

Sedna Programmer's Guide

Contents

1	Client Application Programming Interfaces	1
1.1	Java API	1
1.1.1	Sessions	1
1.1.2	Transactions	2
1.1.3	Statements	3
1.1.4	Results	4
1.1.5	Exceptions	4
1.1.6	Code example	4
1.2	C API	8
1.2.1	Connecting to a Database	9
1.2.2	Transactions	10
1.2.3	Executing Queries and Updates and Retrieving Result Data	11
1.2.4	Loading Data	12
1.2.5	Error Handling	13
1.2.6	Example Code	13
1.3	Scheme API	17
1.3.1	Sessions	17
1.3.2	Transactions	18
1.3.3	Executing Queries	18
1.3.4	Bulk load from stream	20
1.3.5	Higher-level function for a transaction	20
1.3.6	Error handling	20
1.3.7	Code example	21
2	Database Language	23
2.1	Data Manipulation Language	23
2.1.1	XQuery Support	23
2.1.2	Update Language	24
2.1.3	Bulk load	26

2.2	XQuery Functions	26
2.2.1	Standard XQuery Functions	26
2.2.2	Index-scan Functions	27
2.2.3	SQL Connection	27
2.3	Data Definition Language	32
2.3.1	Managing Documents	32
2.3.2	Managing Collections	34
2.3.3	Managing Indices	35

1 Client Application Programming Interfaces

The Sedna client application programming interfaces (APIs) provides programmatic access to Sedna from client applications developed in host programming languages. This section describes the client APIs distributed with Sedna.

1.1 Java API

The Java API provides programmatic access to XML data from the Java programming language. Using the Java API, applications written in the Java can access one or more databases of the Sedna DBMS and manipulate database data using the database language described in Section 2.

Application working with the Sedna DBMS through the Java API operates with the following notions of the Java API: session, transaction, statement, result.

1.1.1 Sessions

To start working with Sedna application has to open a session via establishing an authenticated connection with the server. The Java API defines the `SednaConnection` interface to represent a session.

To open a session application uses static method `getConnection` of the `DatabaseManager` class.

```
public static SednaConnection getConnection(java.lang.String url,
                                           java.lang.String DBName,
                                           java.lang.String user,
                                           java.lang.String password)
                                           throws DriverException
```

Parameters:

`url` - the name of the computer where the Sedna DBMS is running. This parameter may contain a port number. If the port number is not specified, the default port number (5050) is used.

`DBName` - the name of the database to connect.

`user` - user name.

`password` - user password.

If the connection is established and authentication succeeds the method returns an object that implements the `SednaConnection` interface. Otherwise, `DriverException` is thrown.

Application can close session using the `close` method of the `SednaConnection` interface.

```
public void close() throws DriverException
```

If the server does not manage to close connection properly the `close` method throws `DriverException`.

The `isClosed` method retrieves whether this connection has been closed or not. A connection is closed if the method `close` has been called on it or if certain fatal errors have occurred.

```
public boolean isClosed()
```

1.1.2 Transactions

Application can execute queries and updates against specified database only in the scope of a transaction. That is, once a session has been opened, application can begin a transaction, execute statements and commit this transaction.

In the session transactions are processed sequentially. That is, application must commit a begun transaction before beginning a new one.

To specify transaction boundaries application uses methods of the `SednaConnection` interface: `begin`, `commit`.

The `begin` method begins a new transaction.

```
public void begin() throws DriverException
```

If the transaction has not begun successfully the `begin` method throws `DriverException`.

The `commit` method commits a transaction.

```
public void commit() throws DriverException
```

If the transaction has not been committed successfully the `commit` method throws `DriverException`.

The Java API does not provide auto-commit mode for transactions. That is, every transaction must be explicitly started (by means of `begin`) and committed (by means of `commit`). If session is closed (by means of `close`) before a transaction committed, server does rollback for that transaction, and `close` throws `DriverException`.

1.1.3 Statements

The `SednaStatement` interface provides methods for loading documents into the database, executing statements of the database language defined in Section 2 and retrieving results that statements produce. `SednaStatement` object is created using the `createStatement` method of the `SednaConnection` interface:

```
public SednaStatement createStatement() throws DriverException
```

`SednaStatement` object may only be created on an open connection. Otherwise, the `createStatement` method throws `DriverException`.

To load the document into the database use:

```
public void loadDocument(InputStream in, String doc_name)
                        throws DriverException, IOException
```

The `in` parameter is an input stream to get the document from. The `doc_name` parameter is the name for this document in the database.

To load the document into the specified collection of the database use:

```
public void loadDocument(InputStream in, String doc_name, String col_name)
                        throws DriverException, IOException
```

The `in` parameter is an input stream to get the document from. The `doc_name` parameter is the name for this document in the database. The `col_name` parameter is the name of the collection to load the document into.

To execute a statement the `execute` methods of the `SednaStatement` are used.

```
public boolean execute(java.lang.String queryText)
                        throws DriverException
```

The `queryText` parameter contains the text of the statement.

```
public boolean execute(java.io.InputSteram in) throws DriverException
```

The `in` parameter is some input stream to read an XQuery statement from.

Some statements (such as XQuery statements or the retrieve metadata command) produce the result. In case of such statements, the `execute` methods return true and the result can be obtained as described in Section 1.1.4. In case of statements that do not produce the result (such as updates or bulk load), the `execute` methods return false.

1.1.4 Results

The `SednaSerializedResult` interface represents the result of the statement evaluation. Application can obtain the result using the `getSerializedResult` method of the `SednaStatement` interface.

```
public SednaSerializedResult getSerializedResult()
```

The result of the non-XQuery statement evaluation is retrieved as a sequence with only one item, where the item is a string. For example, in case of the retrieve descriptive schema command the result is a sequence with an item that is a descriptive schema represented as a string. The result of the XQuery statement evaluation is retrieved as a sequence of items, where every item is represented as a string.

Application can use the `next` methods of the `SednaSerializedResult` interface to iterate over the result sequence:

```
public String next() throws DriverException
```

The method returns an item of the result sequence as a string. If the sequence has ended it returns null. It throws `DriverException` in the case of error.

```
public int next(java.io.Writer writer) throws DriverException
```

The method writes an item of the result sequence to some output stream using `writer`. It returns 0 if an item was retrieved and written successful, and 1 if the result sequence has ended. It throws `DriverException` in the case of error.

Please, notice, that in the current version of the Sedna DBMS application has to execute statements and use the results of their execution sequentially. To explain this note, suppose application execute a statement, obtains a result of that statement execution and iterates over this result. If, after that, application executes next statement, it cannot iterate over the result of the previous statement any more.

1.1.5 Exceptions

The `DriverException` class provides information about errors that occur while application works with the Sedna DBMS through the Java API. `DriverException` is also thrown when application loses its connection with the server.

