Calling sa.engine from C

Stream Analyze Sweden AB
Sweden

Version 2.0
2022-01-13
`sa_client_2.0.pdf`

One important property of sa.engine is that it is designed to be called from other systems using sa.engine APIs in a several programming languages. There are predefined APIs for interfacing code in C99, C++, Lisp, Java, Python and JavaScript. The system furthermore provides primitives for defining APIs to any other programming language based on the C99 API. This document describes the external interfaces between external programs in C and sa.engine.

Table of contents

1. Introduction 3
   1.1. Peers 3
   1.2. Object handles 4
   1.3. Connections 5
2. Executing OSQL queries 6
   2.1. The callback API to object streams 7
   2.2. The run API to object streams 8
3. Calling OSQL functions 9
   3.1. Type resolution 10
4. Error handling 10
5. Multi-threaded clients 11
6. Data objects 12
   6.1. Tuples 12
   6.2. Integers 13
   6.3. Floating point numbers 13
   6.4. Strings 13
   6.5. Generic objects 13
   6.6. Vectors 14
   6.7. Records 14
   6.8. Binary areas 14
7. References 15

1. Introduction

In https://streamanalyze.com/under-the-hood/ you find an overview of the sa.engine system. In [4] there is a more detailed introduction to the sa.engine kernel system.

There are two main kinds of external interfaces to sa.engine, the *client* and the *plugin* interfaces:

- With the *client interface* a program implemented in some programming language calls sa.engine. The client interface allows OSQL queries and function calls to be shipped from application programs to either i) *remote* sa.engine servers or ii) an *embedded* sa.engine system running in the same process as the application. This document describes the client API for the programming language C. There are similar APIs for the programming languages Java [2], Lisp [3], C++, Python, and JavaScript.

- With the *plugin interface* OSQL functions are implemented as code in some programming language. These *foreign OSQL functions* are executed in the same process and address space as sa.engine. The client interface can be used also in foreign OSQL function implementations. The plugin interface for C is documented separately [1].

The result of an OSQL query or function call is an *object stream*, which is a possibly infinite stream of objects. The client interface provides primitives to *consume* the elements in such object streams by using callback functions or methods provided by the application.

The client API of sa.engine for C is presented in this document through a number of example programs whose source codes are in the folder `sa.engine/demo/client/C/` of an installed sa.engine system. In that folder you will find a number of examples for how to compile, use and validate C client code using the API. You are assumed to be familiar with OSQL.

The client interface is defined by the header file

```
#include "sa_client.h"
```

1.1. Peers

When calling sa.engine from application programs, the application must be connected to some sa.engine *peer* [4]. A peer can be one of

a) an *embedded* sa.engine system in the same process as the application,
b) an sa.engine *stream server*, *SAS*, coordinating communication with other peers,
c) an sa.engine *edge client*, *EC*, running on an edge device registered in a SAS, or
d) a *nameserver*, which is a SAS that keeps track of all peers in a federation of sa.engine peers.

Local connections

A particular peer is the embedded sa.engine system. This is called a *local connection* between the application program and the embedded sa.engine. Many client threads can concurrently access the embedded sa.engine (Sec. 5); such local connections are thread safe. The easiest way to get started is to use the local connection.

Before using the local connection, an embedded sa.engine must be initialized by:

```
sa_engine_init(int argc, char** argv);
```

The function initializes an embedded sa.engine system that runs in the same process and memory address space as the client application. An embedded sa.engine with default settings is initialized by calling:

```
sa_engine_init(1, NULL);
```

The arguments `argc` and `argv` are working like command line parameters for initializing the embedded sa.engine. In an OS console window you get a list of available command line parameters by running:

```
sa.engine -h
```

Remote connections

The client may also run as a client connected to a remote sa.engine peer running on the same or some other computer. This is called using a *remote connection* from the application to the peer. With the remote connection several applications (and application threads) running in different locations can remotely access the same peer concurrently. The applications and the SAS run as separate programs so that the server will survive client crashes and vice versa.

