='remote:employee',user='sysdba',password='masterkey')
cg = fdb.ConnectionGroup((con,con2))
with TransactionContext(cg):
   cur1 = cg.cursor(con)
   cur2 = cg.cursor(con2)
   cur1.execute("insert into T (PK,C1) values (3,'Local')")
   cur2.execute("insert into T (PK,C1) values (3,'Remote')")
```

1.2.6 Database Events

What they are

The Firebird engine features a distributed, interprocess communication mechanism based on messages called *database events*. A database event is a message passed from a trigger or stored procedure to an application to announce the occurrence of a specified condition or action, usually a database change such as an insertion, modification, or deletion of a record. The Firebird event mechanism enables applications to respond to actions and database changes made by other, concurrently running applications without the need for those applications to communicate directly with one another, and without incurring the expense of CPU time required for periodic polling to determine if an event has occurred.

Why use them

Anything that can be accomplished with database events can also be implemented using other techniques, so why bother with events? Since you've chosen to write database-centric programs in Python rather than assembly language, you probably already know the answer to this question, but let's illustrate.

A typical application for database events is the handling of administrative messages. Suppose you have an administrative message database with a *message's* table, into which various applications insert timestamped status reports. It may be desirable to react to these messages in diverse ways, depending on the status they indicate: to ignore them, to initiate the update of dependent databases upon their arrival, to forward them by e-mail to a remote administrator, or even to set off an alarm so that on-site administrators will know a problem has occurred.

It is undesirable to tightly couple the program whose status is being reported (the *message producer*) to the program that handles the status reports (the *message handler*). There are obvious losses of flexibility in doing so. For example, the message producer may run on a separate machine from the administrative message database and may lack access rights to the downstream reporting facilities (e.g., network access to the SMTP server, in the case of forwarded e-mail notifications). Additionally, the actions required to handle status reports may themselves be time-consuming and error-prone, as in accessing a remote network to transmit e-mail.

In the absence of database event support, the message handler would probably be implemented via *polling*. Polling is simply the repetition of a check for a condition at a specified interval. In this case, the message handler would check in an infinite loop to see whether the most recent record in the *messages* table was more recent than the last message it had handled. If so, it would handle the fresh message(s); if not, it would go to sleep for a specified interval, then loop.

The *polling-based* implementation of the message handler is fundamentally flawed. Polling is a form of busy-wait; the check for new messages is performed at the specified interval, regardless of the actual activity level of the message producers. If the polling interval is lengthy, messages might not be handled within a reasonable time period after their arrival; if the polling interval is brief, the message handler program (and there may be many such programs) will waste a large amount of CPU time on unnecessary checks.

The database server is necessarily aware of the exact moment when a new message arrives. Why not let the message handler program request that the database server send it a notification when a new message arrives? The message handler can then efficiently sleep until the moment its services are needed. Under this *event-based* scheme, the message handler becomes aware of new messages at the instant they arrive, yet it does not waste CPU time checking in vain for new messages when there are none available.

How events are exposed

1. Server Process ("An event just occurred!")

   To notify any interested listeners that a specific event has occurred, issue the *POST_EVENT* statement from Stored Procedure or Trigger. The *POST_EVENT* statement has one parameter: the name of the event to post.

In the preceding example of the administrative message database, *POST_EVENT* might be used from an *after insert* trigger on the *messages* table, like this:

```
create trigger trig_messages_handle_insert
  for messages
    after insert
as
begin
  POST_EVENT 'new_message';
end
```

Note: The physical notification of the client process does not occur until the transaction in which the *POST_EVENT* took place is actually committed. Therefore, multiple events may *conceptually* occur before the client process is *physically* informed of even one occurrence. Furthermore, the database engine makes no guarantee that clients will be informed of events in the same groupings in which they conceptually occurred. If, within a single transaction, an event named *event_a* is posted once and an event named *event_b* is posted once, the client may receive those posts in separate "batches", despite the fact that they occurred in the same conceptual unit (a single transaction). This also applies to multiple occurrences of *the same* event within a single conceptual unit: the physical notifications may arrive at the client separately.

---

2. Client Process ("Send me a message when an event occurs.")

Note: If you don't care about the gory details of event notification, skip to the section that describes FDB's Python-level event handling API.

The Firebird C client library offers two forms of event notification. The first form is *synchronous* notification, by way of the function `isc_wait_for_event()`. This form is admirably simple for a C programmer to use, but is inappropriate as a basis for FDB's event support, chiefly because it's not sophisticated enough to serve as the basis for a comfortable Python-level API. The other form of event notification offered by the database client library is *asynchronous*, by way of the functions `isc_que_events()` (note that the name of that function is misspelled), `isc_cancel_events()`, and others. The details are as nasty as they are numerous, but the essence of using asynchronous notification from C is as follows:

(a) Call `isc_event_block()` to create a formatted binary buffer that will tell the server which events the client wants to listen for.

(b) Call `isc_que_events()` (passing the buffer created in the previous step) to inform the server that the client is ready to receive event notifications, and provide a callback that will be asynchronously invoked when one or more of the registered events occurs.

(c) [The thread that called `isc_que_events()` to initiate event listening must now do something else.]

(d) When the callback is invoked (the database client library starts a thread dedicated to this purpose), it can use the `isc_event_counts()` function to determine how many times each of the registered events has occurred since the last call to `isc_event_counts()` (if any).

(e) [The callback thread should now "do its thing", which may include communicating with the thread that called `isc_que_events()`.]

(f) When the callback thread is finished handling an event notification, it must call `isc_que_events()` again in order to receive future notifications. Future notifications will invoke the callback again, effectively "looping" the callback thread back to Step 4.

API for Python developers

The FDB database event API is comprised of the following: the method *Connection.event_conduit()* and the class *EventConduit*.

The *EventConduit* class serve as "conduit" through which database event notifications will flow into the Python program. It's not designed to be instantiated directly by the Python programmer. Instead, use the *Connection.event_conduit()* method to create *EventConduit* instances. *event_conduit* is a method of *Connection* rather than a module-level function or a class constructor because the database engine deals with events in the context of a particular database (after all, *POST_EVENT* must be issued by a stored procedure or a trigger).

Connection.event_conduit() takes a sequence of string event names as parameter, and returns *EventConduit* instance.

Important: To start listening for events it's necessary (starting from FDB version 1.4.2) to call *EventConduit.begin()* method or use EventConduit's context manager interface.

Immediately when *begin()* method is called, EventConduit starts to accumulate notifications of any events that occur within the conduit's internal queue until the conduit is closed either explicitly (via the *close()* method) or implicitly (via garbage collection).

Notifications about events are aquired through call to *wait()* method, that blocks the calling thread until at least one of the events occurs, or the specified *timeout* (if any) expires, and returns *None* if the wait timed out, or a dictionary that maps *event_name* -> *event_occurrence_count*.

Important: *EventConduit* can act as context manager that ensures execution of *begin()* and *close()* methods. It's strongly advised to use the *EventConduit* with the *with* statement.

Example:

```
with connection.event_conduit( ('event_a', 'event_b') ) as conduit:
    events = conduit.wait()
    process_events(events)
```

If you want to drop notifications accumulated so far by conduit, call *EventConduit.flush()* method.

Example program:

```
import fdb
import threading
import time

# Prepare database
con = fdb.create_database("CREATE DATABASE 'event_test.fdb' USER 'SYSDBA' PASSWORD
↪'masterkey'")
con.execute_immediate("CREATE TABLE T (PK Integer, C1 Integer)")
con.execute_immediate("""CREATE TRIGGER EVENTS_AU FOR T ACTIVE
BEFORE UPDATE POSITION 0
AS
BEGIN
   if (old.C1 <> new.C1) then
      post_event 'c1_updated' ;
END""")
con.execute_immediate("""CREATE TRIGGER EVENTS_AI FOR T ACTIVE
AFTER INSERT POSITION 0
```

(continues on next page)

(continued from previous page)

```
AS
BEGIN
   if (new.c1 = 1) then
     post_event 'insert_1' ;
   else if (new.c1 = 2) then
     post_event 'insert_2' ;
   else if (new.c1 = 3) then
     post_event 'insert_3' ;
   else
     post_event 'insert_other' ;
END""")
con.commit()
cur = con.cursor()

# Utility function
def send_events(command_list):
   for cmd in command_list:
      cur.execute(cmd)
   con.commit()

print "One event"
#      =========
timed_event = threading.Timer(3.0,send_events,args=[["insert into T (PK,C1) values (1,
↪1)",]])
events = con.event_conduit(['insert_1'])
events.begin()
timed_event.start()
e = events.wait()
events.close()
print e

print "Multiple events"
#      ===============
cmds = ["insert into T (PK,C1) values (1,1)",
        "insert into T (PK,C1) values (1,2)",
        "insert into T (PK,C1) values (1,3)",
        "insert into T (PK,C1) values (1,1)",
        "insert into T (PK,C1) values (1,2)",]
timed_event = threading.Timer(3.0,send_events,args=[cmds])
events = self.con.event_conduit(['insert_1','insert_3'])
events.begin()
timed_event.start()
e = events.wait()
events.close()
print e

print "20 events"
#      =========
cmds = ["insert into T (PK,C1) values (1,1)",
        "insert into T (PK,C1) values (1,2)",
        "insert into T (PK,C1) values (1,3)",
        "insert into T (PK,C1) values (1,1)",
        "insert into T (PK,C1) values (1,2)",]
timed_event = threading.Timer(1.0,send_events,args=[cmds])
events = con.event_conduit(['insert_1','A','B','C','D',
                            'E','F','G','H','I','J','K','L','M',
                            'N','O','P','Q','R','insert_3'])
```

(continues on next page)

(continued from previous page)

```
events.begin()
timed_event.start()
time.sleep(3)
e = events.wait()
events.close()
print e

print "Flush events"
#      ============
timed_event = threading.Timer(3.0,send_events,args=[["insert into T (PK,C1) values (1,
↪1)",]])
events = con.event_conduit(['insert_1'])
events.begin()
send_events(["insert into T (PK,C1) values (1,1)",
             "insert into T (PK,C1) values (1,1)"])
time.sleep(2)
events.flush()
timed_event.start()
e = events.wait()
events.close()
print e

# Finalize
con.drop_database()
con.close()
```

Output:

```
One event
{'insert_1': 1}
Multiple events
{'insert_3': 1, 'insert_1': 2}
20 events
{'A': 0, 'C': 0, 'B': 0, 'E': 0, 'D': 0, 'G': 0, 'insert_1': 2, 'I': 0, 'H': 0, 'K':␣
↪0, 'J': 0, 'M': 0,
 'L': 0, 'O': 0, 'N': 0, 'Q': 0, 'P': 0, 'R': 0, 'insert_3': 1, 'F': 0}
Flush events
{'insert_1': 1}
```

1.2.7 Working with Services

Database server maintenance tasks such as user management, load monitoring, and database backup have traditionally been automated by scripting the command-line tools **gbak**, **gfix**, **gsec**, and **gstat**.

The API presented to the client programmer by these utilities is inelegant because they are, after all, command-line tools rather than native components of the client language. To address this problem, Firebird has a facility called the *Services API*, which exposes a uniform interface to the administrative functionality of the traditional command-line tools.

The native Services API, though consistent, is much lower-level than a Pythonic API. If the native version were exposed directly, accomplishing a given task would probably require more Python code than scripting the traditional command-line tools. For this reason, FDB presents its own abstraction over the native API via the *fdb.services* module.

Services API Connections

All Services API operations are performed in the context of a *connection* to a specific database server, represented by the *fdb.services.Connection* class. Similarly to database connections, FDB provides *connect()* constructor function to create such connections.

This constructor has three keyword parameters:

host The network name of the computer on which the database server is running.

user The name of the database user under whose authority the maintenance tasks are to be performed.

password User's password.

Since maintenance operations are most often initiated by an administrative user on the same computer as the database server, *host* defaults to the local computer, and *user* defaults to *SYSDBA*.

The three calls to *fdb.services.connect()* in the following program are equivalent:

```
from fdb import services

con = services.connect(password='masterkey')
con = services.connect(user='sysdba', password='masterkey')
con = services.connect(host='localhost', user='sysdba', password='masterkey')
```

Note: Like database connections, it's good practice to *close()* them when you don't need them anymore.

Connection object provides number of methods that could be divided into several groups:

- *Server Configuration and State*: To get information about server configuration, active attachments or users, or to get content of server log.
- *Database options*: To set various database parameters like size of page cache, access mode or SQL dialect.
- *Database maintenance*: To perform backup, restore, validation or other database maintenance tasks.
- *User maintenance*: To get or change information about users defined in security database, to create new or remove users.
- *Trace service*: To start, stop, pause/resume or list Firebird *trace sessions*.
- *Text ouput from Services*: Some services like *backup* or *trace* may return significant amount of text. This output is not returned directly by method that starts the service, but through separate methods that emulate read from text file, or provide iterator protocol support on *Connection*.

Server Configuration and State

get_service_manager_version()

To help client programs adapt to version changes, the service manager exposes its version number as an integer.

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> print con.get_service_manager_version()
2
```

fdb.services is a thick wrapper of the Services API that can shield its users from changes in the underlying C API, so this method is unlikely to be useful to the typical Python client programmer.

get_server_version()

Returns the server's version string

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> print con.get_server_version()
LI-V2.5.2.26536 Firebird 2.5
```

At first glance, this method appears to duplicate the functionality of the *fdb.Connection.server_version* property, but when working with Firebird, there is a difference. *fdb.Connection.server_version* is based on a C API call (*isc_database_info()*) that existed long before the introduction of the Services API. Some programs written before the advent of Firebird test the version number in the return value of *isc_database_info()*, and refuse to work if it indicates that the server is too old. Since the first stable version of Firebird was labeled 1.0, this pre-Firebird version testing scheme incorrectly concludes that (e.g.) Firebird 1.0 is older than Interbase 5.0.

Firebird addresses this problem by making *isc_database_info()* return a "pseudo-InterBase" version number, whereas the Services API returns the true Firebird version, as shown:

```
# 64-bit Linux Firebird 2.5.1 SuperServer
import fdb
con = fdb.connect(dsn='employee', user='sysdba', password='masterkey')
print 'Interbase-compatible version string:', con.server_version
svcCon = fdb.services.connect(password='masterkey')
print 'Actual Firebird version string:     ', svcCon.get_server_version()
```

Output (on Firebird 2.5.1/Linux64):

```
Interbase-compatible version string: LI-V6.3.1.26351 Firebird 2.5
Actual Firebird version string:      LI-V2.5.1.26351 Firebird 2.5
```

get_architecture()

Returns platform information for the server, including hardware architecture and operating system family.

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> print con.get_architecture()
Firebird/linux AMD64
```

get_home_directory()

Returns the equivalent of the RootDirectory setting from `firebird.conf`.

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> print con.get_home_directory()
/opt/firebird/
```

get_security_database_path()

Returns the location of the server's core security database, which contains user definitions and such. Name of this database is `security2.fdb` (Firebird 2.0 and later) or `security.fdb` (Firebird 1.5).

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> print con.get_security_database_path()
/opt/firebird/security2.fdb
```

get_lock_file_directory()

Returns the directory location for Firebird lock files.

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> print con.get_lock_file_directory()
/tmp/firebird/
```

get_server_capabilities()

Returns tuple of capability info codes for each capability reported by Firebird server. Following constants are defined in *fdb.services* for convenience:

- CAPABILITY_MULTI_CLIENT
- CAPABILITY_REMOTE_HOP
- CAPABILITY_SERVER_CONFIG
- CAPABILITY_QUOTED_FILENAME
- CAPABILITY_NO_SERVER_SHUTDOWN

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> print con.get_server_capabilities()
(2L, 4L, 512L, 256L)
>>> fdb.services.CAPABILITY_MULTI_CLIENT in con.get_server_capabilities()
True
>>> fdb.services.CAPABILITY_QUOTED_FILENAME in con.get_server_capabilities()
False
```

get_message_file_directory()

To support internationalized error messages/prompts, the database engine stores its messages in a file named `firebird.msg`. The directory in which this file resides can be determined with this method.

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> print con.get_message_file_directory()
/opt/firebird/
```

get_connection_count()

Returns the number of active connections to databases managed by the server. This count only includes database connections (such as open instances of *fdb.Connection*), not services manager connections (such as open instances of *fdb.services.Connection*).

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> db1 = fdb.connect(dsn='employee',user='sysdba',password='masterkey')
>>> db2 = fdb.connect(dsn='employee',user='sysdba',password='masterkey')
>>> print con.get_connection_count()
2
```

get_attached_database_names()

Returns a list of the names of all databases to which the server is maintaining at least one connection. The database names are not guaranteed to be in any particular order.

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> db1 = fdb.connect(dsn='employee',user='sysdba',password='masterkey')
>>> db2 = fdb.connect(dsn='employee',user='sysdba',password='masterkey')
>>> print con.get_attached_database_names()
['/opt/firebird/examples/empbuild/employee.fdb']
```

get_log()

Request the contents of the server's log file (`firebird.log`).

This method is so-called *Async method* that only initiates log transfer. Actual log content could be read by one from many methods for *text ouput from Services* that *Connection* provides .

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.get_log()
>>> log = con.readlines()
```

Tip: You can use *fdb.log* module for parsing and further data processing.

Database options

set_default_page_buffers()

Sets individual page cache size for Database.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.set_default_page_buffers('employee',100)
```

set_sweep_interval()

Sets treshold for automatic sweep.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.set_sweep_interval('employee',100000)
```

set_reserve_page_space()

Sets data page space reservation policy.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
# Use all space
>>> con.set_reserve_page_space('employee',False)
```

set_write_mode()

Sets Disk Write Mode: Sync (forced writes) or Async (buffered). Following constants are defined in *fdb.services* for convenience:

- WRITE_FORCED
- WRITE_BUFFERED

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
# Disable Forced Writes
>>> con.set_write_mode('employee',services.WRITE_BUFFERED)
```

set_access_mode()

Sets Database Access mode: Read Only or Read/Write. Following constants are defined in *fdb.services* for convenience:

- ACCESS_READ_WRITE
- ACCESS_READ_ONLY

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
# Set database to R/O mode
>>> con.set_access_mode('employee',services.ACCESS_READ_ONLY)
```

set_sql_dialect()

Sets SQL Dialect for Database.

Warning: Changing SQL dialect on existing database is not recommended. Only newly created database objects would respect new dialect setting, while objects created with previous dialect remain unchanged. That may have dire consequences.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
# Use SQL dialect 1
>>> con.set_sql_dialect('employee',1)
```

Database maintenance

get_limbo_transaction_ids()

Returns list of transactions in limbo.

commit_limbo_transaction()

Resolves limbo transaction with commit.

rollback_limbo_transaction()

Resolves limbo transaction with rollback.

get_statistics()

Request database statisctics. Report is in the same format as the output of the gstat command-line utility. This method has one required parameter, the location of the database on which to compute statistics, and six optional boolean parameters for controlling the domain of the statistics.

This method is so-called *Async method* that only initiates report processing. Actual report could be read by one from many methods for *text ouput from Services* that *Connection* provides .

Note: Until statistical report is not fully fetched from service (or ignored via *wait()*), any attempt to start another asynchronous service will fail with exception.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.get_statistics('employee')
>>> stat_report = con.readlines()
```

Tip: You can use *fdb.gstat* module for parsing and further data processing.

---

backup()

Request logical (GBAK) database backup. Produces report about backup process.

This method is so-called *Async method* that only initiates backup process. Actual report could be read by one from many methods for *text ouput from Services* that *Connection* provides .

Note: Until backup report is not fully fetched from service (or ignored via *wait()*), any attempt to start another asynchronous service will fail with exception.

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.backup('employee', '/home/data/employee.fbk', metadata_only=True,␣
↪collect_garbage=False)
>>> backup_report = con.readlines()
```

Note: *backup()* creates a backup file on server host. Alternatively you can use *local_backup()* to create a backup file on local machine.

---

restore()

Request database restore from logical (GBAK) backup. Produces report about restore process.

This method is so-called *Async method* that only initiates restore process. Actual report could be read by one from many methods for *text ouput from Services* that *Connection* provides .

Note: Until restore report is not fully fetched from service (or ignored via *wait()*), any attempt to start another asynchronous service will fail with exception.

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.restore('/home/data/employee.fbk', '/home/data/empcopy.fdb')
>>> restore_report = con.readlines()
```

Note: *restore()* uses a backup file on server host. Alternatively you can use *local_restore()* to use a backup file on local machine.

---

nbackup()

Perform physical (NBACKUP) database backup.

Note: Method call will not return until sweep is finished.

---

nrestore()

Perform restore from physical (NBACKUP) database backup.

Note: Method call will not return until sweep is finished.

---

shutdown()

Database shutdown. Following constants are defined in *fdb.services* for convenience:

For shutdow mode:

- SHUT_SINGLE
- SHUT_MULTI
- SHUT_FULL

For shutdown method:

- SHUT_FORCE
- SHUT_DENY_NEW_TRANSACTIONS
- SHUT_DENY_NEW_ATTACHMENTS

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
```

(continues on next page)

(continued from previous page)

```
# Shutdown database to single-user maintenance mode
>>> con.shutdown('empoyee', services.SHUT_SINGLE, services.SHUT_FORCE, 0)
# Go to full shutdown mode, disabling new attachments during 5 seconds
>>> con.shutdown('empoyee', services.SHUT_FULL, services.SHUT_DENY_NEW_
↪ATTACHMENTS, 5)
```

bring_online()

Bring previously shut down database back online. Following constants are defined in *fdb.services* for convenience:

For on-line mode:

- SHUT_NORMAL
- SHUT_SINGLE
- SHUT_MULTI

```
# 64-bit Linux Firebird 2.5.1 SuperServer
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
# Enable multi-user maintenance
>>> con.bring_online('employee', services.SHUT_MULTI)
# Enable single-user maintenance
>>> con.bring_online('employee', services.SHUT_SINGLE)
# Return to normal state
>>> con.bring_online('employee')
```

activate_shadow()

Activates Database Shadow(s).

sweep()

Performs Database Sweep.

Note: Method call will not return until sweep is finished.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.sweep('empoyee')
```

repair()

Database Validation and Repair.

Note: Method call will not return until action is finished.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
# Just validate
>>> con.repair('empoyee', ignore_checksums=True, read_only_validation=True)
```

(continues on next page)

(continued from previous page)

```
# Mend the database
>>> con.repair('empoyee', ignore_checksums=True, mend_database=True)
```

User maintenance

get_users()

Returns information about specified user or all users as a list of *User* instances.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> users = con.get_users()
>>> for user in users:
...     print user.name
...     print user.first_name, user.middle_name, user.last_name
...     print user.user_id, user.group_id
SYSDBA
Sql Server Administrator
0 0
```

add_user()

Adds new user. Requires instance of *User* with **at least** its `name` and `password` attributes specified as non-empty values. All other attributes are optional.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> user = services.User('NewUser')
>>> user.password = 'secret'
>>> user.first_name = 'John'
>>> user.last_name = 'Doe'
>>> con.add_users(User)
```

modify_user()

Modifycation of user information. Requires instance of *User* with **at least** its `name` attribute specified as non-empty value.

Note: Sets `first_name`, `middle_name` and `last_name` to their actual values, and ignores the `user_id` and `group_id` attributes regardless of their values. `password` is set **only** when it has value.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> user = services.User('SYSDBA')
# Change pasword
>>> user.password = 'Pa$$w0rd'
>>> con.modify_user(User)
```

remove_user()

Removes user. Requires User name or instance of *User* with **at least** its `name` attribute specified as non-empty value.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.remove_user('NewUser')
```

user_exists()

Checks for user's existence. Requires User name or instance of *User* with **at least** its `name` attribute specified as non-empty value.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.user_exists('NewUser')
False
```

Trace service

Tip: You can use *fdb.trace* module for parsing and further data processing.

trace_start()

Starts new trace session. Requires trace *configuration* and returns *Session ID*.

Trace session output could be retrieved through *readline()*, *readlines()*, iteration over *Connection* or ignored via call to *wait()*.

Note: Until session output is not fully fetched from service (or ignored via *wait()*), any attempt to start another asynchronous service including call to any *trace_* method will fail with exception.

```
import fdb

svc = fdb.services.connect(password='masterkey')
# Because trace session blocks the connection, we need another one to stop␣
↪trace session!
svc_aux = fdb.services.connect(password='masterkey')

trace_config = """<database 'employee'>
    enabled true
    log_statement_finish true
    print_plan true
    include_filter %%SELECT%%
    exclude_filter %%RDB$%%
    time_threshold 0
    max_sql_length 2048
</database>
"""
trace_id = svc.trace_start(trace_config,'test_trace_1')
trace_log = []
# Get first 10 lines of trace output
for i in range(10):
   trace_log.append(svc.readline())
```

(continues on next page)

(continued from previous page)

```
# Stop trace session
svc_aux.stop_trace(trace_id)
```

trace_stop()

Stops trace session specified by ID.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.trace_stop(15)
Trace session ID 15 stopped
```

trace_suspend()

Suspends trace session specified by ID.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.trace_suspend(15)
Trace session ID 15 paused
```

trace_resume()

resumes trace session specified by ID.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.trace_resume(15)
Trace session ID 15 resumed
```

trace_list()

Returns information about existing trace sessions as dictionary mapping *SESSION_ID -> SESSION_PARAMS*. Session parameters is another dictionary with next keys:

name (string) (optional) Session name if specified.

date (datetime.datetime) Session start date and time.

user (string) Trace user name.

flags (list of strings) Session flags.

```
>>> from fdb import services
>>> con = services.connect(host='localhost', user='sysdba', password=
↪'masterkey')
>>> con.trace_list()
{53: {'date': datetime.datetime(2012, 10, 5, 10, 45, 4),
      'flags': ['active', ' admin', ' trace'],
      'user': 'SYSDBA'}}
```

Text ouput from Services

Some services like *backup* or *trace* may return significant amount of text. Rather than return the whole text as single string value by methods that provide access to these services, FDB isolated the transfer process to separate methods:

- *readline()* - Similar to file.readline(), returns next line of output from Service.
- *readlines()* - Like file.readlines(), returns list of output lines.
- Iteration over *Connection* object, because *Connection* has built-in support for iterator protocol.
- Using *callback* method provided by developer. Each *Connection* method that returns its result asynchronously accepts an optional parameter *callback*, which must be a function that accepts one string parameter. This method is then called with each output line coming from service.
- *wait()* - Waits for Sevice to finish, ignoring rest of the output it may produce.

Warning: Until output is not fully fetched from service, any attempt to start another asynchronous service will fail with exception! This constraint is set by Firebird Service API.

You may check the status of asynchronous Services using *Connection.fetching* attribute or *Connection.isrunning()* method.

In cases when you're not interested in output produced by Service, call *wait()* to wait for service to complete.

Examples:

```
import fdb
svc = fdb.services.connect(password='masterkey')

print "Fetch materialized"
print "=================="
print "Start backup"
svc.backup('employee', 'employee.fbk')
print "svc.fetching is", svc.fetching
print "svc.running is", svc.isrunning()
report = svc.readlines()
print "%i lines returned" % len(report)
print "First 5 lines from output:"
for i in xrange(5):
   print i,report[i]
print "svc.fetching is", svc.fetching
print "svc.running is", svc.isrunning()
print
print "Iterate over result"
print "==================="
svc.backup('employee', 'employee.fbk')
output = []
for line in svc:
   output.append(line)
print "%i lines returned" % len(output)
print "Last 5 lines from output:"
for line in output[-5:]:
   print line
print
print "Callback"
print "========"

output = []

# Callback function
def fetchline(line):
   output.append(line)
```

(continues on next page)

(continued from previous page)

```
svc.backup('employee', 'employee.fbk', callback=fetchline)
print "%i lines returned" % len(output)
print "Last 5 lines from output:"
for line in output[-5:]:
   print line
```

Output:

```
Fetch materialized
==================
Start backup
svc.fetching is True
svc.running is True
558 lines returned
First 5 lines from output:
0 gbak:readied database employee for backup
1 gbak:creating file employee.fbk
2 gbak:starting transaction
3 gbak:database employee has a page size of 4096 bytes.
4 gbak:writing domains
svc.fetching is False
svc.running is False

Iterate over result
===================
558 lines returned
Last 5 lines from output:
gbak:writing referential constraints
gbak:writing check constraints
gbak:writing SQL roles
gbak:writing names mapping
gbak:closing file, committing, and finishing. 74752 bytes written

Callback
========
558 lines returned
Last 5 lines from output:
gbak:writing referential constraints
gbak:writing check constraints
gbak:writing SQL roles
gbak:writing names mapping
gbak:closing file, committing, and finishing. 74752 bytes written
```

1.2.8 Working with database schema

Description of database objects like tables, views, stored procedures, triggers or UDF functions that represent database schema is stored in set of system tables present in every database. Firebird users can query these tables to get information about these objects and their relations. But querying system tables is inconvenient, as it requires good knowledge how this information is structured and requires significant amount of Python code. Changes in system tables between Firebird versions further add to this complexity. Hence FDB provides set of classes (isolated in separate module *fdb.schema*) that transform information stored in system tables into set of Python objects that surface the vital information in meaningful way, and additionaly provide set of methods for most commonly used operations or checks.

Database schema could be accessed in three different ways, each suitable for different use case:

- By direct creation of *fdb.schema.Schema* instances that are then *binded* to particular `Connection` instance. This method is best if you want to work with schema only occasionally, or you want to keep connections as lightweight as possible.
- Accessing *fdb.Connection.schema* property. This method is more convenient than previous one, and represents a compromise between convenience and resource consumption because `Schema` instance is not created until first reference and is managed by connection itself. Individual metadata objects are not loaded from system tables until first reference.
- Using *ConnectionWithSchema* instead `Connection` by specifying *connection_class=ConnectionWithSchema* parameter to *connect()* or *create_database()*. This *Connection* descendant loads all database metadata immediately and provides directly all attributes and methods provided by *Schema* class. This method is most suitable in case you want to work with database metadata extensively.

Examples:

1. Using Schema instance:

```
>>> import fdb
>>> con = fdb.connect(dsn='employee',user='sysdba', password='masterkey')
>>> schema = fdb.schema.Schema()
>>> schema.bind(con)
>>> [t.name for t in schema.tables]
['COUNTRY', 'JOB', 'DEPARTMENT', 'EMPLOYEE', 'SALES', 'PROJECT', 'EMPLOYEE_PROJECT',
↪'PROJ_DEPT_BUDGET',
 'SALARY_HISTORY', 'CUSTOMER']
```

2. Using Connection.schema:

```
>>> import fdb
>>> con = fdb.connect(dsn='employee',user='sysdba', password='masterkey')
>>> [t.name for t in con.schema.tables]
['COUNTRY', 'JOB', 'DEPARTMENT', 'EMPLOYEE', 'SALES', 'PROJECT', 'EMPLOYEE_PROJECT',
↪'PROJ_DEPT_BUDGET',
 'SALARY_HISTORY', 'CUSTOMER']
```

3. Using ConnectionWithSchema:

```
>>> import fdb
>>> con = fdb.connect(dsn='employee',user='sysdba', password='masterkey',
    connection_class=fdb.ConnectionWithSchema)
>>> [t.name for t in con.tables]
['COUNTRY', 'JOB', 'DEPARTMENT', 'EMPLOYEE', 'SALES', 'PROJECT', 'EMPLOYEE_PROJECT',
↪'PROJ_DEPT_BUDGET',
 'SALARY_HISTORY', 'CUSTOMER']
```

Note: Individual metadata information (i.e. information about *domains*, *tables* etc.) is loaded on first access and cached for further reference until it's *clared* or *reload* is requested.

Because once loaded information is cached, it's good to *clar* it when it's no longer needed to conserve memory.

Available information

The *Schema* provides information about:

- **Database:** *Owner name*, *default character set*, *description*, *security class*, *nbackup backup history* and whether database consist from *single or multiple files*.

