Sedna Programmer's Guide

Contents

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.

```
SednaConnection getConnection(String url,
                              String DBName,
                              String user,
                              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()
```

Setting connection into debug mode allows getting debug information when XQuery query fails due to some reason (see 2.7.2 for details). To set the connection into debug mode use `setDebugMode` method of `SednaConnection` interface:

```
public void setDebugMode(boolean debug)
```

Parameters:

`debug` - set to `true` to turn debug mode on; set to `false` to turn debug mode off. Debug mode is off by default.

Sedna supports `fn:trace` function for debugging purpose also (see 2.7.1 for details). By default trace output is included into XQuery query result. You can turn trace output on/off using `setTraceOutput` method of the `SednaConnection` interface:

```
public void setTraceOutput(boolean doTrace)
```

Parameters:

`doTrace` - set to `true` to turn trace output on; set to `false` to turn trace output off. Trace output is on by default.

1.1.2 Transactions Management

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 (or rollback) 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`.

To rollback transaction the `rollback` method is used.

```
public void rollback() throws DriverException
```

If the transaction has not been rollback successfully the `rollback` 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(String queryText)
    throws DriverException
```

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

```
public boolean execute(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.

The results of XQuery queries to the Sedna DBMS can be represented either in XML or SXML [11]. To specify the type of the result use the following extended versions of the `execute` method:

```
boolean execute(InputStream in,
                ResultType resultType)
    throws DriverException, IOException;

boolean execute(String queryText,
                ResultType resultType)
    throws DriverException, IOException;
```

The `resultType` parameter is either `ResultType.XML` or `ResultType.SXML`.

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(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 an example program that uses the Java API to work with Sedna DBMS. This application connects to the Sedna DBMS, opens a session. The session consists of one transaction. Application loads data from the file `region.xml` and executes the XQuery statement. When statement is executed, application drop the document, commits the transaction and closes the session.

```
import ru.ispras.sedna.driver.*;
class Client {
  public static void main(String args[]) {

    SednaConnection con = null;
    try {
      /* Get a connection */
      con = DatabaseManager.getConnection("localhost",
```

```
                                        "testdb",
                                        "SYSTEM",
                                        "MANAGER");

  /* Begin a new transaction */
  con.begin();

  /* Create statement */
  SednaStatement st = con.createStatement();

  /* Load XML into the database */
  System.out.println("Loading data ...");
  boolean res;
  res = st.execute("LOAD 'C:/region.xml' 'region'");

  System.out.println("Document 'region.xml' "+
      "has been loaded successfully");
  /* Execute query */
  System.out.println("Executing query");
  res = st.execute("doc('region')/*/*");

  /* Print query results */
  printQueryResults(st);

  /* Remove document */
  System.out.println("Removing document ...");
  res = st.execute("DROP DOCUMENT 'region'");

  System.out.println("Document 'region' " +
             "has been dropped successfully");
  /* Commit current transaction */
  con.commit();
}
catch(DriverException e) {
    e.printStackTrace();
}
finally {
  /* Properly close connection */
  try { if(con != null) con.close(); }
  catch(DriverException e) {
    e.printStackTrace();
  }
}
```

```
  }

  /* Pretty printing for query results */
  private static void printQueryResults(SednaStatement st)
    throws DriverException {
    int count = 1;
    String item;
    SednaSerializedResult pr = st.getSerializedResult();
    while ((item = pr.next()) != null) {
      System.out.println(count + " item: " + item);
      count++;
    }
  }
}
```

The full-version source code of this example program can be found at:

```
[win:] INSTALL_DIR\examples\api\java\Client.java
[nix:] INSTALL_DIR/examples/api/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 `testdb` by running the following command:

   ```
   se_cdb testdb
   ```

   If the database is created successfully it prints "The database 'testdb' has been created successfully".

3. Start the testdb database by running the following command:

   ```
   se_sm testdb
   ```

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

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

1. To compile the example, run the script:

```
[win:] Clientbuild.bat
[nix:] ./Clientbuild.sh
```

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

2. To run the compiled example, use the script:

```
[win:] Client.bat
[nix:] ./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"`, `"sp_defs.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 `"libsedna.h"`, `sp_defs.h` files are stored.

For convenience three versions of `libsedna` are provided on the Windows operating system:

1. `libsednamt.lib` - static multi-threaded version built with /MT option. Use it if you compile your project with /MT[d] option.
2. `libsednamd.lib` - static multi-threaded version built with /MD option. Use it if you compile your project with /MD[d] option.
3. `sednamt.dll` - dynamic version. `sednamt.lib` is import library.

On Unix-like operating systems the following versions of `libsedna` are provided:

1. `libsedna.so` - dynamic shared library.
2. `libsedna.a` - static version of the library.
3. `libsedna_pic.a` - static version of the library with PIC enabled. You may need it to build drivers for Sedna which are based on `libsedna`.

1.2.1 Errors Handling

C API provides set of functions for sessions and transactions management, query and update statements execution, etc. If the function fails it returns negative value. In this case application can obtain the error message and code which help to understand the reason of the error occurred.

To get the last error message use `SEgetLastErrorMsg` function:

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

Parameters:

- `conn` - a pointer to an instance of the `SednaConnection` type (see 1.2.2, Connecting to a Database section for details on how to obtain a connection instance).

The function `SEgetLastErrorCode` returns the last error code occurred in the session:

```
int SEgetLastErrorCode(struct SednaConnection *conn)
```

Parameters:

- `conn` - a pointer to an instance of the `SednaConnection` type (see 1.2.2, Connecting to a Database section for details on how to obtain a connection instance).

1.2.2 Connecting to a Database

Before working with Sedna an application has to declare variable of the `SednaConnection` type and initialize it in the following manner:

```
struct SednaConnection conn = SEDNA_CONNECTION_INITIALIZER;
```

Note 1 *The initialization with* `SEDNA_CONNECTION_INITIALIZER` *is mandatory for Sedna version 0.5 and earlier.*

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)
```

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 to.
- `login` - user login.
- `password` - user password.

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_SESSION_OPEN` - connection to the database is established, authentication passed successfully.

If the function fails, the return value is negative and session is not opened:

- `SEDNA_AUTHENTICATION_FAILED` - authentication failed.
- `SEDNA_OPEN_SESSION_FAILED` - failed to open session.
- `SEDNA_ERROR` - some error occurred.

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 an application finished it's work with the database, it must close the session. `SEclose` finishes the session and closes the connection to the server. `SEclose` also frees resources that were equipped by the call to `SEconnect` function, that is for every successful call to `SEconnect` there must be a call to `SEclose` in the client program. You must call `SEclose` both when application finishes its work with the database, and when application cannot work with the database anymore due to some error.

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

Parameters:

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

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_SESSION_CLOSED` - session closed successfully.

If the function fails, the return value is negative:

- `SEDNA_CLOSE_SESSION_FAILED` - session closed with errors.
- `SEDNA_ERROR` - some error occurred.

1.2.3 Setting Session Options

An application can set attributes that govern aspects of a session using `SEsetConnectionAttr`:

```
int SEsetConnectionAttr(struct SednaConnection *conn,
                        enum SEattr attr,
                        const void* attrValue,
                        int attrValueLength)
```

Parameters:

- `conn` - a pointer to an instance of the SednaConnection type, associated with a session to be closed.
- `attr` - an attribute to set (one of the predefined Sedna connection attributes listed below).
- `attrValue` - a pointer to the value to be associated with the attribute.
- `attrValueLength` - a length of the value in bytes.

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_SET_ATTRIBUTE_SUCCEEDED` - the attribute was set successfully.

If the function fails, the return value is negative:

- `SEDNA_ERROR` - some error occurred.

Supported attributes:

- `SEDNA_ATTR_AUTOCOMMIT` Autocommit mode is the default transaction management mode of the Sedna server (`SEDNA_AUTOCOMMIT_OFF` is the default value of this attribute). Every XQuery or update statement is committed or rolled back when it completes. If a statement completes successfully, it is committed; if it encounters any error, it is rolled back. A connection to an instance of the Sedna database operates in autocommit mode whenever this default mode has not been overridden by setting this attribute into `SEDNA_AUTOCOMMIT_OFF` value.

Atrribute values	*Value size*
`SEDNA_ATTR_AUTOCOMMIT_ON`, `SEDNA_ATTR_AUTOCOMMIT_OFF`	sizeof(int)

- `SEDNA_ATTR_SESSION_DIRECTORY` connection attribute defines the `session directory`. If this attribute is set, paths in the `LOAD` statement 2.4 or `LOAD MODULE` are evaluated relative to the session directory.

Atrribute values	*Size of value*
path to directory	*length of path*

- SEDNA_ATTR_DEBUG connection attribute turns on/off query debug mode. Query debug mode is off by default. **Note:** SEDNA_ATTR_DEBUG connection attribute must be set only after SEconnect has been called on the conn.

Atrribute values	*Value size*
SEDNA_ATTR_DEBUG_ON, SEDNA_ATTR_DEBUG_OFF	sizeof(int)

- SEDNA_ATTR_CONCURRENCY_TYPE connection attribute changes the mode of the next transactions. Transaction can be set to run as READ-ONLY (SEDNA_READONLY_TRANSACTION) or UPDATE-transaction (SEDNA_UPDATE_TRANSACTION). READ-ONLY transactions have one major benefit: they never wait for other transactions (they do not have to acquire any document/collection locks). However they might access slightly obsolete state of the database (for example, they probably would not see the most recent committed updates). You should use READ-ONLY transactions in a highly concurrent environment. Notice that the current transaction, if any, will be forcefully committed.

Atrribute values	*Value size*
SEDNA_READONLY_TRANSACTION, SEDNA_UPDATE_TRANSACTION	sizeof(int)

- SEDNA_ATTR_QUERY_EXEC_TIMEOUT connection attribute allows to set the limit on query execution time. If set, for each next query in this session, query execution will be stopped if it lasts longer than timeout set. In this case transaction in bounds of which the query run is rollback. By default (value 0) there is no any timeout for query execution, that is a query can be executed as long as needed.

Atrribute values	*Value size*
time in seconds	sizeof(int)

- SEDNA_ATTR_MAX_RESULT_SIZE connection attribute allows to set the limit on query result size. If this attribute is set, the server will cut the result data if its size exceeds the specified limit. By default, result data that is passed from server in response to user query can be of unlimited size.

Atrribute values	*Value size*
size in bytes	sizeof(int)

- SEDNA_LOG_AMOUNT connection attribute changes the mode of logical logging for the following transactions in the same session. Transaction can be set to run in full log mode (SEDNA_LOG_FULL) or reduced log mode (SEDNA_LOG_LESS). The former means transaction writes much less log info during bulk loads. Also, when such transaction commits the checkpoint is made, which might greatly reduce recovery time. There is a caveat, however: such transaction always runs in exclusive mode, which means there can be

no concurrent transactions. Before it starts it waits for other concurrent transactions to finish. In turn, all other transactions will not start until exlusive transaction finishes. You should use this option with care, since it effectively stalls any concurrent activity. The main purpose of such transactions is to bulk-load data. The other possible use-case includes transactions performing heavy update operations. Since checkpoint will be made when such transaction commits, it might reduce recovery time in case of database crash. Otherwise, you should not use this option since you will not gain anything. Notice also that the current transaction in the same session, if any, will be forcefully committed. The default value for this attribute is `SEDNA_LOG_FULL`.

Atrribute values	*Value size*
`SEDNA_LOG_LESS`, `SEDNA_LOG_FULL`	sizeof(int)

An application can retrieve current value of hte connection attributes using `SEgetConnectionAttr`:

