

# sa.engine Overview

Stream Analyze Sweden AB

Version 2.0

2022-01-14

sa\_system\_2.0.pdf

This document is an extended whitepaper for the sa.engine system. It contains general descriptions of the system's features and architecture. See also the on-line whitepaper in <https://streamanalyze.com/under-the-hood/>.

## 1. Features

**sa.engine** is a platform and extensible toolbox by [Stream Analyze Sweden](#) for easy and interactive development and deployment of applications that process and analyze real-time streams of data in distributed and mobile environments. Streams that are produced by sensors on mobile or edge devices can be analyzed on-line interactively. An edge device can be, e.g., an Android unit, a desktop computer, a Raspberry Pi, or MCUs like MangOH Red or ARM M4. sa.engine has a very small footprint (from 100kB to 5MB depending on configuration) and is hardware and OS agnostic and fully independent of any third party software.

The combination in sa.engine of a main-memory database, a computational engine, a data stream processor, and an inference engine implemented on edge devices allows **edge analytics** directly on the edge devices rather than the contemporary approach of first uploading all data from devices to a server and then do the analytics centrally on a server or in the cloud. The approach allows for drastic data reduction and scaleout by defining and running directly on the devices analysis models that reduce the data streams so that only population analyses and management of collections of devices need to be made centrally.

On server computers, in order to collect data from edges and combine and forward the data to other server-based systems and users, sa.engine systems can also be configured as **stream servers**. For example, whenever the analysis model in some edge device detects strong vibrations by doing a computation over the latest readings from its accelerometer, a data stream containing the frequency spectrum of the vibrations along with the geographical position of the device is transmitted to a stream server. If the stream server receives many such streams at the same time from devices in geographical positions close to each other, it may indicate an earthquake. The stream server may furthermore forward the received data to other systems, e.g. for permanent central storage or batch analysis.

To analyze streaming data interactively on a very high and user-oriented level, sa.engine allows analysts and engineers to develop computations, filters and transformations over data using a query language called **OSQL** (Object Stream Query Language). With OSQL computations and filters over real-time streaming data are defined as mathematical formulas and expressions, called **stream models**. A stream model is a set of definitions of mathematical functions, filters, and other expressions over a stream of measurements. With

OSQL the models can be specified on a very high level without deep programming knowledge; you specify **what** to do rather than writing detailed programs expressing **how** to execute the models. The user needs not worry about details on how to efficiently implement programs that integrate large numbers of edge devices and servers.

An OSQL query that continuously makes computations or filtering over measurements in a data stream is called a **continuous query, CQ**. The system allows analysts to interactively specify CQs for continuously analyzing measurements flowing through edge devices and stream servers in real-time. The result of a CQ is a real-time **object stream** of processed and filtered measurements, for example a CQ returning a stream of the position vectors of a device measured every second when it is close to a given geo-position.

Both stream models and user data are stored in very fast object-oriented **in-memory databases** residing on the edge devices and in the stream servers. These databases are important for data stream processing, which usually involves matching in real-time fast flowing stream objects against data in a database. For example, to locally determine that the frequency spectrum of a measured vibration may later destroy an edge device, the frequencies measured by vibration sensors on a device is matched against the local database of resonance frequencies in the device.

In order to combine object streams from several edges, sa.engine supports **fusion queries** that combine object streams. An example of a fusion query is a CQ observing when several edge devices in an area detect strong vibrations at the same time. The user is alerted when the fusion query produces results, perhaps together with a visualization in real-time of the magnitude of the observed vibrations. A user can then interactively send new queries on-the-fly to affected devices to find out details of their causes.

Fusion queries require the integrated data streams to be comparable even though the involved object streams from different devices may represent the same or similar data in different ways. For example, one device may represent temperature in Fahrenheit while another one uses Celsius. To be able to combine such **heterogeneous** data streams from different devices, sa.engine allows **mediator** models to be defined as queries and functions that harmonize arriving heterogeneous object streams by transforming them to a universal model, called an **ontology**, in stream servers that integrate data streams from different edges. Mediation can be, e.g., mapping local names of sensors to a universally known nomenclature, measurement unit conversions, or calibrations of local measurements.