- **Facilities:** Available *character sets*, *collations*, BLOB *filters*, database *files* and *shadows*.
- **User database objects:** *exceptions*, *generators*, *domains*, *tables* and their *constraints*, *indices*, *views*, *triggers*, *procedures*, user *roles*, *user defined functions* and *packages*.
- **System database objects:** *generators*, *domains*, *tables* and their constraints, *indices*, *views*, *triggers*, *procedures*, *functions* and *backup history*.
- **Relations between objects:** Through direct links between metadata objects and *dependencies*.
- **Privileges:** *All* privileges, or privileges granted for specific *table*, *table column*, *view*, *view column*, *procedure* or *role*. It's also possible to get all privileges *granted to* specific user, role, procedure, trigger or view.

Metadata objects

Schema information is presented as Python objects of various classes with common parent class *BaseSchemaItem* (except *Schema* itself), that defines several common attributes and methods:

Attributes:

- *name*: Name of database object or None if object doesn't have a name.
- *description*: Description (documentation text) for object or None if object doesn't have a description.
- *actions*: List of supported SQL operations on schema object instance.

Methods:

- `accept_visitor()`: *Visitor Pattern support*.
- *issystemobject()*: Returns True if this database object is system object.
- *get_quoted_name()*: Returns quoted (if necessary) name of database object.
- *get_dependents()*: Returns list of all database objects that *depend* on this one.
- *get_dependencies()*: Returns list of database objects that this object *depend* on.
- *get_sql_for()*: Returns *SQL command string* for specified action on database object.

There are next schema objects: *Collation*, *CharacterSet*, *DatabaseException*, *Sequence* (Generator), *Domain*, *Index*, *Table*, *TableColumn*, *Constraint*, *View*, *ViewColumn*, *Trigger*, *Procedure*, *ProcedureParameter*, *Function*, *FunctionArgument*, *Role*, *Dependency*, *DatabaseFile*, *Shadow*, *Package*, *Filter*, *BackupHistory* and *Privilege*.

Visitor Pattern support

Changed in version 2.0: Class *fdb.schema.SchemaVisitor* was replaced with *fdb.utils.Visitor* class.

Visitor Pattern is particularly useful when you need to process various objects that need special handling in common algorithm (for example display information about them or generate SQL commands to create them in new database). Each metadata objects (including *Schema*) descend from *Visitable* class and thus support *accept()* method that calls visitor's *visit()* method. This method dispatch calls to specific class-handling method or *fdb.utils.Visitor.default_action()* if there is no such special class-handling method defined in your visitor class. Special class-handling methods must have a name that follows *visit_<class_name>* pattern, for example method that should handle *Table* (or its descendants) objects must be named as *visit_Table*.

Next code uses visitor pattern to print all DROP SQL statements necessary to drop database object, taking its dependencies into account, i.e. it could be necessary to first drop other - dependant objects before it could be dropped.

```
import fdb
# Object dropper
class ObjectDropper(fdb.utils.Visitor):
    def __init__(self):
        self.seen = []
    def drop(self, obj):
        self.seen = []
        obj.accept(self) # You can call self.visit(obj) directly here as well
    def default_action(self, obj):
        if not obj.issystemobject() and 'drop' in obj.actions:
           for dependency in obj.get_dependents():
                d = dependency.dependent
                if d and d not in self.seen:
                    d.accept(self)
           if obj not in self.seen:
                print obj.get_sql_for('drop')
                self.seen.append(obj)
    def visit_TableColumn(self, column):
        column.table.accept(self)
    def visit_ViewColumn(self, column):
        column.view.accept(self)
    def visit_ProcedureParameter(self, param):
        param.procedure.accept(self)
    def visit_FunctionArgument(self, arg):
        arg.function.accept(self)

# Sample use:

con = fdb.connect(dsn='employee',user='sysdba', password='masterkey')
table = con.schema.get_table('JOB')
dropper = ObjectDropper()
dropper.drop(table)
```

Will produce next result:

```
DROP PROCEDURE ALL_LANGS
DROP PROCEDURE SHOW_LANGS
DROP TABLE JOB
```

Object dependencies

Close relations between metadata object like *ownership* (Table vs. TableColumn, Index or Trigger) or *cooperation* (like FK Index vs. partner UQ/PK Index) are defined directly using properties of particular schema objects. Besides close releations Firebird also uses *dependencies*, that describe functional dependency between otherwise independent metadata objects. For example stored procedure can call other stored procedures, define its parameters using domains or work with tables or views. Removing or changing these objects may/will cause the procedure to stop working correctly, so Firebird tracks these dependencies. FDB surfaces these dependencies as *Dependency* schema objects, and all schema objects have *get_dependents()* and *get_dependencies()* methods to get list of *Dependency* instances that describe these dependencies.

Dependency object provides names and types of dependent/depended on database objects, and access to their respective schema Python objects as well.

Enhanced list of objects

New in version 2.0.

Whenever possible, schema module uses enhanced *ObjectList* list descendant for collections of metadata objects. This enhanced list provides several convenient methods for advanced list processing:

- filtering - *filter()*, *ifilter()* and *ifilterfalse()*
- sorting - *sort()*
- extracting/splitting - *extract()* and *split()*
- testing - *contains()*, *all()* and *any()*
- reporting - *ecount()*, *report()* and *ireport()*
- fast key access - *key*, *frozen*, *freeze()* and *get()*

SQL operations

FDB doesn't allow you to change database metadata directly using schema objects. Instead it supports generation of DDL SQL commands from schema objects using *get_sql_for()* method present on all schema objects except Schema itself. DDL commands that could be generated depend on object type and context (for example it's not possible to generate all DDL commands for system database objects), and list of DDL commands that could be generated for particular schema object could be obtained from its *actions* attribute.

Possible *actions* could be: create, recreate, create_or_alter, alter, drop, activate, deactivate, recompute and declare. Some actions require/allow additional parameters.

Table 1: SQL actions

Schema class	Action	Parameter	Required	Description
Collation	create			
	drop			
	comment			
CharacterSet	alter	collation	Yes	*Collation* instance or collation name
	comment			
DatabaseException	create			
	recreate			
	alter	message	Yes	string.
	create_or_alter			
	drop			
	comment			
Sequence	create			
	alter	value	Yes	integer
	drop			
	comment			
Domain	create			
	alter			One from next parameters required
		name	No	string
		default	No	string definition or None to drop default
		check	No	string definition or None to drop check
		datatype	No	string SQL datatype definition
	drop			
	comment			

Continued on next page

Table 1 – continued from previous page

Schema class	Action	Parameter	Required	Description
Constraint	create			
	drop			
Index	create			
	activate			
	deactivate			
	recompute			
	drop			
	comment			
Table	create			
		no_pk	No	Do not generate PK constraint
		no_unique	No	Do not generate unique constraints
	recreate			
		no_pk	No	Do not generate PK constraint
		no_unique	No	Do not generate unique constraints
	drop			
	comment			
TableColumn	alter			One from next parameters required
		name	No	string
		datatype	No	string SQL type definition
		position	No	integer
		expression	No	string with COMPUTED BY expression
		restart	No	None or initial value
	drop			
	comment			
View	create			
	recreate			
	alter	columns	No	string or list of strings
		query	Yes	string
		check	No	True for WITH CHECK OPTION clause
	create_or_alter			
	drop			
	comment			
Trigger	create	inactive	No	Create inactive trigger
	recreate			
	create_or_alter			
	alter			Requires parameters for either header or body definition.
		fire_on	No	string
		active	No	bool
		sequence	No	integer
		declare	No	string or list of strings
		code	No	string or list of strings
	drop			
	comment			
Procedure	create	no_code	No	True to supress procedure body from output
	recreate	no_code	No	True to supress procedure body from output
	create_or_alter	no_code	No	True to supress procedure body from output
	alter	input	No	Input parameters
		output	No	Output parameters

Continued on next page

Table 1 – continued from previous page

Schema class	Action	Parameter	Required	Description
		declare	No	Variable declarations
		code	Yes	Procedure code / body
	drop			
	comment			
Role	create			
	drop			
	comment			
Function	declare			
	drop			
	create	no_code	No	Generate PSQL function code or not
	create_or_alter	no_code	No	Generate PSQL function code or not
	recreate	no_code	No	Generate PSQL function code or not
	alter	arguments	No	Function arguments
		returns	Yes	Function return value
		declare	No	Variable declarations
		code	Yes	PSQL function body / code
	comment			
DatabaseFile	create			
Shadow	create			
	drop	preserve	No	Preserve file or not
Privilege	grant	grantors	No	List of grantor names. Generates GRANTED BY clause if grantor is not in list.
	revoke	grantors	No	List of grantor names. Generates GRANTED BY clause if grantor is not in list.
		grant_option	No	True to get REVOKE of GRANT/ADMIN OPTION only. Raises ProgrammingError if privilege doesn't have such option.
Package	create	body	No	(bool) Generate package body
	recreate	body	No	(bool) Generate package body
	create_or_alter	body	No	(bool) Generate package body
	alter	header	No	(string_or_list) Package header
	drop	body	No	(bool) Drop only package body

Examples:

```
>>> import fdb
>>> con = fdb.connect(dsn='employee',user='sysdba', password='masterkey')
>>> t = con.schema.get_table('EMPLOYEE')
>>> print t.get_sql_for('create')
CREATE TABLE EMPLOYEE
(
  EMP_NO EMPNO NOT NULL,
  FIRST_NAME "FIRSTNAME" NOT NULL,
  LAST_NAME "LASTNAME" NOT NULL,
  PHONE_EXT VARCHAR(4),
  HIRE_DATE TIMESTAMP DEFAULT 'NOW' NOT NULL,
  DEPT_NO DEPTNO NOT NULL,
  JOB_CODE JOBCODE NOT NULL,
  JOB_GRADE JOBGRADE NOT NULL,
  JOB_COUNTRY COUNTRYNAME NOT NULL,
  SALARY SALARY NOT NULL,
  FULL_NAME COMPUTED BY (last_name || ', ' || first_name),
```

(continues on next page)

(continued from previous page)

```
  PRIMARY KEY (EMP_NO)
)
>>> for i in t.indices:
...     if 'create' in i.actions:
...         print i.get_sql_for('create')
...
CREATE ASCENDING INDEX NAMEX
   ON EMPLOYEE (LAST_NAME,FIRST_NAME)
>>> for c in [x for x in t.constraints if x.ischeck() or x.isfkey()]:
...     print c.get_sql_for('create')
...
ALTER TABLE EMPLOYEE ADD FOREIGN KEY (DEPT_NO)
  REFERENCES DEPARTMENT (DEPT_NO)
ALTER TABLE EMPLOYEE ADD FOREIGN KEY (JOB_CODE,JOB_GRADE,JOB_COUNTRY)
  REFERENCES JOB (JOB_CODE,JOB_GRADE,JOB_COUNTRY)
ALTER TABLE EMPLOYEE ADD CHECK ( salary >= (SELECT min_salary FROM job WHERE
                        job.job_code = employee.job_code AND
                        job.job_grade = employee.job_grade AND
                        job.job_country = employee.job_country) AND
            salary <= (SELECT max_salary FROM job WHERE
                        job.job_code = employee.job_code AND
                        job.job_grade = employee.job_grade AND
                        job.job_country = employee.job_country))
>>> p = con.schema.get_procedure('GET_EMP_PROJ')
>>> print p.get_sql_for('recreate',no_code=True)
RECREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
END
>>> print p.get_sql_for('create_or_alter')
CREATE OR ALTER PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
     FOR SELECT proj_id
             FROM employee_project
             WHERE emp_no = :emp_no
             INTO :proj_id
     DO
             SUSPEND;
END
>>> print p.get_sql_for('alter',input=['In1 INTEGER','In2 VARCHAR(5)'],
... output='Out1 INETEGER,\nOut2 VARCHAR(10)',declare=['declare variable i integer =
↪1;'],
... code=['/* body */','Out1 = i',"Out2 = 'Value'"])
ALTER PROCEDURE GET_EMP_PROJ (
  In1 INTEGER,
  In2 VARCHAR(5)
)
RETURNS (Out1 INETEGER,
Out2 VARCHAR(10))
AS
  declare variable i integer = 1;
BEGIN
  /* body */
  Out1 = i
```

(continues on next page)

(continued from previous page)

```
  Out2 = 'Value'
END
```

1.2.9 Working with user privileges

User or database object privileges are part of database metadata accesible through FDB *Schema* support. Each discrete privilege is represented by *Privilege* instance. You can access either *all* privileges, or privileges granted for specific *table*, *table column*, *view*, *view column*, *procedure* or *role*. It's also possible to get all privileges *granted to* specific user, role, procedure, trigger or view.

Privilege class supports *get_sql_for()* method to generate GRANT and REVOKE SQL statements for given privilege. If you want to generate grant/revoke statements for set of privileges (for example all privileges granted on specific object or grated to specific user), it's more convenient to use function *get_grants()* that returns list of minimal set of SQL commands required for task.

Examples:

```
>>> import fdb
>>> con = fdb.connect(dsn='employee',user='sysdba', password='masterkey')
>>> t = con.schema.get_table('EMPLOYEE')
>>> for p in t.privileges:
...     print p.get_sql_for('grant')
...
GRANT SELECT ON EMPLOYEE TO SYSDBA WITH GRANT OPTION
GRANT INSERT ON EMPLOYEE TO SYSDBA WITH GRANT OPTION
GRANT UPDATE ON EMPLOYEE TO SYSDBA WITH GRANT OPTION
GRANT DELETE ON EMPLOYEE TO SYSDBA WITH GRANT OPTION
GRANT REFERENCES ON EMPLOYEE TO SYSDBA WITH GRANT OPTION
GRANT SELECT ON EMPLOYEE TO PUBLIC WITH GRANT OPTION
GRANT INSERT ON EMPLOYEE TO PUBLIC WITH GRANT OPTION
GRANT UPDATE ON EMPLOYEE TO PUBLIC WITH GRANT OPTION
GRANT DELETE ON EMPLOYEE TO PUBLIC WITH GRANT OPTION
GRANT REFERENCES ON EMPLOYEE TO PUBLIC WITH GRANT OPTION
>>> for p in fdb.schema.get_grants(t.privileges):
...     print p
...
GRANT DELETE, INSERT, REFERENCES, SELECT, UPDATE ON EMPLOYEE TO PUBLIC WITH GRANT␣
↪OPTION
GRANT DELETE, INSERT, REFERENCES, SELECT, UPDATE ON EMPLOYEE TO SYSDBA WITH GRANT␣
↪OPTION
```

Normally generated GRANT/REVOKE statements don't contain grantor's name, because GRANTED BY clause is supported only since Firebird 2.5. If you want to get GRANT/REVOKE statements including this clause, use *grantors* parameter for *get_sql_for* and *get_grants*. This parameter is a list of grantor names, and GRANTED BY clause is generated only for privileges not granted by user from this list. It's useful to suppress GRANTED BY clause for SYSDBA or database owner.

1.2.10 Working with monitoring tables

Starting from Firebird 2.1 (ODS 11.1) its possible to monitor server-side activity happening inside a particular database. The engine offers a set of "virtual" tables (so-called "monitoring tables") that provides the user with a snapshot of the current activity within the given database. FDB provides access to this information through set of classes (isolated in separate module *fdb.monitor*) that transform information stored in monitoring tables into set

of Python objects that surface the information in meaningful way, and additionaly provide set of methods for available operations or checks.

Like database schema, monitoring tables could be accessed in two different ways, each suitable for different use case:

- By direct creation of *fdb.monitor.Monitor* instances that are then binded to particular *Connection* instance. This method is best if you want to work with monitoring data only occasionally, or you want to keep connections as lightweight as possible.
- Accessing *fdb.Connection.monitor* property. This method is more convenient than previous one, and represents a compromise between convenience and resource consumption because *Monitor* instance is not created until first reference and is managed by connection itself.

Examples:

1. Using Monitor instance:

```
>>> import fdb
>>> con = fdb.connect(dsn='employee',user='sysdba', password='masterkey')
>>> monitor = fdb.monitor.Monitor()
>>> monitor.bind(con)
>>> monitor.db.name
'/opt/firebird/examples/empbuild/employee.fdb'
```

2. Using Connection.monitor:

```
>>> import fdb
>>> con = fdb.connect(dsn='employee',user='sysdba', password='masterkey')
>>> con.monitor.db.name
'/opt/firebird/examples/empbuild/employee.fdb'
```

Available information

The *Monitor* provides information about:

- *Database*.
- *Connections* to database and *current* connection.
- *Transactions*.
- Executed *SQL statements*.
- PSQL *callstack*.
- Page and row *I/O statistics*, including memory usage.
- *Context variables*.

Activity snapshot

The key term of the monitoring feature is an *activity snapshot*. It represents the current state of the database, comprising a variety of information about the database itself, active attachments and users, transactions, prepared and running statements, and more.

A snapshot is created the first time any of the monitoring information is being accessed from in the given *Monitor* instance and it is preserved until *closed*, *clared* or *refreshed*, in order that accessed information is always consistent.

There are two ways to refresh the snapshot:

1. Call *Monitor.clear()* method. New snapshot will be taken on next access to monitoring information.
2. Call *Monitor.refresh()* method to take the new snapshot immediately.

Important: In both cases, any instances of information objects your application may hold would be obsolete. Using them may result in error, or (more likely) provide outdated information.

Note: Individual monitoring information (i.e. information about *connections*, *transactions* etc.) is loaded from activity snapshot on first access and cached for further reference until it's *clared* or snapshot is *refreshed*.

Because once loaded information is cached, it's good to *clear* it when it's no longer needed to conserve memory.

1.2.11 Driver hooks

New in version 2.0.

FDB provides internal notification mechanism that allows installation of custom *hooks* into certain driver tasks. This mechanism consists from next functions:

- *fdb.add_hook()* that instals hook function for specified *hook_type*.
- *fdb.remove_hook()* that uninstalls previously installed hook function for specified *hook_type*.
- *fdb.get_hooks()* that returns list of installed hook routines for specified *hook_type*.

FDB provides next *hook types* (exposed as constants in *fdb* namespace):

fdb.**HOOK_API_LOADED**
: This hook is invoked once when instance of *fbclient_API* is created. It could be used for additional initialization tasks that require Firebird API.

  Hook routine must have signature: *hook_func(api)*. Any value returned by hook is ignored.

fdb.**HOOK_DATABASE_ATTACHED**
: This hook is invoked just before *Connection* (or subclass) instance is returned to the client application.

  Hook routine must have signature: *hook_func(connection)*. Any value returned by hook is ignored.

fdb.**HOOK_DATABASE_ATTACH_REQUEST**
: This hook is invoked after all parameters are preprocessed and before *Connection* is created.

  Hook routine must have signature: *hook_func(dsn, dpb)* where *dpb* is *ParameterBuffer* instance. It may return *Connection* (or subclass) instance or None. First instance returned by any hook of this type will become the return value of caller function and other hooks of the same type are not invoked.

fdb.**HOOK_DATABASE_DETACH_REQUEST**
: This hook is invoked before connection is closed.

  Hook must have signature: *hook_func(connection)*. If any hook function returns True, connection is not closed.

fdb.**HOOK_DATABASE_CLOSED**
: This hook is invoked after connection is closed.

  Hook must have signature: *hook_func(connection)*. Any value returned by hook is ignored.

fdb.**HOOK_SERVICE_ATTACHED**
: This hook is invoked before *fdb.services.Connection* instance is returned.

  Hook must have signature: hook_func(connection). Any value returned by hook is ignored.

Installed hook functions are invoked by next fdb code.

fdb.load_api() hooks:

- Event *HOOK_API_LOADED*

fdb.connect() hooks:

- Event *HOOK_DATABASE_ATTACH_REQUEST*
- Event *HOOK_DATABASE_ATTACHED*

fdb.create_database() hooks:

- Event *HOOK_DATABASE_ATTACHED*

fdb.Connection.close() hooks:

- Event *HOOK_DATABASE_DETACH_REQUEST*
- Event *HOOK_DATABASE_CLOSED*

fdb.services.connect() hooks:

- Event *HOOK_SERVICE_ATTACHED*

1.3 Compliance to PyDB API 2.0

Full text of Python Database API 2.0 (PEP 249) is available at http://www.python.org/dev/peps/pep-0249/

1.3.1 Unsupported Optional Features

`fdb.Cursor.nextset()`

This method is not implemented because the database engine does not support opening multiple result sets simultaneously with a single cursor.

1.3.2 Supported Optional Features

- *Connection.Error*, *Connection.ProgrammingError*, etc.

  All exception classes defined by the DB API standard are exposed on the Connection objects as attributes (in addition to being available at module scope).

- *Cursor.connection*

  This read-only attribute return a reference to the Connection object on which the cursor was created.

1.3.3 Nominally Supported Optional Features

fdb.Cursor

arraysize

As required by the spec, the value of this attribute is observed with respect to the *fetchmany* method. However, changing the value of this attribute does not make any difference in fetch efficiency because the database engine only supports fetching a single row at a time.

setinputsizes()

Although this method is present, it does nothing, as allowed by the spec.

setoutputsize()

Although this method is present, it does nothing, as allowed by the spec.

1.3.4 Extensions and Caveats

FDB offers a large feature set beyond the minimal requirements of the Python DB API. This section attempts to document only those features that overlap with the DB API, or are too insignificant to warrant their own subsection elsewhere.

fdb.connect()

This function supports the following optional keyword arguments in addition to those required by the spec:

role For connecting to a database with a specific SQL role.

Example:

```
fdb.connect(dsn='host:/path/database.db', user='limited_user',
   password='pass', role='MORE_POWERFUL_ROLE')
```

charset For explicitly specifying the character set of the connection. See Firebird Documentation for a list of available character sets, and *Unicode Fields and FDB* section for information on handling extended character sets with FDB.

Example:

```
fdb.connect(dsn='host:/path/database.db', user='sysdba',
    password='pass', charset='UTF8')
```

sql_dialect The SQL dialect is feature for backward compatibility with Interbase® 5.5 or earlier. The default dialect is *3* (the most featureful dialect, default for Firebird). If you want to connect to *legacy* databases, you must explicitly set this argument's value to *1*. Dialect *2* is a transitional dialect that is normally used only during ports from IB < 6 to IB >= 6 or Firebird. See Firebird documentation for more information about SQL Dialects.

Example:

```
fdb.connect(dsn='host:/path/database.db', user='sysdba',
   password='pass', dialect=1)
```

fdb.Connection

charset

(read-only) The character set of the connection (set via the *charset* parameter of *fdb.connect()*). See Firebird Documentation for a list of available character sets, and *Unicode Fields and FDB* section for information on handling extended character sets with FDB.

sql_dialect

This integer attribute indicates which SQL dialect the connection is using. You should not change a connection's dialect; instead, discard the connection and establish a new one with the desired dialect. For more information, see the documentation of the *sql_dialect* argument of the *connect* function.

server_version

(read-only) The version string of the database server to which this connection is connected.

execute_immediate()

Executes a statement without caching its prepared form. The statement must *not* be of a type that returns a result set. In most cases (especially cases in which the same statement – perhaps a parameterized statement – is executed repeatedly), it is better to create a cursor using the connection's *cursor* method, then execute the statement using one of the cursor's execute methods.

Arguments:

sql String containing the SQL statement to execute.

`commit(retaining=False)() rollback(retaining=False, savepoint=None)()`

The *commit* and *rollback* methods accept an optional boolean parameter *retaining* (default *False*) that indicates whether the transactional context of the transaction being resolved should be recycled. For details, see the Advanced Transaction Control: Retaining Operations section of this document. The *rollback* method accepts an optional string parameter *savepoint* that causes the transaction to roll back only as far as the designated savepoint, rather than rolling back entirely. For details, see the Advanced Transaction Control: Savepoints section of this document.

fdb.Cursor

description

FDB makes absolutely no guarantees about *description* except those required by the Python Database API Specification 2.0 (that is, *description* is either *None* or a sequence of 7-element sequences). Therefore, client programmers should *not* rely on *description* being an instance of a particular class or type. FDB provides several named positional constants to be used as indices into a given element of *description* . The contents of all *description* elements are defined by the DB API spec; these constants are provided merely for convenience.

```
DESCRIPTION_NAME
DESCRIPTION_TYPE_CODE
DESCRIPTION_DISPLAY_SIZE
DESCRIPTION_INTERNAL_SIZE
DESCRIPTION_PRECISION
DESCRIPTION_SCALE
DESCRIPTION_NULL_OK
```

Here is an example of accessing the *name* of the first field in the *description* of cursor *cur*:

```
nameOfFirstField = cur.description[0][fdb.DESCRIPTION_NAME]
```

For more information, see the documentation of Cursor.description in the DB API Specification.

rowcount

Although FDB's *Cursor's implement this attribute, the database engine's own support for the determination of "rows affected"/"rows selected" is quirky. The database engine only supports the determination of rowcount for 'INSERT*, *UPDATE*, *DELETE*, and *SELECT* statements. When stored procedures become involved, row count figures are usually not available to the client. Determining rowcount for *SELECT* statements is problematic: the rowcount is reported as zero until at least one row has been fetched from the result set, and the rowcount is misreported if the result set is larger than 1302 rows. The server apparently marshals result sets internally in batches of 1302, and will misreport the rowcount for result sets larger than

1302 rows until the 1303rd row is fetched, result sets larger than 2604 rows until the 2605th row is fetched, and so on, in increments of 1302. As required by the Python DB API Spec, the rowcount attribute "is -1 in case no executeXX() has been performed on the cursor or the rowcount of the last operation is not determinable by the interface".

fetchone()

fetchmany()

fetchall()

FDB makes absolutely no guarantees about the return value of the *fetchone* / *fetchmany* / *fetchall* methods except that it is a sequence indexed by field position. FDB makes absolutely no guarantees about the return value of the *fetchonemap* / *fetchmanymap* / *fetchallmap* methods (documented below) except that it is a mapping of field name to field value. Therefore, client programmers should *not* rely on the return value being an instance of a particular class or type.

1.4 FDB Reference

1.4.1 Main driver namespace

Constants

__version__ Current driver version, string.

PyDB API 2.0 globals

apilevel String constant stating the supported DB API level (2.0).

threadsafety Integer constant stating the level of thread safety the interface supports. Curretly *1* = Threads may share the module, but not connections.

paramstyle String constant stating the type of parameter marker formatting expected by the interface. *'qmark'* = Question mark style, e.g. '...WHERE name=?'

Constants for work with driver hooks

- HOOK_API_LOADED
- HOOK_DATABASE_ATTACHED
- HOOK_DATABASE_ATTACH_REQUEST
- HOOK_DATABASE_DETACH_REQUEST
- HOOK_DATABASE_CLOSED
- HOOK_SERVICE_ATTACHED

Helper constants for work with `Cursor.description` content

- DESCRIPTION_NAME
- DESCRIPTION_TYPE_CODE

- DESCRIPTION_DISPLAY_SIZE
- DESCRIPTION_INTERNAL_SIZE
- DESCRIPTION_PRECISION
- DESCRIPTION_SCALE
- DESCRIPTION_NULL_OK

Helper Transaction Parameter Block (TPB) constants

ISOLATION_LEVEL_READ_COMMITED_LEGACY R/W WAIT READ COMMITTED NO RECORD VERSION transaction.

ISOLATION_LEVEL_READ_COMMITED R/W WAIT READ COMMITTED RECORD VERSION transaction.

ISOLATION_LEVEL_REPEATABLE_READ Same as ISOLATION_LEVEL_SNAPSHOT.

ISOLATION_LEVEL_SNAPSHOT R/W WAIT SNAPSHOT transaction.

ISOLATION_LEVEL_SERIALIZABLE R/W WAIT SERIALIZABLE transaction.

ISOLATION_LEVEL_SNAPSHOT_TABLE_STABILITY Same as ISOLATION_LEVEL_SERIALIZABLE.

ISOLATION_LEVEL_READ_COMMITED_RO R/O WAIT READ COMMITTED RECORD VERSION transaction.

MAX_BLOB_SEGMENT_SIZE

charset_map Python dictionary that maps Firebird character set names (key) to Python character sets (value).