```
int SEgetConnectionAttr(struct SednaConnection *conn,
                        enum SEattr attr,
                        void* attrValue,
                        int* attrValueLength);
```

Parameters:

- `conn` - a pointer to an instance of the `SednaConnection` type, associated with a session to be closed.
- `attr` - an attribute to retrieve.
- `attrValue` - a pointer to memory in which to return the current value of the attribute specified by `attr`.
- `attrValueLength` - a length of the retrieved value in bytes.

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_GET_ATTRIBUTE_SUCCEEDED` - the attribute was retrieved successfully.

If the function fails, the return value is negative:

- `SEDNA_ERROR` - some error occurred.

To reset all connection attributes to default values use:

```
int SEresetAllConnectionAttr(struct SednaConnection *conn);
```

Parameters:

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

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_RESET_ATTRIBUTES_SUCCEEDED` - attributes has been reset successfully.

If the function fails, the return value is negative:

- `SEDNA_ERROR` - some error occurred.

1.2.4 Transactions Management

An 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, an application can begin a transaction, execute statements and commit this transaction. In a session transactions are processed sequentially. That is, application must commit an ongoing transaction before beginning a new one.

There are two ways to manage transactions in Sedna sessions: *autocommit mode* and *manual-commit mode*:

- **Autocommit mode**. Each individual statement is committed when it completes successfully. When running in autocommit mode no other transaction management functions are needed. By default, Sedna sessions are run in autocommit mode.
- **Manual-commit mode**. Transaction boundaries are specified explicitly by means of `SEbegin`, `SEcommit` and `SErollback` functions. All statements between the call to `SEbegin` and `SEcommit`/`SErollback` are included in the same transaction.

An application can switch between the two modes using `SEsetConnectionAttr` and `SEgetConnectionAttr` functions (see 1.2.3) for `SEDNA_ATTR_AUTOCOMMIT` attribute.

To specify transaction boundaries application uses `SEbegin`, `SEcommit` and `SErollback` functions. `SEbegin` function starts new transaction in the provided session:

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

Parameters:

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

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_BEGIN_TRANSACTION_SUCCEEDED` - transaction has been successfully started.

If the function fails, the return value is negative:

- `SEDNA_BEGIN_TRANSACTION_FAILED` - failed to start a transaction.
- `SEDNA_ERROR` - some error occurred.

`SEcommit` function commits the current transaction:

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

Parameters:

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

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_COMMIT_TRANSACTION_SUCCEEDED` - transaction has been committed.

If the function fails, the return value is negative:

- `SEDNA_COMMIT_TRANSACTION_FAILED` - failed to commit transaction.
- `SEDNA_ERROR` - some error occurred.

`SErollback` function rollbacks the current transaction:

```
int SErollback(SednaCommection* conn)
```

Parameters:

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

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_ROLLBACK_TRANSACTION_SUCCEEDED` - transaction has been rollbacked.

If the function fails, the return value is negative:

- `SEDNA_ROLLBACK_TRANSACTION_FAILED` - failed to rollback transaction.
- `SEDNA_ERROR` - some error occurred.

1.2.5 Getting Connection and Transaction Status

An application can obtain the connection status by `SEconnectionStatus` function:

```
int SEconnectionStatus(SednaConnection* conn)
```

Parameters:

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

Return values:

- `SEDNA_CONNECTION_OK` - specified connection is open and functions ok.
- `SEDNA_CONNECTION_CLOSED` - specified connection is closed. This could be either after the call to `SEclose` function, or before the call to `SEconnect` function.
- `SEDNA_CONNECTION_FAILED` - specified connection has been failed. (**Note:** in this case you should call `SEclose` function to release resources).

An application may obtain the transaction status by `SEtransactionStatus` function:

```
int SEtransactionStatus(SednaConnection* conn)
```

Parameters:

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

Return values:

- `SEDNA_TRANSACTION_ACTIVE` - specified connection runs transaction.
- `SEDNA_NO_TRANSACTION` - specified connection does not run transaction. This could be for example when previous transaction has been committed and a new one has not begun yet.

1.2.6 Executing Queries and Updates

There are two functions to execute a statement (query or update): `SEexecute` function and `SEexecuteLong` function. First one reads statement from a C-string, the second reads long statement from a provided file. To get trace (`fn:trace` XQuery function) and debug information application may implement custom debug handler and set it using function: `SEsetDebugHandler`.

```
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 or XUpdate statement.

Return values:

If the function succeeds, the return value is positive:

- `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 2.4) succeeded.

If the function fails, the return value is negative:

- `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 the statement is really long, and you prefer to pass it to the Sedna directly from a file use `SEexecuteLong` function.

```
int SEexecuteLong(SednaConnection* conn,
                  const char* query_file_path)
```

Parameters:

- `conn` - a pointer to an instance of the `SednaConnection` type.
- `query_file` - a path to the file with a statement to execute.

Return values:

If the function succeeds, the return value is positive:

- `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 2.4) succeeded.

If the function fails, the return value is negative:

- `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 or SEexecuteLong function return 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 the current item of the sequence.

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

Parameters:

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

Return values:

If the function succeeds, the return value is positive:

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

If the function fails or the is no result items available to retrieve, the return value is negative:

- 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.

SEgetData function retrieves current item from the result sequence:

```
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.

Return values:

If the function succeeds, the return value is non-negative:

- 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).

If the function fails, the return value is negative:

- `SEDNA_GET_DATA_FAILED` - failed to get data.
- `SEDNA_ERROR` - some error occurred.

Since version 1.5 Sedna supports reporting tracing information (`fn:trace` XQuery function). To handle tracing information while retrieving result data use debug handler `debug_handler_t` and `SEsetDebugHandler` function:

```
void SEsetDebugHandler(struct SednaConnection *conn,
                       debug_handler_t debug_handler)
```

Parameters:

- `conn` - a pointer to an instance of the `SednaConnection` type.
- `debug_handler` - a pointer to your own defined function of the following type: `void (*debug_handler_t)(int subtype, const char *msg)` where `subtype` is a type of debug information (currently only `SEDNA_QUERY_TRACE_INFO` is supported), `msg` is a buffer with debug information.

For example the following debug handler prints out debug iformation to the stdout:

```
void my_debug_handler(enum SEdebugType subtype,
                      const char *msg) {
  printf("TRACE: ");
  printf("subtype(%d), msg: %s\n", subtype, msg);
}
```

If the debug handler is not defined by the application, trace information is ignored.

1.2.7 Loading Data

XML data can be loaded into a database using `"LOAD"` statement of the Data Manipulation Language (see 2.4). 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 an XML document:

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

Parameters:

- `conn` - a pointer to an instance of the `SednaConnection` type.
- `buf` - a buffer with chunk of an XML document to load.
- `bytes_to_load` - number of bytes to load.
- `doc_name` - name of the document in a database the data loads to.
- `col_name` - name of the collection in the case if document is loaded into the collection, `NULL` if document is loaded as a standalone one.

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_DATA_CHUNK_LOADED` - chunk of an XML document loaded successfully.

If the function fails, the return value is negative:

- `SEDNA_ERROR` - some error occurred. Data is not loaded.

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

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

Parameters:

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

Return values:

If the function succeeds, the return value is positive:

- `SEDNA_BULK_LOAD_SUCCEEDED` - XML document was successfully loaded into the database.

If the function fails, the return value is negative:

- `SEDNA_BULK_LOAD_FAILED` - failed to load XML document into the database.
- `SEDNA_ERROR` - some error occurred.

1.2.8 Example Code

```
#include "libsedna.h"
#include "stdio.h"

int handle_error(SednaConnection* conn,
                 const char* op,
                 int close_connection) {
    printf("%s failed: \n%s\n", op, SEgetLastErrorMsg(conn));
    if(close_connection == 1) SEclose(conn);
    return -1;
}

int main() {
  struct SednaConnection conn = SEDNA_CONNECTION_INITIALIZER;
  int bytes_read, res, value;
  char buf[1024];