A federation of sa.engine peers coordinated by a special peer called the *nameserver* can be set up. Before a new peer can be started a nameserver must have been previously started. It can be started in an OS console window of your computer with the command:

```
sa.engine -N
```

To start a sa.engine SAS named `mysas` on the current computer you can issue the console command:

```
sa.engine -S mysas
```

To close down the nameserver and all peers issue the command:

```
sa.engine -K
```

1.2. Object handles

All access to objects inside sa.engine from C is made through *object handles* which are indirect identifiers for physical data structures, called *storage objects*. Object handles in general are declared as C type `ohandle` in the header file `sa_client.h`.

Notice that object handles *must always* be initialized to `nil`, declared like this:

```
ohandle myhandle = nil;
```

There are specialized object handle C data types for different kinds of objects such as object streams, tuples, or connections, for example:

```
sa_stream mystream = nil;
sa_tuple mytuple = nil;
sa_connection myconnection = nil;
```

In order to make the application code both fast and independent of the internal representation of object handles, the object handles are always manipulated through a set of C macros and functions. The interface is connected to an automatic garbage collector inside sa.engine so that data no longer used is reclaimed when using those macros/functions.

Storage objects have an associated data type called a *storage type* represented an integer. The storage type tag of object handle `h` can be accessed with the function:

```
int sa_typetag(ohandle h);
```

For a given storage type tag `tt`, the name of the corresponding storage type is retrieved by:

```
char *sa_storagetype(int tt);
```

The following storage types are predefined as macros:

Storage type name	Type tag	Represents
INTEGER	INTEGERTYPE	Integers
REAL	REALTYPE	64-bit real numbers
STRING	STRINGTYPE	Strings
ARRAY	ARRAYTYPE	1D arrays of object handles
RECORD	RECORDTYPE	Records of key/value pairs
BINARY	BINARYTYPE	Binary areas (buffers)
OID	SURROGATETYPE	OSQL objects

1.3. Connections

Before an application program can call sa.engine using the client interface it has to establish a *connection* to a *peer* running the system.

The connection itself is accessed through an object handle declared as:

```
sa_connection c = nil;
```

A new connection `c` to an sa.engine system is created by:

```
int sa_connect(sa_connection *c, const char *peer)
```

The connection object `c` holds the necessary information for calling sa.engine primitives and exchange data between the application and the connected sa.engine system. `sa_connect()` returns 0 if the connection was established OK; otherwise it returns an error number. If an error occurred it can be printed on standard output by calling `sa_print_error()`.

The `peer` identifies which sa.engine system to connect to. It is specified as a string. The format of the string `peer` is one of:

```
""
"peer"
"peer@host"
"peer@host:portno"
```

The empty string "" establishes a local connection to the embedded sa.engine.

Non-empty `peer` strings established a remote connection to an sa.engine peer running as a separate process on the same or some other computer reachable through the local computer network.

If just `"peer"` is specified it is the name of a local edge client or SAS known by the nameserver running on the same computer as the client. The local nameserver can be reached using the peer name `"nameserver"`.

If `"peer@host"` is specified a connection is established to a peer managed by the sa.engine name server of the specified `host`. Specifying `peer@localhost` is equivalent to just `peer`.

The nameserver by default listens on port 35021; the format `"peer@host:portno"` is used when the nameserver on that host uses some other port. Under Windows, Linux and OSX the OS environment variables `NAMESERVERHOST` and `NAMSERVERPORT` can be set to `host` and `portno`, respectively, before the system is started.

2. Executing OSQL queries

To execute OSQL commands to the embedded sa.engine system, call:

```
int sa_command(const char *stmt);
```

It executes the OSQL statement `stmt` and ignores the result. The result is 0 if the command was successful.