ODS version numbers introduced by Firebird engine version

- ODS_FB_20
- ODS_FB_21
- ODS_FB_25
- ODS_FB_30

Translation dictionaries

IMPLEMENTATION_NAMES Dictionary to map Implementation codes to names

PROVIDER_NAMES Dictionary to map provider codes to names

DB_CLASS_NAMES Dictionary to map database class codes to names

Firebird API constants and globals

- frb_info_att_charset
- isc_dpb_activate_shadow
- isc_dpb_address_path

- isc_dpb_allocation
- isc_dpb_begin_log
- isc_dpb_buffer_length
- isc_dpb_cache_manager
- isc_dpb_cdd_pathname
- isc_dpb_connect_timeout
- isc_dpb_damaged
- isc_dpb_dbkey_scope
- isc_dpb_debug
- isc_dpb_delete_shadow
- isc_dpb_dummy_packet_interval
- isc_dpb_encrypt_key
- isc_dpb_force_write
- isc_dpb_garbage_collect
- isc_dpb_gbak_attach
- isc_dpb_gfix_attach
- isc_dpb_gsec_attach
- isc_dpb_gstat_attach
- isc_dpb_interp
- isc_dpb_lc_ctype
- isc_dpb_lc_messages
- isc_dpb_no_garbage_collect
- isc_dpb_no_reserve
- isc_dpb_num_buffers
- isc_dpb_number_of_users
- isc_dpb_old_dump_id
- isc_dpb_old_file
- isc_dpb_old_file_size
- isc_dpb_old_num_files
- isc_dpb_old_start_file
- isc_dpb_old_start_page
- isc_dpb_old_start_seqno
- isc_dpb_online
- isc_dpb_online_dump
- isc_dpb_overwrite
- isc_dpb_page_size
- isc_dpb_password
- isc_dpb_password_enc
- isc_dpb_reserved
- isc_dpb_sec_attach
- isc_dpb_set_db_charset
- isc_dpb_set_db_readonly
- isc_dpb_set_db_sql_dialect
- isc_dpb_set_page_buffers
- isc_dpb_shutdown
- isc_dpb_shutdown_delay
- isc_dpb_sql_dialect
- isc_dpb_sql_role_name
- isc_dpb_sweep
- isc_dpb_sweep_interval
- isc_dpb_sys_user_name
- isc_dpb_sys_user_name_enc
- isc_dpb_trace
- isc_dpb_user_name

- isc_dpb_verify
- isc_dpb_version1
- isc_dpb_working_directory
- isc_dpb_no_db_triggers
- isc_dpb_nolinger,
- isc_info_active_tran_count
- isc_info_end
- isc_info_truncated
- isc_info_sql_stmt_type
- isc_info_sql_get_plan
- isc_info_sql_records
- isc_info_req_select_count
- isc_info_req_insert_count
- isc_info_req_update_count
- isc_info_req_delete_count
- isc_info_blob_total_length
- isc_info_blob_max_segment
- isc_info_blob_type
- isc_info_blob_num_segments
- fb_info_page_contents
- isc_info_active_transactions
- isc_info_allocation
- isc_info_attachment_id
- isc_info_backout_count
- isc_info_base_level
- isc_info_bpage_errors
- isc_info_creation_date
- isc_info_current_memory
- isc_info_db_class
- isc_info_db_id
- isc_info_db_provider
- isc_info_db_read_only
- isc_info_db_size_in_pages
- isc_info_db_sql_dialect
- isc_info_delete_count
- isc_info_dpage_errors
- isc_info_expunge_count
- isc_info_fetches
- isc_info_firebird_version
- isc_info_forced_writes
- isc_info_implementation
- isc_info_insert_count
- isc_info_ipage_errors
- isc_info_isc_version
- isc_info_limbo
- isc_info_marks
- isc_info_max_memory
- isc_info_next_transaction
- isc_info_no_reserve
- isc_info_num_buffers
- isc_info_ods_minor_version
- isc_info_ods_version
- isc_info_oldest_active
- isc_info_oldest_snapshot

- isc_info_oldest_transaction
- isc_info_page_errors
- isc_info_page_size
- isc_info_ppage_errors
- isc_info_purge_count
- isc_info_read_idx_count
- isc_info_read_seq_count
- isc_info_reads
- isc_info_record_errors
- isc_info_set_page_buffers
- isc_info_sql_stmt_commit
- isc_info_sql_stmt_ddl
- isc_info_sql_stmt_delete
- isc_info_sql_stmt_exec_procedure
- isc_info_sql_stmt_get_segment
- isc_info_sql_stmt_insert
- isc_info_sql_stmt_put_segment
- isc_info_sql_stmt_rollback
- isc_info_sql_stmt_savepoint
- isc_info_sql_stmt_select
- isc_info_sql_stmt_select_for_upd
- isc_info_sql_stmt_set_generator
- isc_info_sql_stmt_start_trans
- isc_info_sql_stmt_update
- isc_info_sweep_interval
- isc_info_tpage_errors
- isc_info_tra_access
- isc_info_tra_concurrency
- isc_info_tra_consistency
- isc_info_tra_id
- isc_info_tra_isolation
- isc_info_tra_lock_timeout
- isc_info_tra_no_rec_version
- isc_info_tra_oldest_active
- isc_info_tra_oldest_interesting
- isc_info_tra_oldest_snapshot
- isc_info_tra_read_committed
- isc_info_tra_readonly
- isc_info_tra_readwrite
- isc_info_tra_rec_version
- fb_info_tra_dbpath
- isc_info_update_count
- isc_info_user_names
- isc_info_version
- isc_info_writes
- isc_tpb_autocommit
- isc_dpb_version2
- fb_info_implementation
- fb_info_page_warns
- fb_info_record_warns
- fb_info_bpage_warns
- fb_info_dpage_warns
- fb_info_ipage_warns
- fb_info_ppage_warns

- fb_info_tpage_warns
- fb_info_pip_errors
- fb_info_pip_warns
- isc_tpb_commit_time
- isc_tpb_concurrency
- isc_tpb_consistency
- isc_tpb_exclusive
- isc_tpb_ignore_limbo
- isc_tpb_lock_read
- isc_tpb_lock_timeout
- isc_tpb_lock_write
- isc_tpb_no_auto_undo
- isc_tpb_no_rec_version
- isc_tpb_nowait
- isc_tpb_protected
- isc_tpb_read
- isc_tpb_read_committed
- isc_tpb_rec_version
- isc_tpb_restart_requests
- isc_tpb_shared
- isc_tpb_verb_time
- isc_tpb_version3
- isc_tpb_wait
- isc_tpb_write
- charset_map
- XSQLDA_PTR
- ISC_SHORT
- ISC_LONG
- ISC_SCHAR
- ISC_UCHAR
- ISC_QUAD
- ISC_DATE
- ISC_TIME
- SHRT_MIN
- SHRT_MAX
- USHRT_MAX
- INT_MIN
- INT_MAX
- LONG_MIN
- LONG_MAX
- SQL_TEXT
- SQL_VARYING
- SQL_SHORT
- SQL_LONG
- SQL_FLOAT
- SQL_DOUBLE
- SQL_D_FLOAT
- SQL_TIMESTAMP
- SQL_BLOB
- SQL_ARRAY
- SQL_QUAD
- SQL_TYPE_TIME
- SQL_TYPE_DATE
- SQL_INT64

- SQL_BOOLEAN
- SUBTYPE_NUMERIC
- SUBTYPE_DECIMAL
- MAX_BLOB_SEGMENT_SIZE
- ISC_INT64
- XSQLVAR
- ISC_TEB
- RESULT_VECTOR
- ISC_STATUS
- ISC_STATUS_ARRAY
- ISC_STATUS_PTR
- ISC_EVENT_CALLBACK
- ISC_ARRAY_DESC
- blr_varying
- blr_varying2
- blr_text
- blr_text2
- blr_short
- blr_long
- blr_int64
- blr_float
- blr_d_float
- blr_double
- blr_timestamp
- blr_sql_date
- blr_sql_time
- blr_cstring
- blr_quad
- blr_blob
- blr_bool
- SQLDA_version1
- isc_segment
- isc_db_handle
- isc_tr_handle
- isc_stmt_handle
- isc_blob_handle
- sys_encoding

Exceptions

exception fdb.**Error**
Bases: exceptions.Exception

Exception that is the base class of all other error exceptions. You can use this to catch all errors with one single 'except' statement. Warnings are not considered errors and thus should not use this class as base.

exception fdb.**InterfaceError**
Bases: fdb.fbcore.Error

Exception raised for errors that are related to the database interface rather than the database itself.

exception fdb.**DatabaseError**
Bases: fdb.fbcore.Error

Exception raised for errors that are related to the database.

exception fdb.**DataError**
Bases: fdb.fbcore.DatabaseError

Exception raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc.

exception fdb.**OperationalError**
Bases: fdb.fbcore.DatabaseError

Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc.

exception fdb.**IntegrityError**
Bases: fdb.fbcore.DatabaseError

Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails.

exception fdb.**InternalError**
Bases: fdb.fbcore.DatabaseError

Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc.

exception fdb.**ProgrammingError**
Bases: fdb.fbcore.DatabaseError

Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc.

exception fdb.**NotSupportedError**
Bases: fdb.fbcore.DatabaseError

Exception raised in case a method or database API was used which is not supported by the database

exception fdb.**TransactionConflict**
Bases: fdb.fbcore.DatabaseError

exception fdb.**ParseError**
Bases: exceptions.Exception

This is the exception inheritance layout:

```
StandardError
|__Warning
|__Error
   |__InterfaceError
   |__ParseError
   |__DatabaseError
      |__DataError
      |__OperationalError
      |__IntegrityError
      |__InternalError
      |__ProgrammingError
      |__NotSupportedError
```

Functions

connect

fdb.**connect**(*dsn=''*, *user=None*, *password=None*, *host=None*, *port=None*, *database=None*, *sql_dialect=3*, *role=None*, *charset=None*, *buffers=None*, *force_write=None*, *no_reserve=None*, *db_key_scope=None*, *isolation_level='\x03\t\x06\x0f\x11'*, *connection_class=None*, *fb_library_name=None*, *no_gc=None*, *no_db_triggers=None*, *no_linger=None*)

Establish a connection to database.

Parameters

- **dsn** – Connection string in format [host[/port]]:database
- **user** (*string*) – User name. If not specified, fdb attempts to use ISC_USER envar.
- **password** (*string*) – User password. If not specified, fdb attempts to use ISC_PASSWORD envar.
- **host** (*string*) – Server host machine specification.
- **port** (*integer*) – Port used by Firebird server.
- **database** (*string*) – Database specification (file spec. or alias)
- **sql_dialect** – SQL Dialect for connection.
- **role** (*string*) – User role.
- **charset** (*string*) – Character set for connection.
- **buffers** (*integer*) – Page case size override for connection.
- **force_writes** (*integer*) – Forced writes override for connection.
- **no_reserve** (*integer*) – Page space reservation override for connection.
- **db_key_scope** (*integer*) – DBKEY scope override for connection.
- **isolation_level** (*0, 1, 2 or 3*) – Default transaction isolation level for connection **(not used)**.
- **connection_class** (subclass of *Connection*) – Custom connection class
- **fb_library_name** (*string*) – Full path to Firebird client library. See *load_api()* for details.
- **no_gc** (*integer*) – No Garbage Collection flag.
- **no_db_triggers** (*integer*) – No database triggers flag (FB 2.1).
- **no_linger** (*integer*) – No linger flag (FB3).

Returns Connection to database.

Return type *Connection* instance.

Raises

- ***ProgrammingError*** – For bad parameter values.
- ***DatabaseError*** – When connection cannot be established.

Important: You may specify the database using either *dns* or *database* (with optional *host*), but not both.

Examples:

```
con = fdb.connect(dsn='host:/path/database.fdb', user='sysdba',
                  password='pass', charset='UTF8')
con = fdb.connect(host='myhost', database='/path/database.fdb',
                  user='sysdba', password='pass', charset='UTF8')
```

Hooks:

Event *HOOK_DATABASE_ATTACH_REQUEST*: Executed after all parameters are preprocessed and before *Connection* is created. Hook must have signature: hook_func(dsn, dpb) where *dpb* is *ParameterBuffer* instance.

Hook may return *Connection* (or subclass) instance or None. First instance returned by any hook will become the return value of this function and other hooks are not called.

Event *HOOK_DATABASE_ATTACHED*: Executed before *Connection* (or subclass) instance is returned. Hook must have signature: hook_func(connection). Any value returned by hook is ignored.

create_database

fdb.**create_database**(*sql=''*, *sql_dialect=3*, *dsn=''*, *user=None*, *password=None*, *host=None*, *port=None*, *database=None*, *page_size=None*, *length=None*, *charset=None*, *files=None*, *connection_class=None*, *fb_library_name=None*)

Creates a new database. Parameters could be specified either by supplied "CREATE DATABASE" statement, or set of database parameters.

Parameters

- **sql** – "CREATE DATABASE" statement.
- **sql_dialect** (*1 or 3*) – SQL Dialect for newly created database.
- **dsn** – Connection string in format [host[/port]]:database
- **user** (*string*) – User name. If not specified, fdb attempts to use ISC_USER envar.
- **password** (*string*) – User password. If not specified, fdb attempts to use ISC_PASSWORD envar.
- **host** (*string*) – Server host machine specification.
- **port** (*integer*) – Port used by Firebird server.
- **database** (*string*) – Database specification (file spec. or alias)
- **page_size** (*integer*) – Database page size.
- **length** (*integer*) – Database size in pages.
- **charset** (*string*) – Character set for connection.
- **files** (*string*) – Specification of secondary database files.
- **connection_class** (subclass of *Connection*) – Custom connection class
- **fb_library_name** (*string*) – Full path to Firebird client library. See *load_api()* for details.

Returns Connection to the newly created database.

Return type *Connection* instance.

Raises

- ***ProgrammingError*** – For bad parameter values.
- ***DatabaseError*** – When database creation fails.

Example:

```
con = fdb.create_database("create database '/temp/db.fdb' user 'sysdba' password
↪'pass'")
con = fdb.create_database(dsn='/temp/db.fdb',user='sysdba',password='pass',page_
↪size=8192)
```

Hooks:

Event' HOOK_DATABASE_ATTACHED': Executed before *Connection* (or subclass) instance is returned. Hook must have signature: hook_func(connection). Any value returned by hook is ignored.

load_api

fdb.**load_api**(*fb_library_name=None*)

Initializes bindings to Firebird Client Library unless they are already initialized. Called automatically by *fdb.connect()* and *fdb.create_database()*.

Parameters **fb_library_name** (*string*) – (optional) Path to Firebird Client Library. When it's not specified, FDB does its best to locate appropriate client library.

Returns *fdb.ibase.fbclient_API* instance.

Hooks:

Event HOOK_API_LOADED: Executed after api is initialized. Hook routine must have signature: hook_func(api). Any value returned by hook is ignored.

hook-related functions

fdb.**add_hook**(*hook_type*, *func*)

Instals hook function for specified hook_type.

Parameters

- **hook_type** – One from HOOK_* constants
- **func** – Hook routine to be installed

Important: Routine must have a signature required for given hook type. However it's not checked when hook is installed, and any issue will lead to run-time error when hook routine is executed.

---

fdb.**remove_hook**(*hook_type*, *func*)

Uninstalls previously installed hook function for specified hook_type.

Parameters

- **hook_type** – One from HOOK_* constants
- **func** – Hook routine to be uninstalled

If hook routine wasn't previously installed, it does nothing.

fdb.**get_hooks**(*hook_type*)

Returns list of installed hook routines for specified hook_type.

Parameters **hook_type** – One from HOOK_* constants

Returns List of installed hook routines.

fdb.**is_dead_proxy**(*obj*)
Return True if object is a dead `weakref.proxy()`.

Classes

Connection

class fdb.**Connection**(*db_handle*, *dpb=None*, *sql_dialect=3*, *charset=None*, *isolation_level='x03tx06x0fx11'*)
Represents a connection between the database client (the Python process) and the database server.

Important: DO NOT create instances of this class directly! Use only *`connect()`* or *`create_database()`* to get Connection instances.

Parameters

- **db_handle** – Database handle provided by factory function.
- **dpb** – Database Parameter Block associated with database handle.
- **sql_dialect** (*integer*) – SQL Dialect associated with database handle.
- **charset** (*string*) – Character set associated with database handle.

exception DataError
Exception raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc.

args

message

exception DatabaseError
Exception raised for errors that are related to the database.

args

message

exception Error
Exception that is the base class of all other error exceptions. You can use this to catch all errors with one single 'except' statement. Warnings are not considered errors and thus should not use this class as base.

args

message

exception IntegrityError
Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails.

args

message

exception InterfaceError
Exception raised for errors that are related to the database interface rather than the database itself.

args

message

exception InternalError
Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc.

args

message

exception NotSupportedError
Exception raised in case a method or database API was used which is not supported by the database

args

message

exception OperationalError
Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc.

args

message

exception ProgrammingError
Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc.

args

message

exception Warning
Base class for warning categories.

args

message

begin(*tpb=None*)
Starts a transaction explicitly. Operates on *main_transaction*. See *Transaction.begin()* for details.

Parameters **tpb** (*TPB* instance, list/tuple of *isc_tpb_** constants or *bytestring*) – (Optional) Transaction Parameter Buffer for newly started transaction. If not specified, *default_tpb* is used.

close()
Close the connection now (rather than whenever *__del__* is called). The connection will be unusable from this point forward; an *Error* (or subclass) exception will be raised if any operation is attempted with the connection. The same applies to all cursor and transaction objects trying to use the connection.

Also closes all *EventConduit*, *Cursor* and *Transaction* instances associated with this connection.

Raises ***ProgrammingError*** – When connection is a member of a *ConnectionGroup*.

Hooks:

Event HOOK_DATABASE_DETACH_REQUEST: Executed before connection is closed. Hook must have signature: hook_func(connection). If any hook function returns True, connection is not closed.

Event HOOK_DATABASE_CLOSED: Executed after connection is closed. Hook must have signature: hook_func(connection). Any value returned by hook is ignored.

commit(*retaining=False*)

Commit pending transaction to the database. Operates on *main_transaction*. See *Transaction.commit()* for details.

Parameters **retaining** (*boolean*) – (Optional) Indicates whether the transactional context of the transaction being resolved should be recycled.

Raises ***ProgrammingError*** – If Connection is *closed*.

cursor()

Return a new *Cursor* instance using the connection associated with *main_transaction*. See *Transaction.cursor()* for details.

Raises ***ProgrammingError*** – If Connection is *closed*.

database_info(*info_code*, *result_type*, *page_number=None*)

Wraps the Firebird C API function *isc_database_info*.

For documentation, see the IB 6 API Guide section entitled "Requesting information about an attachment" (p. 51).

Note that this method is a VERY THIN wrapper around the FB C API function *isc_database_info*. This method does NOT attempt to interpret its results except with regard to whether they are a string or an integer.

For example, requesting *isc_info_user_names* will return a string containing a raw succession of length-name pairs. A thicker wrapper might interpret those raw results and return a Python tuple, but it would need to handle a multitude of special cases in order to cover all possible isc_info_* items.

Parameters

- **info_code** (*integer*) – One of the *isc_info_** constants.
- **result_type** (*string*) – Must be either 'b' if you expect a binary string result, or 'i' if you expect an integer result.
- **page_number** (*integer*) – Page number for *fb_info_page_contents* info code.

Raises

- ***DatabaseError*** – When error is returned from server.
- ***OperationalError*** – When returned information is bigger than SHRT_MAX.
- ***InternalError*** – On unexpected processing condition.
- **ValueError** – On illegal *result_type* value.

See also:

Extracting data with the database_info function is rather clumsy. See *db_info()* for higher-level means of accessing the same information.

Note: Some of the information available through this method would be more easily retrieved with the Services API (see submodule *fdb.services*).

---

db_info(*request*)

Higher-level convenience wrapper around the *database_info()* method that parses the output of *database_info* into Python-friendly objects instead of returning raw binary buffers in the case of complex result types.

Parameters **request** – Single *fdb.isc_info_** info request code or a sequence of such codes.

Returns Mapping of (info request code -> result).

Raises

- **ValueError** – When requested code is not recognized.
- ***OperationalError*** – On unexpected processing condition.

drop_database()
Drops the database to which this connection is attached.

Unlike plain file deletion, this method behaves responsibly, in that it removes shadow files and other ancillary files for this database.

Raises

- ***ProgrammingError*** – When connection is a member of a *ConnectionGroup*.
- ***DatabaseError*** – When error is returned from server.

event_conduit(*event_names*)
Creates a conduit through which database event notifications will flow into the Python program.

Parameters **event_names** – A sequence of string event names.

Returns An *EventConduit* instance.

execute_immediate(*sql*)
Executes a statement in context of *main_transaction* without caching its prepared form.

Automatically starts transaction if it's not already started.

Parameters **sql** (*string*) – SQL statement to execute.

Important: **The statement must not be of a type that returns a result set.** In most cases (especially cases in which the same statement – perhaps a parameterized statement – is executed repeatedly), it is better to create a cursor using the connection's cursor method, then execute the statement using one of the cursor's execute methods.

Parameters **sql** (*string*) – SQL statement to execute.

Raises

- ***ProgrammingError*** – When connection is closed.
- ***DatabaseError*** – When error is returned from server.

get_active_transaction_count()
Return count of currently active transactions.

get_active_transaction_ids()
Return list of transaction IDs for all currently active transactions.

get_page_contents(*page_number*)
Return content of specified database page as binary string.

Parameters **page_number** (*int*) – Page sequence number.

get_table_access_stats()
Return current stats for access to tables.

Returns List of *fbcore._TableAccessStats* instances.

isreadonly()
Returns True if database is read-only.

rollback(*retaining=False*, *savepoint=None*)
Causes the the database to roll back to the start of pending transaction. Operates on *main_transaction*. See *Transaction.rollback()* for details.

Parameters

- **retaining** (*boolean*) – (Optional) Indicates whether the transactional context of the transaction being resolved should be recycled.
- **savepoint** (*string*) – (Optional) Causes the transaction to roll back only as far as the designated savepoint, rather than rolling back entirely.

Raises ***ProgrammingError*** – If Connection is *closed*.

savepoint(*name*)
Establishes a named SAVEPOINT for current transaction. Operates on *main_transaction*. See *Transaction.savepoint()* for details.

Parameters **name** (*string*) – Name for savepoint.

Raises ***ProgrammingError*** – If Connection is *closed*.

Example:

```
con.savepoint('BEGINNING_OF_SOME_SUBTASK')
...
con.rollback(savepoint='BEGINNING_OF_SOME_SUBTASK')
```

trans(*default_tpb=None*)
Creates a new *Transaction* that operates within the context of this connection. Cursors can be created within that Transaction via its *cursor()* method.

Parameters **default_tpb** (*TPB* instance, list/tuple of *isc_tpb_** constants or *bytestring*) – (optional) Transaction Parameter Block for newly created Transaction. If not specified, *default_tpb* is used.

Raises ***ProgrammingError*** – If Connection is *closed*.

trans_info(*request*)
Pythonic wrapper around *transaction_info()* call. Operates on *main_transaction*. See *Transaction.trans_info()* for details.

transaction_info(*info_code*, *result_type*)
Returns information about active transaction. Thin wrapper around Firebird API *isc_transaction_info* call. Operates on *main_transaction*. See *Transaction.transaction_info()* for details.

attachment_id

charset

closed

creation_date

current_memory

database_name

database_sql_dialect

db_class_id

default_tpb

engine_version

firebird_version

forced_writes

group

implementation_id

io_stats

main_transaction

max_memory

monitor

next_transaction

oat

ods

ods_minor_version

ods_version

oit

ost

page_cache_size

page_size

pages_allocated

provider_id

query_transaction

schema

server_version

site_name

space_reservation

sql_dialect

sweep_interval

transactions

version

ConnectionWithSchema

class fdb.**ConnectionWithSchema**(*db_handle*, *dpb=None*, *sql_dialect=3*, *charset=None*, *isolation_level='x03tx06x0fx11'*)

Connection descendant that exposes all attributes of encapsulated *Schema* instance directly as connection attributes, except *close()* and *bind()*, and those attributes that are already defined by Connection class.

Note: Use *connection_class* parametr of *connect()* or *create_database()* to create connections with direct schema interface.

Note: For list of methods see *fdb.schema.Schema*.

Cursor

class fdb.**Cursor**(*connection*, *transaction*)

Represents a database cursor, which is used to execute SQL statement and manage the context of a fetch operation.

Important: DO NOT create instances of this class directly! Use only *Connection.cursor()*, *Transaction.cursor()* and *ConnectionGroup.cursor()* to get Cursor instances that operate in desired context.

Note: Cursor is actually a high-level wrapper around *PreparedStatement* instance(s) that handle the actual SQL statement execution and result management.

Tip: Cursor supports the iterator protocol, yielding tuples of values like *fetchone()*.

Important: The association between a Cursor and its *Transaction* and *Connection* is set when the Cursor is created, and cannot be changed during the lifetime of that Cursor.

Parameters

- **connection** – *Connection* instance this cursor should be bound to.
- **transaction** – *Transaction* instance this cursor should be bound to.

callproc(*procname*, *parameters=None*)

Call a stored database procedure with the given name.

The result of the call is available through the standard fetchXXX() methods.

Parameters

- **procname** (*string*) – Stored procedure name.
- **parameters** (*List or Tuple*) – (Optional) Sequence of parameters. Must contain one entry for each argument that the procedure expects.

Returns parameters, as required by Python DB API 2.0 Spec.

Raises

- **TypeError** – When parameters is not List or Tuple.
- ***ProgrammingError*** – When more parameters than expected are suplied.

- **DatabaseError** – When error is returned by server.

close()
Close the cursor now (rather than whenever __del__ is called).

Closes any currently open *PreparedStatement*. However, the cursor is still bound to *Connection* and *Transaction*, so it could be still used to execute SQL statements.

> **Warning:** FDB's implementation of Cursor somewhat violates the Python DB API 2.0, which requires that cursor will be unusable after call to *close*; and an Error (or subclass) exception should be raised if any operation is attempted with the cursor.
>
> If you'll take advantage of this anomaly, your code would be less portable to other Python DB API 2.0 compliant drivers.

execute(*operation*, *parameters=None*)
Prepare and execute a database operation (query or command).

Note: Execution is handled by *PreparedStatement* that is either supplied as *operation* parameter, or created internally when *operation* is a string. Internally created PreparedStatements are stored in cache for later reuse, when the same *operation* string is used again.

Returns self, so call to execute could be used as iterator.

Parameters

- **operation** (string or *PreparedStatement* instance) – SQL command specification.
- **parameters** (*List or Tuple*) – (Optional) Sequence of parameters. Must contain one entry for each argument that the operation expects.

Raises

- **ValueError** – When operation PreparedStatement belongs to different Cursor instance.
- **TypeError** – When parameters is not List or Tuple.
- **ProgrammingError** – When more parameters than expected are suplied.
- **DatabaseError** – When error is returned by server.

executemany(*operation*, *seq_of_parameters*)
Prepare a database operation (query or command) and then execute it against all parameter sequences or mappings found in the sequence *seq_of_parameters*.

Note: This function simply calls *execute()* in a loop, feeding it with parameters from *seq_of_parameters*. Because *execute* caches *PreparedStatements*, calling *executemany* is equally efective as direct use of prepared statement and calling *execute* in a loop directly in application.

Returns self, so call to executemany could be used as iterator.

Parameters

- **operation** (string or *PreparedStatement* instance) – SQL command specification.

- **seq_of_parameters** (*List or Tuple*) – Sequence of sequences of parameters. Must contain one sequence of parameters for each execution that has one entry for each argument that the operation expects.

Raises

- **ValueError** – When operation PreparedStatement belongs to different Cursor instance.
- **TypeError** – When seq_of_parameters is not List or Tuple.
- ***ProgrammingError*** – When there are more parameters in any sequence than expected.
- ***DatabaseError*** – When error is returned by server.

fetchall()

Fetch all (remaining) rows of a query result.

Returns List of tuples, where each tuple is one row of returned values.

Raises

- ***DatabaseError*** – When error is returned by server.
- ***ProgrammingError*** – When underlying *PreparedStatement* is closed, statement was not yet executed, or unknown status is returned by fetch operation.

fetchallmap()

Fetch all (remaining) rows of a query result like *fetchall()*, except that it returns a list of mappings of field name to field value, rather than a list of tuples.

Returns List of *fbcore._RowMapping* instances, one such instance for each row.

Raises

- ***DatabaseError*** – When error is returned by server.
- ***ProgrammingError*** – When underlying *PreparedStatement* is closed, statement was not yet executed, or unknown status is returned by fetch operation.

fetchmany(*size=1*)

Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a list of tuples). An empty sequence is returned when no more rows are available. The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's arraysize determines the number of rows to be fetched. The method does try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

Parameters **size** (*integer*) – Max. number of rows to fetch.

Returns List of tuples, where each tuple is one row of returned values.

Raises

- ***DatabaseError*** – When error is returned by server.
- ***ProgrammingError*** – When underlying *PreparedStatement* is closed, statement was not yet executed, or unknown status is returned by fetch operation.

fetchmanymap(*size=1*)

Fetch the next set of rows of a query result, like *fetchmany()*, except that it returns a list of mappings of field name to field value, rather than a list of tuples.

Parameters **size** (*integer*) – Max. number of rows to fetch.

Returns List of *fbcore._RowMapping* instances, one such instance for each row.

Raises

- ***DatabaseError*** – When error is returned by server.
- ***ProgrammingError*** – When underlying *PreparedStatement* is closed, statement was not yet executed, or unknown status is returned by fetch operation.

fetchone()
Fetch the next row of a query result set.

Returns tuple of returned values, or None when no more data is available.

Raises

- ***DatabaseError*** – When error is returned by server.
- ***ProgrammingError*** – When underlying *PreparedStatement* is closed, statement was not yet executed, or unknown status is returned by fetch operation.

fetchonemap()
Fetch the next row of a query result set like *fetchone()*, except that it returns a mapping of field name to field value, rather than a tuple.

Returns *fbcore._RowMapping* of returned values, or None when no more data is available.

Raises

- ***DatabaseError*** – When error is returned by server.
- ***ProgrammingError*** – When underlying *PreparedStatement* is closed, statement was not yet executed, or unknown status is returned by fetch operation.

iter()
Equivalent to the *fetchall()*, except that it returns iterator rather than materialized list.

Returns Iterator that yields tuple of values like *fetchone()*.

itermap()
Equivalent to the *fetchallmap()*, except that it returns iterator rather than materialized list.

Returns Iterator that yields *fbcore._RowMapping* instance like *fetchonemap()*.

next()
Return the next item from the container. Part of *iterator protocol*.

Raises **StopIteration** – If there are no further items.

prep(*operation*)
Create prepared statement for repeated execution.

Note: Returned *PreparedStatement* instance is bound to its Cursor instance via strong reference, and is not stored in Cursor's internal cache of prepared statements.

Parameters **operation** (*string*) – SQL command

Returns *PreparedStatement* instance.

Raises

- ***DatabaseError*** – When error is returned by server.
- ***InternalError*** – On unexpected processing condition.

set_stream_blob(*blob_name*)
Specify a BLOB column(s) to work in *stream* mode instead classic, materialized mode for already executed statement.

Parameters **blob_name** (*string or list*) – Single name or sequence of column names. Name must be in format as it's stored in database (refer to *description* for real value).

Important: BLOB name is **permanently** added to the list of BLOBs handled as *stream* BLOBs by current *PreparedStatement* instance. If instance is stored in internal cache of prepared statements, the same command executed repeatedly will retain this setting.

Parameters **blob_name** (*string*) – Name of BLOB column.

Raises ***ProgrammingError*** –

set_stream_blob_treshold(*size*)
Specify max. blob size for materialized blobs. If size of particular blob exceeds this threshold, returns streamed blob (*BlobReader*) instead string. Value -1 means no size limit (use at your own risk). Default value is 64K

Parameters **size** (*integer*) – Max. size for materialized blob.

setinputsizes(*sizes*)
Required by Python DB API 2.0, but pointless for Firebird, so it does nothing.

setoutputsize(*size*, *column=None*)
Required by Python DB API 2.0, but pointless for Firebird, so it does nothing.

arraysize = 1

connection

description

name

plan

rowcount

transaction

Transaction

class fdb.**Transaction**(*connections*, *default_tpb=None*, *default_action='commit'*)
Represents a transaction context, which is used to execute SQL statement.

Important: DO NOT create instances of this class directly! *Connection* and *ConnectionGroup* manage Transaction internally, surfacing all important methods directly in their interfaces. If you want additional transactions independent from *Connection.main_transaction*, use *Connection.trans()* method to obtain such *Transaction* instance.

Parameters

- **connections** (*iterable*) – Sequence of (up to 16) *Connection* instances.

- **default_tpb** (*TPB* instance, list/tuple of *isc_tpb_** constants or *bytestring*) – Transaction Parameter Block for this transaction. If *None* is specified, uses *ISOLATION_LEVEL_READ_COMMITED*.
- **default_action** (*string 'commit' or 'rollback'*) – Action taken when active transaction is ended automatically (during *close()* or *begin()*).

Raises ***ProgrammingError*** – When zero or more than 16 connections are given.

begin(*tpb=None*)

Starts a transaction explicitly.

Parameters **tpb** (*TPB* instance, list/tuple of *isc_tpb_** constants or *bytestring*) – (optional) Transaction Parameter Block for newly created Transaction. If not specified, *default_tpb* is used.

Note: Calling this method directly is never required; a transaction will be started implicitly if necessary.

Important: If the physical transaction is unresolved when this method is called, a *commit()* or *rollback()* will be performed first, accordingly to *default_action* value.

Raises

- ***DatabaseError*** – When error is returned by server.
- ***ProgrammingError*** – When TPB is in usupported format, or transaction is permanently *closed*.

close()

Permanently closes the Transaction object and severs its associations with other objects (*Cursor* and *Connection* instances).

Important: If the physical transaction is unresolved when this method is called, a *commit()* or *rollback()* will be performed first, accordingly to *default_action* value.

---

commit(*retaining=False*)

Commit any pending transaction to the database.

Note: If transaction is not active, this method does nothing, because the consensus among Python DB API experts is that transactions should always be started implicitly, even if that means allowing a *commit()* or *rollback()* without an actual transaction.

Parameters **retaining** (*boolean*) – Indicates whether the transactional context of the transaction being resolved should be recycled.

Raises ***DatabaseError*** – When error is returned by server as response to commit.

cursor(*connection=None*)

Creates a new *Cursor* that will operate in the context of this Transaction.

Parameters **connection** (*Connection* instance) – **Required only when** Transaction is bound to multiple *Connections*, to specify to which *Connection* the returned Cursor should be bound.

Raises ***ProgrammingError*** – When transaction operates on multiple *Connections* and: *connection* parameter is not specified, or specified *connection* is not among *Connections* this Transaction is bound to.

execute_immediate(*sql*)

Executes a statement without caching its prepared form on all connections this transaction is bound to.

Automatically starts transaction if it's not already started.

Parameters **sql** (*string*) – SQL statement to execute.

Important: **The statement must not be of a type that returns a result set.** In most cases (especially cases in which the same statement – perhaps a parameterized statement – is executed repeatedly), it is better to create a cursor using the connection's cursor method, then execute the statement using one of the cursor's execute methods.

Parameters **sql** (*string*) – SQL statement to execute.

Raises ***DatabaseError*** – When error is returned from server.

isreadonly()
Returns True if transaction is Read Only.

prepare()
Manually triggers the first phase of a two-phase commit (2PC).

Note: Direct use of this method is optional; if preparation is not triggered manually, it will be performed implicitly by *commit()* in a 2PC.

---

rollback(*retaining=False*, *savepoint=None*)
Rollback any pending transaction to the database.

Note: If transaction is not active, this method does nothing, because the consensus among Python DB API experts is that transactions should always be started implicitly, even if that means allowing a *commit()* or *rollback()* without an actual transaction.

Parameters

- **retaining** (*boolean*) – Indicates whether the transactional context of the transaction being resolved should be recycled. Mutually exclusive with 'savepoint'.
- **savepoint** (*string*) – Savepoint name. Causes the transaction to roll back only as far as the designated savepoint, rather than rolling back entirely. Mutually exclusive with 'retaining'.

Raises

- ***ProgrammingError*** – If both *savepoint* and *retaining* are specified.
- ***DatabaseError*** – When error is returned by server as response to rollback.

savepoint(*name*)
Establishes a savepoint with the specified name.