  /* Turn off autocommit mode */
  value = SEDNA_AUTOCOMMIT_OFF;
  res = SEsetConnectionAttr(&conn, SEDNA_ATTR_AUTOCOMMIT,
                            (void*)&value, sizeof(int));

  /* Connect to the database */
  res = SEconnect(&conn, "localhost", "test_db",
                  "SYSTEM", "MANAGER");
  if(res != SEDNA_SESSION_OPEN)
    return handle_error(&conn, "Connection", 0);

  /* Begin a new transaction */
  res = SEbegin(&conn);
  if(res != SEDNA_BEGIN_TRANSACTION_SUCCEEDED)
    return handle_error(&conn, "Transaction begin", 1);

  /* Load file "region.xml" into the document "region" */
  res = SEexecute(&conn, "LOAD 'region.xml' 'region'");
  if(res != SEDNA_BULK_LOAD_SUCCEEDED)
    return handle_error(&conn, "Bulk load", 1);

  /* Execute XQuery statement */
  res = SEexecute(&conn, "doc('region')/*/*");
  if(res != SEDNA_QUERY_SUCCEEDED)
    return handle_error(&conn, "Query", 1);
```

```
  /* Iterate and print the result sequence */
  while((res = SEnext(&conn)) != SEDNA_RESULT_END) {
    if (res == SEDNA_ERROR)
      return handle_error(&conn, "Getting item", 1);

    do {
      bytes_read = SEgetData(&conn, buf, sizeof(buf) - 1);
      if(bytes_read == SEDNA_ERROR)
        return handle_error(&conn, "Getting item", 1);
      buf[bytes_read] = '\0';
      printf("%s\n", buf);
    } while(bytes_read > 0);
  }

  /* Drop document "region" */
  res = SEexecute(&conn, "DROP DOCUMENT 'region'");
  if(res != SEDNA_UPDATE_SUCCEEDED)
    return handle_error(&conn, "Drop document", 1);

  /* Commit transaction */
  res = SEcommit(&conn);
  if(res != SEDNA_COMMIT_TRANSACTION_SUCCEEDED)
    return handle_error(&conn, "Commit", 1);

  /* Close connection */
  res = SEclose(&conn);
  if(res != SEDNA_SESSION_CLOSED)
    return handle_error(&conn, "Close", 0);

  return 0;
}
```

The full version of this example program can be found in:

```
[win:] INSTALL_DIR\examples\api\c\Client.c
[nix:] INSTALL_DIR/examples/api/c/Client.c
```

Here `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 `testdb` by running the following command:

   ```
   se_cdb testdb
   ```

   If the database is created successfully it prints "The database 'testdb' has been created successfully".

3. Start the testdb database by running the following command:

   ```
   se_sm testdb
   ```

   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. To compile the example use:

   ```
   [win:] Clientbuild.bat
   [nix:] ./Clientbuild.sh
   ```

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

2. To run the compiled example use:

   ```
   [win:] Client.exe
   [nix:] ./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 [10], an implementation of SQL 1992 for Scheme. The results of XQuery statements to the Sedna DBMS via the Scheme API can be represented either in XML or SXML [11].

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 `"SYSTEM"` for the value of this parameter.
- `password` is your password for the session. You can use the `"MANAGER"` 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 Manage 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 subsubsection 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.

1.3.3 Executing Queries and Updates

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 following kinds of statements:

- **XQuery statement** – for querying data from the database, without modifying it;
- **Update statement** – 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;
- **Database management statement** – to create index, trigger, retrieve metadata, etc.

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 3 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* [10] (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 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 statement result in that case exactly corresponds to the term of a *node-set* in the XPath implementation in Scheme SXPath [14].

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 section 2.4.

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 [12]. The SRFI-12 exception mechanism is natively supported in the Chicken Scheme compiler [13]. In the Scheme API driver, we also provide the SRFI-12 implementation for PLT and Gambit.

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" "testdb"
                             "SYSTEM" "MANAGER"))
; Begin a transaction
(sedna:begin-transaction conn)

; Bulk load
(call/cc
 (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 "doc('region')/*/*")))

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

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

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

```
; Close the connection
(sedna:disconnect-from-database conn)
```

You can find the full version of this example and the Scheme API driver code in:

```
[win:] INSTALL_DIR\examples\api\scheme
[nix:] INSTALL_DIR/examples/api/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 `testdb` by running the following command:

   ```
   se_cdb testdb
   ```

   If the database is created successfully it prints "The database 'testdb' has been created successfully".

3. Start the testdb database by running the following command:

   ```
   se_sm testdb
   ```

   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\api\scheme
[nix:] INSTALL_DIR/examples/api/scheme
```

into the folder where the Scheme API driver code is located

```
[win:] INSTALL_DIR\driver\scheme
[nix:] 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
```

For running the example with a different Scheme implementation (Chicken or Gambit), uncomment the corresponding line at the beginning of the example code in `"client.scm"` and follow the instructions of the chosen Scheme implementation in running the program.

2 Database Language

2.1 XQuery Support

The Sedna query language is XQuery [3] developed by W3C. Sedna conforms to January 2007 specification of XQuery except the following features:

- `copy-namespaces` declaration works only in `preserve`, `inherit` mode regardless of actual prolog values;
- `fn:normalize-unicode` function is not conformant (it always returns the same string or raises exception);
- External variables are not supported. However they are supported by XQJ driver by Charles Foster (see download page);
- Regular expressions (e.g. `fn:matches()` use them) are based on PCRE [7] syntax which differs from the one defined in W3C specifications.

Sedna also has full support for two optional XQuery features:

- **Full Axis**. The following optional axes are supported: `ancestor`, `ancestor-or-self`, `following`, `following-sibling`, `preceding`, and `preceding-sibling`;
- **Module Feature** allows a query Prolog to contain a Module Import and allows library modules to be created.

Sedna passes "XML Query Test Suite (XQTS)" and has official "almost passed" status. The detailed report can be found at http://www.w3.org/XML/Query/test-suite/XQTSReport.html

2.2 XQuery Options and Extensions

2.2.1 Controlling Serialization

Serialization is the process of converting XML nodes evaluated by XQuery into a stream of characters. In Sedna serialization is carried out when the result of a query is returned to the user. You can control the serialization by setting the *serialization parameters* specified in [5]. Currently, Sedna supports the following serialization parameters:

Parameter name	Values	Description
indent	"yes" or "no" (default yes)	Output indented
cdata-section-elements	Element list e.g. 'name;data'	Text within specified elements appears within CDATA section

To set a serialization parameter, use the `output` option in a query prolog. The `output` option is in the Sedna namespace (http://www.modis.ispras.ru/sedna) which is the predefined namespace in Sedna so you can omit its declaration. The value of the `output` option must have the following structure "`parameter-name=value; parameter-name=value`". Consider the following examples:

```
declare namespace se = "http://www.modis.ispras.ru/sedna";
declare option se:output "indent=yes; cdata-section-elements=script";
<x><script>K&amp;R C</script><b>element</b></x>
```

As mentioned above, you may omit the Sedna namespace declaration:

```
declare option se:output "indent=yes; cdata-section-elements=script";
<x><script>K&amp;R C</script><b>element</b></x>
```

This query is evaluated as follows:

```
<x>
  <script><![CDATA[K&R C]]></script>
  <b>element</b>
</x>
```

2.2.2 Value index-scan Functions

In the current version of Sedna, query executor does not use indices automatically. Use the following functions to enforce executor to employ 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`. A 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. The value of the parameter must be equal to one of the following: '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 `$value2` search values. A Sedna error is raised if the search values can not be cast to the atomic type of the index. This function provides the `$range` parameter of the `xs:string` type to set the type of the scan. The value of the string must be equal to one of the following: 'INT' (interval), 'SEG' (segment), 'HINTR' (right half-interval), 'HINTL' (left half-interval).

For example, to select the names of people who live in the London city employing the "people" index defined in section 2.5.3, use the following expression:

```
index-scan("people", "London", "EQ")/name
```

2.2.3 Full-Text Search Functions

Please read section 2.5.4 before reading this section.

In the current version of Sedna, query executor does not use full-text indices automatically. Use the following functions to enforce executor to employ indices.

```
ftindex-scan($title as xs:string,
             $query as xs:string,
             $options as xs:string) as node()*
```

The `ftindex-scan` function scans the full-text index with the `$title` name and returns the sequence of items which satisfy the `$query`. If dtSearch [15] is used as full-text search backend, use dtSearch request language [16] to specify the query. DtSearch options `dtsSearchAnyWords` or `dtsSearchAllWords` may be specified in `$options`.

For example, you can employ the "articles" index defined in section 2.5.4 to select the titles of articles that contain word "apple" but not "pear":

```
ftindex-scan("articles", "apple and not pear")/title
```

If native full-text indices are used, the following constructs can be used as parts of query:

- phrases: several words in single or double quotes will be searched as a phrase
- binary operators: parts of the query separated by whitepace are treated as conjunction, disjunction is specified by `'OR'`. For example: query `'apple juice'` will return only nodes contaitinig both words. while query `'apple OR juice'` will return nodes containing any of these words.
- stemming: if index was created with stemtype=both option, tilde must be appended to the word in order to use stemming, for example in the query `'apple juice~'` stemming will be used only for the second word.
- contains: to search for words inside some tag, use CONTAINS; the query `'title CONTAINS word'` will return nodes in which word 'word' occurs as part of tag 'title'.

All keywords (like CONTAINS and OR) must be upper-cased.