The function

```
int sa_query(sa_stream *s, sa_connection c, const char *q);
```

executes the OSQL query `q` in the connection `c` returning the result as the object stream handle `s` declared as:

```
sa_stream s = nil;
```

The result of `sa_query()` is 0 if the call succeeded and an error number otherwise. The object stream `s` represents the future result of the query, which can be retrieved in two ways either by calling a callback C function for each result tuple from the call (Sec. 2.1) or by running the query q to completion by using the run API (Sec. 2.2)

2.1. The callback API to object streams

In the *callback API* the application program passes a *callback function* `cb` to the *mapper* function:

```
int sa_map(sa_stream s, sa_callback sb, void *xa)
```

The callback function `cb` must have the signature:

```
int cb(sa_tuple tpl, void *xa)
```

The callback function is called for each element tuple `tpl` of the object stream `s` as they arrive, with `xa` passed unchanged from `sa_map()` to enable passing state between the mapper and the callback function. The callback function is executed in the same thread as `sa_map()`. For example, the following program in `sa.engine/demo/client/C/QueryRange.c` prints the natural numbers 1,2 and 3 by executing the query "`range(1,3)`" in the embedded sa.engine system:

```
#include "sa_client.h"
int printint(sa_tuple res, void *xa)
{
  long i;
  if(sa_getlongelem(&i, res, 0))
  {
    sa_print_error();
    return FALSE;
  }
  printf("%d\n", i);
  return TRUE;
}
int main(int argc, char **argv)
{
  sa_connection c = nil;
  sa_stream s = nil;
  sa_engine_init(1, NULL);
  if(sa_connect(&c, "")
     || sa_query(&s, c, "range(1,3)")
     || sa_map(s, printint, NULL)) sa_print_error();
  sa_free(&c, &s, NULL);
  return 0;
}
```

First in the main program the handles `c` and `s` are declared and initialized. Then the embedded sa.engine is started with default parameters and connected through the local connection `c`. After that the query is issued returning the object stream `s`.

The callback function `printint()` is then applied on each element tuple in `s`. In `printint()` the function

```
int sa_getlongelem(long *i, sa_tuple tpl, int pos);
```

retrieves in variable `i` the integer element in position `pos` (enumerated from 0) of each tuple `tpl` in the object stream `s`.

If `sa_getlongelem()` fails, it returns an error number. A standard error message for the latest error in the current thread is printed by calling `sa_print_error()`.

If the callback function returns `TRUE` the mapping will continue; if `FALSE` is returned it will be terminated.

Finally, the call to the function

```
sa_free(ohandle *h…)
```

in the main program instructs the garbage collection that the application does not need to hold references the object handles `c` and `s`. The garbage collector will free the objects if no other application or system component hold references to them.

2.2. The run API to object streams

A common case is that the object stream is immediate processed over of all its elements. This is called the *run API*. With the run API the client application calls the function:

```
int sa_run(sa_tuple *res, sa_stream s);
```

The function `sa_run()` runs the entire object stream to finish. The function returns the *last* element of the completely processed object stream `s`. `sa_run()` will thus loop forever if `s` is an infinite object stream. By returning the last element `sa_run()` will not use more memory than occupied by a single object stream element, which makes it scale over very long object streams. The run API is very practical for queries returning single values (i.e. a finite object stream containing one object) such as `sqrt(2)`, in which case the single element of the call or query is returned from `sa_run()`, for example in `sa.engine/demo/client/QuerySqrt.c`:

```
#include "sa_client.h"
int main(int argc, char **argv)
{
  double res;
  sa_connection c = nil;
  sa_stream s = nil;
  sa_tuple restpl = nil;
  sa_engine_init(1, NULL);
  if(sa_connect(&c, "")
     || sa_query(&s, c, "sqrt(2)")
     || sa_run(&restpl, s)
     || sa_getdoubleelem(&res, restpl, 0)) sa_print_error();
  else printf("The square root of 2 is: %g\n", res);
  sa_free(&c, &s, &restpl, NULL);
  return 0;
}
```

Notice that not only pure queries can be passed to `sa_query()` but any OSQL statement. The run API is recommended for immediately executing OSQL statements with side effects.