1.1.6 Code example

In this section we provide fragments of an example program that uses the Java API to work with Sedna DBMS.

```

import ru.ispras.sedna.driver.*;

import java.io.*;
import java.net.*;
import java.util.*;

    ...
    ...

    System.out.println("Client started\n");

// Set up arguments for the call to the getConnection method.
    String url = "localhost";
    String dbname = "sample-db";
    String user = "guest";
    String password = "guest";
    String item;
    int count;

    SednaConnection con = null;
// Get a connection
    try{
        con = DatabaseManager.getConnection(url, dbname, user, password);

// begin transaction
        con.begin();

// creates statements
        SednaStatement st1 = con.createStatement();
        SednaStatement st2 = con.createStatement();

// execute a statement. the statement is a bluk load.
        System.out.println("Loading data");
        boolean call_res =
            st1.execute("LOAD \"region.xml\" \"region\"");
        if(!call_res) // if call_res is false the statement was an update
        {
            System.out.println("Bulk load succeeded");
            System.out.println("=====\n");
        }
    }

```

```

// execute query.
System.out.println("Executing query");
call_res =
    st2.execute("document(\"region\")/*/*");
if(call_res) // if call_res is true the statement
              // was not an update and we can use
              // SerializedResult object
{
    System.out.println("Result:");
    SednaSerializedResult pr1 = st2.getSerializedResult();
    item = null;
    item = pr1.next();
    count = 1;
    while(item!=null)
    {
        System.out.println(count+" item: "+item);
        item = pr1.next();
        System.out.println("=====\n");
        count++;
    }
}

// execute update.
System.out.println("Executing update");
call_res =
    st2.execute("UPDATE delete document(\"region\")//africa");
System.out.println("=====\n");

// execute query.
System.out.println("Executing query");
call_res = st2.execute("document(\"region\")/*/*");
if(call_res) // if call_res is true the statement was not
              // an update and we can use PlaneResult object
{
    System.out.println("Result:");
    SednaSerializedResult pr3 = st2.getSerializedResult();
    item = null;
    item = pr3.next();
    count = 1;
    while(item!=null)
    {
        System.out.println(count+" item: "+item);
    }
}

```

```

        item = pr3.next();
        System.out.println("=====\n");
        count++;
    }

    // commit the transaction
    con.commit();
}

// close the connection
con.close();
}

catch(DriverException e)
{

    System.out.println(e.toString());
}

...

```

The source code of this example program can be found in

[win:] INSTALL_DIR\examples\applications\java\Client.java

[linux:] INSTALL_DIR/examples/applications/java/Client.java

where INSTALL_DIR refers to the directory where Sedna is installed.

Before running the example make sure that the Sedna DBMS is installed and do the following steps:

1. Start Sedna by running the following command:

```
se_gov
```

If Sedna is started successfully it prints "GOVERNOR has been started in the background mode".

2. Create a new database **sample-db** by running the following command:

```
se_cdb sample-db
```

If the database is created successfully it prints "The database 'sample-db' has been created successfully".

3. Start the sample-db database by running the following command:

```
se_sm sample-db
```

If the database is started successfully it prints "SM has been started in the background mode".

You can compile and run the example by following the steps listed below:

1. In the file `Client.java`, the `url` parameter denotes the name of the computer where the server is running. If Sedna DBMS is not running on the same computer edit the file `Client.java` to set the value of the `url` parameter. This will look as something like the following:

```
String url = "computer-name";
```

if Sedna DBMS is running on the same computer you do not need to edit this parameter.

2. To compile the example, run the script

```
[win:] Clientbuild.bat
```

```
[linux:] Clientbuild.sh
```

located in the same folder as `Client.java`.

3. To run the compiled example, use the script

```
[win:] Client.bat
```

```
[linux:] Client.sh
```

located in the same folder as `Client.java`.

1.2 C API

libsedna is the C application programmer's interface to Sedna XML DBMS. libsedna is a set of library functions that allow client programs to access one or more databases of Sedna XML DBMS and manipulate database data using database language (XQuery and XUpdate) described in Section 2.

libsedna library is supplied with two header files: "`libsedna.h`" and "`libbase.h`". Client programs that use libsedna must include the header file `libsedna.h`, must link with the libsedna library and provide the compiler with the path to the directory where "`libbase.h`" and "`libsedna.h`" files are stored.

Sedna XML DBMS is supplied with libsednamt library also. libsednamt library is libsedna library compiled in multithreaded mode. Applications that need to be compiled in multithreaded mode must link with libsednamt library.

Below in the example we consider libsedna (single-threaded library).

1.2.1 Connecting to a Database

To start working with Sedna an application has to open a session via establishing an authenticated connection with the server using SEconnect:

```
int SEconnect(SednaConnection* conn,
              const char* url,
              const char* db_name,
              const char* login,
              const char* password)
```

This function opens a session to the database, using the following parameters:

conn - is a pointer to an instance of SednaConnection type, that is associated with a session. The instance of SednaConnection type is initialized by the SEconnect if the session is open successfully.

url - the name of the computer where the Sedna DBMS is running. This parameter may contain a port number. If the port number is not specified, the default port number (5050) is used.

db_name - the name of the database to connect.

login - user name.

password - user password.

SEconnect returns:

positive values:

SEDNA_SESSION_OPEN - connection to the database is established, authentication is passed successfully, and session is opened.

negative values:

SEDNA_AUTHENTICATION_FAILED - authentication was not passed, session is not open.

SEDNA_OPEN_SESSION_FAILED - failed to open session.

SEDNA_ERROR - some error is occurred, session is not open.

To access multiple databases at one time or to better process its complex logic an application can have several sessions open at one time.

When the application finishes its work with the database, it must close the session using SEclose:

```
int SEclose(SednaConnection* conn)
```

parameters:

conn - a pointer to an instance of the SednaConnection type, associated with a session to be closed.

SEclose returns:

SEDNA_SESSION_CLOSED - session is closed successfully.

SEDNA_CLOSE_SESSION_FAILED - session was closed with errors.

SEDNA_ERROR - some error occurred.

1.2.2 Transactions

Application can execute queries and updates against the specified database only in the scope of a transaction. That is, once a session has been opened, application can begin a transaction, execute statements and commit this transaction. In a session transactions are processed sequentially. That is, application must commit a begun transaction before beginning a new one. To specify transaction boundaries application uses functions `SEbegin`, `SEcommit`:

```
int SEbegin(SednaConnection* conn)
```

parameters:

`conn` - a pointer to an instance of the `SednaConnection` type.

`SEbegin` returns:

positive value:

`SEDNA_BEGIN_TRANSACTION_SUCCEEDED` - transaction is begun.

negative value:

`SEDNA_BEGIN_TRANSACTION_FAILED` - failed to begin a transaction.

`SEDNA_ERROR` - some error occurred.

```
int SEcommit(SednaConnection* conn)
```

parameters:

`conn` - a pointer to an instance of the `SednaConnection` type.

`SEcommit` returns:

positive value:

`SEDNA_COMMIT_TRANSACTION_SUCCEEDED` - transaction is committed.

negative value:

`SEDNA_COMMIT_TRANSACTION_FAILED` - failed to commit transaction.

`SEDNA_ERROR` - some error occurred.