The `ftscan` function returns those items of the input sequence `$seq` which satisfy the query `$query`. The function does not use indices and can be applied to any sequence of nodes, even those that are not indexed. The query `$query` is evaluated over the text representation constructed according to the `$type` and `$customization_rules` parameters. The values of the parameters are the same as those used when a full-text index is created (see section 2.5.4 for details).

```
ftscan($seq as node()*,
       $query as xs:string,
       $type as xs:string,
       $customization_rules as xs:string) as node()*
```

For example, you can select the titles of articles that contain word "apple" but not "pear" *without* using indices and using special customization rules as follows:

```
ftscan(doc("foo")/library//article,
       "apple and not pear",
       "customized-value",
       (("b","string-value"),("a","delimited-value")))/title
```

2.2.4 SQL Connection

SQL Connection allows access to relational databases from XQuery using SQL. The resulting relations are represented on-the-fly as sequences 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 referred 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 using one of the `sql:connect` functions:

```
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` functions. If connection could not be established, a 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. It should be one of the following form:

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

  ";" after `<database>` or `<server>` is required if there are some driver options following it. Driver options must be in the following form:

  ```
  <option>=<value>{;<option>=<value>}
  ```

  List of available options depends on the ODBC driver used. One of the common options is "Port" which is used to specify the port on which the database server is configured to listen. For example:

  ```
  odbc:MySQL ODBC 3.51 Driver://localhost/somedb;Port=1234
  ```

- `$user` is your user name for the session.
- `$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`. A 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'");
```

The result will be something like this:

```
<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.A Sedna error is raised on an erroneous statement.

The `sql:execute` have the following arguments:

- `$connection` is a connection handle, returned by `sql:connect` function;
- `$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 behaviour 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 functions 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. A Sedna error is raised on an erroneous statement.

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

- `$connection` is a connection handle, created by `sql:connect` function;
- `$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.

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 the query result.

The `sql:execute` function have the following arguments:

- `$prepared-statement` is a prepared statement handle created by `$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 `()`.

To execute a prepared update statement you may use `exec-update` function:

```
function sql:exec-update($prepared-statement as xs:integer,
                         $param1 as item()?,
                         ...) 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/testdb",
              "user-name",
              "user-password",
              <sql:option name="manual-commit" value="1"/>)
return
  for $person in doc("auction")/person
  return (
    sql:execute($connection, "<do something with person>"),
    if (fn:all-is-ok($connection, $person)) then
      (
        sql:execute($connection, "<do something 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` function commits all changes made during the last transaction in the database connection specified by connection handle `$connection` and closes transaction. A Sedna error is raised if operation cannot be completed.

Function `sql:rollback` rolls back all changes made during the last transaction in the database connection specified by the connection handle `$connection` and closes transaction. A Sedna error is raised if operation cannot be completed.

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

2.2.5 External Functions

External function is a notion defined in the XQuery specification [3] as follows: "External functions are functions that are implemented outside the query environment". Support for external functions allows you to extend XQuery by implementing functions in other languages.

Sedna provides a server programming API to write external functions in the C/C++ language. External functions in Sedna are limited to dealing with sequences of atomic values. External functions are compiled and linked in the form of shared libraries (i.e. `.dll` files in Windows or `.so` files in Linux/FreeBSD and `.dylib` in Mac OS) and loaded by the server on demand. Although the Sedna XQuery executor evaluates queries in a lazy manner, all external function calls are evaluated in an eager manner.

Using External Functions

To use an external function you need to declare this function in prologue with `external` keyword instead of function body. Then it may be used normally:

```
declare function se:f($a as xs:integer) as $xs:integer external;
f(10)
```

Creating External Functions

External functions must be written in C/C++. To implement a new XQuery function `func` you should write the following C (or C++) functions: `func`, `func_init` and `func_deinit`. When executor decides that it needs to use an external function, first it initializes this function by calling `func_init`, after that it will call `func` to compute results as many times as needed. When some external function is not needed anymore, executor calls `func_deinit` (which probably will free any memory allocated by `func_init`). Each one of the three functions receives an `SEDNA_EF_INIT`[1] structure as a parameter. This structure has several fields that are initialized by executor before any `func_init` or `func_deinit` calls:

```
typedef struct sedna_ef_init
{
    void *(*sedna_malloc)(size_t);
    void (*sedna_free)(void *);
    SEDNA_SEQUENCE_ITEM *item_buf;
} SEDNA_EF_INIT;
```

The fields of this structure may be used in your implementation:

[1] All needed types and constants are defined in the sedna_ef.h file, located in the include directory of the Sedna distribution. See Section "Sedna Directory Structure" in [1] to learn where the include directory is located.

- `sedna_malloc` is a pointer to a malloc function which must be used to allocate memory for function results, this memory will be automatically freed by the query executor. It may also be used to allocate memory for internal use, such memory must be freed manually using the `sedna_free` function.

- `sedna_free` is a pointer to free function that releases memory allocated using `sedna_malloc` function.

- `item_buf` is a pointer to a preallocated `SEDNA_SEQUENCE_ITEM` which may be used to store results (this allows to avoid using `sedna_malloc` function when result is a single atomic non-string value)

`func`, `func_init` and `func_deinit` must have specific signatures:

- `func()` (required) – computes external function results. This function has the following signature:

```
SEDNA_SEQUENCE_ITEM *func(SEDNA_EF_INIT *init,
                          SEDNA_EF_ARGS *args,
                          char * error_msg_buf);
```

  - `init` is a pointer to the `SEDNA_EF_INIT` structure which was passed to `func_init` function (if written);
  - `args` is a pointer to the `SEDNA_EF_ARGS` structure which contains all function arguments;
  - `error_msg_buf` is a pointer to the string buffer used for specifying error message if function invocation fails. Maximum message length is `SEDNA_ERROR_MSG_BUF_SIZE` bytes, including the null character `'\0'`.

- `func_init()` (optional) – the initialization function. It can be used to allocate any memory required by the main function. This function has the following signature:

```
void func_init(SEDNA_EF_INIT *init, char * error_msg_buf);
```

  - `init` is a pointer to the `SEDNA_EF_INIT` structure (the pointer to this structure will be passed then to `func` and `func_deinit` functions);
  - `error_msg_buf` is a pointer to the string buffer used for specifying error message if function invocation fails. Maximum message length is `SEDNA_ERROR_MSG_BUF_SIZE`, including the null character `'\0'`.

- `func_deinit()` (optional) – the deinitialization function. It should deallocate any memory allocated by the initialization function. This function has the following signature:

```
void func_init(SEDNA_EF_INIT *init, char * error_msg_buf);
```

- `init` is a pointer to the `SEDNA_EF_INIT` structure which was passed to `func_init` function (if written);
- `error_msg_buf` is a pointer to the string buffer used for specifying error message if function invocation fails. Maximum message length is `SEDNA_ERROR_MSG_BUF_SIZE`, including the null character `'\0'`.

When `func`, `func_init` or `func_deinit` is being executed `error_msg_buf` contains an empty string. If function succeedes, it should leave this value empty. In case of error a non-empty string (error description) must be placed in `error_msg_buf` (if you place an empty string in `error_msg_buf` executor assumes that function execution was successful).

Each shared library must also export an null-terminated array with the names of the XQuery functions defined by this library:

```
char const *ef_names[] = {"func", NULL};
```

The file `sedna_ef.h` defines several types for representing function arguments and results:

- `SEDNA_ATOMIC_TYPE` – represents an atomic type, defined as:

```
typedef enum sedna_atomic_type {
     SEDNATYPE_integer,
     SEDNATYPE_float,
     SEDNATYPE_double,
     SEDNATYPE_string
} SEDNA_ATOMIC_TYPE;
```

- `SEDNA_ATOMIC_VALUE` – represents an atomic value, defined as:

```
typedef int     SEDNA_integer;
typedef float   SEDNA_float;
typedef double  SEDNA_double;
typedef char   *SEDNA_string;
typedef struct sedna_atomic_value {
    SEDNA_ATOMIC_TYPE type;
    union {
        SEDNA_integer   val_integer;
        SEDNA_float     val_float;
        SEDNA_double    val_double;
        SEDNA_string    val_string;
    };
} SEDNA_ATOMIC_VALUE;
```

Memory for values that are pointers (i.e. `SEDNA_string`) MUST be allocated using the malloc function passed in the `SEDNA_EF_INIT` structure.

- `SEDNA_SEQUENCE_ITEM` – represents a node in a linked list of atomic values, defined as:

```
typedef struct sedna_sequence_item {
    SEDNA_ATOMIC_VALUE data;
    struct sedna_sequence_item *next;
} SEDNA_SEQUENCE_ITEM;
```

Linked lists are used to represent sequences of atomic values. An empty sequence is presented by a `NULL` pointer. If `func` needs to return a sequence of values, memory for nodes MUST be allocated using the malloc function passed in `SEDNA_EF_INIT` structure.

- `SEDNA_EF_ARGS` – represents an array of arguments passed to a function, defined as:

```
typedef struct sedna_ef_args {
    int length;
    SEDNA_SEQUENCE_ITEM **args;
} SEDNA_EF_ARGS;
```

Location of External Function Libraries

Compiled libraries must be placed in the directory `lib` that is (1) in the same directory where the directory `data` with database data is located or (2) in the directory `<db_name>_files` where database data are stored[2]. Libraries that are database-independent should be placed in (1). Libraries that are database-specific should be placed in (2). Overloaded functions are not allowed. If two libraries located in (1) and (2) contain functions with the same name, a function from the library in (2) is called. If libraries in the same directory (1 or 2) contain functions with the same name, it is not specified which one is called.