The system includes a library of over 1000 predefined OSQL-functions for math/stat computations, object stream filtering and transformation, signal processing, model and data management, and much more. The function library is continuously extended for new customer needs and it is easy to define and deploy new user functions on-the-fly.

Existing algorithms and libraries implemented in a regular programming language, such as C or Java, can be plugged into the system as **foreign functions** that implement OSQL functions in some external programming language using language specific APIs. The foreign functions can then be transparently used in OSQL queries and expressions. For example, a large library of basic mathematical, statistical, and machine learning algorithms are implemented as foreign OSQL functions in C and Lisp. New foreign functions can easily be developed.

In order to access external data streams, sa.engine provides **wrapper** functionality, which is APIs that enable processing incoming data stream objects as they arrive in order to inject them into the sa.engine kernel so that the accessed stream can be used in CQs expressed in OSQL. The wrappers are defined as OSQL functions that return object streams from wrapped data sources. There is a library of predefined wrappers to interoperate with common data infrastructures such as relational databases through [JDBC](#) and data processing systems through [Kafka](#), [Azure IoT Hub](#), or [MQTT](#). New wrappers can easily be developed.

Common **machine learning** algorithms such as [DBSCAN](#), [DenStream](#), [k-NN](#), and [k-means](#) are available as predefined OSQL models and the user can easily extend this with other algorithms defined in OSQL or as foreign functions. Learning and inference are supported on both edges and servers.

Machine learning requires **pre-processing** of sensor data before applying the learned inference algorithm, followed by **post-processing** of the inferred knowledge. With sa.engine both pre and post-processing are easily expressed using the powerful object stream filtering, transformation and math/stat analytics capabilities of OSQL.

The system includes a **deep learning** subsystem, SANN, where neural network models defined by Tensorflow/Tensorboard can be automatically translated into an internal binary SANN representation and pushed out to the edge databases. Once stored in an edge database the SANN inference engine can analyze local sensor readings to detect anomalies.

SANN furthermore allows to continue training and even building neural networks on edge devices. Thus, centrally trained models can be further retrained and modified on edge devices to adapt their behavior to their environment.

## 2. Scaling the system

A unique property of the sa.engine system is that it is independent of other systems, while at the same time providing powerful extensibility mechanisms to enable tight integration with other software on different hardware platforms. These properties have enabled to port the system to a large variety of hardware and software environments. The agnosticism has enabled to **scale down** the system to run on small edge devices. The number of small edge devices can be very large by running large numbers of scaled down sa.engine systems. The architecture provides a mechanism to keep track of all running sa.engine systems. For this, the system can **scale out** to run in many copies on large multi-cores, clusters and clouds. It is always the same sa.engine kernel software running in all these system configurations. In this chapter the scale-out architecture is first overviewed before describing the details of the kernel system architecture.

### 2.1 Scaling down

The configuration of the sa.engine kernel system can be scaled down to run directly on small devices with limited or no OS support. The system is agnostic to the hardware, OS, and communication infrastructure used. The system can even run completely **stand-alone** on a device or computer without communicating with other systems. In its most basic **Microkernel** version it requires no OS; it runs on **bare metal** (Figure 1). The smallest

configuration of the Microkernel requires less than 100K RAM and 200K DRAM.

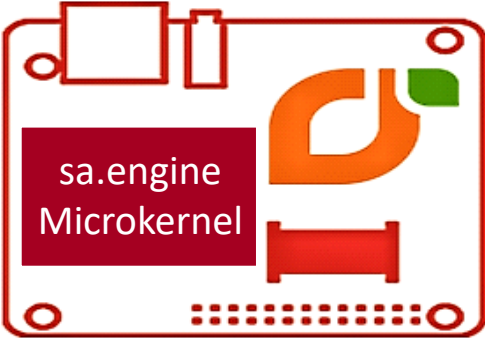


Figure 1: Stand-alone sa.engine Microkernel on device

In order to tightly integrate the system with other software running on devices, the system can also run as an embedded engine inside another **embedding system** (Figure 2).