For the common case that a query `q` is run immediately, as above, there is a combined call:

```
int sa_runquery(sa_tuple *res, sa_connection c, const char *q);
```

3. Calling OSQL functions

The time to dynamically compile and optimize a query by sa.engine can be rather long, so a better way is to directly call OSQL functions using the function:

```
int sa_call(sa_stream *s, sa_connection c, const char *fn, sa_tuple args);
```

Here `fn` is the name of an OSQL function to call and `args` are the actual arguments in the call represented as a tuple. The following code in file `sa.engine/demo/client/C/CallRange.c` calls the OSQL function `range(1,3)`:

```
#include "sa_client.h"
int printint(sa_tuple resl, void *xa)
{
  long i;
  if(sa_getlongelem(&i, resl, 0)) {
    sa_print_error();
    return FALSE;
  }
  printf("%d\n", (int)i);
  return TRUE;
}
int main(int argc, char **argv)
{
  sa_connection c = nil;
  sa_stream s = nil;
  sa_tuple argl = nil;
  sa_engine_init(1, NULL);
  if(sa_connect(&c, "")
     || sa_maketuple(&argl, 2)
     || sa_setlongelem(argl, 0, 1)
     || sa_setlongelem(argl, 1, 3)
     || sa_call(&s, c, "range", argl)
     || sa_map(s, printint, NULL)) sa_print_error();
  sa_free(&c, &s, &argl, NULL);
  return 0;
}
```

Tuples are used for holding arguments (such as `argl` in the main program) and results (such as `resl` in the callback function `printint()`) of OSQL function calls.

In the example a new tuple is created by calling the function:

```
int sa_maketuple(sa_tuple *tpl, int sz);
```

It creates a new tuple object `tpl` of size `sz`.

The function

```
int sa_setlongelem(sa_tuple tpl, int pos, long val);
```

sets element `pos` in tuple `tpl` to the integer `val`.

For the common case that a called function `fn` is run immediately there is a combined call:

```
int sa_runcall(sa_tuple *res, sa_connection c, const char *fn,
               sa_tuple argl);
```

3.1. Type resolution

OSQL functions can be *overloaded*, meaning that they have several different function definitions, called *resolvents*, depending on the types of their argument. The resolvents have internal names assigned by the system. In OSQL you can retrieve the resolvents of any OSQL function *fn* by calling the system function *resolvents(Function fn) -> Bag of Function*. For example, the following query retrieves the names of the resolvents of the generic function `plus` (implementing the infix operator +):

```
name(resolvents(thefunction("plus")))
```

You can retrieve their *signatures*, i.e. their names and types of arguments and results, with:

```
signature(resolvents(thefunction("plus")))
```

When an OSQL function is called from C using the generic name, the system has to retrieve the applicable resolvent for the given generic function and the arguments in the call. This is called *type resolution*, and causes some overhead. The overhead can be avoided by specifying the full internal name of the applicable resolvent as `fn` in `sa_call()` or `sa_runcall()` above. The following OSQL query retrieves the correct resolvent when calling the generic function named `fn` with the arguments $a_1, \ldots, a_n$:

```
name(resolve_call(fn,[a1,…,an]))
```

For example:

```
name(resolve_call("plus",[1,2]))
⇨  "NUMBER.NUMBER.PLUS->NUMBER"
signature(resolve_call("plus",[1,2]))
⇨  "plus(Number x,Number y)->Number r"
```

So, if you know that your arguments to `plus` in your call to `sa_call()` or `sa_runcall()` are always going to be numbers, you can use the string `"NUMBER.NUMBER.PLUS->NUMBER"` rather than just `"plus"` to speed up the call.

4. Error handling

The following functions are all thread safe and can be called at any time in the program or even in C interrupt handlers. They are all based on either inspecting error codes and messages. In order to signal that errors have happened the programmer can set these codes and messages, e.g. in signal handlers, without involving any exception handling or OS system calls. The raised error codes can then be inspected by the application program. The error information is maintained in the current thread for multi-threaded applications and globally for single-threaded applications.

By contrast, when extending the system kernel by foreign function in plugins [1], the programmer has access to the exception handling of the sa.engine kernel and can throw raised errors.

The following function returns an identifier of latest error that occurred:

```
int sa_errno(void);
```

The error number may vary between different configurations of sa.engine. If the error number is -1 it indicates that the error message is dynamic and specific only for the latest error. If 0 is returned there is no error currently raised.