To rollback transaction use `SErollback` function:

```
int SErollback(SednaConnection* conn)
```

parameters:

`conn` - a pointer to an instance of the `SednaConnection` type.

`SErollback` returns:

positive value:

`SEDNA_ROLLBACK_TRANSACTION_SUCCEEDED` - rollback transaction succeeded.

negative value:

`SEDNA_ROLLBACK_TRANSACTION_FAILED` - failed to rollback transaction.

`SEDNA_ERROR` - some error occurred.

1.2.3 Executing Queries and Updates and Retrieving Result Data

To execute a statement (query or update) use `SEexecute` function:

```
int SEexecute(SednaConnection* conn, const char* query)
```

parameters:

`conn` - a pointer to an instance of the `SednaConnection` type.

`query` - a null-terminated string with an XQuery query or XUpdate update statement.

returns:

positive values:

`SEDNA_QUERY_SUCCEEDED` - specified query succeeded and result data can be retrieved.

`SEDNA_UPDATE_SUCCEEDED` - specified update succeeded.

`SEDNA_BULK_LOAD_SUCCEEDED` - specified update (bulk load- see Section 2) succeeded.

negative values:

`SEDNA_QUERY_FAILED` - specified query failed.

`SEDNA_UPDATE_FAILED` - specified update failed.

`SEDNA_BULK_LOAD_FAILED` - bulk load failed.

`SEDNA_ERROR` - some error occurred.

if `SEexecute` function returns `SEDNA_QUERY_SUCCEEDED`, the result data can be retrieved. The result of XQuery query evaluation is a sequence of items, where every item is represented as a string. Use the `SEnext` function to iterate over the sequence and `SEgetData` function to retrieve current item of the sequence.

```
int SEnext(SednaConnection* conn)
```

parameters:

`conn` - a pointer to an instance of the `SednaConnection` type.

returns:

positive values:

`SEDNA_NEXT_ITEM_SUCCEEDED` - moving to a next item succeeded, and the item can be retrieved.

negative values:

`SEDNA_NEXT_ITEM_FAILED` - failed to get next item.

`SEDNA_RESULT_END` - the result sequence is ended, no result data to retrieve.

`SEDNA_NO_ITEM` - there was no succeeded query that produced the result data, no result data to retrieve.

`SEDNA_ERROR` - some error occurred.

```
int SEgetData(SednaConnection* conn, char* buf, int bytes_to_read)
```

parameters:
conn - a pointer to an instance of the SednaConnection type.
buf - pointer to the buffer that receives the data got from the server.
bytes_to_read - number of bytes to be read from the server into the buffer.
returns:
positive values:
number of bytes actually read from the server and put into the buffer.
zero - no data was read from the server and put into the buffer because of the item end. (use **SEnext** to move to the next item of the result)
negative values:
SEDNA_GET_DATA_FAILED - failed to get data.
SEDNA_ERROR - some error occurred.

1.2.4 Loading Data

XML data can be loaded into a database using "LOAD" statement of the Data Manipulation Language (see 2.1.3). Besides, libsedna library provides **SEloadData** and **SEendLoadData** functions to load well-formed XML documents divided into parts of any convenient size.

SEloadData functions loads a chunk of XML document:

```
int SEloadData(SednaConnection* conn,
               const char* buf,
               int bytes_to_load,
               const char* doc_name)
```

parameters:
conn - a pointer to an instance of the SednaConnection type.
buf - a buffer with chunk of XML document.
bytes_to_load - number of bytes to load.
doc_name - name of the document in database the data loads to.
returns:
positive values:
SEDNA_DATA_CHUNK_LOADED - chunk of XML document was loaded.
negative values:
SEDNA_ERROR - some error occurred. Data were not loaded.

When the whole document is loaded using **SEloadData**, application must use **SEendLoadData** to notify server that transfer of XML document is finished:

```
int SEendLoadData(SednaConnection* conn)
```

parameters:
conn - a pointer to an instance of the SednaConnection type.
returns:

positive values:

SEDNA_BULK_LOAD_SUCCEEDED - XML document was successfully loaded into the database.

negative values:

SEDNA_BULK_LOAD_FAILED - Failed to load XML document into the database.

SEDNA_ERROR - some error occurred.

1.2.5 Error Handling

If the function returns negative value, application can get the error info in the form of string using SEgetLastErrorMsg function:

```
char* SEgetLastErrorMsg(SednaConnection* conn)
```

parameters:

conn - a pointer to an instance of the SednaConnection type.

The function returns the information about the last error occurred in this session.

1.2.6 Example Code

```
/* This is an example application that works with Sedna XML DBMS through
 * C-API using libsedna library.
 * The application opens a session to "sample-db" database,
 * begins a transaction, loads document
 * "region.xml" to "sample-db" database, executes XQuery query,
 * retrieves every item of the
 * result data, commits the transaction and closes the session.
 */

#include "libsedna.h"
#include <stdio.h>

int main()
{
    SednaConnection conn;
    printf("Client started.\n");

    const char* url = "localhost";
    const char* db_name = "sample-db";
    const char* login = "guest";
    const char* password = "guest";
```

```

//connecting to database "sample-db" with login "guest", password "guest"
int res = SEconnect(&conn, url, db_name, login, password);

if(res != SEDNA_SESSION_OPEN)
{
    printf("Session starting failed %s\n", SEgetLastErrorMsg(&conn));
    return -1;
}

//begin transaction
res = SEbegin(&conn);
if(res != SEDNA_BEGIN_TRANSACTION_SUCCEEDED)
{
    printf("Begin transaction failed %s\n", SEgetLastErrorMsg(&conn));
    return -1;
}

    // load data from file "region.xml" into the document "regions"
    res = SEexecute(&conn, "LOAD \"region.xml\" \"regions\"");
if(res != SEDNA_BULK_LOAD_SUCCEEDED)
{
    printf("Bulk load failed %s\n", SEgetLastErrorMsg(&conn));

    //closing session
    SEclose(&conn);
    return -1;
}

// execute XQuery query
    res = SEexecute(&conn, "document(\"regions\")/*/*");
if(res != SEDNA_QUERY_SUCCEEDED)
{
    printf("Query failed %s\n", SEgetLastErrorMsg(&conn));
    //closing session
    SEclose(&conn);
    return -1;
}

//iterate over the result sequece and retrieve the result data
char buf[1024];
int bytes_read;
res = SEnext(&conn);

```

```

while((res != SEDNA_RESULT_END)&&(res != SEDNA_ERROR))
{
    bytes_read = SEgetData(&conn, buf, 1024);
    buf[bytes_read] = '\0';
    printf("result item: %s\n", buf);
    res = SEnext(&conn);
}

if(res == SEDNA_RESULT_END)
{
    printf("Result end \n");
}

//committing the transaction
res = SEcommit(&conn);
if(res != SEDNA_COMMIT_TRANSACTION_SUCCEEDED)
{
    printf("Commit transaction failed %s\n", SEgetLastErrorMsg(&conn));
    return -1;
}

//closing session
res = SEclose(&conn);
if(res != SEDNA_SESSION_CLOSED)
{
    printf("Session was closed with errors %s\n", SEgetLastErrorMsg(&conn));
    return -1;
}
return 1;
}

```

The source code of this example program can be found in

[win:] INSTALL_DIR\examples\applications\c\Client.c

[linux:] INSTALL_DIR/examples/applications/c/Client.c

INSTALL_DIR refers to the directory where Sedna is installed.