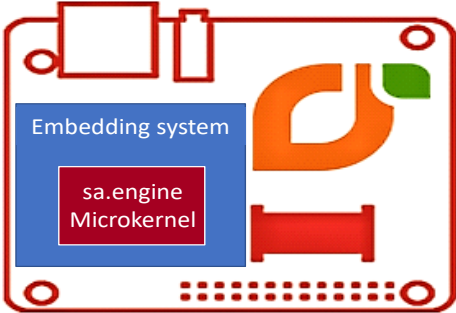


Figure 2: Microkernel embedded in other software on device

## 2.2 Scaling out

In order to manage massive amounts of scaled down sa.engine systems on IoT devices, the system can be scaled out to large distributed systems of communicating SA servers that can run on clusters, clouds, PCs or virtually any kind of computer. Figure 3 illustrates how edge devices with sa.engines on-board, called **edge engines (ECs)**, are managed by **SA servers (SASes)** running in a cloud.

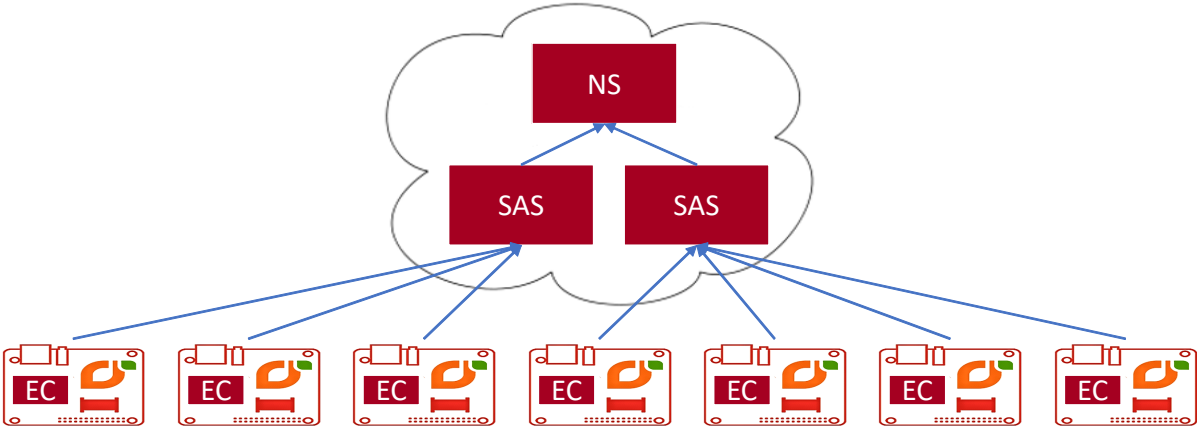


Figure 3: Edge clients connected to SASes in cloud

Depending on the size and configuration of an edge device, an edge client can be an sa.engine microkernel, an embedded microkernel, and even complete sa.engine system. The tiny configuration in Figure 3 has seven edge clients. The unique identity of each EC needs to be registered with an SAS running on some server in a cloud, container, PC, or as a separate process on the same computer as the EC. In Figure 3 there are two SASes running in a cloud where the seven ECs are registered. There is a special SAS keeping track of all the unique identities of each SAS called the **nameserver**, *NS*. Through the NS and the SASes all ECs can be reached.

Different instances of the sa.engine kernel running on different computers can communicate with SASes running on the same or some other computer. In particular each EC communicates with the SAS where it is registered. The SAS keeps some meta-data about its ECs and can forward data and streams to other SASes.

It should be noticed that an edge client is not required to be continuously connected with its SAS, it only needs to be registered with it. The actual connection to the device and generation of object streams is started and finished only when so required. If there is no active communication with its SAS the edge client runs autonomously.

The overall term sa.engine **peer** is used to denote any kernel system running either as an NS, SAS or EC. The universe of all peers is called a **federation**. In Figure 3 there is a federation of ten peers.