Note: If transaction is bound to multiple connections, savepoint is created on all of them.

Important: Because savepoint is created not through Firebird API (there is no such API call), but by executing *SAVEPOINT* <*name*> SQL statement, calling this method starts the transaction if it was not yet started.

Parameters **name** (*string*) – Savepoint name.

trans_info(*request*)
Pythonic wrapper around *transaction_info()* call.

Parameters **request** – One or more information request codes (see *transaction_info()* for details). Multiple codes must be passed as tuple.

Returns Decoded response(s) for specified request code(s). When multiple requests are passed, returns a dictionary where key is the request code and value is the response from server.

transaction_info(*info_code*, *result_type*)
Return information about active transaction.

This is very thin wrapper around Firebird API *isc_transaction_info* call.

Parameters

- **info_code** (*integer*) – One from the *isc_info_tra_** constants.
- **result_type** (*'b' for binary string or 'i' for integer*) – Code for result type.

Raises

- ***ProgrammingError*** – If transaction is not active.
- ***OperationalError*** – When result is too large to fit into buffer of size SHRT_MAX.
- ***InternalError*** – On unexpected processing condition.
- **ValueError** – When illegal result type code is specified.

active

closed

cursors

default_action

default_tpb = '\x03\t\x06\x0f\x11'

isolation

lock_timeout

oat

oit

ost

transaction_id

PreparedStatement

class fdb.**PreparedStatement**(*operation*, *cursor*, *internal=True*)

Represents a prepared statement, an "inner" database cursor, which is used to manage the SQL statement execution and context of a fetch operation.

Important: DO NOT create instances of this class directly! Use only *Cursor.prep()* to get PreparedStatement instances.

Note: PreparedStatements are bound to *Cursor* instance that created them, and using them with other Cursor would report an error.

---

close()

Drops the resources associated with executed prepared statement, but keeps it prepared for another execution.

set_stream_blob(*blob_spec*)

Specify a BLOB column(s) to work in *stream* mode instead classic, materialized mode.

Parameters **blob_spec** (*string or list*) – Single name or sequence of column names. Name must be in format as it's stored in database (refer to *description* for real value).

Important: BLOB name is **permanently** added to the list of BLOBs handled as *stream* BLOBs by this instance.

Parameters **blob_spec** (*string*) – Name of BLOB column.

set_stream_blob_treshold(*size*)

Specify max. blob size for materialized blobs. If size of particular blob exceeds this threshold, returns streamed blob (*BlobReader*) instead string. Value -1 means no size limit (use at your own risk). Default value is 64K

Parameters **size** (*integer*) – Max. size for materialized blob.

NO_FETCH_ATTEMPTED_YET = -1

RESULT_SET_EXHAUSTED = 100

closed

cursor = None

description

n_input_params = 0

n_output_params = 0

name

plan

rowcount

sql

statement_type = 0

ConnectionGroup

class fdb.**ConnectionGroup**(*connections=()*)

Manager for distributed transactions, i.e. transactions that span multiple databases.

Tip: ConnectionGroup supports *in* operator to check membership of connections.

Parameters **connections** (*iterable*) – Sequence of *Connection* instances.

See also:

See *add()* for list of exceptions the constructor may throw.

add(*con*)

Adds active connection to the group.

Parameters **con** – A *Connection* instance to add to this group.

Raises

- **TypeError** – When *con* is not *Connection* instance.
- ***ProgrammingError*** – When *con* is already member of this or another group, or *closed*. When this group has unresolved transaction or contains 16 connections.

begin(*tpb=None*)

Starts distributed transaction over member connections.

Parameters **tpb** (*TPB* instance, list/tuple of *isc_tpb_** constants or *bytestring*) – (Optional) Transaction Parameter Buffer for newly started transaction. If not specified, *default_tpb* is used.

Raises ***ProgrammingError*** – When group is empty or has active transaction.

clear()

Removes all connections from group.

Raises ***ProgrammingError*** – When transaction is active.

commit(*retaining=False*)

Commits distributed transaction over member connections using 2PC.

Note: If transaction is not active, this method does nothing, because the consensus among Python DB API experts is that transactions should always be started implicitly, even if that means allowing a *commit()* or *rollback()* without an actual transaction.

Parameters **retaining** (*boolean*) – Indicates whether the transactional context of the transaction being resolved should be recycled.

Raises ***ProgrammingError*** – When group is empty.

contains(*con*)

Returns True if specified connection belong to this group.

Parameters **con** – *Connection* instance.

count()
Returns number of *Connection* objects that belong to this group.

cursor(*connection*)
Creates a new *Cursor* that will operate in the context of distributed transaction and specific *Connection* that belongs to this group.

Note: Automatically starts transaction if it's not already started.

Parameters **connection** – *Connection* instance.

Raises ***ProgrammingError*** – When group is empty or specified *connection* doesn't belong to this group.

disband()
Forcefully deletes all connections from connection group.

Note: If transaction is active, it's canceled (**rollback**).

Note: Any error during transaction finalization doesn't stop the disband process, however the exception catched is eventually reported.

---

execute_immediate(*sql*)
Executes a statement on all member connections without caching its prepared form.

Automatically starts transaction if it's not already started.

Parameters **sql** (*string*) – SQL statement to execute.

Important: **The statement must not be of a type that returns a result set.** In most cases (especially cases in which the same statement – perhaps a parameterized statement – is executed repeatedly), it is better to create a cursor using the connection's cursor method, then execute the statement using one of the cursor's execute methods.

Parameters **sql** (*string*) – SQL statement to execute.

Raises ***DatabaseError*** – When error is returned from server.

members()
Returns list of connection objects that belong to this group.

prepare()
Manually triggers the first phase of a two-phase commit (2PC). Use of this method is optional; if preparation is not triggered manually, it will be performed implicitly by commit() in a 2PC.

remove(*con*)
Removes specified connection from group.

Parameters **con** – A *Connection* instance to remove.

Raises ***ProgrammingError*** – When *con* doesn't belong to this group or transaction is active.

rollback (*retaining=False*, *savepoint=None*)
Rollbacks distributed transaction over member connections.

Note: If transaction is not active, this method does nothing, because the consensus among Python DB API experts is that transactions should always be started implicitly, even if that means allowing a *commit()* or *rollback()* without an actual transaction.

Parameters **retaining** (*boolean*) – Indicates whether the transactional context of the transaction being resolved should be recycled.

Raises ***ProgrammingError*** – When group is empty.

savepoint (*name*)
Establishes a named SAVEPOINT on all member connections. See *Transaction.savepoint()* for details.

Parameters **name** (*string*) – Name for savepoint.

Raises ***ProgrammingError*** – When group is empty.

default_tpb

TransactionContext

class fdb.**TransactionContext** (*transaction*)
Context Manager that manages transaction for object passed to constructor.

Performs *rollback* if exception is thrown inside code block, otherwise performs *commit* at the end of block.

Example:

```
with TransactionContext(my_transaction):
    cursor.execute('insert into tableA (x,y) values (?,?)',(x,y))
    cursor.execute('insert into tableB (x,y) values (?,?)',(x,y))
```

Parameters **transaction** – Any object that supports *begin()*, *commit()* and *rollback()*.

transaction = None

EventConduit

class fdb.**EventConduit** (*db_handle*, *event_names*)
Represents a conduit through which database event notifications will flow into the Python program.

Important: DO NOT create instances of this class directly! Use only *Connection.event_conduit()* to get EventConduit instances.

Notifications of any events are not accumulated until *begin()* method is called.

From the moment the *begin()* method is called, notifications of any events that occur will accumulate asynchronously within the conduit's internal queue until the conduit is closed either explicitly (via the *close()* method) or implicitly (via garbage collection).

EventConduit implements context manager protocol to call method *begin()* and *close()* automatically.

Example:

```
with connection.event_conduit( ('event_a', 'event_b') ) as conduit:
    events = conduit.wait()
    process_events(events)
```

Parameters

- **db_handle** – Database handle.
- **event_names** – List of strings that represent event names.

begin()
Starts listening for events.

Must be called directly or through context manager interface.

close()
Cancels the standing request for this conduit to be notified of events.

After this method has been called, this EventConduit object is useless, and should be discarded.

flush()
Clear any event notifications that have accumulated in the conduit's internal queue.

wait(*timeout=None*)
Wait for events.

Blocks the calling thread until at least one of the events occurs, or the specified timeout (if any) expires.

Parameters **timeout** (*integer or float*) – Number of seconds (use a float to indicate fractions of seconds). If not even one of the relevant events has occurred after timeout seconds, this method will unblock and return None. The default timeout is infinite.

Returns *None* if the wait timed out, otherwise a dictionary that maps *event_name* -> *event_occurrence_count*.

Example:

```
>>>conduit = connection.event_conduit( ('event_a', 'event_b') )
>>>conduit.begin()
>>>conduit.wait()
{
 'event_a': 1,
 'event_b': 0
}
```

In the example above *event_a* occurred once and *event_b* did not occur at all.

closed

BlobReader

class fdb.**BlobReader**(*blobid*, *db_handle*, *tr_handle*, *is_text*, *charset*)
BlobReader is a "file-like" class, so it acts much like a file instance opened in *rb* mode.

Important: DO NOT create instances of this class directly! BlobReader instances are returned automatically in place of output BLOB values when *stream* BLOB access is requested via *PreparedStatement.set_stream_blob()*.

Tip: BlobReader supports iterator protocol, yielding lines like *readline()*.

close()
Closes the Reader. Like file.close().

Raises ***DatabaseError*** – When error is returned by server.

flush()
Flush the internal buffer. Like file.flush(). Does nothing as it's pointless for reader.

get_info()
Return information about BLOB.

Returns Tuple with values: blob_length, segment_size, num_segments, blob_type

Meaning of individual values:

Blob_length Total blob length in bytes

Segment_size Size of largest segment

Num_segments Number of segments

Blob_type isc_bpb_type_segmented or isc_bpb_type_stream

next()
Return the next line from the BLOB. Part of *iterator protocol*.

Raises **StopIteration** – If there are no further lines.

read(*size=-1*)
Read at most size bytes from the file (less if the read hits EOF before obtaining size bytes). If the size argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately. Like file.read().

Raises ***ProgrammingError*** – When reader is closed.

Note: Performs automatic conversion to *unicode* for TEXT BLOBs, if used Python is v3 or *connection charset* is defined.

readline()
Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). An empty string is returned when EOF is encountered immediately. Like file.readline().

Raises ***ProgrammingError*** – When reader is closed.

Note: Performs automatic conversion to *unicode* for TEXT BLOBs, if used Python is v3 or *connection charset* is defined.

readlines(*sizehint=None*)
Read until EOF using *readline()* and return a list containing the lines thus read. The optional sizehint argument (if present) is ignored. Like `file.readlines()`.

Note: Performs automatic conversion to *unicode* for TEXT BLOBs, if used Python is v3 or *connection charset* is defined.

---

seek(*offset*, *whence=0*)
Set the file's current position, like stdio's *fseek()*. See `file.seek()` details.

Parameters

- **offset** (*integer*) – Offset from specified position.
- **whence** (*os.SEEK_SET, os.SEEK_CUR or os.SEEK_END*) – (Optional) Context for offset.

Raises ***ProgrammingError*** – When reader is closed.

Warning: If BLOB was NOT CREATED as *stream* BLOB, this method raises *DatabaseError* exception. This constraint is set by Firebird.

tell()
Return current position in BLOB, like stdio's *ftell()* and `file.tell()`.

closed

mode

TPB

class fdb.**TPB**
Helper class for convenient and safe construction of custom Transaction Parameter Blocks.

clear()

copy()
Returns a copy of self.

render()
Create valid *transaction parameter block* according to current values of member attributes.

Returns (string) TPB block.

access_mode

isolation_level

lock_resolution

lock_timeout

table_reservation

TableReservation

class fdb.**TableReservation**

A dictionary-like helper class that maps "TABLE_NAME": (sharingMode, accessMode). It performs validation of values assigned to keys.

copy()

get(*key*, *default=None*)

items()

iteritems()

iterkeys()

itervalues()

keys()

render()

Create valid *table access parameter block* according to current key/value pairs.

Returns (string) Table access definition block.

values()

ParameterBuffer

class fdb.**ParameterBuffer**(*charset*)

Helper class for construction of Database (and other) parameter buffers. Parameters are stored in insertion order.

add_byte(*byte*)

Add byte value to buffer.

Parameters **byte** – Value to be added.

add_byte_parameter(*code*, *value*)

Add byte value to parameter buffer.

Parameters

- **code** – Firebird code for the parameter
- **value** – Parameter value (0-255)

add_integer_parameter(*code*, *value*)

Add integer value to parameter buffer.

Parameters

- **code** – Firebird code for the parameter
- **value** (*int*) – Parameter value

add_parameter_code(*code*)

Add parameter code to parameter buffer.

Parameters **code** – Firebird code for the parameter

add_string(*value*)

Add string value to buffer.

Parameters **value** – String to be added.

add_string_parameter(*code*, *value*)
Add string to parameter buffer.

Parameters

- **code** – Firebird code for the parameter
- **value** (*string*) – Parameter value

add_word(*word*)
Add two byte value to buffer.

Parameters **word** – Value to be added.

clear()
Clear all parameters stored in parameter buffer.

get_buffer()
Get parameter buffer content.

Returns Byte string with all inserted parameters.

get_length()
Returns actual total length of parameter buffer.

Internally used classes exposed to driver users

_RowMapping

class fdb.fbcore.**_RowMapping**(*description*, *row*)
An internal dictionary-like class that wraps a row of results in order to map field name to field value.

> **Warning:** We make ABSOLUTELY NO GUARANTEES about the return value of the *fetch(one|many|all)* methods except that it is a sequence indexed by field position, and no guarantees about the return value of the *fetch(one|many|all)map* methods except that it is a mapping of field name to field value.
>
> Therefore, client programmers should NOT rely on the return value being an instance of a particular class or type.

get(*field_name*, *default_value=None*)

items()

iteritems()

iterkeys()

itervalues()

keys()

values()

EventBlock

class fdb.fbcore.**EventBlock**(*queue*, *db_handle*, *event_names*)
Represents Firebird API structure for block of events.

close()
Close this block canceling managed events.

count_and_reregister()
Count event occurences and reregister interest in futrther notifications.

buf_length = 0

closed

event_buf = None

event_id = 0

event_names = []

result_buf = None

TableAccessStats

class fdb.fbcore.**_TableAccessStats**(*table_id*)
An internal class that wraps results from *get_table_access_stats()*

1.4.2 Services

Constants

shutdown_mode codes for `Connection.shutdown()` and `Connection.bring_online()`

- SHUT_LEGACY
- SHUT_NORMAL
- SHUT_MULTI
- SHUT_SINGLE
- SHUT_FULL

shutdown_method codes for `Connection.shutdown()`

- SHUT_FORCE
- SHUT_DENY_NEW_TRANSACTIONS
- SHUT_DENY_NEW_ATTACHMENTS

mode codes for `Connection.setWriteMode()`

- WRITE_FORCED
- WRITE_BUFFERED

mode codes for `Connection.setAccessMode()`

- ACCESS_READ_WRITE
- ACCESS_READ_ONLY

`Connection.get_server_capabilities()` return codes

- CAPABILITY_MULTI_CLIENT
- CAPABILITY_REMOTE_HOP
- CAPABILITY_SERVER_CONFIG
- CAPABILITY_QUOTED_FILENAME
- CAPABILITY_NO_SERVER_SHUTDOWN

'stats' codes for `Connection.backup()`/`Connection.restore()`

- STATS_TOTAL_TIME
- STATS_TIME_DELTA
- STATS_PAGE_READS
- STATS_PAGE_WRITES

Functions

connect

`fdb.services.`**`connect`**(*host='service_mgr'*, *user=None*, *password=None*)

Establishes a connection to the Services Manager.

Parameters

- **`host`** (*`string`*) – (optional) Host machine specification. Local by default.
- **`user`** (*`string`*) – (optional) Administrative user name. Defaults to content of environment variable *'ISC_USER'* or *'SYSDBA'*.
- **`password`** (*`string`*) – Administrative user password. Default is content of environment variable *'ISC_PASSWORD'*.

Note: By definition, a Services Manager connection is bound to a particular host. Therefore, the database specified as a parameter to methods such as *getStatistics* MUST NOT include the host name of the database server.

Hooks:

Event *HOOK_SERVICE_ATTACHED*: Executed before *`Connection`* instance is returned. Hook must have signature: hook_func(connection). Any value returned by hook is ignored.

Classes

Connection

class fdb.services.**Connection**(*host*, *user*, *password*, *charset=None*)
Represents a sevice connection between the database client (the Python process) and the database server.

Important: DO NOT create instances of this class directly! Use only *connect()* to get Connection instances.

Tip: Connection supports the iterator protocol, yielding lines of result like *readline()*.

---

activate_shadow(*database*)
Activate Database Shadow(s).

Parameters **database** (*string*) – Database filename or alias.

add_user(*user*)
Add new user.

Parameters **user** (*User*) – Instance of *User* with **at least** its `name` and `password` attributes specified as non-empty values. All other attributes are optional.

Note: This method ignores the `user_id` and `group_id` attributes of *User* regardless of their values.

---

backup(*source_database*, *dest_filenames*, *dest_file_sizes=()*, *ignore_checksums=0*, *ignore_limbo_transactions=0*, *metadata_only=0*, *collect_garbage=1*, *transportable=1*, *convert_external_tables=0*, *compressed=1*, *no_db_triggers=0*, *callback=None*, *stats=None*)
Request logical (GBAK) database backup. **(ASYNC service)**

Parameters

- **source_database** (*string*) – Source database specification.
- **dest_filenames** (*string or tuple of strings*) – Backup file(s) specification.
- **dest_file_sizes** (*tuple of integers*) – (optional) specification of backup file max. sizes.
- **ignore_checksums** (*integer*) – *1* to ignore checksums.
- **ignore_limbo_transactions** (*integer*) – *1* to ignore limbo transactions.
- **metadata_only** (*integer*) – *1* to create only metadata backup.
- **collect_garbage** (*integer*) – *0* to skip garbage collection.
- **transportable** (*integer*) – *0* to do not create transportable backup.
- **convert_external_tables** (*integer*) – *1* to convert external table to internal ones.
- **compressed** (*integer*) – *0* to create uncompressed backup.
- **no_db_triggers** (*integer*) – *1* to disable database triggers temporarily.

- **callback** (*function*) – Function to call back with each output line. Function must accept only one parameter: line of output.
- **stats** (*list*) – List of arguments for run-time statistics, see STATS_* constants.

If *callback* is not specified, backup log could be retrieved through *readline()*, *readlines()*, iteration over *Connection* or ignored via call to *wait()*.

Note: Until backup report is not fully fetched from service (or ignored via *wait()*), any attempt to start another asynchronous service will fail with exception.

---

bring_online(*database*, *online_mode=0*)
Bring previously shut down database back online.

Parameters

- **database** (*string*) – Database filename or alias.
- **online_mode** (*integer*) – (Optional) One from following constants: SHUT_LEGACY, SHUT_SINGLE, SHUT_MULTI or SHUT_NORMAL (**Default**).

See also:

See also *shutdown()* method.

close()
Close the connection now (rather than whenever *__del__* is called). The connection will be unusable from this point forward; an Error (or subclass) exception will be raised if any operation is attempted with the connection.

commit_limbo_transaction(*database*, *transaction_id*)
Resolve limbo transaction with commit.

Parameters

- **database** (*string*) – Database filename or alias.
- **transaction_id** (*integer*) – ID of Transaction to resolve.

get_architecture()
Get Firebird Server architecture.

Returns string Architecture (example: 'Firebird/linux AMD64').

get_attached_database_names()
Get list of attached databases.

Returns list Filenames of attached databases.

get_connection_count()
Get number of attachments to server.

Returns integer Number of attachments.

get_home_directory()
Get Firebird Home (installation) Directory.

Returns string Directory path.

get_limbo_transaction_ids(*database*)
Get list of transactions in limbo.

Parameters **database** (*string*) – Database filename or alias.

Returns list Transaction IDs.

Raises *InternalError* – When can't process the result buffer.

get_lock_file_directory()
Get directory location for Firebird lock files.

Returns string Directory path.

get_log(*callback=None*)
Request content of Firebird Server log. **(ASYNC service)**

Parameters **callback** (*function*) – Function to call back with each output line. Function must accept only one parameter: line of output.

If *callback* is not specified, log content could be retrieved through *readline()*, *readlines()*, iteration over *Connection* or ignored via call to *wait()*.

Note: Until log content is not fully fetched from service (or ignored via *wait()*), any attempt to start another asynchronous service will fail with exception.

---

get_message_file_directory()
Get directory with Firebird message file.

Returns string Directory path.

get_security_database_path()
Get full path to Firebird security database.

Returns string Path (path+filename) to security database.

get_server_capabilities()
Get list of Firebird capabilities.

Returns tuple Capability info codes for each capability reported by server.

Next fdb.services constants define possible info codes returned:

```
CAPABILITY_MULTI_CLIENT
CAPABILITY_REMOTE_HOP
CAPABILITY_SERVER_CONFIG
CAPABILITY_QUOTED_FILENAME
CAPABILITY_NO_SERVER_SHUTDOWN
```

Example:

```
>>>fdb.services.CAPABILITY_REMOTE_HOP in svc.get_server_capabilities()
True
```

get_server_version()
Get Firebird version.

Returns string Firebird version (example: 'LI-V2.5.2.26536 Firebird 2.5').

get_service_manager_version()
Get Firebird Service Manager version number.

Returns integer Version number.

get_statistics(*database*, *show_only_db_log_pages=0*, *show_only_db_header_pages=0*, *show_user_data_pages=1*, *show_user_index_pages=1*, *show_system_tables_and_indexes=0*, *show_record_versions=0*, *callback=None*, *tables=None*)

Request database statisctics. **(ASYNC service)**

Parameters

- **database** (*string*) – Database specification.
- **show_only_db_log_pages** (*integer*) – *1* to analyze only log pages.
- **show_only_db_header_pages** (*integer*) – *1* to analyze only database header. When set, all other parameters are ignored.
- **show_user_data_pages** (*integer*) – *0* to skip user data analysis.
- **show_user_index_pages** (*integer*) – *0* to skip user index analysis.
- **show_system_tables_and_indexes** (*integer*) – *1* to analyze system tables and indices.
- **show_record_versions** (*integer*) – *1* to analyze record versions.
- **callback** (*function*) – Function to call back with each output line. Function must accept only one parameter: line of output.
- **tables** (*string_or_list*) – table name or iterable of table names.

If *callback* is not specified, statistical report could be retrieved through *readline()*, *readlines()*, iteration over *Connection* or ignored via call to *wait()*.

Note: Until report is not fully fetched from service (or ignored via *wait()*), any attempt to start another asynchronous service will fail with exception.

---

get_users(*user_name=None*)

Get information about user(s).

Parameters **user_name** (*string*) – (Optional) When specified, returns information only about user with specified user name.

Returns list *User* instances.

isrunning()

Returns True if service is running.

Note: Some services like *backup()* or *sweep()* may take time to comlete, so they're called asynchronously. Until they're finished, no other async service could be started.

---

local_backup(*source_database*, *backup_stream*, *ignore_checksums=0*, *ignore_limbo_transactions=0*, *metadata_only=0*, *collect_garbage=1*, *transportable=1*, *convert_external_tables=0*, *compressed=1*, *no_db_triggers=0*)

Request logical (GBAK) database backup into local byte stream. **(SYNC service)**

Parameters

- **source_database** (*string*) – Source database specification.
- **backup_stream** – Backup stream.
- **ignore_checksums** (*integer*) – *1* to ignore checksums.

- **ignore_limbo_transactions** (*integer*) – *1* to ignore limbo transactions.
- **metadata_only** (*integer*) – *1* to create only metadata backup.
- **collect_garbage** (*integer*) – *0* to skip garbage collection.
- **transportable** (*integer*) – *0* to do not create transportable backup.
- **convert_external_tables** (*integer*) – *1* to convert external table to internal ones.
- **compressed** (*integer*) – *0* to create uncompressed backup.
- **no_db_triggers** (*integer*) – *1* to disable database triggers temporarily.

local_restore(*backup_stream*, *dest_filenames*, *dest_file_pages=()*, *page_size=None*, *cache_buffers=None*, *access_mode_read_only=0*, *replace=0*, *deactivate_indexes=0*, *do_not_restore_shadows=0*, *do_not_enforce_constraints=0*, *commit_after_each_table=0*, *use_all_page_space=0*, *no_db_triggers=0*, *metadata_only=0*)

Request database restore from logical (GBAK) backup stored in local byte stream. **(SYNC service)**

Parameters

- **backup_stream** – Backup stream.
- **dest_filenames** (*string or tuple of strings*) – Database file(s) specification.
- **dest_file_pages** (*tuple of integers*) – (optional) specification of database file max. # of pages.
- **page_size** (*integer*) – (optional) Page size.
- **cache_buffers** (*integer*) – (optional) Size of page-cache for this database.
- **access_mode_read_only** (*integer*) – *1* to create R/O database.
- **replace** (*integer*) – *1* to replace existing database.
- **deactivate_indexes** (*integer*) – *1* to do not activate indices.
- **do_not_restore_shadows** (*integer*) – *1* to do not restore shadows.
- **do_not_enforce_constraints** (*integer*) – *1* to do not enforce constraints during restore.
- **commit_after_each_table** (*integer*) – *1* to commit after each table is restored.
- **use_all_page_space** (*integer*) – *1* to use all space on data pages.
- **no_db_triggers** (*integer*) – *1* to disable database triggers temporarily.
- **metadata_only** (*integer*) – *1* to restore only database metadata.

modify_user(*user*)

Modify user information.

Parameters **user** (*User*) – Instance of *User* with **at least** its `name` attribute specified as non-empty value.

Note: This method sets `first_name`, `middle_name` and `last_name` to their actual values, and ignores the `user_id` and `group_id` attributes regardless of their values. `password` is set **only** when it has value.

---

nbackup(*source_database*, *dest_filename*, *nbackup_level=0*, *no_db_triggers=0*)
Perform physical (NBACKUP) database backup.

Parameters

- **source_database** (*string*) – Source database specification.
- **dest_filename** – Backup file specification.
- **nbackup_level** (*integer*) – Incremental backup level.
- **no_db_triggers** (*integer*) – *1* to disable database triggers temporarily.

Note: Method call will not return until action is finished.

next()
Return the next result line from service manager. Part of *iterator protocol*.

Raises **StopIteration** – If there are no further lines.

no_linger(*database*)
Set one-off override for database linger.

Parameters **database** (*string*) – Database filename or alias.

nrestore(*source_filenames*, *dest_filename*, *no_db_triggers=0*)
Perform restore from physical (NBACKUP) database backup.

Parameters

- **source_filenames** (*string or tuple of strings*) – Backup file(s) specification.
- **dest_filename** – Database file specification.
- **no_db_triggers** (*integer*) – *1* to disable database triggers temporarily.

Note: Method call will not return until action is finished.

readline()
Get next line of textual output from last service query.

Returns string Output line.

readlines()
Get list of remaining output lines from last service query.

Returns list Service output.

Raises ***ProgrammingError*** – When service is not in *fetching* mode.

remove_user(*user*)
Remove user.

Parameters **user** (string or *User*) – User name or Instance of *User* with **at least** its `name` attribute specified as non-empty value.

repair(*database*, *read_only_validation=0*, *ignore_checksums=0*, *kill_unavailable_shadows=0*, *mend_database=0*, *validate_database=1*, *validate_record_fragments=1*)
Database Validation and Repair.

Parameters

- **database** (*string*) – Database filename or alias.
- **read_only_validation** (*integer*) – *1* to prevent any database changes.
- **ignore_checksums** (*integer*) – *1* to ignore page checksum errors.
- **kill_unavailable_shadows** (*integer*) – *1* to kill unavailable shadows.
- **mend_database** (*integer*) – *1* to fix database for backup.
- **validate_database** (*integer*) – *0* to skip database validation.
- **validate_record_fragments** (*integer*) – *0* to skip validation of record fragments.

Note: Method call will not return until action is finished.

---

restore (*source_filenames*, *dest_filenames*, *dest_file_pages=()*, *page_size=None*, *cache_buffers=None*, *access_mode_read_only=0*, *replace=0*, *deactivate_indexes=0*, *do_not_restore_shadows=0*, *do_not_enforce_constraints=0*, *commit_after_each_table=0*, *use_all_page_space=0*, *no_db_triggers=0*, *metadata_only=0*, *callback=None*, *stats=None*)

Request database restore from logical (GBAK) backup. **(ASYNC service)**

Parameters

- **source_filenames** (*string or tuple of strings*) – Backup file(s) specification.
- **dest_filenames** (*string or tuple of strings*) – Database file(s) specification.
- **dest_file_pages** (*tuple of integers*) – (optional) specification of database file max. # of pages.
- **page_size** (*integer*) – (optional) Page size.
- **cache_buffers** (*integer*) – (optional) Size of page-cache for this database.
- **access_mode_read_only** (*integer*) – *1* to create R/O database.
- **replace** (*integer*) – *1* to replace existing database.
- **deactivate_indexes** (*integer*) – *1* to do not activate indices.
- **do_not_restore_shadows** (*integer*) – *1* to do not restore shadows.
- **do_not_enforce_constraints** (*integer*) – *1* to do not enforce constraints during restore.
- **commit_after_each_table** (*integer*) – *1* to commit after each table is restored.
- **use_all_page_space** (*integer*) – *1* to use all space on data pages.
- **no_db_triggers** (*integer*) – *1* to disable database triggers temporarily.
- **metadata_only** (*integer*) – *1* to restore only database metadata.
- **callback** (*function*) – Function to call back with each output line. Function must accept only one parameter: line of output.
- **stats** (*list*) – List of arguments for run-time statistics, see STATS_* constants.

If *callback* is not specified, restore log could be retrieved through *readline()*, *readlines()*, iteration over *Connection* or ignored via call to *wait()*.

Note: Until restore report is not fully fetched from service (or ignored via `wait()`), any attempt to start another asynchronous service will fail with exception.

rollback_limbo_transaction(*database*, *transaction_id*)
Resolve limbo transaction with rollback.

Parameters

- **database** (*string*) – Database filename or alias.
- **transaction_id** (*integer*) – ID of Transaction to resolve.

set_access_mode(*database*, *mode*)
Set Database Access mode: Read Only or Read/Write

Parameters

- **database** (*string*) – Database filename or alias.
- **mode** (*integer*) – One from following constants: `ACCESS_READ_WRITE` or `ACCESS_READ_ONLY`

set_default_page_buffers(*database*, *n*)
Set individual page cache size for Database.

Parameters

- **database** (*string*) – Database filename or alias.
- **n** (*integer*) – Number of pages.

set_reserve_page_space(*database*, *reserve_space*)
Set data page space reservation policy.

Parameters

- **database** (*string*) – Database filename or alias.
- **reserve_space** (*boolean*) – *True* to reserve space, *False* to do not.

set_sql_dialect(*database*, *dialect*)
Set SQL Dialect for Database.

Parameters

- **database** (*string*) – Database filename or alias.
- **dialect** (*integer*) – *1* or *3*.

set_sweep_interval(*database*, *n*)
Set treshold for automatic sweep.

Parameters

- **database** (*string*) – Database filename or alias.
- **n** (*integer*) – Sweep treshold, or *0* to disable automatic sweep.

set_write_mode(*database*, *mode*)
Set Disk Write Mode: Sync (forced writes) or Async (buffered).

Parameters

- **database** (*string*) – Database filename or alias.

- **mode** (*integer*) – One from following constants: WRITE_FORCED or WRITE_BUFFERED

shutdown(*database*, *shutdown_mode*, *shutdown_method*, *timeout*)
Database shutdown.

Parameters

- **database** (*string*) – Database filename or alias.
- **shutdown_mode** (*integer*) – One from following constants: SHUT_LEGACY, SHUT_SINGLE, SHUT_MULTI or SHUT_FULL.
- **shutdown_method** (*integer*) – One from following constants: SHUT_FORCE, SHUT_DENY_NEW_TRANSACTIONS or SHUT_DENY_NEW_ATTACHMENTS.
- **timeout** (*integer*) – Time in seconds, that the shutdown must complete in.