Before running the example make sure that the Sedna DBMS is installed and do the following steps:

1. Start Sedna by running the following command:

```
se_gov
```

If Sedna is started successfully it prints "GOVERNOR has been started in the background mode".

2. Create a new database **sample-db** by running the following command:

```
se_cdb sample-db
```

If the database is created successfully it prints "The database 'sample-db' has been created successfully".

3. Start the sample-db database by running the following command:

```
se_sm sample-db
```

If the database is started successfully it prints "SM has been started in the background mode".

You can compile and run the example by following the steps listed below:

1. In the file `Client.c`, the `url` parameter specifies the name of the computer where the server is running. If Sedna DBMS is not running on the same computer edit the file `Client.c` to set the value of the `url` parameter. This will look as something like the following:

```
const char* url = "computer-name";
```

if Sedna DBMS is running on the same computer you do not need to edit this parameter.

2. To compile the example use

```
[win:] Clientbuild.bat
```

```
[linux:] Clientbuild.sh
```

located in the same folder as `Client.c`.

3. To run the compiled example use

```
[win:] Client.exe
```

```
[linux:] Client
```

located in the same folder as `Client.c`.

1.3 Scheme API

Scheme API allows querying and managing XML data from an application written in Scheme. Scheme API follows the spirit of SchemeQL [8], an implementation of SQL 1992 for Scheme. The results of XQuery queries to the Sedna DBMS via the Scheme API can be represented either in XML or SXML [9].

1.3.1 Sessions

For working with the Sedna DBMS from Scheme, you should first establish a connection between the Scheme API driver and the Sedna DBMS. Here are two functions to manage connections:

```
> (sedna:connect-to-database host db-name user password)
: String String String String -> connection-object
```

Establishes a connection between the client application and the Sedna DBMS. Returns a 'connection' object which encapsulate information about the connection. The arguments are strings that denote connection parameters:

- **host** is the host where the Sedna DBMS is running. If it is on the same machine as your Scheme application, you can use "localhost" for the value of this parameter.
- **db-name** is the name of the database you want to work with. You are to establish a separate connection for each database you would like to work with.
- **user** is your user name for the session. You can use "guest" for the value of this parameter.
- **password** is your password for the session. You can use the "guest" value of this parameter.

To disconnect from the database, you can use the following function:

```
> (sedna:disconnect-from-database connection)
: connection-object -> void
```

Closes the connection represented by the **connection** object. If server fails to close the connection, the function closes the connection forcibly from the client side and raises the exception, as discussed in subsubsection 1.3.6.

1.3.2 Transactions

After the connection with a database is established and the session is begun, you can run zero or more transactions in this session. Transactions are to be run sequentially, with no more than a single transaction at a time, so you should commit your running transaction before starting a new one.

To begin a new transaction, the following function is provided:

```
> (sedna:begin-transaction connection)
: connection-object -> void
```

It accepts the `connection` object (earlier created by `sedna:connect-to-database` function) and starts a new transaction. If the transaction could not be created, the exception is raised, as discussed in subsection 1.3.6.

To end your running transaction, you are provided with the following function:

```
> (sedna:end-transaction connection action)
: connection-object, symbol -> void
```

If `action` is `'COMMIT` the transaction in the given connection will be committed, if `'ROLLBACK` is given, the transaction will be rolled back.

Note 1 *In the current version of the Sedna DBMS, transaction rollback is not supported. Calling `sedna:end-transaction` with a `'ROLLBACK` action would currently result to an exception.*

1.3.3 Executing Queries

Within a transaction, you can execute zero or more queries to the database.

```
> (sedna:execute-query connection query)
: connection-object, string -> result
```

The first argument is the `connection` object, earlier created by `sedna:connect-to-database` function. The `query` is represented as a string and can express one of the 3 kinds of queries:

- XQuery query – for querying data from the database, without modifying it;
- Update command – for making modifications to the database you work with;
- Bulk load command – for loading a new XML document from the local file to the database.

If an error occurs at the server side during query execution (e.g. the requested document not found), the function raises an exception that contains the message about the error occurred.

In the successful case of query execution, `sedna:execute-query` returns `#t` for the last 2 kinds of queries, to denote a successful update made to the database. The XQuery `query` results to a sequence of items, which are evaluated lazily and are represented as a pair:

```
xquery-result ::= (cons current-item promise)
```

This way of result representation is very close to the notion of SchemeQL *cursor* [8] (with the only difference in that the Scheme API driver returns XQuery items instead of table rows returned by SchemeQL). The first member of the pair is the `current-item` represented in SXML, and the second member of the pair holds a promise (which can be forced) to evaluate and return the next item in the result sequence.

To iterate over the result sequence, you can use the function:

```
> (sedna:next xquery-result)
: xquery-result -> xquery-result or '()
```

which forces the evaluation of the following items in the result sequence, until the end of the sequence is reached.

Such a design allows you to process a query result in a lazy stream-wise fashion and provides you with an ability to process large query results, which would not otherwise fit in the main memory.

However, for query results that are not very large, you may find it convenient to evaluate them all at once and represent the result sequence as a Scheme list. Scheme API provides a function that converts the `xquery-result` into the list that contains all items of the result sequence:

```
> (sedna:result->list xquery-result)
: xquery-result -> (listof item)
```

To obtain the result sequence in the form of the list, you can execute your queries as a superposition of the above considered functions:

```
(sedna:result->list
 (sedna:execute-query connection query))
```

It should be noted that the XQuery query result in that case exactly corresponds to the term of a *node-set* in the XPath implementation in Scheme SXPath [12].

If you want to obtain your query results in XML instead of SXML, you can use the function

```
> (sedna:execute-query-xml connection query)
: connection-object, string -> result
```

It is the counterpart of earlier discussed `sedna:execute-query` and has the same signature, but represents query results in XML. The function returns a sequence of items, in the form of `xquery-result` discussed above, but the `current-item` is now a string containing the representation for the current item in the form of XML.

1.3.4 Bulk load from stream

The following wrapper function provides a natural way to load an input stream containing an XML document into your database:

```
> (sedna:bulk-load-from-xml-stream
      connection port document-name . collection-name)
: connection-object, input-port, string [, collection-name] -> boolean
```

As for `sedna:execute-query`, the first argument here is the `connection` object, earlier created by `sedna:connect-to-database` function. Argument `port` is a Scheme input port and is supposed to contain a well-formed XML document. Argument `document-name` specifies the name that will be given to the XML document within a database. If the 4-th argument `collection-name` is supplied, the XML document is loaded into the collection which name is specified by the `collection-name` argument. If the 4-th argument of the function call is not supplied, the XML document is loaded into the database as a standalone document.