In general, a peer can be one of the following

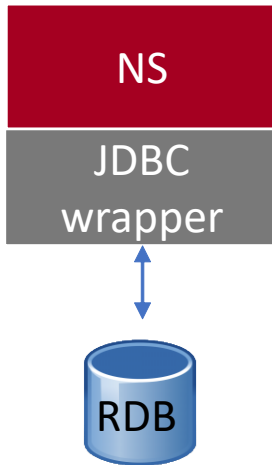
1. It can be an **embedded engine** running on some device or computer.
2. It can be an **edge client**, *EC*, running on an edge device registered in a SAS.
3. It can be an **SA server**, *SAS*, that coordinates communication with other peers.
4. It can also be a **nameserver**, *NS*, which is a SAS that keeps track of all peers in a federation of sa.engine peers. The nameserver usually wraps a regular database where meta-data about the peers is persisted.

In practice there can be a massive number of devices accessible through edge clients. It is therefore necessary to be able to scale out the numbers of peers to handle extremely large numbers of edge clients, from tens of thousands up to billions. This is handled by scaling out the number of edge clients registered in each SAS to thousands of edge clients and defining a hierarchy of several SAS levels.

The federation of peers must persist even when systems go down. This is achieved by connecting the nameserver to a wrapped **relational database**<sup>1</sup>, *RDB*, as illustrated by Figure 4.

---

<sup>1</sup> The database system need not be a relational database but can be any kind of scalable database system which satisfactory update capacity.



*Figure 4: Accessing a relational database from the nameserver*

The **JDBC wrapper** is a general plug-in (Section 3.1) to sa.engine that enables access to any relational database using the standard [JDBC](#) API. In this case the relational database holds the identifiers of all peers in the federation, along with other meta-data such as what kind of equipment is accessible through each edge client, what kind of sensors they access, etc.

An NS without a relational database back-end can also persist its database but it will not scale to millions of devices. Custom meta-data repositories other than relational databases holding information of the edge devices can easily be defined as wrappers (Section 3.1) allowing transparent access through the NS.

The SASes and the NS can run on many different kinds of hardware and software configurations. In the simplest case they all run on a regular PC. In more scalable configurations each SAS can run in a separate container (e.g. Docker) and the NS along with the relational database on a dedicated cluster. With such a scale-out, if the number of clients registered with each peer is 1000 and with two levels of SASes up to  $10^9$  peers can be handled. The relational database is not a bottleneck here as a limited amount of meta-data per peer will be stored there.

### 3. The core system

The sa.engine core system provides general data analytics and inference capabilities to the device or computer where it is running. The core system is written in C99 and is not depending on any other system. It is agnostic to hardware, OS, and communication infrastructure used by the device or the computer. This has enabled porting it to many different kinds of devices, computers and OSes.

The system is designed to be easily integrated with other systems and extended with models and plug-ins. Figure 5 illustrates how the sa.engine core system is extensible by user models calling interfaces to external systems and accessing different kinds of data sources.

