J Intell Inf Syst
DOI 10.1007/s10844-016-0427-2

Scalable validation of industrial equipment using a functional DSMS

Cheng Xu[1] · Elisabeth Källström[2] · Tore Risch[1] · John Lindström[3] · Lars Håkansson[4] · Jonas Larsson[2]

Received: 2 March 2016 / Revised: 5 August 2016 / Accepted: 8 August 2016

Abstract A stream validation system called SVALI is developed in order to continuously validate correct behavior of industrial equipment. A functional data model allows the user to define meta-data, analyses, and queries about the monitored equipment in terms of types and functions. Two different approaches to validate that sensor readings in a data stream indicate correct equipment behavior are supported: with the model-and-validate approach anomalies are detected based on a physical model, while with learn-and-validate anomalies are detected by comparing streaming data with a model of normal behavior learnt during a training period. Both models are expressed on a high level using the functional data model and query language. The experiments show that parallel stream processing enables SVALI to scale very well with respect to system throughput and response time. The paper is

✉ Cheng Xu
cheng.xu@it.uu.se

Elisabeth Källström
elisabeth.kallstrom@volvo.com

Tore Risch
tore.risch@it.uu.se

John Lindström
john.lindstrom@ltu.se

Lars Håkansson
Lars.Hakansson@bth.se

Jonas Larsson
jonas.jl.larsson@volvo.com

[1] Department of Information Technology, Uppsala University, Uppsala, Sweden

[2] Volvo Construction Equipment, Eskilstuna, Sweden

[3] ProcessIT Innovations, Luleå University of Technology, Luleå, Sweden

[4] Department of Applied Signal Processing, Blekinge Institute of Technology, Karlskrona, Sweden

Published online: 18 August 2016

based on a real world application for wheel loader slippage detection at Volvo Construction Equipment implemented in SVALI.

Keywords Data stream management · Distributed stream systems · Data stream validation · Parallelization · Anomaly detection

1 Introduction

Traditional database management systems (DBMSs) store data records persistently and enables execution of queries over the current state of the database on demand. This fits well for business applications such as bank and accounting systems. However, in the last decades, more and more data is generated in real-time, e.g. data from stock markets, real-time traffic control, human internet interactions, sensors installed on machines, etc. Such data continuously generated in real-time is called *data streams*. The rate at which data streams are produced is often very high e.g. megabytes per second, which makes it infeasible to first store streaming data on disk and then query it. Furthermore, business decisions and production systems rely on short response times so the delay caused by first storing the data in a database before querying and analyzing it may be infeasible. For example, monitoring the healthiness of different components in industrial equipment requires the system to return the result within seconds. Data stream management systems (DSMSs), such as AURORA (Abadi et al. 2003), STREAM (Motwani et al. 2002), and SCSQ (Zeitler and Risch 2011), are designed to deal with this kind of applications. Instead of ad-hoc queries over static tables, queries over streams are *continuous queries* (CQs) since they are running until they are explicitly terminated and will produce a result stream as long as they are active.

In order to deliver quality services for industrial equipment it should be continuously monitored to detect and predict failures. As the complexity of the equipment increases, more and more research is conducted to automatically and remotely detect the abnormal behavior of machines (Namburu et al. 2006). Volvo Construction Equipment (Volvo CE) has installed a component called *automatic transmission clutches* to monitor the health of the clutch material of their L90F wheel loaders. Various sensors measuring different signal variables are installed on the L90F machines and data from the sensors are delivered following the CANBUS protocol (Canbus, http://en.wikipedia.org/wiki/CAN_bus), which is an industry standard protocol to communicate with the data buses in engines and other machines. Statistical computations over the data are required in real-time to detect and predict anomalies so that corresponding actions can be taken to reduce the cost of maintenance. Furthermore, when the number of wheel loaders increases it is also important that the processing scales.

The Stream VALIdator (SVALI) system is a DSMS to efficiently validate anomalies of measurements in data streams using CQs, e.g. to monitor correct behavior of equipment such as Volvo CE wheel loaders. Such validation will involve defining as CQs more or less complex mathematical models that identify and predict non-expected behaviors based on streams of measurements from sensors installed in the equipment. The CQs are natural to express as formulas involving functions and variables over numerical entities such as numbers and vectors, i.e. domain calculus, rather than the traditional tuple calculus based relational database model where variables range over rows in tables. To facilitate complex mathematical models over sensored numerical measurements, SVALI provides a *functional data model* where CQs can be expressed as functions over sets, numbers, vectors, and

streams. Variables in SVALI queries can be bound to objects from any domain, i.e. SVALI queries are based on an object-oriented and functional domain calculus. SVALI provides a library of built-in numerical vector and aggregate functions to build the models. To utilize existing numerical libraries, SVALI is extensible by calling in queries *foreign functions* written in regular programming languages such as C, Java, or Python.