By allowing you to specify the input port you would like to use, this function provides a higher-level shortcut for `sedna:execute-query` when bulk load from stream is concerned. For more details on bulk load, see Sect. 2.1.3.

1.3.5 Higher-level function for a transaction

This higher-level function provides a convenient way for executing a transaction consisting of several queries, within a single function call:

```
> (sedna:transaction connection . queries)
: connection-object, string* -> result
```

This function starts a new transaction on the `connection` objects and executes all the `queries` given in order. If no exception occurs, the function commits the transaction and returns the result of the last query. If any exception occurred during query execution, the function sends rollback to the Sedna DBMS and passes along the exception to the application.

1.3.6 Error handling

Error handling in the Scheme API driver is based on the exception mechanism suggested in the (currently withdrawn) SRFI-12 [10]. The SRFI-12 exception mechanism is natively supported in the Chicken Scheme compiler [11]. In the Scheme API driver, we also provide the SRFI-12 implementation for PLT.

1.3.7 Code example

This section presents an example that illustrates the application of the Scheme API driver.

```
; Load the necessary Scheme API driver files
(load "collect-sedna-plt.scm")

; Create a connection
(define conn
  (sedna:connect-to-database "localhost" "sample-db" "guest" "guest"))

; Begin a transaction
(sedna:begin-transaction conn)

; Bulk load
(call/cc ; Thanks to Felix Winkelmann for recommending us to use call/cc here
  (lambda (k)
    (with-exception-handler ; Exception handler
      (lambda (x)
        (display "File already loaded to the database")
        (newline)
        (k #f)))
      (lambda ()
        (sedna:execute-query conn "LOAD 'region.xml' 'regions'")))))

; Execute a statement and represent it as an SXML nodeset
(pp
  (sedna:result->list
    (sedna:execute-query conn "document('region')/*/*")))

; Update statement
(pp
  (sedna:execute-query conn "UPDATE delete document('region')//africa"))

; Querying all regions once again
(pp
  (sedna:result->list
    (sedna:execute-query conn "document('region')/*/*")))

; Commit transaction
(sedna:end-transaction conn 'COMMIT)

; Close the connection
```

```
(sedna:disconnect-from-database conn)
```

You can find the example and the Scheme API driver code in:

```
[win:] INSTALL_DIR\examples\applications\scheme  
[linux:] INSTALL_DIR/examples/applications/scheme
```

where `INSTALL_DIR` refers to the directory where Sedna is installed.

Before running the example make sure that the Sedna DBMS is installed and do the following steps:

1. Start Sedna by running the following command in a command line:

```
se_gov
```

If Sedna is started successfully it prints "GOVERNOR has been started in the background mode".

2. Create a new database `sample-db` by running the following command:

```
se_cdb sample-db
```

If the database is created successfully it prints "The database 'sample-db' has been created successfully".

3. Start the `sample-db` database by running the following command:

```
se_sm sample-db
```

If the database is started successfully it prints "SM has been started in the background mode".

If the Sedna DBMS is running on the same computer as your Scheme application, you don't need to change anything in the example code. If the Sedna DBMS is running on a remote machine, you should use the name of this machine when connecting to the database with `sedna:connect-to-database` function.

For running the example supplied, you should copy all files from the folder

```
[win:] INSTALL_DIR\examples\applications\scheme  
[linux:] INSTALL_DIR/examples/applications/scheme
```

into the folder where the Scheme API driver code is located

```
[win:] INSTALL_DIR\driver\scheme  
[linux:] INSTALL_DIR/driver/scheme
```

where `INSTALL_DIR` refers to the directory where Sedna is installed.

You can use PLT DrScheme GUI to open and run "`client.scm`" in the graphical mode. You can also run the example from your command line, by typing

```
mzscheme -gr client.scm
```

2 Database Language

2.1 Data Manipulation Language

2.1.1 XQuery Support

The Sedna query language is XQuery [2] developed by W3C. Below is the list of the XQuery features, supported in the current release of Sedna.

- Data model [1]
 - Nodes: Element, Document, Text, Namespace and Attribute Nodes.
 - Accessors: dm:node-kind, dm:node-name, dm:string-value, dm:typed-value
 - Atomic types: xs:string , xs:boolean , xs:decimal (*restricted*), xs:float, xs:double, xs:QName,xdt:untypedAtomic and xdt:anyAtomicType .
- Language features [2]
 - Primary Expressions: literals, variable references, parenthesized expressions, context item expression, XQuery comments
 - Path Expressions
 - * Axes: child, descendant, attribute, self, descendant-or-self, parent.
 - * Node Tests: NameTest, Wildcard , TextTest, AnyKindTest.
 - * Other Features: predicates, abbreviated syntax.
 - Sequences: comma operator
 - Arithmetic Expressions
 - Comparisons: value comparison, general comparison, node comparison.
 - Logical Expressions
 - Constructors: direct element constructor, computed element and attribute constructors.
 - FLWOR: for and let clause, where clause, return clause.
 - Function declaration (locally defined user functions are supported).
 - Other Expressions: conditional expression, quantified expression (without type declaration), cast expression.
- Functions and operators [3]: see 2.2.1

2.1.2 Update Language

The update language is based on the XQuery update proposal by Patrick Lehti [4] with the number of improvements.

Note 2 *The result of each update statement, shouldn't break the well-formedness and validness of XML entitites, stored in the database. Otherwise, an error is raised.*

INSERT

The insert statement inserts content at positions identified by the INTO, PRECEDING and FOLLOWING clauses.

UPDATE

```
insert Expr1 (INTO|PRECEEDING|FOLLOWING) Expr2
```

Expr1 identifies the ordered sequence of the nodes to be inserted. The INTO, PRECEDING or FOLLOWING clause identifies the position. For each node in the result sequence of Expr2, the result sequence of Expr1 is inserted to the position identified by the INTO, PRECEDING or FOLLOWING clauses. If the INTO clause is specified, the sequence is appended to the random position of the child sequence for each node in the result of Expr2. If the PRECEDING clause is specified, the sequence is appended before each node in the result of Expr2. If the FOLLOWING clause is specified, the sequence is appended after each node in the result of Expr2.

If one of the following conditions is met an error occurs:

- There are non-element nodes in the result of Expr2 in case of the INTO clause
- There are temporary nodes in Expr2 (a node is considered *temporary*, if it is created as the result of the XQuery constructor evaluation.)

UPDATE

```
insert <warning>High Blood Pressure!</warning>  
preceding doc("hospital")//blood_pressure[systolic>180]
```

DELETE

The DELETE statement serves to remove persistent nodes from the database. It contains a subexpression, that returns the nodes to be deleted.

UPDATE

```
delete Expr
```

Expr identifies the nodes to be removed from the datatbase. Note, that nodes are removed from the database with all their descendants.

If one of the following conditions is met an error occurs:

- There are nodes in the result sequence of `Expr` that are not stored in external memory

```
UPDATE
delete doc("hospital")//blood_pressure[systolic>180]
```

DELETE_UNDEEP

```
UPDATE
delete_undeeep Expr
```

The `DELETE_undeeep` statement removes nodes identified by `Expr`, but in contrast to the `DELETE` statement it leaves the descendants of the nodes in the database.