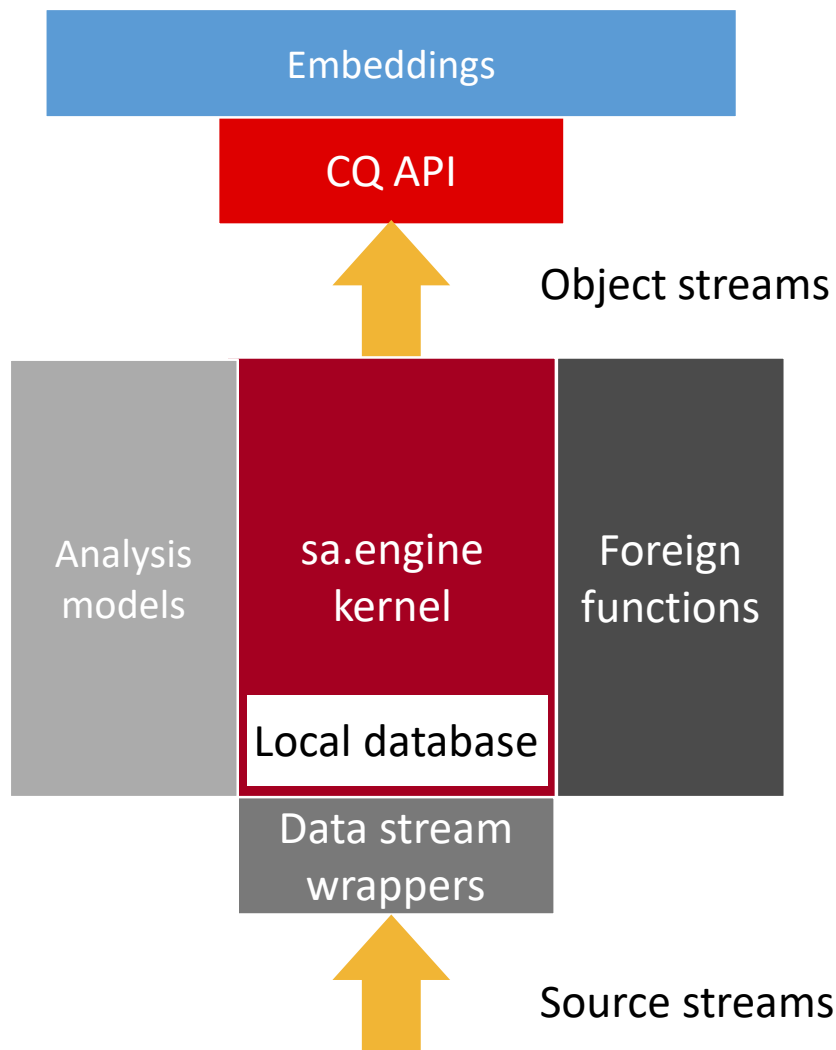


Figure 5: The extensible sa.engine kernel

The main part of the system, the **sa.engine kernel**, is marked red. It provides the generic capabilities needed for real-time data stream analytics. It contains a computational engine, a data stream processor, and an inference engine. An important component of the sa.engine kernel is a main memory **local database** where, e.g., analysis models and device meta-data are stored. The sa.engine kernel includes a powerful query processor and optimizer to search and analyze data and models in the local database.

The yellow arrows indicate streams. Sensors and software produce inflowing **source streams** that are injected into the sa.engine system, which transforms them into one or several outflowing **object streams**.

A source stream may be implemented as accesses to a raw sensor interface on the computer or edge device where the sa.engine kernel is running. A source stream can also be a data stream produced by software. In distributed configurations the source stream may be received from another computer or device than the kernel, in which case some communication infrastructure (e.g. TCP) is used.

Analogously, the outflowing object streams may be sent to other systems and application using some communication infrastructure. Large distributed systems of sa.engine instances

can be configured in which sa.engine peers produce object streams consumed by other sa.engine peers as source streams. On the edge devices the object stream elements can also be sent directly to actuators mounted on the device.

### 3.1 Plug-ins

sa.engine is a highly extensible system where different kinds of plug-ins can be added without changing other parts of the system. In Figure 5 the plug-ins are marked in different shades of gray, where darker shades indicate more tightly and system-oriented plug-in implementations.

**Analysis models** are models that specify transformations, filters, computations and inferences over the source streams to produce object streams as results. The analysis models are specified by engineers and analysts. It is not required to have deep programming skills or detailed knowledge about the sa.engine kernel to define such models. The analysis models are defined using the very powerful domain oriented language OSQL as a glue to combine different kinds of models, algorithms, and engines. An analysis model is constituted as a set of OSQL functions and CQ definitions stored in the local database.

OSQL can be extended through **foreign functions**, which define OSQL functions implemented in some regular programming language (e.g. C [4], Lisp [6] or Java [5]). The foreign functions implement external algorithms, e.g. numerical, statistical, and inference algorithms as plug-ins. They can be plugged into the kernel without modifying the it. The algorithms can be used in analysis models to filter and transform the incoming data streams into derived object streams.

Foreign functions written in C, C++, and Lisp have full access to the kernel system allowing very powerful addition of capabilities to the kernel, for example to access file systems, making OS system calls, running deep learning inference engines, or accessing complex database managers. For high performance, the extensibility of sa.engine permits the direct representation of binary C data structures as OSQL objects without any transformations. See details in [8].