See also:

See also *bring_online()* method.

sweep(*database*)
Perform Database Sweep.

Note: Method call will not return until sweep is finished.

Parameters **database** (*string*) – Database filename or alias.

trace_list()
Get information about existing trace sessions.

Returns dictionary Mapping *SESSION_ID -> SESSION_PARAMS*

Session parameters is another dictionary with next keys:

name (string) (optional) Session name if specified.

date (datetime.datetime) Session start date and time.

user (string) Trace user name.

flags (list of strings) Session flags.

Raises ***OperationalError*** – When server can't perform requested operation.

trace_resume(*trace_id*)
Resume trace session.

Parameters **trace_id** (*integer*) – Trace session ID.

Returns string Text with confirmation that session was resumed.

Raises

- ***DatabaseError*** – When trace session is not resumed.
- ***OperationalError*** – When server can't perform requested operation.

trace_start(*config*, *name=None*)
Start new trace session. **(ASYNC service)**

Parameters

- **config** (*string*) – Trace session configuration.

- **name** (*string*) – (optional) Trace session name.

Returns integer Trace session ID.

Raises ***fdb.DatabaseError*** – When session ID is not returned on start.

Trace session output could be retrieved through *readline()*, *readlines()*, iteration over *Connection* or ignored via call to *wait()*.

Note: Until session output is not fully fetched from service (or ignored via *wait()*), any attempt to start another asynchronous service including call to any *trace_* method will fail with exception.

---

trace_stop(*trace_id*)
Stop trace session.

Parameters **trace_id** (*integer*) – Trace session ID.

Returns string Text with confirmation that session was stopped.

Raises

- ***DatabaseError*** – When trace session is not stopped.
- ***OperationalError*** – When server can't perform requested operation.

trace_suspend(*trace_id*)
Suspend trace session.

Parameters **trace_id** (*integer*) – Trace session ID.

Returns string Text with confirmation that session was paused.

Raises

- ***DatabaseError*** – When trace session is not paused.
- ***OperationalError*** – When server can't perform requested operation.

user_exists(*user*)
Check for user's existence.

Parameters **user** (string or *User*) – User name or Instance of *User* with **at least** its `name` attribute specified as non-empty value.

Returns boolean *True* when the specified user exists.

validate(*database*, *include_tables=None*, *exclude_tables=None*, *include_indices=None*, *exclude_indices=None*, *lock_timeout=None*, *callback=None*)
On-line database validation.

Parameters

- **database** (*string*) – Database filename or alias.
- **include_tables** (*string*) – Pattern for table names to include in validation run.
- **exclude_tables** (*string*) – Pattern for table names to exclude from validation run.
- **include_indices** (*string*) – Pattern for index names to include in validation run.
- **exclude_indices** (*string*) – Pattern for index names to exclude from validation run.
- **lock_timeout** (*integer*) – lock timeout, used to acquire locks for table to validate, in seconds, default is 10 secs. 0 is no-wait, -1 is infinite wait.

- **callback** (*function*) – Function to call back with each output line. Function must accept only one parameter: line of output.

Note: Patterns are regular expressions, processed by the same rules as SIMILAR TO expressions. All patterns are case-sensitive, regardless of database dialect. If the pattern for tables is omitted then all user tables will be validated. If the pattern for indexes is omitted then all indexes of the appointed tables will be validated. System tables are not validated.

If *callback* is not specified, validation log could be retrieved through *readline()*, *readlines()*, iteration over *Connection* or ignored via call to *wait()*.

Note: Until validate report is not fully fetched from service (or ignored via *wait()*), any attempt to start another asynchronous service will fail with exception.

---

wait()
Wait until running service completes, i.e. stops sending data.

QUERY_TYPE_PLAIN_INTEGER = 1

QUERY_TYPE_PLAIN_STRING = 2

QUERY_TYPE_RAW = 3

closed

engine_version

fetching

version

User

class fdb.services.**User**(*name=None*)

load_information(*svc*)
Load information about user from server.

Parameters **svc** (*Connection*) – Open service connection.

Returns True if information was successfuly retrieved, False otherwise.

Raises ***ProgrammingError*** – If user name is not defined.

1.4.3 Database schema

Module globals

Firebird field type codes

- FBT_SMALLINT
- FBT_INTEGER
- FBT_QUAD
- FBT_FLOAT

- FBT_CHAR
- FBT_DOUBLE_PRECISION
- FBT_DATE
- FBT_VARCHAR
- FBT_CSTRING
- FBT_BLOB_ID
- FBT_BLOB
- FBT_SQL_DATE
- FBT_SQL_TIME
- FBT_SQL_TIMESTAMP
- FBT_BIGINT
- FBT_BOOLEAN

Trigger masks

- TRIGGER_TYPE_SHIFT
- TRIGGER_TYPE_MASK
- TRIGGER_TYPE_DML
- TRIGGER_TYPE_DB
- TRIGGER_TYPE_DDL

Trigger type codes

- DDL_TRIGGER_ANY
- DDL_TRIGGER_CREATE_TABLE
- DDL_TRIGGER_ALTER_TABLE
- DDL_TRIGGER_DROP_TABLE
- DDL_TRIGGER_CREATE_PROCEDURE
- DDL_TRIGGER_ALTER_PROCEDURE
- DDL_TRIGGER_DROP_PROCEDURE
- DDL_TRIGGER_CREATE_FUNCTION
- DDL_TRIGGER_ALTER_FUNCTION
- DDL_TRIGGER_DROP_FUNCTION
- DDL_TRIGGER_CREATE_TRIGGER
- DDL_TRIGGER_ALTER_TRIGGER
- DDL_TRIGGER_DROP_TRIGGER
- DDL_TRIGGER_CREATE_EXCEPTION
- DDL_TRIGGER_ALTER_EXCEPTION
- DDL_TRIGGER_DROP_EXCEPTION
- DDL_TRIGGER_CREATE_VIEW
- DDL_TRIGGER_ALTER_VIEW
- DDL_TRIGGER_DROP_VIEW
- DDL_TRIGGER_CREATE_DOMAIN
- DDL_TRIGGER_ALTER_DOMAIN
- DDL_TRIGGER_DROP_DOMAIN
- DDL_TRIGGER_CREATE_ROLE
- DDL_TRIGGER_ALTER_ROLE
- DDL_TRIGGER_DROP_ROLE
- DDL_TRIGGER_CREATE_INDEX
- DDL_TRIGGER_ALTER_INDEX
- DDL_TRIGGER_DROP_INDEX
- DDL_TRIGGER_CREATE_SEQUENCE

- DDL_TRIGGER_ALTER_SEQUENCE
- DDL_TRIGGER_DROP_SEQUENCE
- DDL_TRIGGER_CREATE_USER
- DDL_TRIGGER_ALTER_USER
- DDL_TRIGGER_DROP_USER
- DDL_TRIGGER_CREATE_COLLATION
- DDL_TRIGGER_DROP_COLLATION
- DDL_TRIGGER_ALTER_CHARACTER_SET
- DDL_TRIGGER_CREATE_PACKAGE
- DDL_TRIGGER_ALTER_PACKAGE
- DDL_TRIGGER_DROP_PACKAGE
- DDL_TRIGGER_CREATE_PACKAGE_BODY
- DDL_TRIGGER_DROP_PACKAGE_BODY
- DDL_TRIGGER_CREATE_MAPPING
- DDL_TRIGGER_ALTER_MAPPING
- DDL_TRIGGER_DROP_MAPPING

Collation parameters codes

- COLLATION_PAD_SPACE
- COLLATION_CASE_INSENSITIVE
- COLLATION_ACCENT_INSENSITIVE

Index type names

- INDEX_TYPE_ASCENDING
- INDEX_TYPE_DESCENDING

Relation type codes

- RELATION_TYPE_TABLE
- RELATION_TYPE_VIEW
- RELATION_TYPE_GTT
- RELATION_TYPE_GTT_PRESERVE
- RELATION_TYPE_GTT_DELETE

Procedure parameter type codes

- PROCPAR_DATATYPE
- PROCPAR_DOMAIN
- PROCPAR_TYPE_OF_DOMAIN
- PROCPAR_TYPE_OF_COLUMN

Section codes for `Schema.get_metadata_ddl()`

- SCRIPT_COLLATIONS
- SCRIPT_CHARACTER_SETS
- SCRIPT_UDFS
- SCRIPT_GENERATORS
- SCRIPT_EXCEPTIONS
- SCRIPT_DOMAINS
- SCRIPT_PACKAGE_DEFS
- SCRIPT_FUNCTION_DEFS
- SCRIPT_PROCEDURE_DEFS
- SCRIPT_TABLES
- SCRIPT_PRIMARY_KEYS
- SCRIPT_UNIQUE_CONSTRAINTS
- SCRIPT_CHECK_CONSTRAINTS
- SCRIPT_FOREIGN_CONSTRAINTS
- SCRIPT_INDICES
- SCRIPT_VIEWS
- SCRIPT_PACKAGE_BODIES
- SCRIPT_PROCEDURE_BODIES
- SCRIPT_FUNCTION_BODIES
- SCRIPT_TRIGGERS
- SCRIPT_ROLES
- SCRIPT_GRANTS
- SCRIPT_COMMENTS
- SCRIPT_SHADOWS
- SCRIPT_SET_GENERATORS
- SCRIPT_INDEX_DEACTIVATIONS
- SCRIPT_INDEX_ACTIVATIONS
- SCRIPT_TRIGGER_DEACTIVATIONS
- SCRIPT_TRIGGER_ACTIVATIONS

Lists and disctionary maps

COLUMN_TYPES Dictionary map from filed type codes to type names

INTEGRAL_SUBTYPES List of integral type names, works as map

INDEX_TYPES List of index types

BLOB_SUBTYPES List of blob type names, works as map

TRIGGER_PREFIX_TYPES List of trigger prefix type names, works as map

TRIGGER_SUFFIX_TYPES List of trigger suffix type names, works as map

TRIGGER_DB_TYPES List of db trigger type names, works as map

TRIGGER_DDL_TYPES List of DLL trigger type names, works as map

RESERVED List of reserved Firebird words

NON_RESERVED List of non-reserved Firebird words

SCRIPT_DEFAULT_ORDER List of default sections (in order) for *Schema.get_metadata_ddl()*

Functions

get_grants

fdb.schema.**get_grants**(*privileges*, *grantors=None*)
Get list of minimal set of SQL GRANT statamenets necessary to grant specified privileges.

Parameters

- **privileges** (*list*) – List of *Privilege* instances.
- **grantors** (*list*) – List of standard grantor names. Generates GRANTED BY clause for privileges granted by user that's not in list.

iskeyword

fdb.schema.**iskeyword**(*ident*)
Return True if *ident* is (any) Firebird keyword.

escape_single_quotes

fdb.schema.**escape_single_quotes**(*text*)
Return *text* with any single quotes escaped (doubled).

Classes

Schema

class fdb.schema.**Schema**
Bases: *fdb.utils.Visitable*

This class represents database schema.

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

bind(*connection*)
Bind this instance to specified *Connection*.

Parameters **connection** – *Connection* instance.

Raises ***ProgrammingError*** – If Schema object was set as internal (via `_set_as_internal()`).

clear()
Drop all cached metadata objects.

close()
Sever link to *Connection*.

Raises ***ProgrammingError*** – If Schema object was set as internal (via `_set_as_internal()`).

get_character_set(*name*)
Get *CharacterSet* by name.

Parameters **name** (*string*) – Character set name.

Returns *CharacterSet* with specified name or *None*.

get_character_set_by_id(*id*)
Get *CharacterSet* by ID.

Parameters **name** (*integer*) – CharacterSet ID.

Returns *CharacterSet* with specified ID or *None*.

get_collation(*name*)
Get *Collation* by name.

Parameters **name** (*string*) – Collation name.

Returns *Collation* with specified name or *None*.

get_collation_by_id(*charset_id*, *collation_id*)
Get *Collation* by ID.

Parameters

- **charset_id** (*integer*) – Character set ID.
- **collation_id** (*integer*) – Collation ID.

Returns *Collation* with specified ID or *None*.

get_constraint(*name*)
Get *Constraint* by name.

Parameters **name** (*string*) – Constraint name.

Returns *Constraint* with specified name or *None*.

get_domain(*name*)
Get *Domain* by name.

Parameters **name** (*string*) – Domain name.

Returns *Domain* with specified name or *None*.

get_exception(*name*)
Get *DatabaseException* by name.

Parameters **name** (*string*) – Exception name.

Returns *DatabaseException* with specified name or *None*.

get_function(*name*)
Get *Function* by name.

Parameters **name** (*string*) – Function name.

Returns *Function* with specified name or *None*.

get_generator(*name*)
Get *Sequence* by name.

Parameters **name** (*string*) – Sequence name.

Returns *Sequence* with specified name or *None*.

get_index(*name*)
Get *Index* by name.

Parameters **name** (*string*) – Index name.

Returns *Index* with specified name or *None*.

get_metadata_ddl(*sections=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 21, 23, 24, 25]*)
Return list of DDL SQL commands for creation of specified categories of database objects.

Parameters **sections** (*list*) – List of section identifiers.

Returns List with SQL commands.

Sections identifiers are represented by *SCRIPT_** contants defined in schema module.

Sections are created in the order of occerence in list. Uses *SCRIPT_DEFAULT_ORDER* list when sections are not specified.

get_package(*name*)
Get *Package* by name.

Parameters **name** (*string*) – Package name.

Returns *Package* with specified name or *None*.

get_privileges_of(*user*, *user_type=None*)
Get list of all privileges granted to user/database object.

Parameters

- **user** – User name or instance of class that represents possible user. Allowed classes are *User*, *Table*, *View*, *Procedure*, *Trigger* or *Role*.
- **user_type** (*int*) – **Required if** *user* is provided as string name. Numeric code for user type, see *Schema.enum_object_types*.

Returns List of *Privilege* objects.

Raises ***ProgrammingError*** – For unknown *user_type* code.

get_procedure(*name*)
Get *Procedure* by name.

Parameters **name** (*string*) – Procedure name.

Returns *Procedure* with specified name or *None*.

get_role(*name*)
Get *Role* by name.

Parameters **name** (*string*) – Role name.

Returns *Role* with specified name or *None*.

get_sequence(*name*)
Get *Sequence* by name.

Parameters **name** (*string*) – Sequence name.

Returns *Sequence* with specified name or *None*.

get_table(*name*)
Get *Table* by name.

Parameters **name** (*string*) – Table name.

Returns *Table* with specified name or *None*.

get_trigger(*name*)
Get *Trigger* by name.

Parameters **name** (*string*) – Trigger name.

Returns *Trigger* with specified name or *None*.

get_view(*name*)
Get *View* by name.

Parameters **name** (*string*) – View name.

Returns *View* with specified name or *None*.

ismultifile()
Returns true if database has multiple files.

reload(*data=None*)
Drop all or specified category of cached metadata objects, so they're reloaded from database on next reference.

Parameters **data** (*string*) – *None*, metadata category code or iterable with category codes.

Note: Category codes are defined by *SCHEMA_** globals.

Raises ***fdb.ProgrammingError*** – For undefined metadata category.

Note: Also commits query transaction.

backup_history
ObjectList of all nbackup hisotry records. Items are *BackupHistory* objects.

character_sets
ObjectList of all character sets in database. Items are *CharacterSet* objects.

closed

collations
ObjectList of all collations in database. Items are *Collation* objects.

constraints
ObjectList of all constraints in database. Items are *Constraint* objects.

default_character_set
Default *CharacterSet* for database

dependencies
ObjectList of all dependencies in database. Items are *Dependency* objects.

description
Database description or None if it doesn't have a description.

domains
ObjectList of all user domains in database. Items are *Domain* objects.

enum_character_set_names = {}

enum_determinism_flags = {}

enum_field_subtypes = {}

enum_field_types = {}

enum_function_types = {}

enum_grant_options = {}

enum_index_activity_flags = {}

enum_index_unique_flags = {}

enum_legacy_flags = {}

enum_mechanism_types = {}

enum_object_type_codes = {}

enum_object_types = {}

enum_page_types = {}

enum_param_type_from = {0: 'DATATYPE', 1: 'DOMAIN', 2: 'TYPE OF DOMAIN', 3: 'TYPE

enum_parameter_mechanism_types = {}

enum_parameter_types = {}

enum_privacy_flags = {}

enum_procedure_types = {}

enum_relation_types = {}

enum_system_flag_types = {}

enum_transaction_state_types = {}

enum_trigger_activity_flags = {}

enum_trigger_types = {}

exceptions
ObjectList of all exceptions in database. Items are *DatabaseException* objects.

files
ObjectList of all extension files defined for database. Items are *DatabaseFile* objects.

filters
ObjectList of all user-defined BLOB filters. Items are *Filter* objects.

functions
ObjectList of all user functions defined in database. Items are *Function* objects.

generators
ObjectList of all user generators in database. Items are *Sequence* objects.

indices
ObjectList of all user indices in database. Items are *Index* objects.

linger
Database linger value.

opt_always_quote = False

opt_generator_keyword = 'SEQUENCE'

owner_name
Database owner name.

packages
ObjectList of all packages defined for database. Items are *Package* objects.

privileges
ObjectList of all privileges defined for database. Items are *Privilege* objects.

procedures
ObjectList of all user procedures in database. Items are *Procedure* objects.

roles
ObjectList of all roles in database. Items are *Role* objects.

security_class
Can refer to the security class applied as databasewide access control limits.

sequences
ObjectList of all user generators in database. Items are *Sequence* objects.

shadows
ObjectList of all shadows defined for database. Items are *Shadow* objects.

sysdomains
ObjectList of all system domains in database. Items are *Domain* objects.

sysfunctions
ObjectList of all system functions defined in database. Items are *Function* objects.

sysgenerators
ObjectList of all system generators in database. Items are *Sequence* objects.

sysindices
ObjectList of all system indices in database. Items are *Index* objects.

sysprocedures
ObjectList of all system procedures in database. Items are *Procedure* objects.

syssequences
ObjectList of all system generators in database. Items are *Sequence* objects.

systables
ObjectList of all system tables in database. Items are *Table* objects.

systriggers
ObjectList of all system triggers in database. Items are *Trigger* objects.

sysviews
ObjectList of all system views in database. Items are *View* objects.

tables
ObjectList of all user tables in database. Items are *Table* objects.

triggers
ObjectList of all user triggers in database. Items are *Trigger* objects.

views
ObjectList of all user views in database. Items are *View* objects.

BaseSchemaItem

class fdb.schema.**BaseSchemaItem**(*schema*, *attributes*)
Bases: *fdb.utils.Visitable*

Base class for all database schema objects.

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

name
Database object name or None if object doesn't have a name.

schema = None

Collation

class fdb.schema.**Collation**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents collation.

Supported SQL actions:

- User collation: create, drop, comment
- System collation: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

isaccentinsensitive()
Returns True if collation has ACCENT INSENSITIVE attribute.

isbasedonexternal()
Returns True if collation is based on external collation definition.

iscaseinsensitive()
Returns True if collation has CASE INSENSITIVE attribute.

ispadded()
Returns True if collation has PAD SPACE attribute.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

attributes
Collation attributes.

base_collation
Base Collation object that's extended by this one or None.

character_set
Character set object associated with collation.

description
Database object description or None if object doesn't have a description.

function_name
Not currently used.

id
Collation ID.

name
Database object name or None if object doesn't have a name.

owner_name
Creator user name.

schema = None

security_class
Security class name or None.

specific_attributes
Collation specific attributes.

CharacterSet

class fdb.schema.**CharacterSet**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents character set.

Supported SQL actions: alter(collation=Collation instance or collation name), comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_collation(*name*)
Return *Collation* object with specified name that belongs to this character set.

get_collation_by_id(*id*)
Return *Collation* object with specified id that belongs to this character set.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

bytes_per_character
Size of characters in bytes.

collations
ObjectList of Collations associated with character set.

default_collate
Collate object of default collate.

description
Database object description or None if object doesn't have a description.

id
Character set ID.

name
Database object name or None if object doesn't have a name.

owner_name
Creator user name.

schema = None

security_class
Security class name or None.

DatabaseException

class fdb.schema.**DatabaseException**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents database exception.

Supported SQL actions:

- User exception: create, recreate, alter(message=string), create_or_alter, drop, comment
- System exception: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

id
System-assigned unique exception number.

message
Custom message text.

name
Database object name or None if object doesn't have a name.

owner_name
Creator user name.

schema = None

security_class
Security class name or None.

Sequence

class fdb.schema.**Sequence**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents database generator/sequence.

Supported SQL actions:

- User sequence: create, alter(value=number), drop, comment
- System sequence: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

isidentity()
Returns True for system generators created for IDENTITY columns.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

id
Internal ID number of the sequence.

increment
Sequence increment.

inital_value
Initial sequence value.

name
Database object name or None if object doesn't have a name.

owner_name
Creator user name.

schema = None

security_class
Security class name or None.

value
Current sequence value.

Index

class fdb.schema.**Index**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents database index.

Supported SQL actions:

- User index: create, activate, deactivate, recompute, drop, comment
- System index: activate, recompute, comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

isenforcer()
Returns True if index is used to enforce a constraint.

isexpression()
Returns True if index is expression index.

isinactive()
Returns True if index is INACTIVE.

issystemobject()
Returns True if this database object is system object.

isunique()
Returns True if index is UNIQUE.

actions
List of supported SQL operations on metadata object instance.

constraint
Constraint instance that uses this index or None.

description
Database object description or None if object doesn't have a description.

expression
Source of an expression or None.

id
Internal number ID of the index.

index_type
ASCENDING or DESCENDING.

name
Database object name or None if object doesn't have a name.

partner_index
Associated unique/primary key *Index* instance, or None.

schema = None

segment_names
List of index segment names.

segment_statistics
List of index segment statistics (for ODS 11.1 and higher).

segments
ObjectList of index segments as *TableColumn* instances.

statistics
Latest selectivity of the index.

table
The *Table* instance the index applies to.

TableColumn

class fdb.schema.**TableColumn**(*schema*, *table*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents table column.

Supported SQL actions:

- **User column: alter(name=string,datatype=string_SQLTypeDef,position=number,** expression=computed_by_expr,restart=None_or_init_value), drop, comment
- System column: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_computedby()
Returns (string) extression for column computation or None.

get_dependencies()
Return list of database objects that this object depend on.

get_dependents()
Return list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

has_default()
Returns True if column has default value.

iscomputed()
Returns True if column is computed.

isdomainbased()
Returns True if column is based on user domain.

isidentity()
Returns True for identity type column.

isnullable()
Returns True if column can accept NULL values.

issystemobject()
Returns True if this database object is system object.

iswritable()
Returns True if column is writable (i.e. it's not computed etc.).

actions
List of supported SQL operations on metadata object instance.

collation
Collation object or None.

datatype
Comlete SQL datatype definition.

default
Default value for column or None.

description
Database object description or None if object doesn't have a description.

domain
Domain object this column is based on.

generator
Internal flags.

id
Internam number ID for the column.

identity_type
Internal flags.

name
Database object name or None if object doesn't have a name.

position
Column's sequence number in row.

privileges
ObjectList of *Privilege* objects granted to this object.

schema = None

security_class
Security class name or None.

table
The Table object this column belongs to.

ViewColumn

class fdb.schema.**ViewColumn**(*schema*, *view*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents view column.

Supported SQL actions: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Return list of database objects that this object depend on.

get_dependents()
Return list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

isnullable()
Returns True if column is NULLABLE.

issystemobject()
Returns True if this database object is system object.

iswritable()
Returns True if column is writable.

actions
List of supported SQL operations on metadata object instance.

base_field
The source column from the base relation. Result could be either *TableColumn*, *ViewColumn* or *ProcedureParameter* instance or None.

collation
Collation object or None.

datatype
Comlete SQL datatype definition.

description
Database object description or None if object doesn't have a description.

domain
Domain object this column is based on.

name
Database object name or None if object doesn't have a name.

position
Column's sequence number in row.

privileges
ObjectList of *Privilege* objects granted to this object.

schema = None

security_class
Security class name or None.

view
View object this column belongs to.

Domain

class fdb.schema.**Domain**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents SQl Domain.

Supported SQL actions:

- User domain: create, alter(name=string,default=string_definition_or_None, check=string_definition_or_None,datatype=string_SQLTypeDef), drop, comment
- System domain: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

has_default()
Returns True if domain has default value.

isarray()
Returns True if domain defines an array.

iscomputed()
Returns True if domain is computed.

isnullable()
Returns True if domain is not defined with NOT NULL.

issystemobject()
Return True if this database object is system object.

isvalidated()
Returns True if domain has validation constraint.

actions
List of supported SQL operations on metadata object instance.

character_length
Length of CHAR and VARCHAR column, in characters (not bytes).

character_set
CharacterSet object for a character or text BLOB column, or None.

collation
Collation object for a character column or None.

datatype
Comlete SQL datatype definition.

default
Expression that defines the default value or None.

description
Database object description or None if object doesn't have a description.

dimensions
List of dimension definition pairs if column is an array type. Always empty for non-array columns.

expression
Expression that defines the COMPUTED BY column or None.

external_length
Length of field as it is in an external table. Always 0 for regular tables.

external_scale
Scale factor of an integer field as it is in an external table.

external_type
Data type of the field as it is in an external table.

field_type
Number code of the data type defined for the column.

length
Length of the column in bytes.

name
Database object name or None if object doesn't have a name.

owner_name
Creator user name.

precision
Indicates the number of digits of precision available to the data type of the column.

scale
Negative number representing the scale of NUMBER and DECIMAL column.

schema = None

security_class
Security class name or None.

segment_length
For BLOB columns, a suggested length for BLOB buffers.

sub_type
BLOB subtype.

validation
CHECK constraint for the domain or None.

Dependency

class fdb.schema.**Dependency**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Maps dependency between database objects.

Supported SQL actions: none

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns empty list because Dependency object never has dependencies.

get_dependents()
Returns empty list because Dependency object never has dependents.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

ispackaged()
Returns True if dependency is defined in package.

issystemobject()
Returns True as dependency entries are considered as system objects.

actions
List of supported SQL operations on metadata object instance.

depended_on
Database object on which dependent depends.

depended_on_name
Name of db object on which dependent depends.

depended_on_type
Type of db object on which dependent depends.

dependent
Dependent database object.

dependent_name
Dependent database object name.

dependent_type
Dependent database object type.

description
Database object description or None if object doesn't have a description.

field_name
Name of one column in *depended on* object.

name
Database object name or None if object doesn't have a name.

package
Package instance if dependent depends on object in package or None.

schema = None

Constraint

class fdb.schema.**Constraint**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents table or column constraint.

Supported SQL actions:

- Constraint on user table except NOT NULL constraint: create, drop
- Constraint on system table: none

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

ischeck()
Returns True if it's CHECK constraint.

isdeferrable()
Returns True if it's DEFERRABLE constraint.

isdeferred()
Returns True if it's INITIALLY DEFERRED constraint.

isfkey()
Returns True if it's FOREIGN KEY constraint.

isnotnull()
Returns True if it's NOT NULL constraint.

ispkey()
Returns True if it's PRIMARY KEY constraint.

issystemobject()
Returns True if this database object is system object.

isunique()
Returns True if it's UNIQUE constraint.

actions
List of supported SQL operations on metadata object instance.

column_name
For a NOT NULL constraint, this is the name of the column to which the constraint applies.

constraint_type
primary key/unique/foreign key/check/not null.

delete_rule
For a FOREIGN KEY constraint, this is the action applicable to when primary key is deleted.

description
Database object description or None if object doesn't have a description.

index
Index instance that enforces the constraint. *None* if constraint is not primary key/unique or foreign key.

match_option
For a FOREIGN KEY constraint only. Current value is FULL in all cases.

name
Database object name or None if object doesn't have a name.

partner_constraint
For a FOREIGN KEY constraint, this is the unique or primary key *Constraint* referred.

schema = None

table
Table instance this constraint applies to.

trigger_names
For a CHECK constraint contains trigger names that enforce the constraint.

triggers
For a CHECK constraint contains *Trigger* instances that enforce the constraint.

update_rule
For a FOREIGN KEY constraint, this is the action applicable to when primary key is updated.

Table

class fdb.schema.**Table**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents Table in database.

Supported SQL actions:

- **User table: create (no_pk=bool,no_unique=bool),** recreate (no_pk=bool,no_unique=bool), drop, comment
- System table: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_column(*name*)
Return *TableColumn* object with specified name.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

has_fkey()
Returns True if table has any FOREIGN KEY constraint.

has_pkey()
Returns True if table has PRIMARY KEY defined.

isexternal()
Returns True if table is external table.

isgtt()
Returns True if table is GLOBAL TEMPORARY table.

ispersistent()
Returns True if table is persistent one.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

columns
Returns *ObjectList* of columns defined for table. Items are *TableColumn* objects.

constraints
Returns *ObjectList* of constraints defined for table. Items are *Constraint* objects.

dbkey_length
Length of the RDB$DB_KEY column in bytes.

default_class
Default security class.

description
Database object description or None if object doesn't have a description.

external_file
Full path to the external data file, if any.

flags
Internal flags.

foreign_keys
ObjectList of FOREIGN KEY *Constraint* instances for this table.

format
Internal format ID for the table.

id
Internam number ID for the table.

indices
Returns *ObjectList* of indices defined for table. Items are *Index* objects.

name
Database object name or None if object doesn't have a name.

owner_name
User name of table's creator.

primary_key
PRIMARY KEY *Constraint* for this table or None.

privileges
ObjectList of *Privilege* objects granted to this object.

schema = None

security_class
Security class that define access limits to the table.

table_type
Table type.

triggers
Returns *ObjectList* of triggers defined for table. Items are *Trigger* objects.

View

class fdb.schema.**View**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents database View.

Supported SQL actions:

- User views: create, recreate, alter(columns=string_or_list,query=string,check=bool), create_or_alter, drop, comment
- System views: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_column(*name*)
Return *TableColumn* object with specified name.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

get_trigger(*name*)
Return *Trigger* object with specified name.

has_checkoption()
Returns True if View has WITH CHECK OPTION defined.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

columns
Returns *ObjectList* of columns defined for view. Items are *ViewColumn* objects.

dbkey_length
Length of the RDB$DB_KEY column in bytes.

default_class
Default security class.

description
Database object description or None if object doesn't have a description.

flags
Internal flags.

format
Internal format ID for the view.

id
Internal number ID for the view.

name
Database object name or None if object doesn't have a name.

owner_name
User name of view's creator.

privileges
ObjectList of *Privilege* objects granted to this object.

schema = None

security_class
Security class that define access limits to the view.

sql
The query specification.

triggers
Returns *ObjectList* of triggers defined for view. Items are *Trigger* objects.

Trigger

class fdb.schema.**Trigger**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents trigger.

Supported SQL actions:

- User trigger: create(inactive=bool), recreate, create_or_alter, drop, alter(fire_on=string,active=bool,sequence=int,declare=string_or_list, code=string_or_list), comment
- System trigger: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

get_type_as_string()
Return string with action and operation specification.

isactive()
Returns True if this trigger is active.

isafter()
Returns True if this trigger is set for AFTER action.

isbefore()
Returns True if this trigger is set for BEFORE action.

isdbtrigger()
Returns True if this trigger is database trigger.

isddltrigger()
Returns True if this trigger is DDL trigger.

isdelete()
Returns True if this trigger is set for DELETE operation.

isinsert()
Returns True if this trigger is set for INSERT operation.

issystemobject()
Returns True if this database object is system object.

isupdate()
Returns True if this trigger is set for UPDATE operation.

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

engine_name
Engine name.

entrypoint
Entrypoint.

flags
Internal flags.

name
Database object name or None if object doesn't have a name.

relation
Table or *View* that the trigger is for, or None for database triggers

schema = None

sequence
Sequence (position) of trigger. Zero usually means no sequence defined.

source
PSQL source code.

trigger_type
Numeric code for trigger type that define what event and when are covered by trigger.

valid_blr
Trigger BLR invalidation flag. Coul be True/False or None.