There is a sample external function code available in the folder:

```
[win:] INSTALL_DIR\examples\api\external-functions\c\
[nix:] INSTALL_DIR/examples/api/external-functions/c/
```

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

[2]See Section "Sedna Directory Structure" in [1] to learn where database data are located

2.2.6 Runtime Properties

The `se:get-property` function provides a method for applications to determine in runtime the current values of system parameters, configurable limits, environment information. The name argument specifies the system variable to be queried. The function is defined within the predefined Sedna namespace (`se` prefix) as follows:

```
se:get-property($name as xs:string) as item()
```

The available `names` are as follows:

- `$user` - retrieves string which contains current user name

2.3 Update Language

The update language is based on the XQuery update proposal by Patrick Lehti [6] with the number of improvements.

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

Sedna provides several kinds of update statements:

- `INSERT` statement inserts zero or more nodes into a designated position with respect to a target nodes;
- `DELETE` statement removes target nodes from the database with theirs descendants;
- `DELETE_UNDEEP` statement removes target nodes from the database preserving theirs content;
- `REPLACE` statement replaces target nodes with a new sequence of zero or more nodes;
- `RENAME` statement changes the name of the target nodes.

The syntax and semantics of these expressions are described in the following sections.

Insert Statement

The insert statement inserts result of the given expression at the position identified by the `into`, `preceding` or `following` clauses:

```
UPDATE
insert SourceExpr (into|preceding|following) TargetExpr
```

`SourceExpr` 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 `TargetExpr`, the result sequence of `SourceExpr` 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 `TargetExpr`. If the `preceding` clause is specified, the sequence is appended before each node in the result of `TargetExpr`. If the `following` clause is specified, the sequence is appended after each node in the result of `TargetExpr`.

Error is raised if one of the following conditions is met:

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

For example, the following update statement inserts new *warning* element into all *blood_pressure* elements which have *systolic* value greater than 180:

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

Delete Statement

The `DELETE` statement removes 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 database. Note, that nodes are removed from the database with all their descendants.

Error is raised if one of the following conditions is met:

- There are atomic values in the result of the `Expr` expressions;
- There are temporary nodes in the result of the `Expr` expression evaluation (a node is considered *temporary*, if it is created as the result of the XQuery constructor evaluation).

The following update statement deletes all *blood_pressure* nodes which contain *systolic* value higher than 180:

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

Delete Undeep Statement

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

```
UPDATE
delete_undeep Expr
```

`Expr` identifies the nodes to be removed from the database.

Error is raised if one of the following conditions is met:

- There are atomic values in the result of the `Expr` expressions;
- There are temporary nodes in the result of the `Expr` expression evaluation (a node is considered *temporary*, if it is created as the result of the XQuery constructor evaluation).

Consider the following example. The document named `a.xml` before update:

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

The following delete undeep statement removes B nodes and makes C and D nodes children of the A element:

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

This is how the `a.xml` document will look after the update:

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

Replace Statement

The `REPLACE` statement is used to replace nodes in an XML document in the following manner:

```
UPDATE
replace $var [as type] in SourceExpr
with TargetExpr($var)
```

Replace statement iterates over all the nodes returned by the `SourceExpr`, binding the variable `$var` to each node. For each binding the result of the `TargetExpr($var)` expression is evaluated. Each node returned by the `SourceExpr` is replaced with the returned sequence of nodes. Note that `TargetExpr` is executed over the original document without taking into account intermediate updates performed during execution of this statement.

Error is raised if one of the following conditions is met:

- There are atomic values in the result of the `SourceExpr` or `TargetExpr` expressions;
- There are temporary nodes in the result of the `SourceExpr` expression evaluation (a node is considered *temporary*, if it is created as the result of the XQuery constructor evaluation).

The `$var` variable bound in `replace` clause may have an optional type declaration. If the type of a value bound to the variable does not match the declared type, an error is raised.

In the following example the salary of persons named "John" is doubled.

```
UPDATE
replace $p in doc("foo.xml")/db/person[name="John"]
with
<person>
{($p/@*,
  $p/node()[not(self::salary)],
  for $s in $p/salary
  return <salary>{$s*2}</salary>)}
</person>
```

Rename Statement

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

```
UPDATE
rename TargetExpr on QName
```

Rename statement changes name property of the all nodes returned by the `TargetExpr` expression with a new QName.

Error is raised if one of the following conditions is met:

- There are items which are not element or attribute nodes in the result of the `TargetExpr` expression evaluation;

- There are temporary nodes in the result of the `TargetExpr` expression evaluation (a node is considered *temporary*, if it is created as the result of the XQuery constructor evaluation).

The following expression changes the name of all the `job` elements without changing their contents:

```
UPDATE
rename doc("foo.xml")//job on profession
```

2.4 Bulk Load

To bulk load a stand alone document use the following statements:

```
LOAD "path_to_file" "document_name"
```

The first parameter is a path to the file which contains a document to be loaded. The second parameter is the name for this document in the database.

For example,

```
LOAD "/opt/test.xml" "test"
```

loads file `/opt/test.xml` into database as a stand-alone document with name `test`.

To load document into a collection, use the following statement:

```
LOAD "path_to_file" "document_name" "collection_name"
```

The first parameter is a path to the file which contains a document to be loaded. 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 example,

```
LOAD "/opt/mail-01.xml" "mail-01" "mails"
```

loads file `/opt/mail-01.xml` into collection `mails`.

For performing bulk load not from the source file but from an input stream, use the following statements (first for loading stand alone document, second - for loading into a collection):

```
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.

By default, Sedna removes boundary whitespace according to the boundary-space policy defined in [3]. To control boundary whitespace processing, use `boundary-space declaration` [3] in the prolog of the `LOAD` statement. The following example illustrates a boundary-space declaration that instructs Sedna to preserve whitespace while loading `auction.xml` document:

```
declare boundary-space preserve;
LOAD "auction.xml" "auction"
```

Notice, that heavy bulk-loads might be greatly optimized by setting `SEDNA_LOG_AMOUNT` connection attribute to `SEDNA_LOG_LESS` (see Section 1.2.3 for more information).

2.4.1 CDATA section preserving

It is possible to save the formatting of continuous CDATA sections with `cdata-section-preserve` option.

```
declare option se:bulk-load "cdata-section-preserve=yes";
LOAD "auction.xml" "auction"
```

The `cdata-section-preserve=yes` option makes text nodes within CDATA sections to be serialized within CDATA sections. CDATA section formatting is saved only for the whole text node and this property of text node is inherited when text node is appended. E.g. in the following document fragment CDATA section will be serialized as it appears in document:

```
<a><![CDATA[<example data>]]></a>
```

But the next fragment will not be saved with mixed formatting.

```
<a><![CDATA[<example]]> data<![CDATA[>]]></a>
```

Instead, it will be serialized in the same way as prevoius one, i.e. the whole text will be in CDATA section.

2.5 Data Definition Language

This section describes Sedna Data Definition Language (DDL) that is used to create and manage the database structures that will hold data.

Most of parameters of Sedna DDL are computable and specified as XQuery expressions. The expected type of all the parameters is `xs:string`. All parameters are evaluated and atomized. If the atomized value is not of `xs:string`, a dynamic error is raised.

2.5.1 Managing Standalone Documents

```
CREATE DOCUMENT doc-name-expr
```

The `CREATE DOCUMENT` statement creates a new standalone document with the name that is the result of `doc-name-expr`.

```
DROP DOCUMENT doc-name-expr
```

The `DROP DOCUMENT` statement drops the standalone document with the name that is the result of `doc-name-expr`.

For example, next statement:

```
}
CREATE DOCUMENT "report"
```

creates a documentnamed ”report”, while:

```
DROP DOCUMENT "report"
```

There is a system document `$documents` which lists all available documents and collections. For details on retrieving metadata see 2.5.6 section.

2.5.2 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-expr
```

The `CREATE COLLECTION` statement creates a new collection with the name that is the result of `coll-name-expr`.

For example, `CREATE COLLECTION "mails"` creates a collection named "mails".

XML documents can be loaded into the collection, as previously described in section 2.4.

To access a single document from collection in an XQuery or XML update query, the `document` function accepts a collection name as its second optional argument:

```
doc($doc as xs:string,
    $col as xs:string) as document-node()
```

The function returns the document with the `$doc` name that belongs to the collection named `$col`.

For example, `doc('mail-01', 'mails')` query returns documents with name `mail-01` from collection `mails`.

```
fn:collection($col as xs:string?) as document-node()*
```

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 `$col`. 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.

```
CREATE DOCUMENT doc-name-expr IN COLLECTION coll-name-expr
```

This statement creates a new document named `doc-name-expr` in a collection named `coll-name-expr`.

For example, the following statement:

```
CREATE DOCUMENT 'mail' IN COLLECTION 'mails'
```

creates a document named "mail" in the collection named "mails".

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

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

```
DROP COLLECTION coll-name-expr
```

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

```
RENAME COLLECTION old-name-expr INTO new-name-expr
```

The `RENAME COLLECTION` statement renames collection with the name that is result of the `old-name-expr`. The new name is assigned which is result of the `new-name-expr`. Both result of the `old-name-expr` and result of the `new-name-expr` after atomization applied must be either of type `xs:string` (or derived) or promotable to `xs:string`.

There is a system document `$collections` which lists all available collections. For details on retrieving metadata see 2.5.6 section.

