
StrixStore

A RDF store allowing transactions and DATALOG rule inference

Copyright © 2010 StrixDB. Freely available under the terms of the [StrixDB license](#).

Content

[Overview](#)

[Status and license](#)

[History](#)

[Download and Install](#)

[How to Use](#)

[with SPARQL protocol](#)

[with Lua scripts](#)

[as a regular DLL](#)

[How to get version number](#)

[Transactions and Concurrency](#)

[Lua Binding Reference](#)

[Graph management](#)

[Rules management](#)

[Namespace management](#)

[User Functions](#)

Overview

StrixStore is a disk-based RDF graph database implementing the [SPARQL](#) and [SPARQL/Update](#) standard. It supports transactions with the one-writer-multiple-readers paradigm. **StrixStore** integrates Datalog rules inference with the SPARQL query language.

StrixStore could be used as a SPARQL and SPARQL/Update server used with Apache HTTP server (an Apache httpd module is provided). It could also be used

as a embedded RDF database if launched from Lua (Lua 5.1 module interface is provided) or launched as a standard Windows DLL from a C/C++/Java (C DLL interface is also provided).

This document focuses on the **StrixStore** specificities (Lua bindings) and on its different interfaces. It is not intended to be a SPARQL or SPARQL/Update tutorial.

Status and License

This documentation is based on **StrixStore** version 0.94.3

The current version is a beta release and could be used free of charges for any purpose. The current version is ruled by the terms of the [StrixDB license](#) for beta releases.

History

- 14-07-2010 Initial release v 0.9
 - 02-09-2010 release 0.91: improved XML/RDF and RFC 2396 compliance
 - 06-11-2010 release 0.92: could be used with [APACHE http server](#). StrixServer no more maintained.
 - 27-02-2011 release 0.94: ACID transactions, scheduled backup, delayed transaction commit.
 - 23-03-2011 release 0.94.3: Bug release.
-

Download and installation

StrixStore can be downloaded as Windows binaries from

<http://www.strixDB.com/download.html>. Just unzip the distribution zip file into a folder. The standard distribution includes :

- *StrixStore.dll*
- *lua5.1.dll* a Windows static linked version of Lua
- *webGet.dll* a Lua 5.1 module needed to download http files from Internet (LOAD command of [SPARQL/Update](#)).
- *lua.exe* a Lua console.
- *mod_strixdb* a httpd Apache module,

- *StrixStore.h* the DLL API of **StrixStore** for using it as a standard Windows DLL.
- *sparql.html* a simple html form to test SPARQL queries.

StrixStore doesn't require extra modules or dependencies and need only LUA5.1.dll (DLL provided with the distribution or available at [LuaBinaries](#)) and the Lua module [webGet](#) (needed for the LOAD command of [SPARQL/Update](#)).

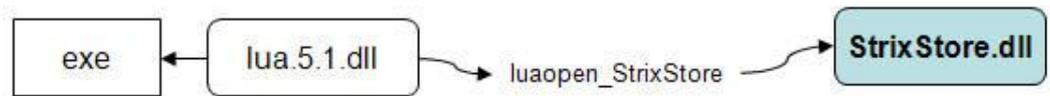
Installation depends of the use of the DLL (see chapter [How to Use](#)). Using from Lua is the simplest way to use **StrixStore** : just use it as a Lua module as bellow....

```
C:\StrixDB\release>lua.exe
Lua 5.1.4 Copyright (C) 1994-2008 Lua.org, PUC-Rio
> require('StrixStore')
>
```

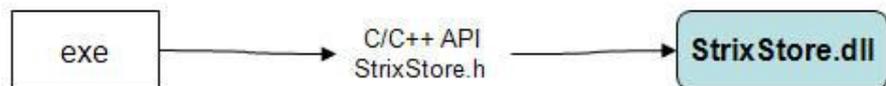
How to use

StrixStore provides 3 different API :

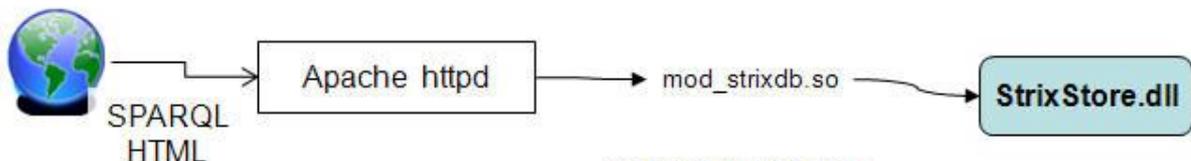
- [httpd](#) (Apache Web Server) module API to deploy a Web server with the SPARQL and SPARQL/Update protocol.
- a Lua API : **StrixStore** could be loaded in Lua with `require('StrixStore')` The Lua functions are described in the [Lua Binding Reference](#) chapter.
- C exported function to use it as a regular DLL from a C/C++/Java program.



using with Lua5.1



using as standard DLL



using with Apache

Notes:

- [Using StrixDB with Apache Server](#) explain how to deploy StrixDB with Apache HTTP Server.
- Compatibility of ISAPI with Microsoft IIS is not well tested and is unsupported

Using with SPARQL protocol

Copy the file **mod_strixdb.so** of StrixDB distribution into the modules folder of Apache Server (with standard installation, this folder is *C:\Program Files\Apache Software Foundation\Apache2.2\modules*).