The following function returns the error message string for the latest error:

```
char *sa_errstr(void)
```

The error message for error number `no` can be retrieved by:

```
char *sa_strerror(int no);
```

The function accesses a small in-memory *error table* associating system error numbers with error message strings and returns the error message associated with the error code `no`. If `no==-1` the latest dynamic error message is returned.

An object handle associated with the latest error can be retrieved by:

```
ohandle sa_errobject(void);
```

To indicate that an error has happened you can call one of the functions:

```
int sa_raise_errormsg(const char *msg, ohandle obj);
int sa_raise_errorno(int no, ohandle obj);
```

The function `sa_raise_errormsg()` looks up the error message table for the error number of `msg`, which is then returned if found. If no matching error number is found -1 is returned and `msg` becomes the latest dynamic error message. Error messages are truncated to max 100 bytes.

If a call to an sa.engine interface function was not successful a standard error message for the latest error in the current C thread can be printed on standard output by calling `sa_print_error()`. The function returns the error number of the error or 0 if no error has occurred. The function is thread safe but will lock the system while executing.

5. ulti-threaded clients

A client program is allowed to call sa.engine from multiple threads managed by the client program. All client interface primitives presented in this chapter can be used for calling either the embedded sa.engine system or remote sa.engine servers. The system is thread safe, meaning that it thereby guarantees that two client threads can run in parallel without crashing sa.engine.

The client interface for multi-threaded applications is defined by the header file:

```
#include "sa_threads.h"
```

Before the current thread can call sa.engine it must first be registered by calling:

```
int sa_thread_initialize(void)
```

The function registers the current thread with sa.engine and allocates some data structures to hold the thread's state.

When the thread is terminated, the resources held by thread should be released by calling:

```
int sa_thread_finalize(void)
```

Notice that the management of the threads is not made by sa.engine; it is in the hand of the OS and the client application. If the client application decides to terminate a thread it can always do so, but it responsible for thereby calling `sa_thread_finalize()` to inform sa.engine that the thread is terminated.

A stream `s` running in some thread can be terminated by calling:

```
int sa_terminate(sa_stream s)
```

The program in `demo/client/C/CAPI.c` gives an example of a multi-threaded client application program. Try it out for three threads connected to the embedded sa.engine by compiling it and calling it with:

```
CAPI "" 3
```

6. Data objects

The client API uses a number of C functions and types documented in this section.

Notice that for debugging the object referenced by handle `h`, it can be printed on standard output by calling:

```
int sa_print(ohandle h);
```

6.1. Tuples

Tuples are represented as storage objects tagged `ARRAYTYPE` (Sec. 1.2). They are used for representing object stream elements (Sec. 2). Tuples are also used for representing argument lists in sa.engine function calls from applications (Sec. 3) as well as 1D arrays (OSQL type *Vector*) of objects (Sec. 6.6).

A handle to a new tuple `tpl` with size `sz` is created with the function:

```
int sa_maketuple(sa_tuple *tpl, int sz);
```

The get the width of a tuple `h`, call the function:

```
int sa_size(ohandle h);
```

The elements of a tuple are enumerated starting at 0 and can be accessed through a number of *tuple access functions* specific for each element type, as described next.

6.2. Integers

To access an integer `res` stored in position `pos` of tuple `tpl`, call the function:

```
int sa_getlongelem(long *res, sa_tuple tpl, int pos);
```

If the element of the tuple is a floating point number it is rounded to the closest integer. An error is generated if there is no number in the specified position of the tuple.

To store integer `val` in element `pos` of tuple `tpl`, call the function:

```
int sa_setlongelem(sa_tuple tpl, int pos, long val);
```

The system also supports double length (64 bits) integers declared by the macro `LONGINT`. To access a 64-bits integer `res` stored in position `pos` of tuple `tpl`, call:

```
int sa_getintelem(LONGINT *res, sa_tuple tpl, int pos);
```

To store the 64-bits integer `val` in element `pos` of tuple `tpl`, call:

```
int sa_setintelem(sa_tuple tpl, int pos, LONGINT val);
```

6.3. Floating point numbers

To get a double precision floating point number `res` stored in position `pos` of a tuple `tpl`, call:

```
sa_getdoubleelem(double *res, sa_tuple tpl, int pos);
```

To store the floating point number `val` in element `pos` of tuple `tpl`, call:

```
sa_setdoubleelem(sa_tuple tpl, int pos, double val);
```

Integers are converted to floating point numbers. An error is generated if there is no number in the specified position of the tuple.