ProcedureParameter

class fdb.schema.**ProcedureParameter**(*schema*, *proc*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents procedure parameter.

Supported SQL actions: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_definition()
Returns SQL definition for parameter.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

has_default()
Returns True if parameter has default value.

isinput()
Returns True if parameter is INPUT parameter.

isnullable()
Returns True if parameter allows NULL.

ispackaged()
Returns True if procedure parameter is defined in package.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

collation
collation for this parameter.

column
TableColumn for this parameter.

datatype
Comlete SQL datatype definition.

default
Default value.

description
Database object description or None if object doesn't have a description.

domain
Domain for this parameter.

mechanism
Parameter mechanism code.

name
Database object name or None if object doesn't have a name.

package
Package this procedure belongs to. Object is *Package* instance or None.

procedure
Name of the stored procedure.

schema = None

sequence
Sequence (position) of parameter.

type_from
Numeric code. See *Schema.enum_param_type_from*.‘

Procedure

class fdb.schema.**Procedure**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents stored procedure.

Supported SQL actions:

- User procedure: create(no_code=bool), recreate(no_code=bool), create_or_alter(no_code=bool), drop, alter(input=string_or_list,output=string_or_list,declare=string_or_list, code=string_or_list), comment
- System procedure: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_param(*name*)
Returns *ProcedureParameter* with specified name or None

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

has_input()
Returns True if procedure has any input parameters.

has_output()
Returns True if procedure has any output parameters.

ispackaged()
Returns True if procedure is defined in package.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

engine_name
Engine name.

entrypoint
Entrypoint.

id
Internal unique ID number.

input_params
ObjectList of input parameters. Instances are *ProcedureParameter* instances.

name
Database object name or None if object doesn't have a name.

output_params
ObjectList of output parameters. Instances are *ProcedureParameter* instances.

owner_name
User name of procedure's creator.

package
Package this procedure belongs to. Object is *Package* instance or None.

privacy
Privacy flag.

privileges
ObjectList of *Privilege* objects granted to this object.

proc_type
Procedure type code. See *enum_procedure_types*.

schema = None

security_class
Security class that define access limits to the procedure.

source
PSQL source code.

valid_blr
Procedure BLR invalidation flag. Coul be True/False or None.

Role

class fdb.schema.**Role**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents user role.

Supported SQL actions:

- User role: create, drop, comment
- System role: comment

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

name
Database object name or None if object doesn't have a name.

owner_name
User name of role owner.

privileges
ObjectList of *Privilege* objects granted to this object.

schema = None

security_class
Security class name or None.

FunctionArgument

class fdb.schema.**FunctionArgument**(*schema*, *function*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represets UDF argument.

Supported SQL actions: none.

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_definition()
Returns SQL definition for parameter.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

has_default()
Returns True if parameter has default value.

isbydescriptor(*any=False*)
Returns True if argument is passed by descriptor.

Parameters **any** (*bool*) – If True, method returns True if any kind of descriptor is used (including BLOB and ARRAY descriptors).

isbyreference()
Returns True if argument is passed by reference.

isbyvalue()
Returns True if argument is passed by value.

isfreeit()
Returns True if (return) argument is declared as FREE_IT.

isnullable()
Returns True if parameter allows NULL.

ispackaged()
Returns True if function argument is defined in package.

isreturning()
Returns True if argument represents return value for function.

issystemobject()
Returns True if this database object is system object.

iswithnull()
Returns True if argument is passed by reference with NULL support.

actions
List of supported SQL operations on metadata object instance.

argument_mechanism
Argument mechanism.

argument_name
Argument name.

character_length
Length of CHAR and VARCHAR column, in characters (not bytes).

character_set
CharacterSet for a character/text BLOB argument, or None.

collation
Collation for this parameter.

column
TableColumn for this parameter.

datatype
Comlete SQL datatype definition.

default
Default value.

description
Database object description or None if object doesn't have a description.

domain
Domain for this parameter.

field_type
Number code of the data type defined for the argument.

function
Function to which this argument belongs.

length
Length of the argument in bytes.

mechanism
How argument is passed.

name
Database object name or None if object doesn't have a name.

package
Package this function belongs to. Object is *Package* instance or None.

position
Argument position.

precision
Indicates the number of digits of precision available to the data type of the argument.

scale
Negative number representing the scale of NUMBER and DECIMAL argument.

schema = None

sub_type
BLOB subtype.

type_from
Numeric code. See *Schema.enum_param_type_from*.‘

Function

class fdb.schema.**Function**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents user defined function.

Supported SQL actions:

- External UDF: declare, drop, comment
- **PSQL UDF (FB 3, not declared in package): create(no_code=bool),** recreate(no_code=bool), create_or_alter(no_code=bool), drop, alter(arguments=string_or_list,returns=string,declare=string_or_list, code=string_or_list)
- System UDF: none

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

has_arguments()
Returns True if function has input arguments.

has_return()
Returns True if function returns a value.

has_return_argument()
Returns True if function returns a value in input argument.

isexternal()
Returns True if function is external UDF, False for PSQL functions.

ispackaged()
Returns True if function is defined in package.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

arguments
ObjectList of function arguments. Items are *FunctionArgument* instances.

description
Database object description or None if object doesn't have a description.

deterministic_flag
Deterministic flag.

engine_mame
Engine name.

entrypoint
Entrypoint in module.

id
Function ID.

legacy_flag
Legacy flag.

module_name
Module name.

name
Database object name or None if object doesn't have a name.

owner_name
Owner name.

package
Package this function belongs to. Object is *Package* instance or None.

private_flag
Private flag.

returns
Returning *FunctionArgument* or None.

schema = None

security_class
Security class.

source
Function source.

valid_blr
BLR validity flag.

DatabaseFile

class fdb.schema.**DatabaseFile**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents database extension file.

Supported SQL actions: create

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

issystemobject()
Returns True.

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

filename
File name.

length
File length in pages.

name
Database object name or None if object doesn't have a name.

schema = None

sequence
File sequence number.

start
File start page number.

Shadow

class fdb.schema.**Shadow**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents database shadow.

Supported SQL actions: create, drop(preserve=bool)

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

isconditional()
Returns True if it's CONDITIONAL shadow.

isinactive()
Returns True if it's INACTIVE shadow.

ismanual()
Returns True if it's MANUAL shadow.

issystemobject()
Returns False.

SHADOW_CONDITIONAL = 16

SHADOW_INACTIVE = 2

SHADOW_MANUAL = 4

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

files
List of shadow files. Items are *DatabaseFile* instances.

flags
Shadow flags.

id
Shadow ID number.

name
Database object name or None if object doesn't have a name.

schema = None

Privilege

class fdb.schema.**Privilege**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents priviledge to database object.

Supported SQL actions: grant(grantors),revoke(grantors,grant_option)

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

has_grant()
Returns True if privilege comes with GRANT OPTION.

isdelete()
Returns True if this is DELETE privilege.

isexecute()
Returns True if this is EXECUTE privilege.

isinsert()
Returns True if this is INSERT privilege.

ismembership()
Returns True if this is ROLE membership privilege.

isreference()
Returns True if this is REFERENCE privilege.

isselect()
Returns True if this is SELECT privilege.

issystemobject()
Returns True.

isupdate()
Returns True if this is UPDATE privilege.

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

field_name
Field name.

grantor
Grantor *User* object.

grantor_name
Grantor name.

name
Database object name or None if object doesn't have a name.

privilege
Privilege code.

schema = None

subject
Priviledge subject. Either *Role*, *Table*, *View* or *Procedure* object.

subject_name
Subject name.

subject_type
Subject type.

user
Grantee. Either *User*, *Role*, *Procedure*, *Trigger* or *View* object.

user_name
User name.

user_type
User type.

Package

class fdb.schema.**Package**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents PSQL package.

Supported SQL actions: create(body=bool), recreate(body=bool), create_or_alter(body=bool),
alter(header=string_or_list), drop(body=bool), alter

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

has_valid_body()

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

body
Package body source.

description
Database object description or None if object doesn't have a description.

functions
ObjectList of package functions. Items are *Function* instances.

header
Package header source.

name
Database object name or None if object doesn't have a name.

owner_name
User name of package creator.

procedures
ObjectList of package procedures. Items are *Procedure* instances.

schema = None

security_class
Security class name or None.

BackupHistory

class fdb.schema.**BackupHistory**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents entry of history for backups performed using the nBackup utility.

Supported SQL actions: None

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

issystemobject()
Returns True.

actions
List of supported SQL operations on metadata object instance.

backup_id
The identifier assigned by the engine.

created
Backup date and time.

description
Database object description or None if object doesn't have a description.

filename
Full path and file name of backup file.

guid
Unique identifier.

level
Backup level.

name
Database object name or None if object doesn't have a name.

schema = None

scn
System (scan) number.

Filter

class fdb.schema.**Filter**(*schema*, *attributes*)
Bases: *fdb.schema.BaseSchemaItem*

Represents userdefined BLOB filter.

Supported SQL actions:

- BLOB filter: declare, drop, comment

- System UDF: none

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

get_dependencies()
Returns list of database objects that this object depend on.

get_dependents()
Returns list of all database objects that depend on this one.

get_quoted_name()
Returns quoted (if necessary) name.

get_sql_for(*action*, ***params*)
Returns SQL command for specified action on metadata object.

Supported actions are defined by *actions* list.

Raises ***ProgrammingError*** – For unsupported action or wrong parameters passed.

issystemobject()
Returns True if this database object is system object.

actions
List of supported SQL operations on metadata object instance.

description
Database object description or None if object doesn't have a description.

entrypoint
The exported name of the BLOB filter in the filter library.

input_sub_type
The BLOB subtype of the data to be converted by the function.

module_name
The name of the dynamic library or shared object where the code of the BLOB filter is located.

name
Database object name or None if object doesn't have a name.

output_sub_type
The BLOB subtype of the converted data.

schema = None

1.4.4 Monitoring information

Constants

Shutdown modes for `DatabaseInfo.shutdown_mode`

- SHUTDOWN_MODE_ONLINE
- SHUTDOWN_MODE_MULTI
- SHUTDOWN_MODE_SINGLE
- SHUTDOWN_MODE_FULL

Backup states for `DatabaseInfo.backup_state`

- BACKUP_STATE_NORMAL
- BACKUP_STATE_STALLED
- BACKUP_STATE_MERGE

States for `AttachmentInfo.state`, `TransactionInfo.state` and `StatementInfo.state`

- STATE_IDLE
- STATE_ACTIVE

Isolation modes for `TransactionInfo.isolation_mode`

- ISOLATION_CONSISTENCY
- ISOLATION_CONCURRENCY
- ISOLATION_READ_COMMITTED_RV
- ISOLATION_READ_COMMITTED_NO_RV

Special timeout values for `TransactionInfo.lock_timeout`

- INFINITE_WAIT
- NO_WAIT

Group codes for `IOStatsInfo.group`

- STAT_DATABASE
- STAT_ATTACHMENT
- STAT_TRANSACTION
- STAT_STATEMENT
- STAT_CALL

Security database

- SEC_DEFAULT
- SEC_SELF
- SEC_OTHER

Classes

Monitor

class fdb.monitor.**Monitor**
Class for access to Firebird monitoring tables.

bind(*connection*)
Bind this instance to specified *Connection*.

Parameters **connection** – *Connection* instance.

Raises ***fdb.ProgrammingError*** – If Monitor object was set as internal (via `_set_as_internal()`) or database has ODS lower than 11.1.

clear()
Drop all cached information objects. Force reload of fresh monitoring information on next reference.

close()
Sever link to *Connection*.

Raises ***fdb.ProgrammingError*** – If Monitor object was set as internal (via `_set_as_internal()`).

get_attachment(*id*)
Get *AttachmentInfo* by ID.

Parameters **id** (*int*) – Attachment ID.

Returns *AttachmentInfo* with specified ID or *None*.

get_call(*id*)
Get *CallStackInfo* by ID.

Parameters **id** (*int*) – Callstack ID.

Returns *CallStackInfo* with specified ID or *None*.

get_statement(*id*)
Get *StatementInfo* by ID.

Parameters **id** (*int*) – Statement ID.

Returns *StatementInfo* with specified ID or *None*.

get_transaction(*id*)
Get *TransactionInfo* by ID.

Parameters **id** (*int*) – Transaction ID.

Returns *TransactionInfo* with specified ID or *None*.

refresh()
Reloads fresh monitoring information.

attachments
ObjectList of all attachments. Items are *AttachmentInfo* objects.

callstack
ObjectList with complete call stack. Items are *CallStackInfo* objects.

closed

db
DatabaseInfo object for attached database.

iostats
ObjectList of all I/O statistics. Items are *IOStatsInfo* objects.

statements
ObjectList of all statements. Items are *StatementInfo* objects.

tablestats
ObjectList of all table record I/O statistics. Items are *TableStatsInfo* objects.

this_attachment
AttachmentInfo object for current connection.

transactions
ObjectList of all transactions. Items are *TransactionInfo* objects.

variables
ObjectList of all context variables. Items are *ContextVariableInfo* objects.

BaseInfoItem

class fdb.monitor.**BaseInfoItem**(*monitor*, *attributes*)
Bases: object

Base class for all database monitoring objects.

monitor = None

stat_id
Internal ID.

DatabaseInfo

class fdb.monitor.**DatabaseInfo**(*monitor*, *attributes*)
Bases: *fdb.monitor.BaseInfoItem*

Information about attached database.

backup_state
Current state of database with respect to nbackup physical backup.

cache_size
Number of pages allocated in the page cache.

created
Creation date and time, i.e., when the database was created or last restored.

crypt_page
Number of page being encrypted.

forced_writes
True if database uses synchronous writes.

iostats
IOStatsInfo for this object.

monitor = None

name
Database pathname or alias.

next_transaction
Transaction ID of the next transaction that will be started.

oat
Transaction ID of the oldest active transaction.

ods
On-Disk Structure (ODS) version number.

oit
Transaction ID of the oldest [interesting] transaction.

ost
Transaction ID of the Oldest Snapshot, i.e., the number of the OAT when the last garbage collection was done.

owner
User name of database owner.

page_size
Size of database page in bytes.

pages
Number of pages allocated on disk.

read_only
True if database is Read Only.

reserve_space
True if database reserves space on data pages.

security_database
Type of security database (Default, Self or Other).

shutdown_mode
Current shutdown mode.

sql_dialect
SQL dialect of the database.

stat_id
Internal ID.

sweep_interval
The sweep interval configured in the database header. Value 0 indicates that sweeping is disabled.

tablestats
Dictionary of *TableStatsInfo* instances for this object.

AttachmentInfo

class fdb.monitor.**AttachmentInfo**(*monitor*, *attributes*)
Bases: *fdb.monitor.BaseInfoItem*

Information about attachment (connection) to database.

isactive()
Returns True if attachment is active.

isgcallowed()
Returns True if Garbage Collection is enabled for this attachment.

isidle()
Returns True if attachment is idle.

isinternal()
Returns True if attachment is internal system attachment.

terminate()
Terminates client session associated with this attachment.

Raises *fdb.ProgrammingError* – If database has ODS lower than 11.2 or this attachement is current session.

auth_method
Authentication method.

character_set
CharacterSet for this attachment.

client_version
Client library version.

id
Attachment ID.

iostats
IOStatsInfo for this object.

monitor = None

name
Database pathname or alias.

remote_address
Remote address.

remote_host
Name of remote host.

remote_os_user
OS user name of client process.

remote_pid
Remote client process ID.

remote_process
Remote client process pathname.

remote_protocol
Remote protocol name.

remote_version
Remote protocol version.

role
Role name.

server_pid
Server process ID.

stat_id
Internal ID.

state
Attachment state (idle/active).

statements
ObjectList of statements associated with attachment. Items are *StatementInfo* objects.

system
True for system attachments.

tablestats
Dictionary of *TableStatsInfo* instances for this object.

timestamp
Attachment date/time.

transactions
ObjectList of transactions associated with attachment. Items are *TransactionInfo* objects.

user
User name.

variables
ObjectList of variables associated with attachment. Items are *ContextVariableInfo* objects.

TransactionInfo

class fdb.monitor.**TransactionInfo**(*monitor*, *attributes*)
Bases: *fdb.monitor.BaseInfoItem*

Information about transaction.

isactive()
Returns True if transaction is active.

isautocommit()
Returns True for autocommited transaction.

isautoundo()
Returns True for transaction with automatic undo.

isidle()
Returns True if transaction is idle.

isreadonly()
Returns True if transaction is Read Only.

attachment
AttachmentInfo instance to which this transaction belongs.

id
Transaction ID.

iostats
IOStatsInfo for this object.

isolation_mode
Transaction isolation mode code.

lock_timeout
Lock timeout.

monitor = None

oldest
Oldest transaction (local OIT).

oldest_active
Oldest active transaction (local OAT).

stat_id
Internal ID.

state
Transaction state (idle/active).

statements
ObjectList of statements associated with transaction. Items are *StatementInfo* objects.

tablestats
Dictionary of *TableStatsInfo* instances for this object.

timestamp
Transaction start date/time.

top
Top transaction.

variables
ObjectList of variables associated with transaction. Items are *ContextVariableInfo* objects.

StatementInfo

class fdb.monitor.**StatementInfo**(*monitor*, *attributes*)
Bases: *fdb.monitor.BaseInfoItem*

Information about executed SQL statement.

isactive()
Returns True if statement is active.

isidle()
Returns True if statement is idle.

terminate()
Terminates execution of statement.

Raises ***fdb.ProgrammingError*** – If this attachement is current session.

attachment
AttachmentInfo instance to which this statement belongs.

callstack
ObjectList with call stack for statement. Items are *CallStackInfo* objects.

id
Statement ID.

iostats
IOStatsInfo for this object.

monitor = None

plan
Explained execution plan.

sql_text
Statement text, if appropriate.

stat_id
Internal ID.

state
Statement state (idle/active).

tablestats
Dictionary of *TableStatsInfo* instances for this object.

timestamp
Statement start date/time.

transaction
TransactionInfo instance to which this statement belongs or None.

CallStackInfo

class fdb.monitor.**CallStackInfo**(*monitor*, *attributes*)
Bases: *fdb.monitor.BaseInfoItem*

Information about PSQL call (stack frame).

caller
Call stack entry (*CallStackInfo*) of the caller.

column
SQL source column number.

dbobject
PSQL object. *Procedure* or *Trigger* instance.

id
Call ID.

iostats
IOStatsInfo for this object.

line
SQL source line number.

monitor = None

package_name
Package name.

stat_id
Internal ID.

statement
Top-level *StatementInfo* instance to which this call stack entry belongs.

timestamp
Request start date/time.

IOStatsInfo

class fdb.monitor.**IOStatsInfo**(*monitor*, *attributes*)
Bases: *fdb.monitor.BaseInfoItem*

Information about page and row level I/O operations, and about memory consumption.

backouts
Number of records where a new primary record version or a change to an existing primary record version is backed out due to rollback or savepoint undo.

backversion_reads
Number of record backversion reads.

conflits
Number of record conflits.

deletes
Number of deleted records.

expunges
Number of records where record version chain is being deleted due to deletions by transactions older than OAT.

fetches
Number of page fetches.

fragment_reads
Number of record fragment reads.

group
Object group code.

idx_reads
Number of records read via an index.

inserts
Number of inserted records.

locks
Number of record locks.

marks
Number of pages with changes pending.

max_memory_allocated
Maximum number of bytes allocated from the operating system by this object.

max_memory_used
Maximum number of bytes used by this object.

memory_allocated
Number of bytes currently allocated at the OS level.

memory_used
Number of bytes currently in use.

monitor = None

owner
Object that owns this IOStats instance. Could be either *DatabaseInfo*, *AttachmentInfo*, *TransactionInfo*, *StatementInfo* or *CallStackInfo* instance.

purges
Number of records where record version chain is being purged of versions no longer needed by OAT or younger transactions.

reads
Number of page reads.

repeated_reads
Number of repeated record reads.

seq_reads
Number of records read sequentially.

stat_id
Internal ID.

updates
Number of updated records.

waits
Number of record waits.

writes
Number of page writes.

TableStatsInfo

class fdb.monitor.**TableStatsInfo**(*monitor*, *attributes*)
Bases: *fdb.monitor.BaseInfoItem*

Information about row level I/O operations on single table.

backouts
Number of records where a new primary record version or a change to an existing primary record version is backed out due to rollback or savepoint undo.

backversion_reads
Number of record backversion reads.

conflits
Number of record conflits.

deletes
Number of deleted records.

expunges
Number of records where record version chain is being deleted due to deletions by transactions older than OAT.

fragment_reads
Number of record fragment reads.

group
Object group code.

idx_reads
Number of records read via an index.

inserts
Number of inserted records.

locks
Number of record locks.

monitor = None

owner
Object that owns this TableStats instance. Could be either *DatabaseInfo*, *AttachmentInfo*, *TransactionInfo*, *StatementInfo* or *CallStackInfo* instance.

purges
Number of records where record version chain is being purged of versions no longer needed by OAT or younger transactions.

repeated_reads
Number of repeated record reads.

row_stat_id
Internal ID.

seq_reads
Number of records read sequentially.

stat_id
Internal ID.

table_name
Table name.

updates
Number of updated records.

waits
Number of record waits.

ContextVariableInfo

class fdb.monitor.**ContextVariableInfo**(*monitor*, *attributes*)
Bases: *fdb.monitor.BaseInfoItem*

Information about context variable.

isattachmentvar()
Returns True if variable is associated to attachment context.

istransactionvar()
Returns True if variable is associated to transaction context.

attachment
AttachmentInfo instance to which this context variable belongs or None.

monitor = None

name
Context variable name.

stat_id
Internal ID.

transaction
TransactionInfo instance to which this context variable belongs or None.

value
Value of context variable.

1.4.5 Firebird trace & audit

Constants

Trace event status codes

- STATUS_OK
- STATUS_FAILED
- STATUS_UNAUTHORIZED
- STATUS_UNKNOWN

Trace event codes

- EVENT_TRACE_INIT
- EVENT_TRACE_SUSPEND
- EVENT_TRACE_END
- EVENT_CREATE_DATABASE
- EVENT_DROP_DATABASE
- EVENT_ATTACH
- EVENT_DETACH
- EVENT_TRANSACTION_START
- EVENT_COMMIT
- EVENT_ROLLBACK
- EVENT_COMMIT_R
- EVENT_ROLLBACK_R
- EVENT_STMT_PREPARE
- EVENT_STMT_START
- EVENT_STMT_END
- EVENT_STMT_FREE
- EVENT_STMT_CLOSE
- EVENT_TRG_START
- EVENT_TRG_END
- EVENT_PROC_START
- EVENT_PROC_END
- EVENT_SVC_START
- EVENT_SVC_ATTACH
- EVENT_SVC_DETACH
- EVENT_SVC_QUERY
- EVENT_SET_CONTEXT
- EVENT_ERROR
- EVENT_WARNING
- EVENT_SWEEP_START
- EVENT_SWEEP_PROGRESS
- EVENT_SWEEP_FINISH
- EVENT_SWEEP_FAILED
- EVENT_BLR_COMPILE
- EVENT_BLR_EXECUTE
- EVENT_DYN_EXECUTE
- EVENT_UNKNOWN

EVENTS List of trace event names in order matching their numeric codes

Classes

Named tuples for information packages

fdb.trace.**AttachmentInfo**(*attachment_id*, *database*, *charset*, *protocol*, *address*, *user*, *role*, *remote_process*, *remote_pid*)

fdb.trace.**TransactionInfo**(*attachment_id*, *transaction_id*, *options*)

fdb.trace.**ServiceInfo**(*service_id*, *user*, *protocol*, *address*, *remote_process*, *remote_pid*)

fdb.trace.**SQLInfo**(*sql_id*, *sql*, *plan*)

fdb.trace.**ParamInfo**(*par_id*, *params*)

fdb.trace.**AccessTuple**(*table*, *natural*, *index*, *update*, *insert*, *delete*, *backout*, *purge*, *expunge*)

Named tuples for individual trace events

fdb.trace.**EventTraceInit**(*event_id*, *timestamp*, *session_name*)

fdb.trace.**EventTraceSuspend**(*event_id*, *timestamp*, *session_name*)

fdb.trace.**EventTraceFinish**(*event_id*, *timestamp*, *session_name*)

fdb.trace.**EventCreate**(*event_id*, *timestamp*, *status*, *attachment_id*, *database*, *charset*, *protocol*, *address*, *user*, *role*, *remote_process*, *remote_pid*)

fdb.trace.**EventDrop**(*event_id*, *timestamp*, *status*, *attachment_id*, *database*, *charset*, *protocol*, *address*, *user*, *role*, *remote_process*, *remote_pid*)

fdb.trace.**EventAttach**(*event_id*, *timestamp*, *status*, *attachment_id*, *database*, *charset*, *protocol*, *address*, *user*, *role*, *remote_process*, *remote_pid*)

fdb.trace.**EventDetach**(*event_id*, *timestamp*, *status*, *attachment_id*, *database*, *charset*, *protocol*, *address*, *user*, *role*, *remote_process*, *remote_pid*)

fdb.trace.**EventTransactionStart**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *options*)

fdb.trace.**EventCommit**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *options*, *run_time*, *reads*, *writes*, *fetches*, *marks*)

fdb.trace.**EventRollback**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *options*, *run_time*, *reads*, *writes*, *fetches*, *marks*)

fdb.trace.**EventCommitRetaining**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *options*, *run_time*, *reads*, *writes*, *fetches*, *marks*)

fdb.trace.**EventRollbackRetaining**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *options*, *run_time*, *reads*, *writes*, *fetches*, *marks*)

fdb.trace.**EventPrepareStatement**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *statement_id*, *sql_id*, *prepare_time*)

fdb.trace.**EventStatementStart**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *statement_id*, *sql_id*, *param_id*)

fdb.trace.**EventStatementFinish**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *statement_id*, *sql_id*, *param_id*, *records*, *run_time*, *reads*, *writes*, *fetches*, *marks*, *access*)

fdb.trace.**EventFreeStatement**(*event_id*, *timestamp*, *attachment_id*, *transaction_id*, *statement_id*, *sql_id*)

fdb.trace.**EventCloseCursor**(*event_id*, *timestamp*, *attachment_id*, *transaction_id*, *statement_id*, *sql_id*)

fdb.trace.**EventTriggerStart**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *trigger*, *table*, *event*)

fdb.trace.**EventTriggerFinish**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *trigger*, *table*, *event*, *run_time*, *reads*, *writes*, *fetches*, *marks*, *access*)

fdb.trace.**EventProcedureStart**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *procedure*, *param_id*)

fdb.trace.**EventProcedureFinish**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *procedure*, *param_id*, *run_time*, *reads*, *writes*, *fetches*, *marks*, *access*)

fdb.trace.**EventServiceAttach**(*event_id*, *timestamp*, *status*, *service_id*)

fdb.trace.**EventServiceDetach**(*event_id*, *timestamp*, *status*, *service_id*)

fdb.trace.**EventServiceStart**(*event_id*, *timestamp*, *status*, *service_id*, *action*, *parameters*)

fdb.trace.**EventServiceQuery**(*event_id*, *timestamp*, *status*, *service_id*, *action*, *parameters*)

fdb.trace.**EventSetContext**(*event_id*, *timestamp*, *attachment_id*, *transaction_id*, *context*, *key*, *value*)

fdb.trace.**EventError**(*event_id*, *timestamp*, *attachment_id*, *place*, *details*)

fdb.trace.**EventServiceError**(*event_id*, *timestamp*, *service_id*, *place*, *details*)

fdb.trace.**EventWarning**(*event_id*, *timestamp*, *attachment_id*, *place*, *details*)

fdb.trace.**EventServiceWarning**(*event_id*, *timestamp*, *service_id*, *place*, *details*)

fdb.trace.**EventSweepStart**(*event_id*, *timestamp*, *attachment_id*, *oit*, *oat*, *ost*, *next*)

fdb.trace.**EventSweepProgress**(*event_id*, *timestamp*, *attachment_id*, *run_time*, *reads*, *writes*, *fetches*, *marks*, *access*)

fdb.trace.**EventSweepFinish**(*event_id*, *timestamp*, *attachment_id*, *oit*, *oat*, *ost*, *next*, *run_time*, *reads*, *writes*, *fetches*, *marks*)

fdb.trace.**EventSweepFailed**(*event_id*, *timestamp*, *attachment_id*)

fdb.trace.**EventBLRCompile**(*event_id*, *timestamp*, *status*, *attachment_id*, *statement_id*, *content*, *prepare_time*)

fdb.trace.**EventBLRExecute**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *statement_id*, *content*, *run_time*, *reads*, *writes*, *fetches*, *marks*, *access*)

fdb.trace.**EventDYNExecute**(*event_id*, *timestamp*, *status*, *attachment_id*, *transaction_id*, *content*, *run_time*)

fdb.trace.**EventUnknown**(*event_id*, *timestamp*, *data*)

TraceParser

class fdb.trace.**TraceParser**

Parser for standard textual trace log. Produces named tuples describing individual trace log entries/events.

Attributes:

Seen_attachments Set of attachment ids that were already processed.

Seen_transactions Set of transaction ids that were already processed.

Seen_services Set of service ids that were already processed.

Sqlinfo_map Dictionary that maps (sql_cmd,plan) keys to internal ids

Param_map Dictionary that maps parameters (statement or procedure) keys to internal ids

Next_event_id Sequence id that would be assigned to next parsed event (starts with 1).

Next_sql_id Sequence id that would be assigned to next parsed unique SQL command (starts with 1).

Next_param_id Sequence id that would be assigned to next parsed unique parameter (starts with 1).

parse(*lines*)

Parse output from Firebird trace session and yield named tuples describing individual trace log entries/events.

Parameters **lines** – Iterable that return lines produced by firebird trace session.

Raises ***ParseError*** – When any problem is found in input stream.

parse_event(*trace_block*)

Parse single trace event.

Parameters **trace_block** (*list*) – List with trace entry lines for single trace event.

Returns Named tuple with parsed event.

1.4.6 GSTAT protocols

Module globals

GSTAT version

- GSTAT_25
- GSTAT_30

Database attribute codes

- ATTR_FORCE_WRITE
- ATTR_NO_RESERVE
- ATTR_NO_SHARED_CACHE
- ATTR_ACTIVE_SHADOW
- ATTR_SHUTDOWN_MULTI
- ATTR_SHUTDOWN_SINGLE
- ATTR_SHUTDOWN_FULL
- ATTR_READ_ONLY
- ATTR_BACKUP_LOCK
- ATTR_BACKUP_MERGE
- ATTR_BACKUP_WRONG

Note: Also works as index to ATTRIBUTES.

ATTRIBUTES List with database attribute names

Functions

empty_str

fdb.gstat.**empty_str**(*str_*)
Return True if string is empty (whitespace don't count) or None

parse

fdb.gstat.**parse**(*lines*)
Parse output from Firebird gstat utility.

Parameters **lines** – Iterable of lines produced by Firebird gstat utility.

Returns *StatDatabase* instance with parsed results.

Raises ***ParseError*** – When any problem is found in input stream.

Classes

Named tuples

fdb.gstat.**FillDistribution**(*d20*, *d40*, *d50*, *d80*, *d100*)

fdb.gstat.**Encryption**(*pages*, *encrypted*, *unencrypted*)

StatDatabase

class fdb.gstat.**StatDatabase**
Bases: object

Firebird database statistics (produced by gstat).

has_encryption_stats()
Return True if instance contains information about database encryption.

has_index_stats()
Return True if instance contains information about indices.

has_row_stats()
Return True if instance contains information about table rows.

has_system()
Return True if instance contains information about system tables.

has_table_stats()
Return True if instance contains information about tables.

Important: This is not the same as check for empty `tables` list. When gstat is run with *-i* without *-d* option, `tables` list contains instances that does not have any other information about table but table name and its indices.

StatTable

class fdb.gstat.**StatTable**
Bases: object

Statisctics for single database table.

StatTable3

class fdb.gstat.**StatTable3**
Bases: *fdb.gstat.StatTable*

Statisctics for single database table (Firebird 3 and above).