You could find more information in document [Using StrixDB with Apache Server](#).

Modify the *httpd.conf* configuration file of Apache as below:

```

LoadModule strixdb_module modules/mod_strixdb.so

StrixRoot "C:/Program Files/StrixDB/"
StrixFilename "D:/RDF/strix.db"
StrixDefaultURI "http://mydefault/graph/uri/"

<Location /strixdb>
    SetHandler strix-db-handler
  
```

```
</Location>
```

Explanations : **LoadModule** says to Apache that we want to use StrixDB, **StrixRoot** refers to StrixDB installation folder, **StrixFilename** is the file used by our RDF store, **StrixDefaultURI** is the default graph URI.

Using with Lua scripts

The use of **StrixStore** from a Lua script could be made from :

- a normal Lua script having loaded StrixStore with `require('StrixStore')`
- a Dynamic Lua page (each file in a script enabled folder. see [StrixServer](#)). Dynamic pages are *.lua files or any file (Lua is executed inside the tags `<?lua ... ?>` similar to the php tags `<?php ... ?>`).
- a regular DLL linked with **StrixStore** (see below) through the function `StrixDB_exec`.

More information available in document [Using StrixDB with Apache Server](#).

Using as regular DLL

You could also use **StrixStore** as an embedded RDF store from a C/C++/Java program. The API is available in the [StrixStore.h](#) file. The exported functions are :

- `StrixDB_open` takes a string describing the parameters to open the RDF store in Lua. Returns NULL if ok or error message.
- `StrixDB_close` close the RDF store.
- `StrixDB_lua_State` returns the `lua_State*` for the calling thread.
- `StrixDB_exec` executes the given string as Lua script
- `StrixDB_sparql` takes a SPARQL query (a string) and returns rows of values (the SELECTed variables)
- `StrixDB_update` executes a SPARQL/Update command.

[test_embed.cpp](#) illustrates use of these functions.

How to get the version number

Used with Lua scripting language, you can get the version number with the command:

```
> assert(require 'StrixStore')
> print(rdf._VERSION)
0.94.3
```

Used has a SPARQL server, you can get the version with the url ?infos as in <http://myserver/sparql?infos> (supposing apache runs for myserver and sparql has the been defined for StrixDB handler

```
<Location /sparql>
  SetHandler strix-db-handler
</Location>
```

Transactions and concurrency

All request to the storage are made inside a transaction. If the request failed (for example with a syntax failure in a graph creation or graph update), a rollback is made. Starting from version 0.94, ACID transactions are supported. This means that an error in SPARQL or graph update, an OS failure or a physical error (for example a power failure inside a write transaction) will leave the database in a coherent state. ACID transactions are unfortunately time expensive : each database (graph) modification need a disk write synchronization. StrixDB support **delayed commit** to reduce the cost of disk synchronization.

StrixDB follow the 1 writer, multiple reader paradigm. Most of the requests need only to read the RDF store : They are made with read rights. Multiple-read transactions could occur together. But only one write transaction is allowed. For this reason, write transactions (transactions modifying the database) could occurs at a given time : write transactions are exclusives.

About concurrency: In all the API (C API, Apache module or Lua module), each call to StrixDB is made inside the thread context. A call could is blocked if : (1) need of a write access and some other transactions are not finished, (2) need of a read access and a write transaction is not finished.

All the API are thread safe.

Delayed commit

Starting from version 0.94, delayed commit are supported. When delayed transaction are specified (timeout of commit delay >0), disk synchronization

(required when a transaction ends) is not immediately done after a transaction but is made :

- when the timeout started at the last write transaction comes at end OR
- when an other thread wants to access as reader or writer to the database.

By default, delayed commit are disabled.

Backup scheduler

Lua Binding Reference

All the functions loaded with **require('StrixStore')** are in a table named **rdf**.

rdf.help()

Print a help summary of all available functions.