Consider the following example:

A document named "a.xml" before update:

```
<A>
<B>
<C/>
<D/>
</B>
</A>
```

```
UPDATE
delete_undeeep doc("a.xml")//B
```

The document after update:

```
<A>
<C/>
<D/>
</A>
```

RENAME

The `RENAME` statement is used to change the qualified name of an element or attribute.

```
UPDATE
rename Expr on QName
```

The following expression changes the name of `<job/>` elements without changing their contents:

```
UPDATE
rename //job[.="bit banger"] on profession
```

2.1.3 Bulk load

To do bulk load use the following statements:

```
LOAD "file_name" "document_name"
```

The first parameter is a file name, it can be an absolute path to the file or only file name if the file is in the local directory. The second parameter is the name for this document in the database.

```
LOAD "file_name" "document_name" "collection_name"
```

The first parameter is a file name, it can be an absolute path to the file or only file name if the file is in the local directory. The second parameter is the name for this document in the database. The third parameter is the collection name to load the document into.

For performing bulk load not from the source file but from an input stream, use the following statements:

```
LOAD STDIN "document_name"
```

```
LOAD STDIN "document_name" "collection_name"
```

Compared to the above bulk load statements, here the "file_name" is replaced by the keyword `STDIN` to denote that the file to be loaded is taken from the input stream. Characters in the input stream must form a well-formed XML document, which is loaded into the database and named as specified by "document_name". If collection_name is set, the document is loaded into the specified collection of the database.

By default, the standard input stream is used. You can redirect a different input stream to be used as an input for bulk load. For example, an XML document produced by some program as its output can be loaded to a Sedna database in a stream-wise fashion. To redirect the input when working from a command line, you can use the functionality provided by your operation system. Java and Scheme APIs provide additional wrappers for bulk load from stream, such that the input stream can be specified by an additional argument of a function call.

2.2 XQuery Functions

2.2.1 Standard XQuery Functions

The following standard XQuery functions specified in [3] are supported:

- count, contains, sum, avg, max, min, true, false, not, empty, exists, distinct-values
- matches, replace (regular expression support based on the PCRE library [5]; see [3] for function signatures and [5] for regular expression syntax)

2.2.2 Index-scan Functions

In the current version of Sedna, query executor will use indecies automatically. Use the following functions to enforce executor using indices.

```
index-scan ($title as xs:string, $value as xdt:anyAtomicType,  
           $mode as xs:string ) as node()*
```

The `index-scan` function scans the index with the `$title` name and returns the sequence of nodes which keys are equal (less than, greater than, greater or equal, less or equal) to the search value `$value`. An Sedna error is raised if the search value can not be cast to the atomic type of the index. The `$mode` parameter of the `xs:string` type is used to set the type of the scan. If the value of the parameter is not equal to one of the following then the Sedna error SE2034 is raised: EQ (equal), LT (less than), GT (greater than), GE (greater or equal), LE (less or equal).

```
index-scan-between ($title as xs:string, $value1 as xdt:anyAtomicType,  
                  $value2 as xdt:anyAtomicType, $range as xs:string )  
                  as node()*
```

The `index-scan-between` scans the index with the `$title` name and returns the sequence of nodes which keys belong to the interval (segment, left half-interval, right half-interval) between the first `$value1` and second `$value2` search values. An Sedna error is raised if the search values can not be cast to the atomic type of the index. This function provides the `$range` parameter of the `xs:string` type to set the type of the scan. If the string is not equal to one of the following then the Sedna error SE2035 is raised: INT (interval), HINTR (segment), SEG (left half-interval), HINTL (right half-interval).

2.2.3 SQL Connection

SQL Connection allows access to relational databases from XQuery using SQL. The resulting ralations are represented on-the-fly as sequenses of XML-elements representing rows. These elements have sub-elements corresponding with the columns returned by the SQL query. and thus can be easy processed in XQuery. All functions dealing with access to SQL data are located in the namespace “`http://modis.ispras.ru/Sedna/SQL`” which is refered as `sql` in the following function declarations and examples.

Connections

In order to execute SQL queries on a RDBMS, you should first establish a connection to it. There are two functions for managing connections:

```

function sql:connect($db-url as xs:string) as xs:integer
function sql:connect($db-url as xs:string, $user as xs:string)
    as xs:integer
function sql:connect($db-url as xs:string, $user as xs:string,
    $password as xs:string) as xs:integer
function sql:connect($db-url as xs:string, $user as xs:string,
    $password as xs:string, $options as element(*) as xs:integer

```

These functions attempt to establish a database connection to the given URL using a user name and password if specified. They return a connection handle which could be then passed to `sql:execute`, `sql:prepare`, `sql:close`, `sql:rollback`, and `sql:commit`

If no connection can be opened, an Sedna error is raised.

All arguments of the `sql:connect` functions except for `$db-url` are optional:

- `$db-url` is the URL of the database to which a connect is established. URL is of the following form:

```
odbc:<driver name>:[//<server>[/<database>][;]][<driver options>]
```

“;” after `<database>` or `<server>` is required if there are some driver options following it

- `$user` is your user name for the session.
- `$password` is your password for the session.
- `$options` is an optional sequence of connection options. Connection options are elements of the form

```
<sql:option name="<option-name>" value="<option-value>" />
```

The only connection option available for the moment is “`manual-commit`” which enables manual commit mode if its value is “`on`”

To disconnect from the database, you can use the following function:

```
function sql:close($connection as xs:integer) as element()?
```

It closes database connection associated with connection handle `$connection`. An Sedna error is raised if operation cannot be completed.

Executing Queries

When a database connection is established you can start executing queries. Two types of query execution are supported: *direct query execution* and *prepared query execution*.

Direct Queries

Simple SQL queries are executed as the following XQuery example shows:

```
declare namespace sql="http://modis.ispras.ru/Sedna/SQL";
let $connection :=
  sql:connect("odbc:MySQL ODBC 3.51 Driver://localhost/somedb", "user",
    "pass")
return
  sql:execute($connection, "SELECT * FROM people WHERE first = 'Peter'");
```

Returning result is like the following:

```
<tuple first="Peter" last="Jackson" city="Wellington"/>
```

There are two functions for direct query execution:

```
function sql:execute($connection as xs:integer, $statement as xs:string)
  as element()*
function sql:execute($connection as xs:integer, $statement as xs:string,
  $query-options as element(*) as element()*
```

These functions execute a SQL query and return a sequence of elements representing the query result. SQL query can be as both a query statement and an update statement. In case of query statement, the result sequence contains an element named 'row' for each row of the query result. Each element contains as many children attributes as there are non-NULL fields in the corresponding result-row. Each attribute has the name of a row field. Fields with NULL values are not included. In case of update statement, empty sequence is returned.

An Sedna error is raised on an erroneous statement.