2.5.3 Managing Value Indices

Sedna supports value indices to index XML element content and attribute values. Index could be based on two different structure types: B+ tree and BST (experimental) (*B*lock *S*tring *T*rie). Below is the description of statements to manage indices.

```
CREATE INDEX title-expr
ON Path1 BY Path2
AS type
[USING tree-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-expr` 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 keys should be cast to. The following types are supported for B-tree: `xs:string`, `xs:integer`, `xs:float`, `xs:double`, `xs:date`, `xs:time`, `xs:dateTime`, `xs_yearMonthDuration`, `xs_dateTimeDuration`. Note that BST supports `xs:string` only.

`tree-type` defines the structure that would be used for index storage. This argument is optional. Index is stored using B+ tree structure by default and it's a good choice in the general case. But there is one more implemented structure that could show great disk-space economy in certain situations. BST is based on prefix tree (or *trie*) conception. Main distinguishing features are:

1. BST can handle strings with any length.
2. If you want to index data by fields that contain strings with repeating prefixes (e.g: URLs, URIs) BST would be a good choice for you.

In the case of appropriate usage BST can store indexes up to 4 times more compressed with the same search speed in comparison with B+ tree.

The following `tree-type` are supported: `"btree"` for B+ tree (default), `"bstrie"` for BST.

Note 3 *BST feature is experimental at this moment;* ***do not use it in production*** *or any critical applications.*

In the following example, people are indexed by the names of their cities. To generate the index keys, city names are cast to the `xs:string` type. B+ tree is used for index storage.

```
CREATE INDEX "people"
ON doc("auction")/site//person BY address/city
AS xs:string
```

The following example is exactly the same as the previous but uses BST for index storage:

```
CREATE INDEX "people"
ON doc("auction")/site//person BY address/city
AS xs:string
USING "bstrie"
```

To remove an index, use the following statement:

```
DROP INDEX title-expr
```

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

Note 4 *In the current version of Sedna, query executor does not use indices automatically. You can enforce the executor to employ indices by using the XQuery index-scan functions specified in section 2.2.2.*

2.5.4 Managing Full-Text Indices

Sedna allows to build full-text indices in order to combine XQuery with full-text search facilities. Resulting indices need to be used explicitly via full-text search functions (see 2.2.3), XQuery full text extensions are not supported.

Sedna can be integrated with dtSearch [15], a commercial text retrieval engine, which provides full-text indices. As dtSearch is a third party commercial product, Sedna does not include dtSearch. If you are interested in using Sedna with dtSearch, please contact us. Below is the description of statements to manage full-text indices in Sedna.

```
CREATE FULL-TEXT INDEX title-expr
ON path
TYPE type
[
WITH OPTIONS options
]
```

The `CREATE INDEX` indexes nodes (specified by `path`) by a text representation of the nodes. The text representations of the nodes are constructed according to `type` parameter value.

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

`path` is an XPath expression without any filter expressions that identifies the nodes of a document or a collection that are to be indexed. An example of the `path` expression is as follows `doc("foo")/library//article`.

`type` specifies how the text representations of nodes are constructed when the nodes are indexed. `type` can have one of the following values:

- `"xml"` – the XML representations of the nodes are used;
- `"string-value"` – the string-values of the nodes are used as obtained using standard XQuery `fn:string` function. The string-value of a node is the concatenated contents of all its descendant text nodes, in document order;
- `"delimited-value"` – the same as `"string-value"` but blank characters are inserted between text nodes;
- `"customized-value" ((element-qname, type) , ... (element-qname, type))` – this option allows specifying types for particular element nodes. Here `element-qname` is a QName of an element,

`type` can have one of the values listed above (i.e. `"xml"`, `"string-value"`, `"delimited-value"`). For those elements that are not specified in the list, the `"xml"` type is used by default.

`options` is a sting of the following form: `"option=value{,option=value}"`. It denotes options used for index constuction. The following options are available:

- `backend` – specifies which implementation of full-text indexes to use. Allowed values are `native` and `dtsearch`, the latter in only available in dtSearch-enabled builds. For dtSearch-enabled builds, default backend is `dtsearch`, for other builds - `native`.

Options for `native` backend:

- `stemming` – specifies stemming language to use.
- `stemtype` – in order to be able to search both stemmed and original words - add `stemtype=both` option, otherwise stemming will be used always if enabled.

In the following example, articles are indexed by their contents represented as XML.

```
CREATE FULL-TEXT INDEX "articles"
ON doc("foo")/library//article
TYPE "xml"
```

The example below illustrates the use of `"customized-value"` type.

```
CREATE FULL-TEXT INDEX "messages"
ON doc("foo")//message
TYPE "customized-value"
     (("b", "string-value"),
     ("a", "delimited-value"))
```

To remove a full-text index, use the following statement:

```
DROP FULL-TEXT INDEX title-expr
```

The `DROP FULL-TEXT INDEX` statement removes the full-text index named `title-expr` from the database.

Note 5 *In the current version of Sedna, query executor does not use full-text indices automatically. You can enforce the executor to employ indices by using the XQuery full-text search functions specified in section 2.2.3.*

2.5.5 Managing Modules

XQuery allows putting functions in library modules, so that they can be shared and imported by any query. A library module contains a module declaration followed by variable and/or function declarations. The module declaration specifies its target namespace URI which is used to identify the module in the database. For more information on modules see the XQuery specification [3].

Before a library module could be imported from an query, it is to be loaded into the database. To load a module, use the following statement.

```
LOAD MODULE "path_to_file", ..., "path_to_file"
```

Each `path_to_file` specifies a path to the file. If only one parameter is supplied, it refers to the file which contains the module definition. The module definition can also be divided into several files. In this case all files must have a module declaration with the same target namespace URI (otherwise an error is raised).

For example, suppose that you have the following module stored in `math.xqlib`.

```
module namespace math = "http://example.org/math";

declare variable $math:pi as xs:decimal := 3.1415926;

declare function math:increment($num as xs:decimal) as xs:decimal {
    $num + 1
};

declare function math:square($num as xs:decimal) as xs:decimal {
    $num * $num
};
```

You can load this module as follows.

```
LOAD MODULE "math.xqlib"
```

Once an library module is loaded into the database, it can be imported into an query using conventional XQuery module import [3]. For example, you can import the above module as follows.

```
import module namespace math = "http://example.org/math";

math:increment(math:square($math:pi))
```

To replace an already loaded module with new one, use the following statement.

```
LOAD OR REPLACE MODULE "path_to_file", ..., "path_to_file"
```

To remove a library module from the database, use the following statement.

```
DROP MODULE "target_namespace_URI"
```

It results in removing the library module with the given target namespace URI from the database.

You can obtain information about modules loaded into the database by querying the system collection named `$modules` as follows `collection("$modules")`.

2.5.6 Retrieving Metadata

You can retrieve various metadata about database objects (such as documents, collections, indexes, etc.) by querying system documents and collections listed below.

Names of the system documents and collections start with `$` symbol. The system documents and collections (except the ones marked with * symbol) are not persistent but generated on the fly. You can query these documents as usual but you cannot update them. Also these documents are not listed in the `$documents` system document.

- `$documents` document – list of all stand-alone documents, collections and in-collection documents (except system meta-documents and collection, like `$documents` document itself);
- `$collections` document – list of all collections;
- `$modules` document – contains list of loaded modules with theirs names;
- `$modules` (*) collection – contains documents with precompiled definitions of XQuery modules;
- `$indexes` document – list of indexes with information about them;
- `$ftindexes` document – list of full-text indexes with information about them (this document is available if Sedna is build with `SE_ENABLE_FTSEARCH` enabled);
- `$triggers` document – list of triggers with information about them (this document is available if Sedna is build with `SE_ENABLE_TRIGGERS` enabled);
- `$db_security_data` (*) document– list of users and privileges on database objects;
- `$schema` document – descriptive schema of all documents and collections with some schema-related information;
- `$errors` document – list of all errors with descriptions;
- `$version` document – version and build numbers;
- `$schema_<name>` document – the descriptive schema of the document or collection named `<name>`;
- `$document_<name>` document – statistical information about the document named `<name>`;

- `$collection_<name>` document – statistical information about the collection named `<name>`.

The statistical information in `$document_<name>` and `$collection_<name>` documents contains the following elements:

- `total_schema_nodes` – the number of the nodes of the descriptive schema;
- `total_schema_text_nodes` – the number of the attribute and text nodes of the descriptive schema;
- `total_nodes` – the number of the nodes of the document (or collection);
- `schema_depth` – the maximal depth of the document (or collection);
- `total_desc_blk` – the number of the descriptor blocks occupied by document (or collection);
- `total_str_blk` – the number of the text blocks occupied by document (or collection);
- `saturation` – fill factor of the blocks (in percents);
- `total_innr_blk` – the number of the descriptor blocks occupied by document (or collection) except first and last blocks in each chain of blocks;
- `total_innr_size` – the size of the inner blocks;
- `innr_blk_saturation` – 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);
- `nid` – the share of the long labeling numbers' (> 11) total size (in percents);
- `indirection` – the share of the indirection records' total size (in percents);
- `total_size` – the total size of the document (or collection), in MBs;
- `string_size` – the total size of the string blocks, in MBs;
- `descriptor_size` – the total size of the descriptor blocks, in MBs;
- `nids_size` – the total size of the long labeling numbers, in MBs;
- `free_space_in_str_blocks` – the total size of the free space in the string blocks;
- `indirection_size` – the total size of the indirection records, in MBs;
- `nids_size` – the share of the indirection records' total size (in percents);
- `STRINGS` – the histogram of the xml data by its size;
- `NID` – the histogram of the labeling numbers its length;

2.6 XQuery Triggers

XQuery triggers support in Sedna is provided as an XQuery extension. To create a trigger into the Sedna database you have to issue the following `CREATE TRIGGER` statement:

```
CREATE TRIGGER trigger-name
( BEFORE | AFTER ) (INSERT | DELETE | REPLACE)
ON path
```

```
( FOR EACH NODE | FOR EACH STATEMENT )
DO {
  Update-statement ($NEW, $OLD,$WHERE);
            . . .
  Update-statement ($NEW, $OLD,$WHERE);
  XQuery-statement ($NEW, $OLD, $WHERE);
}
```

The `DROP TRIGGER` statement drops the trigger with the name which is the result of the `trigger-name-expression`:

```
DROP TRIGGER trigger-name-expression
```

Triggers can be defined to execute either *before* or *after* any `INSERT`, `DELETE` or `REPLACE` operation, either once per modified node (*node-level* triggers), or once per XQuery statement (*statement-level* triggers). If a trigger event occurs, the trigger's action is called at the appropriate time to handle the event.

Create Trigger Parameters:

- `trigger-name` is the unique per database trigger name.
- `ON path` is XPath expression without any filter expression (predicates) that identifies the nodes on which the trigger is set. That means that the trigger fires when corresponding modification (insertion, deletion or replacement) of those nodes occurs. This XPath expression is prohibited to have predicates and parent axes.
- `FOR EACH NODE/FOR EACH STATEMENT`: these key words mean the trigger created is a `node-level` or `statement-level` trigger. With a node-level trigger, the trigger action is invoked once for each node that is affected by the update statement that fired the trigger. In contrast, a statement-level trigger is invoked only once when an appropriate statement is executed, regardless of the number of nodes affected by that statement.
- `BEFORE/AFTER`: triggers are also classified as *before*-triggers and *after*-triggers. `BEFORE` keyword in `CREATE TRIGGER` statement means the trigger created is *before*-trigger; `AFTER` keyword means the trigger created is *after*-trigger. Statement-level-before triggers fire before the statement starts to do anything, while statement-level-after triggers fire at the very end of the statement. Node-level-before triggers fire immediately before a particular node is operated on, while node-level-after triggers fire immediately after the node is operated on (but before any statement-level-after trigger).

- `DO`: trigger action is specified in braces `{}` after the `DO` key word. Action contains zero or more update statements and an XQuery query. It is a mandatory requirement that node-level trigger action ends with an XQuery query, while this is optional for actions of statement-level triggers. It is prohibited to use prolog in statements of the trigger action.

- Transition variables `$NEW`, `$OLD` and `$WHERE` are defined for each node-level trigger firing and can be used in each statement of the trigger action. These tree variables are defined as follows:

  - For `INSERT`: `$NEW` is the node being inserted; `$OLD` is undefined; `$WHERE` is the parent node in case in *insert-into* statement and sibling node in case of *insert-preceding* and *insert-following* statements;
  - For `DELETE`: `$NEW` is undefined; `$OLD` is the node being deleted; `$WHERE` is the parent of the deleted node;
  - For `REPLACE`: `$NEW` is the node being inserted during the replacement; `$OLD` is the node being replaced; `$WHERE` is the parent of the replaced node.

  You cannot use transition variables in statement-level triggers.

XQuery statement in the trigger action of a node-level trigger can *return a node* to the calling executor, if they choose. A node-level-trigger fired before an operation has the following choices:

- It can return `empty sequence` to skip the operation for the current node. This instructs the executor to not perform the node-level operation that invoked the trigger (the insertion or replacement of a particular node).

- For node-level `INSERT` triggers only, the returned node becomes the node that *will be inserted*. This allows the trigger to modify the node being inserted.

- A node-level before trigger that does not intend to cause either of these behaviors must be careful to return as its result the same node that was passed in (that is, the `$NEW` node for `INSERT` and `REPLACE` triggers. For `DELETE` triggers its returned value is ignored in all cases except it is an empty sequence).

The trigger action return value is ignored for node-level triggers fired after an operation, and for all statement-level triggers, and so they may as well return empty sequence.

If more than one trigger is defined for the same event on the same document, the triggers will be fired in alphabetical order by trigger name. In the case of before triggers, the possibly-modified node returned by each trigger becomes the input

to the next trigger. If any before trigger returns empty sequence, the operation is abandoned for that node and subsequent triggers are not fired.

Typically, node-level-before triggers are used for checking or modifying the data that will be inserted or updated. For example, a before trigger might be used to insert the current time node as a child of the inserting node, or to check that two descendants of the inserting node are consistent. Node-level-after triggers are most sensibly used to propagate the updates to other documents, or make consistency checks against other documents. The reason for this division of labor is that an after-trigger can be certain it is seeing the final value of the node, while a before-trigger cannot; there might be other before triggers firing after it. When designing your trigger-application note, that node-level triggers are typically cheaper than statement-level ones.

If a trigger function executes update-statement then these commands may fire other triggers again (cascading triggers). Currently *trigger cascading level* in Sedna is limited to 10.

Note 6 *Currently is it prohibited in a trigger action to update the same document or collection that is being updated by the outer update statement that has fired this trigger.*

	update: INSERT	*update*: DELETE	*update*: REPLACE
trigger event: INSERT	trigger path >= update path		trigger path >= update path
trigger event: DELETE		trigger path >= update path	trigger path >= update path
trigger event: REPLACE			trigger path >= update path

Figure 1: Update and trigger path lengths needed for trigger firing

Note also that hierarchy of the XML data sometimes can affect the trigger firing in a complicated way. For example, if a node is deleted with all its descendant subtree, then a `DELETE`-trigger set on the descendants of the deleting node is fired. In this situation *length of trigger path >= length of update path.* In general, triggers fire according to the table in figure 1.

2.6.1 Trigger Examples

The following trigger is set on insertion of `person` nodes. When some `person` node is inserted, the trigger analyzes its content and modifies it in the following

way. If the person is under 14 years old, the trigger inserts additional child node `age-group` with the text value 'infant': if the person is older than 14 years old - the trigger inserts `age-group` node with value 'adult':

```
CREATE TRIGGER "tr1"
BEFORE INSERT
ON doc("auction")/site//person
FOR EACH NODE
DO {
  if($NEW/age < 14)
  then
    <person>{attribute id {$NEW/@id}}
            {$NEW/*}
              <age-group>infant</age-group>
    </person>
    else
    <person>{attribute id {$NEW/@id}}
            {$NEW/*}
            <age-group>adult</age-group>
    </person>;
}
```

The following trigger *tr2* prohibits (throws exception) stake increase if the person has already more than 3 open auctions:

```
CREATE TRIGGER "tr2"
BEFORE INSERT
ON doc("auction")/site/open_auctions/open_auction/bidder
FOR EACH NODE
DO {
 if(($NEW/increase > 10.5) and
  (count($WHERE/../open_auction
    [bidder/personref/@person=$NEW/personref/@person]) > 3))
 then error(xs:QName("tr2"),"The increase is prohibited")
 else ($NEW);
}
```

The following trigger *tr3* cancels `person` node deletion if there are any open auctions referenced by this person:

```
CREATE TRIGGER "tr3"
BEFORE DELETE
ON doc("auction")/site//person
FOR EACH NODE
```

```
DO {
  if(exists(
    $WHERE//open_auction/bidder/personref/@person=$OLD/@id))
  then ()
  else $OLD;
}
```

The next statement-level trigger *tr4* maintains statistics in the document named *stat*. When this trigger is fired, the update operation is completed - that gives the possibility to make aggregative checks on the updated data. After deletion of any node in the *auction* document, the trigger refreshes statistics in *stat* and throws exception if there are more than 50 persons left:

```
CREATE TRIGGER "tr4"
AFTER DELETE
ON doc("auction")//*
FOR EACH STATEMENT
DO {
  UPDATE replace $b in doc("stat")/stat with
  <stat>
    <open_auctions>
      {count(doc("auction")//open_auction)}
    </open_auctions>
    <closed_auctions>
      {count(doc("auction")//closed_auction)}
    </closed_auctions>
    <persons>
      {count(doc("auction")//person)}
    </persons>
  </stat>;

  UPDATE insert
  if(count(doc("auction")//person) < 10)
  then <warning>
         "Critical number of person left in the auction"
       </warning>
  else ()
  into doc("stat")/stat;
}
```

2.7 Debug and Profile Facilities

Sedna provides several tracing, debug and perfomance profiling tools that can help to monitor and analyze queries and update statements:

- **Trace** – provides a query execution trace intended to be used in debugging queries (section 2.7.1);
- **Debug Mode** – provides stack of physical operations when error is raised (section 2.7.2);
- **Explain Query** – shows complete physical execution plan of the query or update statement (section 2.7.3);
- **Profile Query** – creates complete physical execution plan of the query or update statement with profile information (execution time, number of calls, etc) for each physical operation (section 2.7.4).

2.7.1 Trace

Sedna supports standard XQuery function `fn:trace` [4] providing user helper facility to trace queries. While executing XQuery query using `fn:trace` function intermediate results are shown to the user.

For example in Sedna Terminal the query with `fn:trace` function will provide the following output. Trace information is marked with `##` string:

Query:

```
let $r:= trace(doc("region")/regions/*, "## ")
return $r[id_region="afr"]
```

Output:

```
## <africa><id_region>afr</id_region></africa>

<africa>
  <id_region>afr</id_region>
</africa>

## <asia><id_region>asi</id_region></asia>
## <australia><id_region>aus</id_region></australia>
## <europe><id_region>eur</id_region></europe>
## <namerica><id_region>nam</id_region></namerica>
## <samerica><id_region>sam</id_region></samerica>
```

If you want to use trace facility in your application working through Sedna API you should set your own debug handler as it is shown in 1.2.6 section.

2.7.2 Debug Mode

In addition to trace facility provided by standard XQuery `fn:trace` function (see previous section, 2.7.1) debug mode can be turned on in Sedna Terminal (for details see "Sedna Terminal" section of the Sedna's Administration Guide) or in your application using corresponding Sedna API functions (see 1.2.3 section).

Each query in Sedna is represented and executed internally as a tree of the physical operations. Debug mode is mechanism that allows to get stack of the physical operations after dynamic error was raised. It serves two goals:

- localize error in the query's source code;
- obtain information of the query execution process.

Let us consider the following query to illustrate execution in the debug mode:

Query:

```
(: In this query dynamic error will be raised   :)
(: due to "aaaa" is not castable to xs:integer. :)
declare function local:f($i as item()) as xs:integer
{
   $i cast as xs:integer
};

for $i in (1,2,3,"aaaa")
return local:f($i)
```

Output:

```
1
2
3

<stack xmlns='http://www.modis.ispras.ru/sedna'>
  <operation name='PPCast' line='3' column='4' calls='7'/>
  <operation name='PPFunCall' line='7' column='8' calls='7'/>
  <operation name='PPReturn' line='6' column='5' calls='4'/>
  <operation name='PPQueryRoot' calls='4'/>
</stack>

SEDNA Message: ERROR FORG0001
Invalid value for cast/constructor.
Details: Cannot convert to xs:integer type
Query line: 3, column:4
```

As you can see in the output above each item of the operations stack list consists of the following parts:

- operation name (`PPCast` and `PPFunCall` in the example);
- calls counter - number of calls of the operation;
- corresponding query and column numbers;
- optional additional information (qualified name of the function in the example).

2.7.3 Explain Query

The explain statement has the following syntax:

```
EXPLAIN
{XQuery or Update statement to explain}
```

Each query in Sedna is represented and executed internally as a tree of the physical operations. With the help of `EXPLAIN` statement you can obtain detailed query execution plan which shows how Sedna executes this query.

The following query illustrates `EXPLAIN` statement execution:

Query:

```
explain
declare function local:fact($i as xs:integer) as xs:integer {
    if ($i <= 1)
    then 1
    else $i * local:fact($i - 1)
};

local:fact(10)
```

Output:

```
<prolog xmlns="http://www.modis.ispras.ru/sedna">
  <function id="0" function-name="local:fact" type="xs:integer">
    <arguments>
      <argument descriptor="0" type="xs:integer"/>
    </arguments>
    <operation name="PPIf" position="2:5">

      <operation name="PPLMGeneralComparison"
                 comparison="le" position="2:12">
```

```
        <operation name="PPVariable" descriptor="0"
                   variable-name="i" position="2:9"/>

        <operation name="PPConst" type="xs:integer"
                   value="1" position="2:15"/>
      </operation>
      <operation name="PPConst" type="xs:integer"
                 value="1" position="3:10"/>

      <operation name="PPCalculate" position="4:10">
        <operation name="BinaryOp" operation="*">
          <operation name="LeafAtomOp">

            <operation name="PPVariable" descriptor="0"
                       variable-name="i" position="4:10"/>
          </operation>

          <operation name="LeafAtomOp">
            <operation name="PPFunCall" id="0"
                       function-name="local:fact" position="4:15">
              <operation name="PPCalculate" position="4:26">
                <operation name="BinaryOp" operation="-">
                  <operation name="LeafAtomOp">
                    <operation name="PPVariable" descriptor="0"
                               variable-name="i" position="4:26"/>
                  </operation>
                  <operation name="LeafAtomOp">
                    <operation name="PPConst" type="xs:integer"
                               value="1" position="4:31"/>
                  </operation>
                </operation>
              </operation>
            </operation>
          </operation>
        </operation>
      </operation>

    </operation>
  </function>
</prolog>

<query xmlns="http://www.modis.ispras.ru/sedna">
```

```
  <operation name="PPQueryRoot">
    <operation name="PPFunCall" id="0"
               function-name="local:fact" position="7:1">

      <operation name="PPConst" type="xs:integer"
                 value="10" position="7:12"/>

    </operation>
  </operation>
</query>
```

Explain output consists of two parts (just like any XQuery query): prolog and query body explanations. Prolog part includes complete information about all declarations: namespaces, functions, global variables with complete physical plan for each user defined function and global variable. Query body explanation part describes physical tree of the query.

For each physical operation `EXPLAIN` returns: name of the operation (`PPConst`, `PPFunCall`, etc), corresponding position in the source query (e.g. 4:31 means that operation `PPConst` corresponds to the '1' atomic at the line 4, column 26). Output may also contain additional information depending on the operation type (for example, variable name for some `PPVariable` operations).

2.7.4 Profile Query

The profile statement has the following syntax:

```
PROFILE
{XQuery or Update statement to profile}
```

Each query in Sedna is represented and executed internally as a tree of the physical operations. With the help of `PROFILE` statement you can obtain detailed tree of physical operation and execution time for each of them.

The following query illustrates `PROFILE` statement execution:

Query:

```
profile fn:doc('TestSources/XMarkAuction.xml')//
            person[@id = "person0"]/name
```

Output:

```
<operation name="PPQueryRoot" time="13.426" calls="1">
  <operation name="PPAxisChild" step="child::element(name)"
                                time="13.426" calls="2">
    <operation name="PPReturn" time="13.426" calls="2">
```

```
      <operation name="PPAbsPath" root="document(auction)"
                                  path="descendant-or-self::node()"
                                  time="12.772" calls="85405"/>

      <operation name="PPPred1" time="0.530" calls="85405">

        <operation name="PPAxisChild" step="child::element(person)"
                                      time="0.461" calls="86168">
          <operation name="PPVariable" descr="0"
                     time="0.380" calls="170808"/>
        </operation>

        <operation name="PPEQLGeneralComparison"
                   comparison="eq" time="0.013" calls="1528">

          <operation name="PPAxisAttribute"
                     step="attribute::attribute(id)"
                     time="0.001" calls="1527">

            <operation name="PPVariable" descr="1"
                       time="0.001" calls="1527"/>

          </operation>

          <operation name="PPConst"
                     type="xs:string" value="person0"
                     time="0.001" calls="1527"/>

        </operation>
      </operation>
    </operation>
  </operation>
</operation>
```

Profiling output consists of two parts (just like any XQuery query): prolog and query body explanations. Prolog part includes complete profile information for global variables and user defined functions. Query body profile part describes physical tree of the query and provides execution time and number of calls for each physical operation.

For each physical operation `PROFILE` returns: name of the operation (`PPConst`, `PPFunCall`, etc), corresponding position in the source query (e.g. 4:31 means that operation `PPConst` corresponds to the '1' atomic at the line 4, column 26), execution time of this operation and number of calls. Output may also contain

additional information depending on the operation type (for example, variable name for some `PPVariable` operations).

In above example you can see that `PPAbsPath` operation takes almost all time (12.772 seconds of 13.426 total) and was called 12772 times. Profiling in this case shows that `//` may be very hard to execute and it is much better to use "single" XPath steps everywhere it is possible:

Query:

```
profile fn:doc('TestSources/XMarkAuction.xml')/
          site/people/person[@id = "person0"]/name
```

Output:

```
<operation name="PPQueryRoot" time="0.018" calls="1">

  <operation name="PPAxisChild" step="child::element(name)"
                                time="0.018" calls="2">

    <operation name="PPReturn" time="0.018" calls="2">
      <produces>
        <variable descriptor="0"/>
      </produces>

      <operation name="PPAbsPath"
                 root="document(auction)"
                 path="child::element(site)/child::element(people)"
                 time="0.001" calls="2"/>

      <operation name="PPPred1" time="0.017" calls="2">
        <produces>
          <variable descriptor="1"/>
        </produces>

        <operation name="PPAxisChild" step="child::element(person)"
                                      time="0.001" calls="765">

          <operation name="PPVariable" descr="0"
                     time="0.000" calls="2"/>
        </operation>

        <operation name="PPEQLGeneralComparison"
                   comparison="eq" time="0.013" calls="1528">
```

```
          <operation name="PPAxisAttribute"
                     step="attribute::attribute(id)"
                     time="0.005" calls="1527">

            <operation name="PPVariable" descr="1"
                       time="0.003" calls="1527"/>

          </operation>
          <operation name="PPConst"
                     type="xs:string" value="person0"
                     time="0.000" calls="1527"/>
        </operation>
      </operation>
    </operation>
  </operation>
</operation>
```

References

[1] "Sedna Administration Guide", http://modis.ispras.ru/sedna/adminguide/AdminGuide.html

[2] "XQuery 1.0 and XPath 2.0 Data Model", W3C Recommendation, http://www.w3.org/TR/xpath-datamodel/

[3] "XQuery 1.0: An XML Query Language", W3C Recommendation, http://www.w3.org/TR/xquery/

[4] "XQuery 1.0 and XPath 2.0 Functions and Operators", W3C Recommendation, http://www.w3.org/TR/xpath-functions/

[5] "XSLT 2.0 and XQuery 1.0 Serialization", W3C Recommendation, http://www.w3.org/TR/xslt-xquery-serialization/

[6] Patrick Lehti. "Design and Implementation of a Data Manipulation Processor for a XML Query Language", http://www.lehti.de/beruf/diplomarbeit.pdf

[7] "PCRE - Perl Compatible Regular Expressions", http://www.pcre.org/

[8] 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

[9] Rekouts Maria. "Application Programming Interface for XML DBMS: design and implementation proposal", http://www.modis.ispras.ru/publications.htm

[10] Noel Welsh, Francisco Solsona, Ian Glover. "SchemeUnit and SchemeQL: Two little languages", Scheme Workshop 2002, http://schematics.sourceforge.net/schemeunit-schemeql.ps

[11] Oleg Kiselyov. "SXML Specification, Revision 3.0", http://www.okmij.org/ftp/Scheme/SXML.html

[12] William Clinger, R. Kent Dybvig, Matthew Flatt, and Marc Feeley. "SRFI-12: Exception Handling", http://srfi.schemers.org/srfi-12/srfi-12.html

[13] Felix L. Winkelmann. "Chicken – A practical and portable Scheme system", http://www.call-with-current-continuation.org

[14] 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

[15] dtSearch Engine home page, http://www.dtsearch.com/PLF_engine_2.html

[16] dtSearch Web Help http://support.dtsearch.com/webhelp/dtsearch