6.4. Strings

To copy a string stored in position `pos` of a tuple `tpl` into a buffer `buff` of size `buffsize`, call:

```
int sa_getstringelem(char *buff, size_t buffsize, sa_tuple tpl, int pos);
```

To obtain the length `l` of a string in element `pos` of tuple `tpl`, call:

```
int sa_elemsize(size_t *l, sa_tuple tpl, int pos);
```

To store a string `str` in element `pos` of tuple `tpl`, call:

```
int sa_setstringelem(sa_tuple tpl, int pos, const char *str);
```

6.5. Generic objects

To get a handle to arbitrary object (Sec. 1.2) stored in position `pos` of tuple `tpl`, call:

```
int sa_getelem(ohandle *res, sa_tuple tpl, int pos);
```

To store the object `val` in element `pos` of tuple `tpl`, call:

```
int sa_setelem(sa_tuple tpl, int pos, ohandle val);
```

To assign handle `lhs` to another handle `rhs`, call:

```
void sa_assign(ohandle *lhs, ohandle rhs);
```

6.6. Vectors

As tuples, OSQL vectors (1D arrays of objects) are also represented as storage type objects tagged `ARRAYTYPE` (Sec. 1.2).

The functions `sa_getelem()` and `sa_setelem()` (Sec. 6.5) can be used for accessing and storing vectors in tuples. The same functions can be used for accessing elements in accessed vectors.

The size of the vector is obtained by `sa_size()` (Sec. **Error! Reference source not found.**).

6.7. Records

OSQL records (type `Record`) are represented as storage type objects having the type tag `recordtype` with handles declared in C as `sa_record`.

To access a record `res` stored in position `pos` of tuple `tpl`, use `sa_getelem()` (Sec 6.5)

```
int sa_getelem(sa_record *res, sa_tuple tpl, int pos);
```

To store a record `h` in position `pos` of tuple `tpl` use `sa_setelem()` (Sec. 6.5).

To access the object `res` stored under key `k` in record `r`, call:

```
int sa_getrecord(ohandle *res, sa_record r, const char *k);
```

`res` is set to `nil` if there is no such object in record `r`.

To store object `val` in record `r` under key `k`, call:

```
int sa_putrecord(sa_record r, const char *k, ohandle val);
```

6.8. Binary areas

Binary areas (buffers) are represented in sa.engine as objects of type `Binary` having type tag `BINARYTYPE` with handles declared in C as `sa_binary`.

To copy a binary area stored in position `pos` of a tuple `tpl` into a buffer `buff` of size `buffsize`, call:

```
int sa_getbinaryelem(void *buff, size_t buffsize, size_t *len,
                     sa_tuple tpl, int pos);
```

The parameter `len` is set to the actual length of the fetched binary object. If the area is larger than `buffsize` it is truncated.

To obtain the length `l` of a binary area in element `pos` of tuple `tpl`, call:

```
int sa_elemsize(size_t *l, sa_tuple tpl, int pos);
```

To store a binary area `buff` of size `buffsize` in element `pos` of tuple `tpl`, call:

```
int sa_setbinaryelem(sa_tuple tpl, int pos, const void *buffer, size_t buffersize);
```

7. References

[1] Calling C plugins from sa.engine, Version 2.0, Stream Analyze Sweden AB, *sa_C_pluginAPI_2.0.pdf*.

[2] *sa.engine Java Interfaces*, Version 2.2, Stream Analyze Sweden AB, *sa_JavaAPI_2.2.pdf*, 2020,

[3] sa.engine Lisp Interfaces, Version 2.1, Stream Analyze Sweden AB, *sa_LispAPI_2.1.pdf*, 2020.

[4] sa.engine Overview, Version 2.0, Stream Analyze Sweden AB, *sa_System_2.0.pdf*.