<bool> = rdf.open {<params>}

Take as argument a table specifying RDF store parameters. Returns **true** if the database was successfully opened or created. The parameters are :

file=<string>	the database file on disk	REQUIRED
uri=<string>	the URI of default graph (as specified by SPARQL)	REQUIRED for new database
initFile=<string>	the Lua script to execute at each start of StrixStore	
truncate=<boolean>	if true, the database is deleted just before each start.	default= false
backupFile=<string>	the backup fileName.	
backupPeriod=<integer>	the backup period in hours.	
delay=<integer>	delayed transaction commit time in seconds (delayed disk	default=0

synchronization).
 sparqlFunctions=<lua table> the lua table specify DLL that could be used in SPARQL **FILTER** (or Datalog functions). See chapter [user functions](#)

Advanced parameters

poolSize=<integer>	the number of pages cached in the poolSize. Each page has 4K. Big poolSize improves speed but consumes memory.	default=100*1024
initIndex=<integer>	Size of the initial index (StrixStore is a database bitmap).	default=8*1024*1024
quantum=<integer>	Size of the new allocated quantum (bitmap) when the allocated file is full.	default=512*1024*1024
safe=<boolean>		default=false
noBuffer=<boolean>		default=false
writeThrough=<boolean>	if true, wait disk write acknowledge event for each write transaction (safer but slower with SPARQL/Update transactions).	default=false

An Lua code snippet using **rdf.open**. This snippet is useful for a script that could be used both from a Lua script (not loading by default StrixStore and opening database) or from a Lua dynamic page.

```
require('StrixStore')
if rdf.isOpen()==false then
  rdf.open{file='rdfStore.db',uri='http://myURIRoot/'}
end
```

<bool> = rdf.open (<string>)

Shortcut command used to open a database. This is a shortcut for **rdf.open{file=<string>, uri='http://<hostname>' }**

If the database was not already created, this command create a new database. This created database has default parameters and use the hostname as default **uri**.

rdf.close()

Without comment.. close the RDF store.

<boolean> = rdf.isOpen()

Returns true if RDF store is open, else returns false.

rdf.stats()

Returns informations about memory usage.

rdf.sparql (<query> [,<option>])

Takes as first argument a SPARQL query (a Lua string). Without second argument, returns a Lua iterator. Example:

```
local query = [[PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?r ?name
FROM <test/friends/>
WHERE { ?r foaf:name ?name}]]

for resource,name in rdf.sparql(query) do
    print(resource, name)
end
```

The variables used in the SELECT of the SPARQL query are bounded to the iterator variable by they order of declaration (and without regard of their name).

Three options could be used for this function:

- 'print' print the SPARQL variables (no iterator is returned).
- 'table' returns a table of rows. Each row is a table of variables with the SPAQL query results.
- 'explain' print the SPARQL virtual machine bytecode (no iterator is returned).

rdf.sparqlTyped (<query> [,<option>])

Works as previous function but each object

rdf.update(<query> [,<option>])

Submit the SPARQL/Update query (a Lua string).

The only possible option is 'explain'. With explicit add/retract in query, nothing is to explain. Only the use of WHERE could need a bytecode compilation and produces some results with 'explain' option.

rdf.pragma()

TODO

rdf.pragma(<var>,<boolean>)

TODO

rdf.help ()

Without comment (print help).

rdf.path(<opt>)

returns useful path location.

- <opt> = 'db' returns the path of the RDFstore file
- <opt> = 'loader' returns the path of the loading application
- <opt> = 'StrixStore' returns the path of the installation file of StrixStore.dll

Examples:

```
print( rdf.path('loader') )  
C:/Program Files/Apache Software Foundation/Apache2.2/bin/
```

...means that the loader was in the Apache httpd folder.

For Dynamic Lua pages (used with Apache Server), see also the **apache.root()** function.

Graph management functions

rdf.graph.import { uri=<URI>, file=<URL> [format='xml'|'turtle'] }

If the URL of the file is not local, download the file from internet automatically using HTTP GET and default MS Internet Explorer settings for proxy).

Format is nor required (use of the file extension to decide the format).

- *.rdf and *.xml are associated with XML format
- *.ttl is associated with TURTLE format
- *.nt is associated with N-TRIPLES format.

rdf.graph.create (<URI> , <datas>)

Create a graph of the given <URI> (or replace if already present) and put the **datas** inside.

The datas MUST have the TURTLE format.

rdf.graph.delete (<URI>)

Remove the graph of given URI from the RDF store.

rdf.graph.export { uri=<URI> [,file=<PATH>] [,format='XML'|'turtle'|'triples'] [,compact=0|1|2] }

Export the graph of given URI.

If no file is specified, this function use Lua **print** function. This means :

- standard output if used from a Lua console
- HTTP response if used inside a Lua scripting page (see [Using StrixDB with Apache HTTP Server](#)).