In order to access incoming data streams in CQs, **data stream wrappers** (Figure 5) can be implemented as OSQL functions. Notice that only one such data stream wrapper needs to be implemented for each *kind* of incoming data stream; once implemented for a certain stream kind all such streams can be queried with OSQL. A data stream wrapper is defined as an OSQL wrapper function returning an object stream. The function usually takes arguments, for example to represent the identity of the stream it wraps. Often the wrapper functions are implemented as foreign functions, but in many cases, they are defined completely in OSQL. For example, standard sensor interfaces are often accessible by a file interface, which is provided out-of-the-box in sa.engine.

Data stream wrappers need to physically access an external data stream in order to convert each of its arriving data stream elements to a suitable data format for efficient and flexible processing by the sa.engine kernel. Different streams often represent their elements using different physical data structures, so the data stream wrappers usually convert the external data representations to a format already supported by the system. When needed, new physical data representations can be plugged in for customized representation of C data structures. The representation of specialized binary C formats can be used for very efficient processing of stream elements.



There is a built-in library of data stream wrappers for common infrastructures, e.g. for Kafka, Azure IoT Hub, MQTT, CSV and JSON streams.

The data streams will be infinite when they originate in a sensor. However, they can also be finite. For example, there is a special JDBC data stream wrapper available that handles the finite result from an SQL query passed as a wrapper function parameter through JDBC to a relational database. This wrapper is used for persisting peer meta-data in the nameserver (Section 2.2).

## 3.2 Embeddings

The sa.engine kernel can be an **embedding** (Figure 5) in other applications or systems that access produced object streams produced through a continuous query **CQ API**. The application program or system can run in the same process and address space as the sa.engine kernel, which is often case when running an embedded sa.engine on an edge device. Another common case is that an sa.engine system running on an edge device acts as a client to an sa.engine server running on some other computer or cluster communicated with using TCP or some other communication infrastructure. There are such interfaces to embeddings defined for common infrastructures such as CSV and JSON streams over TCP, Kafka and Azure EventHub.

### 3.3 Components

Figure 6 illustrates the software components of the core system.