StatIndex

class fdb.gstat.**StatIndex**(*table*)
Bases: object

Statisctics for single database index.

StatIndex3

class fdb.gstat.**StatIndex3**(*table*)
Bases: *fdb.gstat.StatIndex*

Statisctics for single database index (Firebird 3 and above).

1.4.7 Firebird server log

Functions

parse

fdb.log.**parse**(*lines*)
Parse Firebird server log and yield named tuples describing individual log entries/events.

Parameters **lines** – Iterable of lines from Firebird server log.

Raises ***ParseError*** – When any problem is found in input stream.

Classes

Named tuples

fdb.log.**LogEntry**(*source_id*, *timestamp*, *message*)

1.4.8 Utilities

Functions

fdb.utils.**update_meta**(*self*, *other*)
Helper function for *LateBindingProperty* class.

fdb.utils.**iter_class_properties**(*cls*)
Iterator that yields *name, property* pairs for all properties in class.

Parameters **cls** (*class*) – Class object.

fdb.utils.**iter_class_variables**(*cls*)
Iterator that yields names of all non-callable attributes in class.

Parameters **cls** (*class*) – Class object.

fdb.utils.**embed_attributes**(*from_class*, *attr*)
Class decorator that injects properties and attributes from another class instance embedded in class instances. Only attributes and properties that are not already defined in decorated class are injected.

param class from_class Class that should extend decorated class.

param string attr Attribute name that holds instance of embedded class within decorated class instance.

Classes

LateBindingProperty

class fdb.utils.**LateBindingProperty**
Bases: property

Property class that binds to getter/setter/deleter methods when **instance** of class that define the property is created. This allows you to override these methods in descendant classes (if they are not private) without necessity to redeclare the property itself in descendant class.

Recipe from Tim Delaney, 2005/03/31 http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/408713

```
class C(object):

    def getx(self):
        print 'C.getx'
        return self._x

    def setx(self, x):
        print 'C.setx'
        self._x = x

    def delx(self):
```

(continues on next page)

(continued from previous page)

```
        print 'C.delx'
        del self._x

    x = LateBindingProperty(getx, setx, delx)

class D(C):

    def setx(self, x):
        print 'D.setx'
        super(D, self).setx(x)

    def delx(self):
        print 'D.delx'
        super(D, self).delx()

c = C()
c.x = 1
c.x
c.x
del c.x

print

d = D()
d.x = 1
d.x
d.x
del d.x
```

This has the advantages that:

1. You get back an actual property object (with attendant memory savings, performance increases, etc);
2. It's the same syntax as using property(fget, fset, fdel, doc) except for the name;
3. It will fail earlier (when you define the class as opposed to when you use it).
4. It's shorter ;)
5. If you inspect the property you will get back functions with the correct __name__, __doc__, etc.

Iterator

class fdb.utils.**Iterator**(*method*, *sentinel=None*)

Generic iterator implementation.

Parameters

- **method** – Callable without parameters that returns next item.
- **sentinel** – Value that when returned by *method* indicates the end of sequence.

next()

EmbeddedProperty

class fdb.utils.**EmbeddedProperty**(*obj*, *prop*)

Bases: property

Property class that forwards calls to getter/setter/deleter methods to respective property methods of another object. This class allows you to "inject" properties from embedded object into class definition of parent object.

Parameters

- **obj** (*string*) – Attribute name with embedded object.
- **prop** (*property*) – Property instance from embedded object.

EmbeddedAttribute

class fdb.utils.**EmbeddedAttribute**(*obj*, *attr*)
Bases: property

Property class that gets/sets attribute of another object. This class allows you to "inject" attributes from embedded object into class definition of parent object.

Parameters

- **obj** (*string*) – Attribute name with embedded object.
- **attr** (*string*) – Attribute name from embedded object.

ObjectList

class fdb.utils.**ObjectList**(*items=None*, *_cls=None*, *key_expr=None*)
Bases: list

List of objects with additional functionality.

New in version 2.0.

Parameters

- **items** (*iterable*) – Sequence to initialize the collection.
- **_cls** – Class or list/tuple of classes. Only instances of these classes would be allowed in collection.
- **key_expr** (*str*) – Key expression. Must contain item referrence as *item*, for example *item.attribute_name*.

Raises **ValueError** – When initialization sequence contains invalid instance.

all(*expr*)
Return True if *expr* is evaluated as True for all list elements.

Parameters **expr** – Boolean expression, a callable accepting one parameter or expression as string referencing list item as *item*.

Example:

```
all(lambda x: x.name.startswith("ABC"))
all('item.name.startswith("ABC")')
```

any(*expr*)
Return True if *expr* is evaluated as True for any list element.

Parameters **expr** – Boolean expression, a callable accepting one parameter or expression as string referencing list item as *item*.

Example:

```
any(lambda x: x.name.startswith("ABC"))
any('item.name.startswith("ABC")')
```

append(*item*)
Add an item to the end of the list.

Raises **TypeError** – When list is frozen or item is not an instance of allowed class

clear()
Remove all items from the list.

Raises **TypeError** – When list is frozen.

contains(*value*, *expr=None*)
Return True if list has any item with default or specified key expression equal to given value.

Parameters

- **value** – Tested key value.
- **expr** – Key value expression, a callable accepting two parameters (item,value) or expression as string referencing list item as *item*.

Examples:

```
# Search using default key expression
contains('ITEM_NAME')
# Search using callable key expression
contains('ITEM_NAME',lambda x: x.name.upper())
# Search using string key expression
contains('ITEM_NAME','item.name.upper()')
```

ecount(*expr*)
Return number of items for which *expr* is evaluated as True.

Parameters **expr** – Boolean expression, a callable accepting one parameter or expression as string referencing list item as *item*.

Example:

```
ecount(lambda x: x.name.startswith("ABC"))
ecount('item.name.startswith("ABC")')
```

extend(*iterable*)
Extend the list by appending all the items in the given iterable.

Raises **TypeError** – When list is frozen or item is not an instance of allowed class

extract(*expr*)
Move items for which *expr* is evaluated as True into new ObjectLists.

Parameters **expr** – Boolean expression, a callable accepting one parameter or expression as string referencing list item as *item*.

Raises **TypeError** – When list is frozen.

Example:

```
extract(lambda x: x.name.startswith("ABC"))
extract('item.name.startswith("ABC")')
```

filter(*expr*)
Return new ObjectList of items for which *expr* is evaluated as True.

Parameters **expr** – Boolean expression, a callable accepting one parameter or expression as string referencing list item as *item*.

Example:

```
filter(lambda x: x.name.startswith("ABC"))
filter('item.name.startswith("ABC")')
```

freeze()
Set list to immutable (frozen) state.

get(*value*, *expr=None*)
Return item with given key value using default or specified key expression, or None if there is no such item.

Uses very fast method to look up value of default key expression in *frozen* list, otherwise it uses slower list traversal.

Parameters

- **value** – Searched value.
- **expr** – Key value expression, a callable accepting two parameters (item,value) or expression as string referencing list item as *item*.

Raises **TypeError** – If key expression is not defined.

Examples:

```
# Search using default key expression
get('ITEM_NAME')
# Search using callable key expression
get('ITEM_NAME',lambda x: x.name.upper())
# Search using string key expression
get('ITEM_NAME','item.name.upper()')
```

ifilter(*expr*)
Return generator that yields items for which *expr* is evaluated as True.

Parameters **expr** – Boolean expression, a callable accepting one parameter or expression as string referencing list item as *item*.

Example:

```
ifilter(lambda x: x.name.startswith("ABC"))
ifilter('item.name.startswith("ABC")')
```

ifilterfalse(*expr*)
Return generator that yields items for which *expr* is evaluated as False.

Parameters **expr** – Boolean expression, a callable accepting one parameter or expression as string referencing list item as *item*.

Example:

```
ifilter(lambda x: x.name.startswith("ABC"))
ifilter('item.name.startswith("ABC")')
```

insert(*index*, *item*)
Insert item before index.

Raises **TypeError** – When list is frozen or item is not an instance of allowed class

ireport(**args*)
Return generator that yields data produced by expression(s) evaluated on list items.

Parameter(s) could be one from:

- A callable accepting one parameter and returning data for output
- One or more expressions as string referencing item as *item*.

Examples:

```
# generator of tuples with item.name and item.size

report(lambda x: (x.name, x.size))
report('item.name','item.size')

# generator of item names

report(lambda x: x.name)
report('item.name')
```

report(**args*)
Return list of data produced by expression(s) evaluated on list items.

Parameter(s) could be one from:

- A callable accepting one parameter and returning data for output
- One or more expressions as string referencing item as *item*.

Examples:

```
# returns list of tuples with item.name and item.size

report(lambda x: (x.name, x.size))
report('item.name','item.size')

# returns list of item names

report(lambda x: x.name)
report('item.name')
```

reverse()
Reverse the elements of the list, in place.

Raises **TypeError** – When list is frozen.

sort(*attrs=None*, *expr=None*, *reverse=False*)
Sort items in-place, optionaly using attribute values as key or key expression.

Parameters

- **attrs** (*list*) – List of attribute names.
- **expr** – Key expression, a callable accepting one parameter or expression as string referencing list item as *item*.

Important: Only one parameter (*attrs* or *expr*) could be specified. If none is present then uses default list sorting rule.

Raises **TypeError** – When list is frozen.

Examples:

```
sort(attrs=['name','degree'])        # Sort by item.name, item.degree
sort(expr=lambda x: x.name.upper()) # Sort by upper item.name
sort(expr='item.name.upper()')       # Sort by upper item.name
```

split(*expr*)
Return two new ObjectLists, first with items for which *expr* is evaluated as True and second for *expr* evaluated as False.

Parameters **expr** – Boolean expression, a callable accepting one parameter or expression as string referencing list item as *item*.

Example:

```
split(lambda x: x.size > 100)
split('item.size > 100')
```

frozen
True if list is immutable

key
Key expression

Visitable

class fdb.utils.**Visitable**
Base class for Visitor Pattern support.

New in version 2.0.

accept(*visitor*)
Visitor Pattern support. Calls *visit(self)* on parameter object.

Parameters **visitor** – Visitor object of Visitor Pattern.

Visitor

class fdb.utils.**Visitor**
Base class for Visitor Pattern visitors.

New in version 2.0.

Descendants may implement methods to handle individual object types that follow naming pattern *visit_<class_name>*. Calls `default_action()` if appropriate special method is not defined.

Important: This implementation uses Python Method Resolution Order (__mro__) to find special handling method, so special method for given class is used also for its decendants.

Example:

```
class Node(object): pass
class A(Node): pass
class B(Node): pass
class C(A,B): pass
```

(continues on next page)

(continued from previous page)

```
class MyVisitor(object):
    def default_action(self, obj):
        print 'default_action '+obj.__class__.__name__

    def visit_B(self, obj):
        print 'visit_B '+obj.__class__.__name__


a = A()
b = B()
c = C()
visitor = MyVisitor()
visitor.visit(a)
visitor.visit(b)
visitor.visit(c)
```

Will create output:

```
default_action A
visit_B B
visit_B C
```

default_action(*obj*)
Default handler for visited objects.

Parameters **obj** (*object*) – Object to be handled.

Note: This implementation does nothing!

visit(*obj*)
Dispatch to method that handles *obj*.

First traverses the *obj.__mro__* to try find method with name following *visit_<class_name>* pattern and calls it with *obj*. Otherwise it calls *default_action()*.

Parameters **obj** (*object*) – Object to be handled by visitor.

1.4.9 ctypes interface to Firebird client library

ctypes interface to *fbclient.so/dll* is defined in submodule *fdb.ibase* and *fdb.blr*.

The *fdb.ibase* is the main module for Firebird API. The *fdb.blr* module contains only constants related to Firebird's low-level Binary Language Representation (BLR).

Constants

C integer limit constants

- SHRT_MIN
- SHRT_MAX
- USHRT_MAX
- INT_MIN
- INT_MAX

- UINT_MAX
- LONG_MIN
- LONG_MAX
- SSIZE_T_MIN
- SSIZE_T_MAX

Type codes

- SQL_TEXT
- SQL_VARYING
- SQL_SHORT
- SQL_LONG
- SQL_FLOAT
- SQL_DOUBLE
- SQL_D_FLOAT
- SQL_TIMESTAMP
- SQL_BLOB
- SQL_ARRAY
- SQL_QUAD
- SQL_TYPE_TIME
- SQL_TYPE_DATE
- SQL_INT64
- SQL_BOOLEAN
- SQL_NULL
- SUBTYPE_NUMERIC
- SUBTYPE_DECIMAL

Internal type codes (for example used by ARRAY descriptor)

- blr_text
- blr_text2
- blr_short
- blr_long
- blr_quad
- blr_float
- blr_double
- blr_d_float
- blr_timestamp
- blr_varying
- blr_varying2
- blr_blob
- blr_cstring
- blr_cstring2
- blr_blob_id
- blr_sql_date
- blr_sql_time
- blr_int64
- blr_blob2
- blr_domain_name
- blr_domain_name2
- blr_not_nullable
- blr_column_name

- blr_column_name2
- blr_bool

Database parameter block stuff

- isc_dpb_version1
- isc_dpb_version2
- isc_dpb_cdd_pathname
- isc_dpb_allocation
- isc_dpb_page_size
- isc_dpb_num_buffers
- isc_dpb_buffer_length
- isc_dpb_debug
- isc_dpb_garbage_collect
- isc_dpb_verify
- isc_dpb_sweep
- isc_dpb_dbkey_scope
- isc_dpb_number_of_users
- isc_dpb_trace
- isc_dpb_no_garbage_collect
- isc_dpb_damaged
- isc_dpb_sys_user_name
- isc_dpb_encrypt_key
- isc_dpb_activate_shadow
- isc_dpb_sweep_interval
- isc_dpb_delete_shadow
- isc_dpb_force_write
- isc_dpb_begin_log
- isc_dpb_quit_log
- isc_dpb_no_reserve
- isc_dpb_user_name
- isc_dpb_password
- isc_dpb_password_enc
- isc_dpb_sys_user_name_enc
- isc_dpb_interp
- isc_dpb_online_dump
- isc_dpb_old_file_size
- isc_dpb_old_num_files
- isc_dpb_old_file
- isc_dpb_old_start_page
- isc_dpb_old_start_seqno
- isc_dpb_old_start_file
- isc_dpb_old_dump_id
- isc_dpb_lc_messages
- isc_dpb_lc_ctype
- isc_dpb_cache_manager
- isc_dpb_shutdown
- isc_dpb_online
- isc_dpb_shutdown_delay
- isc_dpb_reserved
- isc_dpb_overwrite
- isc_dpb_sec_attach
- isc_dpb_connect_timeout

- isc_dpb_dummy_packet_interval
- isc_dpb_gbak_attach
- isc_dpb_sql_role_name
- isc_dpb_set_page_buffers
- isc_dpb_working_directory
- isc_dpb_sql_dialect
- isc_dpb_set_db_readonly
- isc_dpb_set_db_sql_dialect
- isc_dpb_gfix_attach
- isc_dpb_gstat_attach
- isc_dpb_set_db_charset
- isc_dpb_gsec_attach
- isc_dpb_address_path
- isc_dpb_process_id
- isc_dpb_no_db_triggers
- isc_dpb_trusted_auth
- isc_dpb_process_name
- isc_dpb_trusted_role
- isc_dpb_org_filename
- isc_dpb_utf8_filename
- isc_dpb_ext_call_depth
- isc_dpb_auth_block
- isc_dpb_remote_protocol
- isc_dpb_client_version
- isc_dpb_host_name
- isc_dpb_os_user
- isc_dpb_specific_auth_data
- isc_dpb_auth_plugin_list
- isc_dpb_auth_plugin_name
- isc_dpb_config
- isc_dpb_nolinger
- isc_dpb_reset_icu
- isc_dpb_map_attach

Info structural codes

- isc_info_end
- isc_info_truncated
- isc_info_error
- isc_info_data_not_ready
- isc_info_length
- isc_info_flag_end

DB Info item codes

isc_info_db_id [db_filename,site_name[,site_name...]]

isc_info_reads number of page reads

isc_info_writes number of page writes

isc_info_fetches number of reads from the memory buffer cache

isc_info_marks number of writes to the memory buffer cache

isc_info_implementation (implementation code, implementation class)

isc_info_isc_version interbase server version identification string

isc_info_base_level capability version of the server

isc_info_page_size database page size

isc_info_num_buffers number of memory buffers currently allocated

isc_info_limbo limbo transactions

isc_info_current_memory amount of server memory (in bytes) currently in use

isc_info_max_memory maximum amount of memory (in bytes) used at one time since the first process attached to the database

isc_info_allocation number of last database page allocated

isc_info_attachment_id attachment id number

all *_count* codes below return {[table_id]=operation_count,...}; table IDs are in the system table RDB$RELATIONS.

isc_info_read_seq_count number of sequential table scans (row reads) done on each table since the database was last attached

isc_info_read_idx_count number of reads done via an index since the database was last attached

isc_info_insert_count number of inserts into the database since the database was last attached

isc_info_update_count number of database updates since the database was last attached

isc_info_delete_count number of database deletes since the database was last attached

isc_info_backout_count number of removals of a version of a record

isc_info_purge_count number of removals of old versions of fully mature records (records that are committed, so that older ancestor versions are no longer needed)

isc_info_expunge_count number of removals of a record and all of its ancestors, for records whose deletions have been committed

isc_info_sweep_interval number of transactions that are committed between sweeps to remove database record versions that are no longer needed

isc_info_ods_version On-disk structure (ODS) minor major version number

isc_info_ods_minor_version On-disk structure (ODS) minor version number

isc_info_no_reserve 20% page space reservation for holding backup versions of modified records: 0=yes, 1=no

isc_info_forced_writes mode in which database writes are performed: 0=sync, 1=async

isc_info_user_names array of names of all the users currently attached to the database

isc_info_page_errors number of page level errors validate found

isc_info_record_errors number of record level errors validate found

isc_info_bpage_errors number of blob page errors validate found

isc_info_dpage_errors number of data page errors validate found

isc_info_ipage_errors number of index page errors validate found

isc_info_ppage_errors number of pointer page errors validate found

isc_info_tpage_errors number of transaction page errors validate found

isc_info_set_page_buffers number of memory buffers that should be allocated

isc_info_db_sql_dialect dialect of currently attached database

isc_info_db_read_only whether the database is read-only (1) or not (0)

isc_info_db_size_in_pages number of allocated pages

frb_info_att_charset charset of current attachment

isc_info_db_class server architecture

isc_info_firebird_version firebird server version identification string

isc_info_oldest_transaction ID of oldest transaction

isc_info_oldest_active ID of oldest active transaction

isc_info_oldest_snapshot ID of oldest snapshot transaction

isc_info_next_transaction ID of next transaction

isc_info_db_provider for firebird is 'isc_info_db_code_firebird'

isc_info_active_transactions array of active transaction IDs

isc_info_active_tran_count number of active transactions

isc_info_creation_date time_t struct representing database creation date & time

isc_info_db_file_size added in FB 2.1, nbackup-related - size (in pages) of locked db

fb_info_page_contents added in FB 2.5, get raw page contents; takes page_number as parameter;

fb_info_implementation (cpu code, OS code, compiler code, flags, implementation class)

fb_info_page_warns number of page level warnings validate found

fb_info_record_warns number of record level warnings validate found

fb_info_bpage_warns number of blob page level warnings validate found

fb_info_dpage_warns number of data page level warnings validate found

fb_info_ipage_warns number of index page level warnings validate found

fb_info_ppage_warns number of pointer page level warnings validate found

fb_info_tpage_warns number of trabsaction page level warnings validate found

fb_info_pip_errors number of pip page level errors validate found

fb_info_pip_warns number of pip page level warnings validate found

isc_info_version = isc_info_isc_version

Blob information items

- isc_info_blob_num_segments
- isc_info_blob_max_segment
- isc_info_blob_total_length
- isc_info_blob_type

Transaction information items

- isc_info_tra_id
- isc_info_tra_oldest_interesting
- isc_info_tra_oldest_snapshot
- isc_info_tra_oldest_active
- isc_info_tra_isolation
- isc_info_tra_access
- isc_info_tra_lock_timeout
- fb_info_tra_dbpath

isc_info_tra_isolation responses:

- isc_info_tra_consistency
- isc_info_tra_concurrency
- isc_info_tra_read_committed

isc_info_tra_read_committed options:

- isc_info_tra_no_rec_version
- isc_info_tra_rec_version

isc_info_tra_access responses:

- isc_info_tra_readonly
- isc_info_tra_readwrite

SQL information items

- isc_info_sql_select
- isc_info_sql_bind
- isc_info_sql_num_variables
- isc_info_sql_describe_vars
- isc_info_sql_describe_end
- isc_info_sql_sqlda_seq
- isc_info_sql_message_seq
- isc_info_sql_type
- isc_info_sql_sub_type
- isc_info_sql_scale
- isc_info_sql_length
- isc_info_sql_null_ind
- isc_info_sql_field
- isc_info_sql_relation
- isc_info_sql_owner
- isc_info_sql_alias
- isc_info_sql_sqlda_start
- isc_info_sql_stmt_type
- isc_info_sql_get_plan
- isc_info_sql_records
- isc_info_sql_batch_fetch
- isc_info_sql_relation_alias
- isc_info_sql_explain_plan
- isc_info_sql_stmt_flags

SQL information return values

- isc_info_sql_stmt_select
- isc_info_sql_stmt_insert
- isc_info_sql_stmt_update
- isc_info_sql_stmt_delete
- isc_info_sql_stmt_ddl
- isc_info_sql_stmt_get_segment
- isc_info_sql_stmt_put_segment
- isc_info_sql_stmt_exec_procedure
- isc_info_sql_stmt_start_trans
- isc_info_sql_stmt_commit
- isc_info_sql_stmt_rollback
- isc_info_sql_stmt_select_for_upd
- isc_info_sql_stmt_set_generator
- isc_info_sql_stmt_savepoint

Transaction parameter block stuff

- isc_tpb_version1
- isc_tpb_version3
- isc_tpb_consistency
- isc_tpb_concurrency
- isc_tpb_shared
- isc_tpb_protected
- isc_tpb_exclusive
- isc_tpb_wait
- isc_tpb_nowait
- isc_tpb_read
- isc_tpb_write
- isc_tpb_lock_read
- isc_tpb_lock_write
- isc_tpb_verb_time
- isc_tpb_commit_time
- isc_tpb_ignore_limbo
- isc_tpb_read_committed
- isc_tpb_autocommit
- isc_tpb_rec_version
- isc_tpb_no_rec_version
- isc_tpb_restart_requests
- isc_tpb_no_auto_undo
- isc_tpb_lock_timeout

BLOB parameter buffer

- isc_bpb_version1
- isc_bpb_source_type
- isc_bpb_target_type
- isc_bpb_type
- isc_bpb_source_interp
- isc_bpb_target_interp
- isc_bpb_filter_parameter

- isc_bpb_storage
- isc_bpb_type_segmented
- isc_bpb_type_stream
- isc_bpb_storage_main
- isc_bpb_storage_temp

Service parameter block stuff

- isc_spb_current_version
- isc_spb_version
- isc_spb_version3
- isc_spb_user_name
- isc_spb_sys_user_name
- isc_spb_sys_user_name_enc
- isc_spb_password
- isc_spb_password_enc
- isc_spb_command_line
- isc_spb_dbname
- isc_spb_verbose
- isc_spb_options
- isc_spb_address_path
- isc_spb_process_id
- isc_spb_trusted_auth
- isc_spb_process_name
- isc_spb_trusted_role
- isc_spb_verbint
- isc_spb_auth_block
- isc_spb_auth_plugin_name
- isc_spb_auth_plugin_list
- isc_spb_utf8_filename
- isc_spb_client_version
- isc_spb_remote_protocol
- isc_spb_host_name
- isc_spb_os_user
- isc_spb_config
- isc_spb_expected_db

Parameters for isc_action_{add|del|mod|disp)_user:

- isc_spb_sec_userid
- isc_spb_sec_groupid
- isc_spb_sec_username
- isc_spb_sec_password
- isc_spb_sec_groupname
- isc_spb_sec_firstname
- isc_spb_sec_middlename
- isc_spb_sec_lastname
- isc_spb_sec_admin

Parameters for isc_action_svc_backup:

- isc_spb_bkp_file
- isc_spb_bkp_factor
- isc_spb_bkp_length
- isc_spb_bkp_skip_data

- isc_spb_bkp_stat
- isc_spb_bkp_ignore_checksums
- isc_spb_bkp_ignore_limbo
- isc_spb_bkp_metadata_only
- isc_spb_bkp_no_garbage_collect
- isc_spb_bkp_old_descriptions
- isc_spb_bkp_non_transportable
- isc_spb_bkp_convert
- isc_spb_bkp_expand
- isc_spb_bkp_no_triggers

Parameters for isc_action_svc_properties:

- isc_spb_prp_page_buffers
- isc_spb_prp_sweep_interval
- isc_spb_prp_shutdown_db
- isc_spb_prp_deny_new_attachments
- isc_spb_prp_deny_new_transactions
- isc_spb_prp_reserve_space
- isc_spb_prp_write_mode
- isc_spb_prp_access_mode
- isc_spb_prp_set_sql_dialect
- isc_spb_prp_activate
- isc_spb_prp_db_online
- isc_spb_prp_nolinger
- isc_spb_prp_force_shutdown
- isc_spb_prp_attachments_shutdown
- isc_spb_prp_transactions_shutdown
- isc_spb_prp_shutdown_mode
- isc_spb_prp_online_mode

Parameters for isc_spb_prp_shutdown_mode and isc_spb_prp_online_mode:

- isc_spb_prp_sm_normal
- isc_spb_prp_sm_multi
- isc_spb_prp_sm_single
- isc_spb_prp_sm_full

Parameters for isc_spb_prp_reserve_space:

- isc_spb_prp_res_use_full
- isc_spb_prp_res

Parameters for isc_spb_prp_write_mode:

- isc_spb_prp_wm_async
- isc_spb_prp_wm_sync

Parameters for isc_action_svc_validate:

- isc_spb_val_tab_incl
- isc_spb_val_tab_excl
- isc_spb_val_idx_incl
- isc_spb_val_idx_excl
- isc_spb_val_lock_timeout

Parameters for isc_spb_prp_access_mode:

- isc_spb_rpr_commit_trans
- isc_spb_rpr_rollback_trans

- isc_spb_rpr_recover_two_phase
- isc_spb_tra_id
- isc_spb_single_tra_id
- isc_spb_multi_tra_id
- isc_spb_tra_state
- isc_spb_tra_state_limbo
- isc_spb_tra_state_commit
- isc_spb_tra_state_rollback
- isc_spb_tra_state_unknown
- isc_spb_tra_host_site
- isc_spb_tra_remote_site
- isc_spb_tra_db_path
- isc_spb_tra_advise
- isc_spb_tra_advise_commit
- isc_spb_tra_advise_rollback
- isc_spb_tra_advise_unknown
- isc_spb_tra_id_64
- isc_spb_single_tra_id_64
- isc_spb_multi_tra_id_64
- isc_spb_rpr_commit_trans_64
- isc_spb_rpr_rollback_trans_64
- isc_spb_rpr_recover_two_phase_64
- isc_spb_rpr_validate_db
- isc_spb_rpr_sweep_db
- isc_spb_rpr_mend_db
- isc_spb_rpr_list_limbo_trans
- isc_spb_rpr_check_db
- isc_spb_rpr_ignore_checksum
- isc_spb_rpr_kill_shadows
- isc_spb_rpr_full
- isc_spb_rpr_icu

Parameters for isc_action_svc_restore:

- isc_spb_res_skip_data
- isc_spb_res_buffers
- isc_spb_res_page_size
- isc_spb_res_length
- isc_spb_res_access_mode
- isc_spb_res_fix_fss_data
- isc_spb_res_fix_fss_metadata
- isc_spb_res_stat
- isc_spb_res_metadata_only
- isc_spb_res_deactivate_idx
- isc_spb_res_no_shadow
- isc_spb_res_no_validity
- isc_spb_res_one_at_a_time
- isc_spb_res_replace
- isc_spb_res_create
- isc_spb_res_use_all_space

Parameters for isc_spb_res_access_mode:

- isc_spb_res_am_readonly
- isc_spb_res_am_readwrite

Parameters for isc_info_svc_svr_db_info:

- isc_spb_num_att
- isc_spb_num_db

Parameters for isc_info_svc_db_stats:

- isc_spb_sts_data_pages
- isc_spb_sts_db_log
- isc_spb_sts_hdr_pages
- isc_spb_sts_idx_pages
- isc_spb_sts_sys_relations
- isc_spb_sts_record_versions
- isc_spb_sts_table
- isc_spb_sts_nocreation
- isc_spb_sts_encryption

Parameters for isc_action_svc_nbak:

- isc_spb_nbk_level
- isc_spb_nbk_file
- isc_spb_nbk_direct
- isc_spb_nbk_no_triggers

Parameters for trace:

- isc_spb_trc_id
- isc_spb_trc_name
- isc_spb_trc_cfg

Service action items

- isc_action_svc_backup
- isc_action_svc_restore
- isc_action_svc_repair
- isc_action_svc_add_user
- isc_action_svc_delete_user
- isc_action_svc_modify_user
- isc_action_svc_display_user
- isc_action_svc_properties
- isc_action_svc_add_license
- isc_action_svc_remove_license
- isc_action_svc_db_stats
- isc_action_svc_get_ib_log
- isc_action_svc_get_fb_log
- isc_action_svc_nbak
- isc_action_svc_nrest
- isc_action_svc_trace_start
- isc_action_svc_trace_stop
- isc_action_svc_trace_suspend
- isc_action_svc_trace_resume
- isc_action_svc_trace_list
- isc_action_svc_set_mapping
- isc_action_svc_drop_mapping
- isc_action_svc_display_user_adm
- isc_action_svc_validate

Service information items

- isc_info_svc_svr_db_info
- isc_info_svc_get_config
- isc_info_svc_version
- isc_info_svc_server_version
- isc_info_svc_implementation
- isc_info_svc_capabilities
- isc_info_svc_user_dbpath
- isc_info_svc_get_env
- isc_info_svc_get_env_lock
- isc_info_svc_get_env_msg
- isc_info_svc_line
- isc_info_svc_to_eof
- isc_info_svc_timeout
- isc_info_svc_limbo_trans
- isc_info_svc_running
- isc_info_svc_get_users
- isc_info_svc_auth_block
- isc_info_svc_stdin

BLOB action codes

- blb_got_eof
- blb_got_fragment
- blb_got_full_segment
- blb_seek_relative
- blb_seek_from_tail

Implementation codes

- isc_info_db_impl_rdb_vms
- isc_info_db_impl_rdb_eln
- isc_info_db_impl_rdb_eln_dev
- isc_info_db_impl_rdb_vms_y
- isc_info_db_impl_rdb_eln_y
- isc_info_db_impl_jri
- isc_info_db_impl_jsv
- isc_info_db_impl_isc_apl_68K
- isc_info_db_impl_isc_vax_ultr
- isc_info_db_impl_isc_vms
- isc_info_db_impl_isc_sun_68k
- isc_info_db_impl_isc_os2
- isc_info_db_impl_isc_sun4
- isc_info_db_impl_isc_hp_ux
- isc_info_db_impl_isc_sun_386i
- isc_info_db_impl_isc_vms_orcl
- isc_info_db_impl_isc_mac_aux
- isc_info_db_impl_isc_rt_aix
- isc_info_db_impl_isc_mips_ult
- isc_info_db_impl_isc_xenix
- isc_info_db_impl_isc_dg