Analyzing data streaming from sensors on industrial equipment requires low level interfaces capturing streaming measurements. In SVALI such interfaces can be defined as foreign functions called *data stream wrappers*, which iteratively emit data stream elements into the system. For example, the data stream wrapper for the sensors installed in the Volvo CE wheel loaders is implemented as a C function that iteratively emits tuples of measurements received from the equipment based on the CANBUS protocol.

The contributions of the paper are:

1. It is shown how a functional data model can be used for defining meta-data about industrial equipment of different kinds. Numerical models are defined as functions that determine expected measured values computed from streaming data, based on statistics about the behavior of the monitored equipment. Validation models defined in terms of functional meta-data identify deviations from expected behavior.
2. The monitored equipment is often geographically distributed. For example, Volvo CE's wheel loaders are operating at remote excavations sites in different parts of the world. Therefore SVALI is a *distributed* DSMS where many SVALI peers communicating over TCP/IP can be started up at different sites. Each peer produces reduced streams of non-expected measurements, which are continuously emitted to a central SVALI server where anomalies from many sites are collected, combined, and analyzed.
3. To provide security it is required that the SVALI server at the monitoring center is protected behind a firewall and that all monitored equipment is protected behind firewalls. Therefore the software on-board the equipment connects to a SVALI server as a client to register its data stream source. After the registration the on-board software starts emitting stream elements to the server.
4. It is important that the system scales with the number of monitored machines and sites while validation in real-time can be performed with low delays. To investigate the scalability of the system, many instances of SVALI were run on a multi-core computer where the number of received streams (i.e. number of monitored machines), their stream rates, and the number of CQs were scaled.

The paper is organized as follows: Section 2 gives the motivating application scenario from Volvo CE followed by a detailed description of the SVALI system in Section 3. In Section 4 the anomaly detection algorithm used by Volvo CE is described followed by the corresponding SVALI implementation. Section 5 evaluates the scalability of the SVALI system. Section 6 presents related work and, finally, conclusions and future work are discussed in Section 7.

2 Application scenario

In the construction equipment business breakdown of component parts may result in unnecessary stops in machines, leading to customer dissatisfaction. To avoid unnecessary stops and breakdowns, methods to continuously monitor the equipment components, thus enabling proactive measures, predictive maintenance, or graceful degradations, are crucial

to the business. The automatic transmission clutches of the heavy duty equipment is a component whose failure may be costly, hence, an on-board condition monitoring of the clutches based on real time sensor data is desirable.

In automatic transmission, multiple wet clutches are used as in Fig. 1. It consists of steel-core friction discs, separator discs, two shafts, a piston, and automatic trans-mission fluid (ATF), usually referred to as the lubricant (Mäki 2003a). The ATF is the main difference between a dry clutch and a wet clutch. The multiple wet clutch pack is integrated with an electro-mechanical hydraulic actuator, which controls the engagement and disengagement process (Ompusunggu et al. 2013). The components of the electro-mechanical hydraulic actuator include a piston, a returning spring, a control valve, and an oil pump (Ompusunggu et al. 2013).

The L90F Wheel Loader was slightly modified to replicate clutch slippage by installing manual needle valves on the pressure outlet for clutch one and two. The driving was carried out on a steep uphill with one driver and with similar driving style. The monitored CANBUS data are differential speed 1, differential speed 2, output speed, turbine torque, turbine speed, and the gearshift parameters.

3 SVALI - Stream VALIdator

Figure 2 illustrates the architecture of the Stream VALIdator (SVALI) system.

In the figure data streams from different data sources are emitted to a SVALI *monitoring server*. The monitoring server processes queries that transform, combine, and analyze data from many different distributed data sources. Application programs access the monitoring server to perform various analyses.

Each SVALI system manages its own main-memory SVALI *database* that contains an ontology and local data. At each data source a *site SVALI* is running that manages data local to the source. The SVALI database in the monitoring server contains a *global ontology* describing meta-data about all kinds of monitored equipment, while the SVALI database at each site contains a *local ontology* describing the particular monitored equipment.

One kind of data source is data streams from wheel loaders, which are streamed to the monitoring server through a SVALI peer via a *CANBUS interface*. This kind of data source producing online streams is called a *streaming data source*. A SVALI peer encapsulates a streaming data source and a local ontology over which CQs are executed.

Another important kind of data source is CSV files containing logged data streams from monitored equipment. Data streams logged in CSV files can be played-back by SVALI and also streamed to the monitoring server.