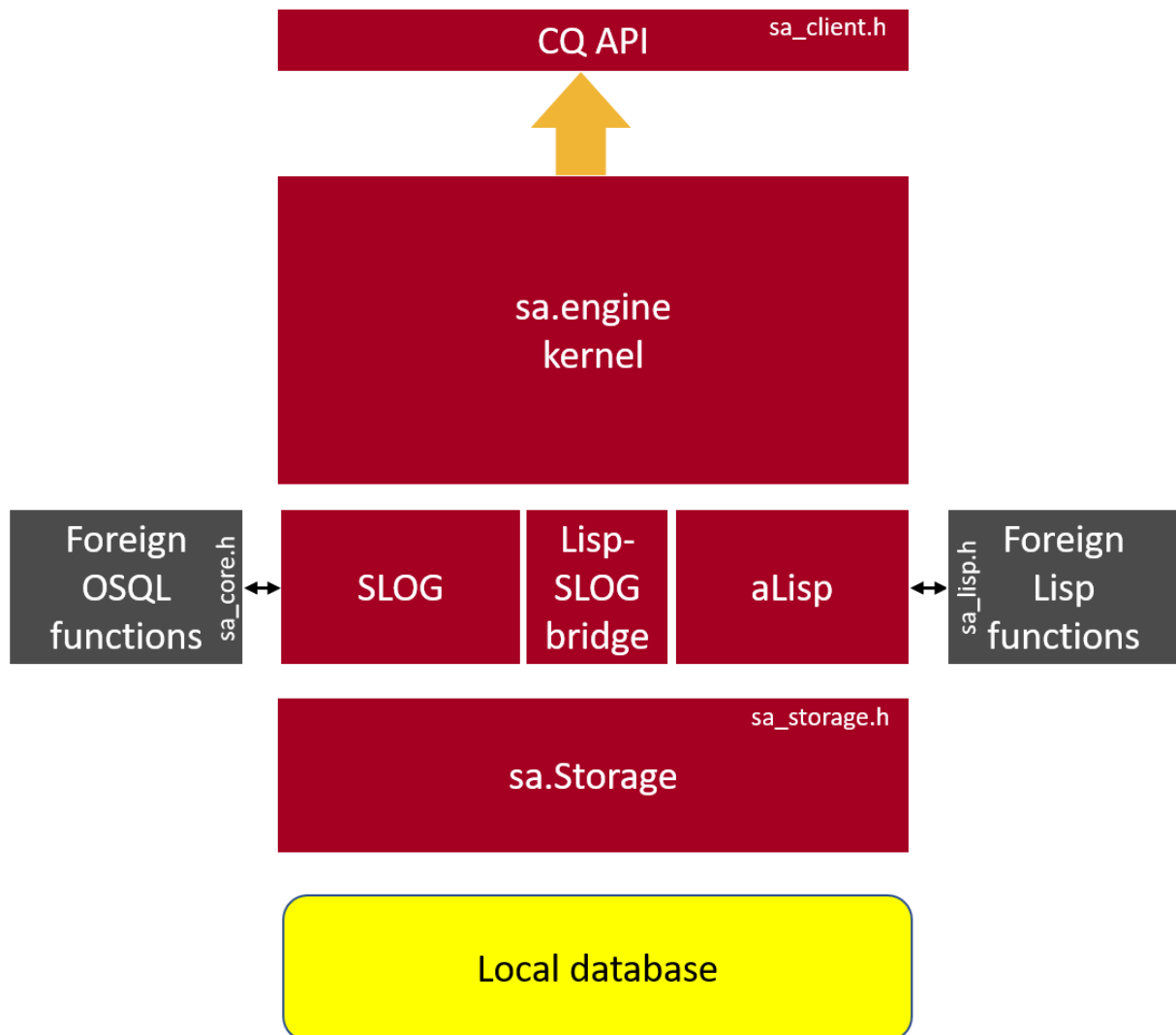


Figure 6: Components of the core system

External applications can access the system through the **CQ API** interface using the C interface *sa\_client.h* [3]. The CQ API is also defined for Lisp [6] and Java [5].

The main memory local database is managed by a storage manager called **sa.Storage** [8], interfaced from C through *sa\_storage.h*. On top of the storage manager there are two interpreters:

- An interpreter for data streams, called **SLOG** (*Streamed LOGic*) processes CQs as new data elements arrive.
- An interpreter for an extended subset of CommonLisp [2] called **aLisp** [1] processes Lisp-code in which large parts of the actual **sa.engine kernel** code is written. It can be accessed from C through *sa\_lisp.h*.

Both SLOG and aLisp can run separately on top of sa.Storage and the **Lisp-SLOG bridge** provides interoperability between them.

Both SLOG and aLisp are extensible by defining **Foreign OSQL functions** (*sa\_core.h* interface) and **Foreign Lisp functions** (*sa\_lisp.h* interface) respectively, in various programming languages [3][4][5][6].

## References

- [1] **aLisp User's Guide**, Version 2.0, Stream Analyze Sweden AB, *sa\_Lisp\_2.0.pdf*, 2020.
- [2] Guy L. Steele Jr.: **Common LISP the language**, Digital Press,  
<http://www.cs.cmu.edu/Groups/AI/html/cltl/clt12.html>
- [3] **Calling sa.engine from C**, Version 2.0, Stream Analyze Sweden, AB,  
*sa\_C\_clientAPI\_2.0.pdf*.
- [4] **Calling C plugins from sa.engine**, Version 2.0, Stream Analyze Sweden AB,  
*sa\_C\_pluginAPI\_2.0.pdf*.
- [5] **sa.engine Java Interfaces**, Version 2.1, Stream Analyze Sweden AB,  
*sa\_JavaAPI\_2.1.pdf*, 2020.
- [6] **sa.engine Lisp Interfaces**, Version 2.1, Stream Analyze Sweden AB,  
*sa\_LispAPI\_2.0.pdf*, 2020.
- [7] **sa.engine Under the hood**, <https://streamanalyze.com/under-the-hood/>
- [8] **sa.storage 2.0 - A main-memory storage manager**, Version 2.0, Stream Analyze Sweden AB, *sa\_Storage\_2.0.pdf*, 2020.