- isc_info_db_impl_isc_hp_mpexl
- isc_info_db_impl_isc_hp_ux68K
- isc_info_db_impl_isc_sgi
- isc_info_db_impl_isc_sco_unix
- isc_info_db_impl_isc_cray
- isc_info_db_impl_isc_imp
- isc_info_db_impl_isc_delta
- isc_info_db_impl_isc_next
- isc_info_db_impl_isc_dos
- isc_info_db_impl_m88K
- isc_info_db_impl_unixware
- isc_info_db_impl_isc_winnt_x86
- isc_info_db_impl_isc_epson
- isc_info_db_impl_alpha_osf
- isc_info_db_impl_alpha_vms
- isc_info_db_impl_netware_386
- isc_info_db_impl_win_only
- isc_info_db_impl_ncr_3000
- isc_info_db_impl_winnt_ppc
- isc_info_db_impl_dg_x86
- isc_info_db_impl_sco_ev
- isc_info_db_impl_i386
- isc_info_db_impl_freebsd
- isc_info_db_impl_netbsd
- isc_info_db_impl_darwin_ppc
- isc_info_db_impl_sinixz
- isc_info_db_impl_linux_sparc
- isc_info_db_impl_linux_amd64
- isc_info_db_impl_freebsd_amd64
- isc_info_db_impl_winnt_amd64
- isc_info_db_impl_linux_ppc
- isc_info_db_impl_darwin_x86
- isc_info_db_impl_linux_mipsel
- isc_info_db_impl_linux_mips
- isc_info_db_impl_darwin_x64
- isc_info_db_impl_sun_amd64
- isc_info_db_impl_linux_arm
- isc_info_db_impl_linux_ia64
- isc_info_db_impl_darwin_ppc64
- isc_info_db_impl_linux_s390x
- isc_info_db_impl_linux_s390
- isc_info_db_impl_linux_sh
- isc_info_db_impl_linux_sheb
- isc_info_db_impl_linux_hppa
- isc_info_db_impl_linux_alpha
- isc_info_db_impl_linux_arm64
- isc_info_db_impl_linux_ppc64el
- isc_info_db_impl_linux_ppc64

Info DB provider codes

- isc_info_db_code_rdb_eln
- isc_info_db_code_rdb_vms

- isc_info_db_code_interbase
- isc_info_db_code_firebird

Info DB class codes

- isc_info_db_class_access
- isc_info_db_class_y_valve
- isc_info_db_class_rem_int
- isc_info_db_class_rem_srvr
- isc_info_db_class_pipe_int
- isc_info_db_class_pipe_srvr
- isc_info_db_class_sam_int
- isc_info_db_class_sam_srvr
- isc_info_db_class_gateway
- isc_info_db_class_cache
- isc_info_db_class_classic_access
- isc_info_db_class_server_access

Request information items

- isc_info_number_messages
- isc_info_max_message
- isc_info_max_send
- isc_info_max_receive
- isc_info_state
- isc_info_message_number
- isc_info_message_size
- isc_info_request_cost
- isc_info_access_path
- isc_info_req_select_count
- isc_info_req_insert_count
- isc_info_req_update_count
- isc_info_req_delete_count

Access path items

- isc_info_rsb_end
- isc_info_rsb_begin
- isc_info_rsb_type
- isc_info_rsb_relation
- isc_info_rsb_plan

Record Source Block (RSB) types

- isc_info_rsb_unknown
- isc_info_rsb_indexed
- isc_info_rsb_navigate
- isc_info_rsb_sequential
- isc_info_rsb_cross
- isc_info_rsb_sort

- isc_info_rsb_first
- isc_info_rsb_boolean
- isc_info_rsb_union
- isc_info_rsb_aggregate
- isc_info_rsb_merge
- isc_info_rsb_ext_sequential
- isc_info_rsb_ext_indexed
- isc_info_rsb_ext_dbkey
- isc_info_rsb_left_cross
- isc_info_rsb_select
- isc_info_rsb_sql_join
- isc_info_rsb_simulate
- isc_info_rsb_sim_cross
- isc_info_rsb_once
- isc_info_rsb_procedure
- isc_info_rsb_skip
- isc_info_rsb_virt_sequential
- isc_info_rsb_recursive
- isc_info_rsb_window
- isc_info_rsb_singular
- isc_info_rsb_writelock
- isc_info_rsb_buffer
- isc_info_rsb_hash

RSB Bitmap expressions

- isc_info_rsb_and
- isc_info_rsb_or
- isc_info_rsb_dbkey
- isc_info_rsb_index

Info request response codes

- isc_info_req_active
- isc_info_req_inactive
- isc_info_req_send
- isc_info_req_receive
- isc_info_req_select
- isc_info_req_sql_stall

Blob Subtypes

- isc_blob_untyped
- isc_blob_text
- isc_blob_blr
- isc_blob_acl
- isc_blob_ranges
- isc_blob_summary
- isc_blob_format
- isc_blob_tra
- isc_blob_extfile

- isc_blob_debug_info
- isc_blob_max_predefined_subtype

Cancel types for fb_cancel_operation

- fb_cancel_disable
- fb_cancel_enable
- fb_cancel_raise
- fb_cancel_abort

Other constants

- DSQL_close
- DSQL_drop
- DSQL_unprepare
- SQLDA_version1
- isc_info_req_select_count
- isc_info_req_insert_count
- isc_info_req_update_count
- isc_info_req_delete_count

flags set in fb_info_crypt_state:

- fb_info_crypt_encrypted
- fb_info_crypt_process

  FB_API_VER (int) Firebird API version number

  MAX_BLOB_SEGMENT_SIZE (int) Max size for BLOB segment

Types

Basic types

`fdb.ibase.`**`STRING`**
alias of `ctypes.c_char_p`

`fdb.ibase.`**`WSTRING`**
alias of `ctypes.c_wchar_p`

`fdb.ibase.`**`FB_API_HANDLE`**
alias of `ctypes.c_uint`

`fdb.ibase.`**`ISC_STATUS`**
alias of `ctypes.c_long`

`fdb.ibase.`**`ISC_STATUS_PTR`**
alias of `fdb.ibase.LP_c_long`

`fdb.ibase.`**`ISC_STATUS_ARRAY`**
alias of `fdb.ibase.c_long_Array_20`

fdb.ibase.**FB_SQLSTATE_STRING**
alias of fdb.ibase.c_char_Array_6

fdb.ibase.**ISC_LONG**
alias of ctypes.c_int

fdb.ibase.**ISC_ULONG**
alias of ctypes.c_uint

fdb.ibase.**ISC_SHORT**
alias of ctypes.c_short

fdb.ibase.**ISC_USHORT**
alias of ctypes.c_ushort

fdb.ibase.**ISC_UCHAR**
alias of ctypes.c_ubyte

fdb.ibase.**ISC_SCHAR**
alias of ctypes.c_char

fdb.ibase.**ISC_INT64**
alias of ctypes.c_long

fdb.ibase.**ISC_UINT64**
alias of ctypes.c_ulong

fdb.ibase.**ISC_DATE**
alias of ctypes.c_int

fdb.ibase.**ISC_TIME**
alias of ctypes.c_uint

class fdb.ibase.**ISC_TIMESTAMP**

timestamp_date
Structure/Union member

timestamp_time
Structure/Union member

class fdb.ibase.**GDS_QUAD_t**

gds_quad_high
Structure/Union member

gds_quad_low
Structure/Union member

fdb.ibase.**GDS_QUAD**
alias of *fdb.ibase.GDS_QUAD_t*

fdb.ibase.**ISC_QUAD**
alias of *fdb.ibase.GDS_QUAD_t*

class fdb.ibase.**ISC_ARRAY_BOUND**

array_bound_lower
Structure/Union member

array_bound_upper
Structure/Union member

class fdb.ibase.**ISC_ARRAY_DESC**

array_desc_bounds
Structure/Union member

array_desc_dimensions
Structure/Union member

array_desc_dtype
Structure/Union member

array_desc_field_name
Structure/Union member

array_desc_flags
Structure/Union member

array_desc_length
Structure/Union member

array_desc_relation_name
Structure/Union member

array_desc_scale
Structure/Union member

class fdb.ibase.**ISC_BLOB_DESC**

blob_desc_charset
Structure/Union member

blob_desc_field_name
Structure/Union member

blob_desc_relation_name
Structure/Union member

blob_desc_segment_size
Structure/Union member

blob_desc_subtype
Structure/Union member

class fdb.ibase.**isc_blob_ctl**

ctl_bpb
Structure/Union member

ctl_bpb_length
Structure/Union member

ctl_buffer
Structure/Union member

ctl_buffer_length
Structure/Union member

ctl_data
Structure/Union member

ctl_from_sub_type
Structure/Union member

ctl_max_segment
Structure/Union member

ctl_number_segments
Structure/Union member

ctl_segment_length
Structure/Union member

ctl_source
Structure/Union member

ctl_source_handle
Structure/Union member

ctl_status
Structure/Union member

ctl_to_sub_type
Structure/Union member

ctl_total_length
Structure/Union member

class fdb.ibase.**bstream**

bstr_blob
Structure/Union member

bstr_buffer
Structure/Union member

bstr_cnt
Structure/Union member

bstr_length
Structure/Union member

bstr_mode
Structure/Union member

bstr_ptr
Structure/Union member

fdb.ibase.**BSTREAM**
alias of *fdb.ibase.bstream*

fdb.ibase.**FB_BLOB_STREAM**
alias of fdb.ibase.LP_bstream

class fdb.ibase.**paramdsc**

dsc_address
Structure/Union member

dsc_dtype
Structure/Union member

dsc_flags
Structure/Union member

dsc_length
Structure/Union member

dsc_scale
Structure/Union member

dsc_sub_type
Structure/Union member

class fdb.ibase.**paramvary**

vary_length
Structure/Union member

vary_string
Structure/Union member

ISC_TEB

class fdb.ibase.**ISC_TEB**

db_ptr
Structure/Union member

tpb_len
Structure/Union member

tpb_ptr
Structure/Union member

XSQLVAR

class fdb.ibase.**XSQLVAR**

aliasname
Structure/Union member

aliasname_length
Structure/Union member

ownname
Structure/Union member

ownname_length
Structure/Union member

relname
Structure/Union member

relname_length
Structure/Union member

sqldata
Structure/Union member

sqlind
Structure/Union member

sqllen
Structure/Union member

sqlname
Structure/Union member

sqlname_length
Structure/Union member

sqlscale
Structure/Union member

sqlsubtype
Structure/Union member

sqltype
Structure/Union member

XSQLDA

class fdb.ibase.**XSQLDA**

sqld
Structure/Union member

sqldabc
Structure/Union member

sqldaid
Structure/Union member

sqln
Structure/Union member

sqlvar
Structure/Union member

version
Structure/Union member

fdb.ibase.**XSQLDA_PTR**
alias of fdb.ibase.LP_XSQLDA

USER_SEC_DATA

class fdb.ibase.**USER_SEC_DATA**

dba_password
Structure/Union member

dba_user_name
Structure/Union member

first_name
Structure/Union member

gid
Structure/Union member

group_name
Structure/Union member

last_name
Structure/Union member

middle_name
Structure/Union member

password
Structure/Union member

protocol
Structure/Union member

sec_flags
Structure/Union member

server
Structure/Union member

uid
Structure/Union member

user_name
Structure/Union member

RESULT_VECTOR

fdb.ibase.**RESULT_VECTOR**
alias of fdb.ibase.c_uint_Array_15

Callbacks

class fdb.ibase.**FB_SHUTDOWN_CALLBACK**
ctypes.CFUNCTYPE(UNCHECKED(c_int), c_int, c_int, POINTER(None))

class fdb.ibase.**ISC_CALLBACK**
ctypes.CFUNCTYPE(None)

class fdb.ibase.**ISC_PRINT_CALLBACK**
ctypes.CFUNCTYPE(None, c_void_p, c_short, STRING)

class fdb.ibase.**ISC_VERSION_CALLBACK**
ctypes.CFUNCTYPE(None, c_void_p, STRING)

class fdb.ibase.**ISC_EVENT_CALLBACK**
ctypes.CFUNCTYPE(None, POINTER(ISC_UCHAR), c_ushort, POINTER(ISC_UCHAR))

class fdb.ibase.**blobcallback**

blob_get_segment
Structure/Union member

blob_handle
Structure/Union member

blob_lseek
Structure/Union member

blob_max_segment
Structure/Union member

blob_number_segments
Structure/Union member

blob_put_segment
Structure/Union member

blob_total_length
Structure/Union member

Other globals

charset_map Dictionary that maps DB CHAR SET NAME to PYTHON CODEC NAME (CANONICAL)

fdb.ibase.**ISC_TRUE**

fdb.ibase.**ISC_FALSE**

Functions

Classes

fbclient_API

class fdb.ibase.**fbclient_API**(*fb_library_name=None*)
Firebird Client API interface object. Loads Firebird Client Library and exposes API functions as member methods. Uses ctypes for bindings.

isc_event_block(*event_buffer*, *result_buffer*, **args*)
Injects variable number of parameters into C_isc_event_block call

BLR Constants

Note: BLR data types are defined in *fdb.ibase*

Main BLR codes

- blr_inner
- blr_left
- blr_right
- blr_full
- blr_gds_code
- blr_sql_code

- blr_exception
- blr_trigger_code
- blr_default_code
- blr_raise
- blr_exception_msg
- blr_exception_params
- blr_sql_state
- blr_version4
- blr_version5
- blr_eoc
- blr_end
- blr_assignment
- blr_begin
- blr_dcl_variable
- blr_message
- blr_erase
- blr_fetch
- blr_for
- blr_if
- blr_loop
- blr_modify
- blr_handler
- blr_receive
- blr_select
- blr_send
- blr_store
- blr_label
- blr_leave
- blr_store2
- blr_post
- blr_literal
- blr_dbkey
- blr_field
- blr_fid
- blr_parameter
- blr_variable
- blr_average
- blr_count
- blr_maximum
- blr_minimum
- blr_total
- blr_add
- blr_subtract
- blr_multiply
- blr_divide
- blr_negate
- blr_concatenate
- blr_substring
- blr_parameter2
- blr_from
- blr_via
- blr_parameter2_old
- blr_user_name
- blr_null

- blr_equiv
- blr_eql
- blr_neq
- blr_gtr
- blr_geq
- blr_lss
- blr_leq
- blr_containing
- blr_matching
- blr_starting
- blr_between
- blr_or
- blr_and
- blr_not
- blr_any
- blr_missing
- blr_unique
- blr_like
- blr_rse
- blr_first
- blr_project
- blr_sort
- blr_boolean
- blr_ascending
- blr_descending
- blr_relation
- blr_rid
- blr_union
- blr_map
- blr_group_by
- blr_aggregate
- blr_join_type
- blr_agg_count
- blr_agg_max
- blr_agg_min
- blr_agg_total
- blr_agg_average
- blr_parameter3
- blr_agg_count2
- blr_agg_count_distinct
- blr_agg_total_distinct
- blr_agg_average_distinct
- blr_function
- blr_gen_id
- blr_prot_mask
- blr_upcase
- blr_lock_state
- blr_value_if
- blr_matching2
- blr_index
- blr_ansi_like
- blr_scrollable
- blr_run_count
- blr_rs_stream

- blr_exec_proc
- blr_procedure
- blr_pid
- blr_exec_pid
- blr_singular
- blr_abort
- blr_block
- blr_error_handler
- blr_cast
- blr_pid2
- blr_procedure2
- blr_start_savepoint
- blr_end_savepoint

Other BLR codes

- blr_domain_type_of
- blr_domain_full
- blr_date
- blr_plan
- blr_merge
- blr_join
- blr_sequential
- blr_navigational
- blr_indices
- blr_retrieve
- blr_relation2
- blr_rid2
- blr_set_generator
- blr_ansi_any
- blr_exists
- blr_record_version
- blr_stall
- blr_ansi_all
- blr_extract
- blr_continue
- blr_forward
- blr_backward
- blr_bof_forward
- blr_eof_backward
- blr_extract_year
- blr_extract_month
- blr_extract_day
- blr_extract_hour
- blr_extract_minute
- blr_extract_second
- blr_extract_weekday
- blr_extract_yearday
- blr_extract_millisecond
- blr_extract_week
- blr_current_date
- blr_current_timestamp
- blr_current_time

- blr_post_arg
- blr_exec_into
- blr_user_savepoint
- blr_dcl_cursor
- blr_cursor_stmt
- blr_current_timestamp2
- blr_current_time2
- blr_agg_list
- blr_agg_list_distinct
- blr_modify2
- blr_current_role
- blr_skip
- blr_exec_sql
- blr_internal_info
- blr_nullsfirst
- blr_writelock
- blr_nullslast
- blr_lowcase
- blr_strlen
- blr_strlen_bit
- blr_strlen_char
- blr_strlen_octet
- blr_trim
- blr_trim_both
- blr_trim_leading
- blr_trim_trailing
- blr_trim_spaces
- blr_trim_characters
- blr_savepoint_set
- blr_savepoint_release
- blr_savepoint_undo
- blr_savepoint_release_single
- blr_cursor_open
- blr_cursor_close
- blr_cursor_fetch
- blr_cursor_fetch_scroll
- blr_croll_forward
- blr_croll_backward
- blr_croll_bof
- blr_croll_eof
- blr_croll_absolute
- blr_croll_relative
- blr_init_variable
- blr_recurse
- blr_sys_function
- blr_auto_trans
- blr_similar
- blr_exec_stmt
- blr_exec_stmt_inputs
- blr_exec_stmt_outputs
- blr_exec_stmt_sql
- blr_exec_stmt_proc_block
- blr_exec_stmt_data_src
- blr_exec_stmt_user

- blr_exec_stmt_pwd
- blr_exec_stmt_tran
- blr_exec_stmt_tran_clone
- blr_exec_stmt_privs
- blr_exec_stmt_in_params
- blr_exec_stmt_in_params2
- blr_exec_stmt_out_params
- blr_exec_stmt_role
- blr_stmt_expr
- blr_derived_expr
- blr_procedure3
- blr_exec_proc2
- blr_function2
- blr_window
- blr_partition_by
- blr_continue_loop
- blr_procedure4
- blr_agg_function
- blr_substring_similar
- blr_bool_as_value
- blr_coalesce
- blr_decode
- blr_exec_subproc
- blr_subproc_decl
- blr_subproc
- blr_subfunc_decl
- blr_subfunc
- blr_record_version2
- blr_gen_id2

1.5 Changelog

1.5.1 Version 2.0

Important: This is initial release of new *"SweetBitter"* driver generation.

During this (v2) generation FDB driver will undergo a transition from development centered around Python 2.7 / Firebird 2.x to development centered around Python 3 / Firebird 3. There are some backward incompatible changes between v2 and v1 generation, and you may expect some also between individual releases of second generation. To *soften* this *bitter* pill, the second generation will have new functionality, enhancements and optimalizations gradually added into each public release.

The second generation is also the last one that will directly support Python 2.7 and will be tested with Firebird 2.

The plan is to move forward with v3 generation (Python 3/Firebird 3+) as soon as v2 code base will become mature.

Improvements

- Hooks.

- New modules for parsing Firebird trace & audit logs (*fdb.trace*), gstat output (*fdb.gstat*) and server log (*fdb.log*)
- Added *fdb.utils.ObjectList* class for improved object collection manipulation.
- Modules *monitor* and *schema* now use new *fdb.utils.ObjectList* for collections of information objects.
- Methods *fdb.Connection.database_info()* and *fdb.Transaction.transaction_info()* now distinguish between text and binary strings with *result_type* code.
- Significant changes to documentation.

Other changes

- Exception fdb.Warning removed as duplicate to standard Warning exception.
- Changes to make pylint more happy about fdb code.
- Parameter *result_type* in methods *fdb.Transaction.transaction_info()* and *fdb.Connection.database_info()* now does not support value 's' for string results as these converted strings to unicode in Python 3 which does not makes sense (it's always binary data, at least partially). Instead new value 'b' is introduced for binary string results.
- Reworked Visitor Pattern support in *schema* module, added classes *fdb.utils.Visitable* and *fdb.utils.Visitor*.
- Method *fdb.schema.Schema.reload()* now takes as parameter numeric metadata category code(s) instead string name.
- Cleanup of unused constants

Bugs Fixed

- (PYFB-72) - exception_from_status function gives an UnicodeDecodeError
- (PYFB-73) - Buffer filled with zero-characters is returned instead of actual content of page when page number more than 64 K
- (Unregistered) - BOOLEAN arrays were not supported
- (Unregistered) - Issues with Python 3 and Windows compatibility from latest 1.x versions.

1.5.2 Version 1.8

- In relation to (PYFB-71) a better memory exhaustion safeguard was implemented for materialized blobs. See *Working with BLOBs* for details.
- Added service support for backup and restore from/to local byte stream. See *local_backup()* and *local_restore()* for details.
- Added attribute *fdb.schema.TableColumn.id* (RDB$FIELD_ID)
- Added method *fdb.BlobReader.get_info()*.

1.5.3 Version 1.7

- (PYFB-66) - Port parameter for connect and create_database is not used
- (PYFB-69) - Can not connect to FB services if set ISC_USER & ISC_PASSWORD by os.environ[...]
- (PYFB-70) - executemany(operation, seq_of_parameters) appears to run slower than it should
- Number of fixes to DDL generators in schema module
- Added support for `Filter` and `BackupHistory` in schema module.
- Added DDL scripts generator `get_metadata_ddl()`.

1.5.4 Version 1.6.1

- (PYFB-68) - Add support for isc_spb_sts_table option
- (PYFB-67) - Cursor fails after use with executemany(). ReferenceError: weakly-referenced object no longer exists

1.5.5 Version 1.6

- New: Extended support for database and transaction info (new attributes and functions on `Connection` and `Transaction`, fixes and improvements to `db_info()` and `database_info()`).
- Fix: Missing character sets for automatic translations.
- (PYFB-64) - cursor.description throws ReferenceError after executemany INSERT

1.5.6 Version 1.5.1

- New `connect()` parameters: no_gc, no_db_triggers and no_linger.
- Direct support for *with* statement (PEP8) in `Connection` class.

1.5.7 Version 1.5

- Initial support for Firebird 3.0
  - BOOLEAN datatype
  - IDENTITY datatype (in schema)
  - Database linger
  - Preserve SHADOW on DROP
  - DDL triggers
  - New and extended system tables
  - New and extended monitoring tables
  - GBAK statistics (service)
  - On-line validation (service)
- (PYFB-60) Cursor: executemany(operation, seq_of_parameters) does PREPARE of <operation> for each parameter from <seq_of_parameters>

1.5.8 Version 1.4.11

- (PYFB-58) Severe performance loss and minor memory leak

1.5.9 Version 1.4.10

- (PYFB-54) Windows 7x64 and FB2.5.2x32 Python2.7: Error in Registry Path. FDB driver does not find the library fbclient.dll
- (PYFB-55) get_sql_for incorrect generate sql query for Views
- (PYFB-56) schema.reload typing mistake for views
- (PYFB-57) role.privileges does not return correct privilege list

1.5.10 Version 1.4.9

- (PYFB-51) <procedure>.get_sql_for('<re>create') returns invalid output parameters
- (PYFB-52) isc_info* types which are _DATABASE_INFO_CODES_WITH_COUNT_RESULTS raises TypeError: 'float' object cannot be interpreted as an integer

1.5.11 Version 1.4.8

- Enhancement to automatic client library location detection on POSIX. Now it also looks at LD_LIBRARY_PATH dir if specified.

1.5.12 Version 1.4.7

- Forgoten debug printout removed. Annoying for Python 2.x users, fatal for 3.x users.

1.5.13 Version 1.4.6

Bugs Fixed

- (PYFB-50) Exception ReferenceError: 'weakly-referenced object no longer exists' in PreparedStatement and Cursor

1.5.14 Version 1.4.5

Bugs Fixed

- (PYFB-49) Memory and DB resource leak due to circular references.

1.5.15 Version 1.4.4

Improvements

- (PYFB-47) Firebird client library path added as optinal parameter to *fdb.connect()* and *fdb.create_database()*.

Bugs Fixed

- Additional fix related to PYFB-43
- Additional correction for unregistered problem with circular ref. between PS and Cursor when explicit PS is executed.

1.5.16 Version 1.4.3

Bugs Fixed

- Previous fix for PYFB-43 was incomplete, corrected.

1.5.17 Version 1.4.2

Improvements

- In relation to PYFB-43 I had to make a **backward incompatible change** to event processing API. Starting from this version *EventConduit* does not automatically starts collection of events upon creation, but it's now necessary to call `begin()` method. To mitigate the inconvenience, EventConduit now supports context manager protocol that ensures calls to begin() and close() via *with* statement.
- In relation to PYFB-39 I have decided to drop support for implicitly cached and reused prepared statements. I never liked this feature as I think it's a sneaky method how to put some performance to badly written applications that in worst case may lead to significant resource consumption on server side when developers are not only lazy but also stupid. It was implemented for the sake of compatibility with KInterbasDB.

  This change has no impact on API, but may affect performance of your applications.

Bugs Fixed

- PYFB-44 - Inserting a datetime.date into a TIMESTAMP column does not work
- PYFB-42 - Python 3.4 and FDB - backup throws an exception
- Unregistered - Fixes in monitor.TransactionInfo

1.5.18 Version 1.4.1

Improvements

- PYFB-40 - fbclient.dll is not found if not in path. Aside from registry lookup, client library isn't loaded until first call to `fdb.connect()`, `fdb.create_database()` or `fdb.load_api()` (which now supports optional specification of Firebird Client Library to load).
- Adjustments for Firebird 3.0 (Alpha1)
- Properties `version` and `engine_version` added to `fdb.services.Connection`

Bugs Fixed

- Unregistered - isolation_level parameter for fdb.connection has no effect.
- Unregistered - Information gathered from monitoring tables is not properly dropped upon refresh request.

1.5.19 Version 1.4

New Features

- *fdb.schema* submodule extended with support for user privileges.

Improvements

- *fdb.services.User.load_information()* method to load information about user from server.
- *fdb.ibase* content cleanup and additions.
- *fdb.blr* submodule with BLR definitions.

Bugs Fixed

- PYFB-37 - Unicode Strings incorrect not allowed for insertion into BLOB SubType 1.

1.5.20 Version 1.3

New Features

- *fdb.monitor* submodule for access to / work with monitoring tables.
- New *fdb.Connection.monitor* property for access to monitoring tables.

Improvements

- *closed* property and *clear()* method for Schema.
- Unit tests reworked.

Bugs Fixed

- Unregistered: Bug in *fdb.schema.Schema.close()* and *fdb.schema.Schema.bind()*.

1.5.21 Version 1.2

New Features

- *fdb.schema* submodule for access to / work with database metadata.
- *fdb.utils* submodule with various helper classes and functions.
- New *fdb.Connection.schema* property for access to database schema.

- New `ConnectionWithSchema` connection class that provides more direct access to database schema than *Connection*.
- New `fdb.Connection.firebird_version`, `fdb.Connection.version` and `fdb.Connection.engine_version` properties.
- New *Connection.ods* read only property that returns ODS version number of connected database. There are also new module-level constants *ODS_FB_20*, *ODS_FB_21* and *ODS_FB_25*.
- New `fdb.Connection.query_transaction` property. This is ReadOnly ReadCommitted transaction that could be active indefinitely without blocking garbage collection. It's used internally to query metadata, but it's generally useful.

Improvements

- Optional PEP 249 (Python DB API 2.0) Extensions
  - *Connection.Error*, *Connection.ProgrammingError*, etc.

    All exception classes defined by the DB API standard are exposed on the Connection objects as attributes (in addition to being available at module scope).
  - *Cursor.connection*

    This read-only attribute return a reference to the Connection object on which the cursor was created.
- *Cursor.transaction* read-only attribute returns a reference to the Transaction object on which the cursor was created.
- Optimized wekref management, especially for *PreparedStatement*.
- *create_database* now supports two methods for database screation. You can specify CREATE DATABASE statement (as before) or provide set of named database parameters (SQL statement is created automatically from them).
- Functions *connection* and *create_database* now take optional keyword parameter *connection_class* to obtain instances of different class instead *Connection*.
- Support for legacy (pre-2.5) shutdown mode with mode *fdb.services.SHUT_LEGACY*.
- `fdb.Cursor.executemany()` returns *self*, so it could be used directly as iterator.
- Documentation inprovements.

Bugs Fixed

- Unregistered: *buffers* parameter of *fdb.connection* doesn't support values greater than 255.
- Unregistered: Lowercase character set name passed to *fdb.connect* may result in wrong funcion of automatic data conversions and other failures (exceptions raised).

1.5.22 Version 1.1.1

Bugs Fixed

- PYFB-35 - Call to fetch after a sql statement without a result should raise exception
- PYFB-34 - Server resources not released on PreparedStatement destruction

1.5.23 Version 1.1

New Features

- *Context Manager* for transactions.

Bugs Fixed

- PYFB-30 - BLOBs are truncated at first zero byte

1.5.24 Version 1.0

Improvements

- Removed dependency on fbclient library to be present at import time (PYFB-24)

Bugs Fixed

- PYFB-25 - Truncate long text from VARCHAR(5000)

1.5.25 Version 0.9.9

New Features

- Firebird ARRAY support.

Other changes

- Cursor.execute() returns Self, so it could be used as iterator.
- Reading output from Services now uses more efficient method to get data from server.

Bugs Fixed

- Fix: precision_cache in Connection works as intended.

1.5.26 Version 0.9.1

Just bugfixes to make FDB work on P3K again.

1.5.27 Version 0.9

New Features

- Documentation; both in-source (in Sphinx autodoc format) and Sphinx (html)
- Services API completely reworked

Other changes

- Unregistered bugs fixed.
- Various optimizations and cleanup
- Object reference graph optimalizations
- Many new tests in test suite

1.5.28 Version 0.8.5

New Features

- Support for Firebird stream BLOBs (see ReleaseNotes for details)
- Documentation (stub, from KInterbasDB 3.3.0)

Bugs Fixed

- Fix for PYFB-17 and PYFB-18 (see our JIRA tracker for details)
- Fixes for automatic unicode conversions + refactoring
- Some optimizations

1.5.29 Version 0.8

New Features

- (PYFB-8) - Support for Firebird Event Notifications

Bugs Fixes

- (PYFB-16) - database_info (isc_info_firebird_version) fails opn amd64 linux
- (PYFB-15) - more than 2 consecutive cursor open execute and iter fail

1.5.30 Version 0.7.2

New Features

- Python 3 Support (thanks to Philippe Makowski)
- Support for Distributed Transactions

And as always, some (unregistered) bugs fixed.

1.5.31 Version 0.7.1

Bug fixes.

1.5.32 Version 0.7

Initial release.

Almost feature-complete (ready for 95% of users), but it could be still buggy (it's beta!), and the code wasn't optimized for size and speed. In all other ways it's ready for wide testing.

What's missing

- Distributed transactions
- ARRAY support
- EVENTs support
- Stream BLOBs
- TRACE service
- Documentation (but you can use KInterbasDB one as FDB is as close to it as possible).
- Python 3.x support (haven't had time to test it, but it shouldn't be hard to make it work there)

1.6 LICENSE

The following contributors hold Copyright (c) over their respective portions of code and documentation:

The main portion of initial code (~95%); Current maintainer:

Pavel Cisar <pcisar@ibphoenix.cz>

Contributors:

[Initial trace API & nbackup support; Python 3 support] Philippe Makowski <pmakowski@espelida.com>

Some code in inital version is Python code from KInterbasDB 3.3.0. As it's very hard to find out who exactly was the original author of used code, here is the full list of KInterbasDB contributors:

[Author of original version; maintained through version 2.0:]

1998-2001 [alex] Alexander Kuznetsov <alexan@users.sourceforge.net>

[Author of ~90% of current code, most of current documentation; maintained through version 3.3:]

2002-2007 [dsr] David S. Rushby <woodsplitter@rocketmail.com>

[Finishing touch to v3.3; New Documentation; Current maintainer:]

2008-2011 [paci] Pavel Cisar <pcisar@ibphoenix.cz>

[Significant Contributors:]

2001-2002 [maz] Marek Isalski <kinterbasdb@maz.nu>

Marek made important first steps in removing the limitations of version 2.0 in preparation for version 3.0.

2001 [eac] Evgeny A. Cherkashin <eugeneai@icc.ru>

Evgeny wrote the first version of the distutils build script, which was included in a 2.x point release.

2001-2002 [janez] Janez Jere <janez.jere@void.si>

Janez contributed several bugfixes, including fixes for the date and time parameter conversion code in preparation for version 3.0.

CHAPTER 2

Indices and tables

- genindex
- modindex

Python Module Index

f

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X