The `sql:execute` have the following arguments:

- `$connection` is a connection handle, returned by `sql:connect`;
- `$statement` is a string containing SQL statement to be executed;
- `$query-options` is a sequence of optional query parameters.

Update queries can be executed using the `sql:exec-update` function:

```
function sql:exec-update($connection as xs:integer,
  $statement as xs:string) as xs:integer
function sql:exec-update($connection as xs:integer,
  $statement as xs:string,
  $query-options as element(*) as xs:integer
```

these functions are similar to `sql:execute`, but return the number of rows affected by an update query (instead of an empty sequence returned by `sql:execute` for update-queries). Function arguments are same as for `sql:execute`.

The behavior of this function is undefined for non-update queries.

Prepared Statements

Sometimes it is more convenient or more efficient to use prepared SQL statements instead of direct query execution. In most cases, when a SQL statement is prepared it will be sent to the DBMS right away, where it will be compiled. This means that the DBMS does not have to compile a prepared statement each time it is executed. Prepared statements can take parameters. This allows using the same statement and supply it with different values each time you execute it, as in the following XQuery example:

```
declare namespace sql="http://modis.ispras.ru/Sedna/SQL";
let $connection :=
  sql:connect("odbc:MySQL ODBC 3.51 Driver://localhost/somedb", "user",
    "pass")
let $statement :=
  sql:prepare($connection, "INSERT INTO people(first, last)
    VALUES (?, ?)")
return (
  sql:execute($statement, "John", "Smith"),
  sql:execute($statement, "Matthew", "Barney")
)
```

this XQuery code inserts two rows into table `people` and returns an empty sequence.

To use prepared statements, first you need to create a prepared statement handle using the `sql:prepare` function:

```
function sql:prepare($connection as xs:integer, $statement as xs:string)
  as xs:integer
function sql:prepare($connection as xs:integer, $statement as xs:string,
  $query-options as element(*)*) as xs:integer
```

these function prepare a SQL statement for later execution and returns a prepared statement handle which can be used in the `sql:execute` and `sql:exec-update` functions.

The `sql:prepare` functions have the following arguments:

- `$connection` is a connection handle, created by `sql:connect`;
- `$statement` is a string containing a SQL statement that may contain one or more `'?'` - IN parameter placeholders;
- `$query-options` is a sequence of optional query parameters.

An Sedna error is raised on an erroneous statement.

There are two prepared statement execution functions, similar to direct query execution:

```
function sql:execute($prepared-statement as xs:integer,
    $param1 as item()?, ...) as element()*
```

this function is similar to `sql:execute` for direct queries and returns a sequence of elements representing query result.

- `$prepared-statement` is a prepared statement handle created by `$sql:prepare`;
- `$param1, ...` are parameters for parametrized statements. The number of parameters specified must exactly match the number of parameters of the prepared statement. NULL values are represented as empty sequences `()`.

```
function sql:exec-update($prepared-statement as xs:integer,
    $param1 as item()?, $param2...) as xs:integer
```

This function is similar to `sql:execute`, but returns the number of rows affected by an update query (instead of an empty sequence returned by `sql:execute` for update-queries). Function arguments are the same as for `sql:execute`.

The behavior of this function is undefined for non-update queries.

Transactions

The default commit mode of connection is auto-commit, meaning that all updates will be committed automatically. If this is not desired behaviour, you can pass manual-commit option to `sql:connect` when you create a connection handle.

In manual commit mode you can specify when updates will be committed or rolled back:

```
declare namespace sql="http://modis.ispras.ru/Sedna/SQL";
let $connection :=
    sql:connect("odbc:MySQL ODBC 3.51 Driver://localhost/somedb", "user",
        "pass", <sql:option name="manual-commit" value="1"/>)
return
    for $person in doc("somedoc.xml")/person
    return (
        sql:execute($connection, "<do something with person>"),
        if (allok($connection, $person)) then
            (
                sql:execute($connection, "<do something else with person>"),
                sql:commit($connection)
            )
        else
            sql:rollback($connection)
    )
```

There are two functions for specifying transaction boundaries - `sql:commit` and `sql:rollback` (transactions are started automatically by queries, these functions only close them):

```
function sql:commit($connection as xs:integer) as element()?
```

`sql:commit` commits all changes made during the last transaction in the database connection specified by connection handle `$connection` and closes transaction.

An Sedna error is raised if operation cannot be completed.

```
function sql:rollback($connection as xs:integer) as element()?
```

rolls back all changes made during the last transaction in the database connection specified by connection the handle `$connection` and closes transaction.

An Sedna error is raised if operation cannot be completed.

2.3 Data Definition Language

2.3.1 Managing Documents

Sedna supports Data Definition Language which allows to manage documents in database. Below is the specification of syntax and semantics of this language.

```
CREATE DOCUMENT "doc-name"
```

The `CREATE DOCUMENT` statement creates a new standalone document with the `doc-name` name in the database.

```
DROP DOCUMENT "doc-name"
```

The `DROP DOCUMENT` statement drops the standalone document with the `doc-name` name from the database.

```
RETRIEVE METADATA FOR (DOCUMENTS|COLLECTIONS) [WITH STATISTICS]
```

The `RETRIEVE METADATA` statement retrieves the metadata information about all XML documents stored in database. The output of this statement is an XML document, which complies with the following DTD:

```
<!ELEMENT XML_DOCUMENTS (Document)*|(Collection)*>
<!ELEMENT Document (#TEXT)?>
<!ATTLIST Document name CDATA #REQUIRED>
<!ELEMENT Collection (#TEXT)?>
<!ATTLIST Collection name CDATA #REQUIRED >
```

The value of the `name` attribute is the name of the document or collection. If the `WITH STATISTICS` command is stated, the following statistical information is retrieved as the content of the `Document` or `Collection` elements:

- `TOTAL NODES IN SCHEMA`. The number of the nodes of the descriptive schema.
- `TOTAL TEXT NODES IN SCHEMA`. The number of the text attribute and document nodes of the descriptive schema.
- `TOTAL NODES`. The number of the nodes of the document (or collection).
- `DEPTH`. The maximal depth of the document (or collection).
- `TOTAL BLOCKS EXCEPT NID and INDIR`. The number of the blocks occupied by document (or collection) except blocks with numberig schema and indirection.
- `TOTAL DESCRIPTOR BLOCKS`. The number of the descriptor blocks occupied by document (or collection) .
- `TOTAL STRING BLOCKS`. The number of the text blocks occupied by document (or collection) .
- `FILL PERCENTAGE`. Fill factor of the blocks (in percents).
- `TOTAL INNER BLOCKS`. The number of the descriptor blocks occupied by document (or collection) except first and last blocks in each chain of blocks.
- `TOTAL INNER NODES SIZE`. The size of the inner blocks.
- `INNER FILL PERCENTAGE`. Fill factor of the inner blocks (in percents).
- `STRINGS`. The share of the string blocks (in percents).
- `DESCRIPTORS`. The share of the descriptor blocks (in percents).
- `NIDS percentage`. The share of the long labeling numbers' (j11) total size (in percents).
- `INDIRECTION percentage`. The share of the indirection records' total size (in percents).
- `TOTAL SIZE`. The total size of the document (or collection), in MGBs.
- `STRINGS size`. The total size of the string blocks , in MGBs.
- `DESCRIPTORS size`. The total size of the descriptor blocks , in MGBs.
- `NIDS size`. The total size of the long labeling numbers , in MGBs.