If no format is provided, it use the file extension to decide the format (see **rdf.graph.import**).

If **compact=1**, export will use the current namespaces (rdf, rdfs, owl, foaf, xsd).

If **compact=2**, export will create all the namespaces for making the output smaller (but the 2 pass process take more time).

rdf.graph.print ()

print the graph meta datas of all graph stored in the RDFstore. Example :

```
> rdf.graph.list()
```

Triples	Time Stamp	URI (source)
0	2010-11-23T00:36:54Z	http://MyStore/
374	2010-11-23T00:36:54Z	http://MyStore/modeles/ (E:/SOMEGRAPH/RDF/modeles-tome4.ttl)
28	2010-11-23T00:36:54Z	http://MyStore/schema/ (E:/SOMEGRAPH/RDF/SOMEGRAPH-schema.ttl)

Meta datas are the number of triples, the **time stamp** (unix time of last graph update or modification), the URI of the graph (this is not the URL... URL is the RDF command to get the graph), the source of the file.

rdf.graph.list ()

returns a Lua table of graph meta datas. The provided datas are the same as for the **rdf.graph.print** command. Example :

```
> table.foreach(rdf.graph.list(),
  function(k,v) print('graph=',k) table.foreach(v, print) print() end
)

graph= http://MyStore/
source
tripleCount 0
blankCount 0
DEFAULT_GRAPH true
uri http://MyStore/
timeStamp 2010-11-23T00:36:54Z

graph= http://MyStore/modeles/
source E:/SOMEGRAPH/RDF/modeles-tome4.ttl
tripleCount 374
blankCount 48
uri http://MyStore/modeles/
timeStamp 2010-11-23T00:36:54Z

graph= http://MyStore/schema/
source E:/SOMEGRAPH/RDF/SOMEGRAPH-schema.ttl
tripleCount 28
blankCount 0
uri http://MyStore/schema/
timeStamp 2010-11-23T00:36:54Z
```

rdf.graph.rename(<URI>, <URI>)

Rename the graph of first <URI> with the second <URI>. Destination <URI> must not be an existing graph.

rdf.graph.copy(<URI>, <URI>)

Copy the graph of first <URI> into the second <URI>. Destination <URI> must not be an existing graph.

```
rdf.graph.clear ( <URI> )
```

Remove all triples from graph of given <URI>.

```
<boolean> = rdf.graph.exists( <URI> )
```

returns true if the graph of given <URI> exists.

```
rdf.graph.triples ( <URI> )
```

print the triples of graph of given <URI>. Has the same result that `rdf.graph.export { uri=<URI>,format='triples'}`

```
<boolean> = rdf.graph.equal( <URI1> , <URI2> )
```

return true if the graph of first <URI> is equivalent to the graph of second <URI>.

Equivalence is calculated with a graph [homomorphism](#) algorithm for all triples using blank nodes.

Avoid using it with graphs having thousand of **blank nodes** (could take a lot of time).

```
rdf.graph.relocate( <URI1>, <URI2> , <URI3> )
```

This function update the graph of the first <URI1>.
All resource of the graph that are child of <URI2> will be relocated to <URI3>.

Exemple: `rdf.graph.relocate(... , 'http://bad/person', 'http://good/people')` will change the triplets

```
http://bad/person/Neron rdfs:label "Emperor Neron"
```

into

```
http://good/people/Neron rdfs:label "Emperor Neron"
```

```
rdf.graph.update { [ { 'add'|'retract', <URI>, <turtleDatas> } ] }
```

This function is provided to update multiple graphs inside a same transaction (to avoid semantic inconsistency between graphs if an error occurs).

This function was created before implementation of SPARQL/Update. Use SPARQL/Update is better recommended (even if more complex syntax).

TO DO EXAMPLE

```
rdf.graph.add {<args>}
```

This function was created before implementation of SPARQL/Update. Use SPARQL/Update is better recommended (even if more complex syntax).

TO DO EXAMPLE

```
rdf.graph.retract {<args>}
```

This function was created before implementation of SPARQL/Update. Use SPARQL/Update is better recommended (even if more complex syntax).

TO DO EXAMPLE

Rules management functions

```
rdf.rules.import {<args>}
```

TO DO

```
rdf.rules.create {<args>}
```

TO DO

```
rdf.rules.export {<args>}
```

TO DO

rdf.rules.list ()

TO DO

rdf.rules.print ()

TO DO p>

Namespace management functions

<string> = rdf.namespace.prefix (<uri>)

TO DO

rdf.namespace.set (<uri>,<prefix>)

TO DO

rdf.namespace.list ()

TO DO p>

User functions