- **FREE SPACE IN STRING BLOCKS size.** The total size of the free space in the string blocks.
- **INDIRECTION size.** The total size of the indirection records , in MGBs.
- **NIDS size.** The share of the indirection records' total size (in percents).
- **Data size variation.** The histogram of the xml data by it's size.
- **NID size variation.** The histogram of the labeling numbers it's length.

RETRIEVE DESCRIPTIVE SCHEMA FOR DOCUMENT "doc-name"

The RETRIEVE DESCRIPTIVE SCHEMA statement retrieves the descriptive schema of the XML document with the `doc-name` name.

The output of a descriptive schema is an XML document, which complies with the following DTD:

```
<!ELEMENT TREE_DESCRIPTIVE_SCHEMA NODE>
<!ELEMENT NODE (NODE)*>
<!ATTLIST NODE name CDATA #IMPLIED
              type (document|element|text|attribute)>
```

The root of the XML document is the `TREE_DESCRIPTIVE_SCHEMA` element, which contains the tree representation of a descriptive schema. The `NODE` element describes a node of descriptive schema. It has two attributes: `name` and `type`, which define the name and the type of the node correspondingly. There are four types of nodes in descriptive schema: document, element, attribute and text. More on descriptive schema see in [6].

2.3.2 Managing Collections

Sedna provides a mechanism of organizing multiple documents into a collection. Collection provides a uniform way of writing XQuery and XML update statements addressed to multiple documents at once. Collections are preferable for situations when a group of documents is to be queried/updated by means of not referring to their individual document names, but according to some conditions over their content.

In a Sedna database, a document can be either a standalone one (when it doesn't belong to any collection) or belonging to some collection. Compared to standalone documents, all documents within a given collection have a common descriptive schema. The common descriptive schema (which can be considered as a union of individual descriptive schemas for all documents that belong to a collection) allows addressing XQuery and XML update queries to all members of a collection.

Below is the specification of syntax and semantic of statements that manage collections.

```
CREATE COLLECTION "coll-name"
```

The `CREATE COLLECTION` statement creates a new collection with the `coll-name` name in the database.

XML documents can be loaded into the collection, as previously described in Sect.2.1.3. To access a single document from collection in an XQuery or XML update query, the `document` function accepts `coll-name` as its second optional argument:

```
document(doc-name, coll-name)
```

The function call to the `document` function returns the document with `doc-name` name that belongs to the collection named `coll-name`.

```
collection(coll-name)
```

The function `collection` can be called from any place within an XQuery or XML update query where the function call is allowed. The `collection` function returns the sequence of all documents that belong to the collection named `coll-name`. The relative order of documents in a sequence returned by `collection` function is currently undefined in Sedna.

Conventional XQuery predicates can be used for filtering the sequence returned by the `collection` function call, for selecting certain documents that satisfy the desired condition.

```
DROP DOCUMENT "doc-name" IN COLLECTION "coll-name"
```

The `DROP DOCUMENT IN COLLECTION` statement drops the document named `doc-name` located in the collection named `coll-name`.

```
DROP COLLECTION "coll-name"
```

The `DROP COLLECTION` statement drops the collection with the `coll-name` name from the database. If the collection contains any documents, these documents are dropped as well.

2.3.3 Managing Indices

Sedna supports value indices based on B-tree to index XML element content and attribute values. Below is the description of statements to manage indices.

```
CREATE INDEX title  
ON Path1 BY Path2  
AS type
```

The `CREATE INDEX` creates an index on nodes (specified by `Path1`) by keys (specified by `Path2`).

`Path1` is an XPath expression without any filter expressions that identifies the nodes of a document or a collection that are to be indexed. `Path2` is an XPath expression without any filter expressions that specifies the relative path to the nodes whose string-values are used as keys to identify the nodes returned by the `Path1` expression. The `Path2` expression should not start with `/` or `//`. The full path from the root of documents (that may be in a collection) to the key nodes is `Path1/Path2`.

For instance, let `Path1` be `doc(a)/b/c` and `Path2` be `d`. Let `X` be the node returned by the `Path1` expression, and `Y` be one of the nodes returned by the `doc(a)/b/c/d` expression. If `Y` is the descendant of `X`, then the value of `Y` is used as the key for searching the node `X`.

`title` is the title of the index created. It should be unique for each index in the database.

`type` is an atomic type which the value of the nodes should be cast to.

`DROP INDEX title`

The `DROP INDEX` statement removes the index named `title` from the database.

Note 3 *In the current version of Sedna, query executor will not use indices automatically. You can enforce the executor to utilize indices by using the XQuery index-scan funtions specified in Section 2.2.2.*

References

- [1] “XQuery 1.0 and XPath 2.0 Data Model”, W3C Working Draft, 12 November 2003
- [2] “XQuery 1.0: An XML Query Language”, W3C Working Draft, 12 November 2003
- [3] “XQuery 1.0 and XPath 2.0 Functions and Operators”, W3C Working Draft, 12 November 2003
- [4] Patrick Lehti. “Design and Implementation of a Data Manipulation Processor for a XML Query Language”, <http://www.ipsi.fraunhofer.de/lehti/>
- [5] <http://www.pcre.org/>
- [6] Maxim Grinev, Andrey Fomichev, Sergey Kuznetsov, Kostantin Antipin, Alexander Boldakov, Dmitry Lizorkin, Leonid Novak, Maria Rekouts, Peter Pleshachkov. “Sedna: A Native XML DBMS”, <http://www.modis.ispras.ru/publications.htm>

- [7] Rekouts Maria. “Application Programming Interface for XML DBMS: design and implementation proposal”, <http://www.modis.ispras.ru/publications.htm>
- [8] Noel Welsh, Francisco Solsona, Ian Glover. “SchemeUnit and SchemeQL: Two little languages”, Scheme Workshop 2002, <http://schematics.sourceforge.net/schemeunit-schemeql.ps>
- [9] Oleg Kiselyov. “SXML Specification, Revision 3.0”, <http://www.okmij.org/ftp/Scheme/SXML.html>
- [10] William Clinger, R. Kent Dybvig, Matthew Flatt, and Marc Feeley. “SRFI-12: Exception Handling”, <http://srfi.schemers.org/srfi-12/srfi-12.html>
- [11] Felix L. Winkelmann. “Chicken – A practical and portable Scheme system”, <http://www.call-with-current-continuation.org/chicken.pdf>
- [12] Kirill Lisovsky, Dmitry Lizorkin. “XML Path Language (XPath) and its functional implementation SXPath”, Russian Digital Libraries Journal , 2003, Volume 6, Issue 4, <http://www.elbib.ru/index.phtml?page=elbib/eng/journal/2003/part4/LL>