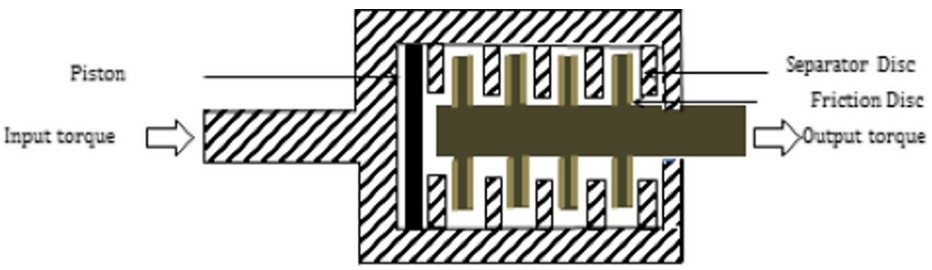

Fig. 1 A multiple wet clutch pack

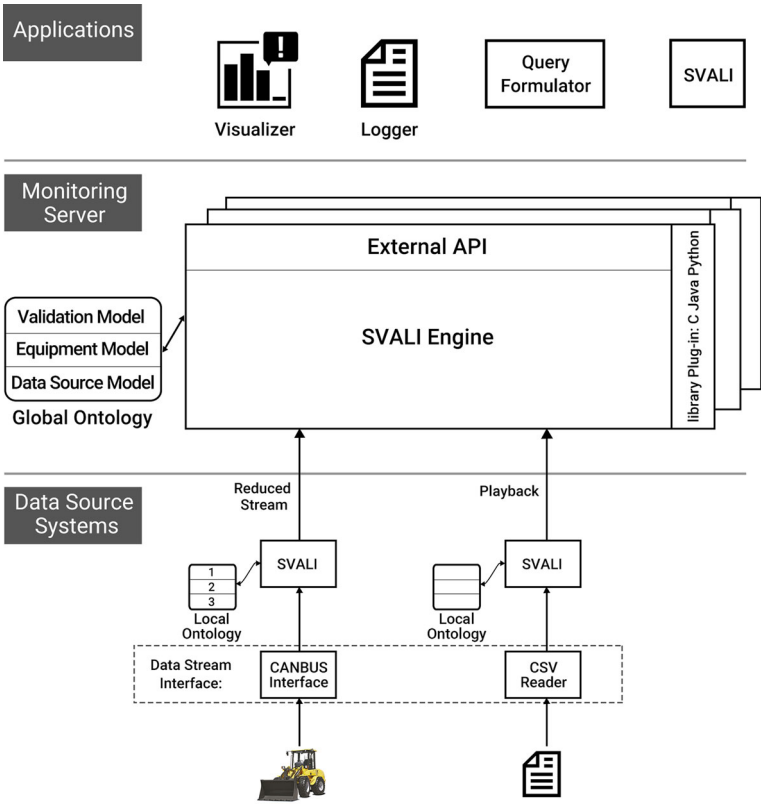

Fig. 2 The SVALI architecture

The ontologies are organized in three levels, as illustrated by Fig. 2. The *equipment model* is a common meta-data model that describes general properties common to all kinds of equipment, e.g. meta-data about sensor models and wheel loaders. The *data source model* maps raw data from a particular kind of data source to the common meta-data model. The *validation model* identifies anomalies in each kind of monitored equipment in terms of the data source and common meta-data models.

For example, the data source model of wheel loaders, the *wheel loader model*, maps data from raw data streams and log files into the common meta-data model. The validation model of wheel loaders includes a statistical model that identifies clutch slippages based on streams from sensors monitored through a CANBUS interface.

To handle computations in CQs that cannot be expressed as built-in functions, the SVALI engine provides an *algorithm plug-in* mechanism. The plug-ins can be used to implement specific algorithms, like indexing, computations, matching, optimization, and classification functions in Java, Python, or C.

The *applications* are other systems accessing the monitoring server by sending CQs to it through the SVALI *external API*. An application can be, e.g., a *visualizer* that graphically displays data streams derived from malfunctioning equipment to indicate what is wrong, a *query formulator* (Bauleo et al. 2014) with which CQs are constructed graphically, or a *stream logger* that saves derived streams on disk.

3.1 The functional data model of SVALI

SVALI is built on top of the functional database management system AMOS (Katchaounov et al. 2003) extending it with stream primitives, windowing operators, and validation functionality.

The basic primitives of SVALIs functional model are *objects* and *functions*. SVALI has two kinds of objects, *literal* and *surrogate objects*, where literals are immutable objects like numbers and string while surrogate objects are mutable objects represented by OIDs (object IDs) managed by the system. Objects can also be collections, where one important kind of collections in SVALI is called *stream* with the following properties: A stream is a sequence of stream elements representing measurements where a *time stamp* defines when the measurement was made. The stream elements are ordered by their time stamps, streams are continuously extended, and can potentially be unbounded. A stream has a *pace*, which is determined by the time stamps of the stream elements.

A query in SVALI defined as a *select-from-where* expression where variables can be bound to objects from any domain and functions can be used in the condition.

Functions can be of three kinds:

1. *Stored functions* model attributes of entities and relationships between entities.
2. *Derived functions* define rules or views as queries over other functions. Derived functions are similar to views in relational DBMS, but can be parameterized similar to prepared queries in JDBC.
3. *Foreign functions* are parameterized functions defined in an external program-ming languages such as C, Java, or Python.

Functions returning a stream as result are called *stream functions*. A CQ is defined by executing a query calling stream functions. To illustrate how regular queries and CQs can be defined, consider the simplified global meta-database in Fig. 3 of the scenario in Section 2.

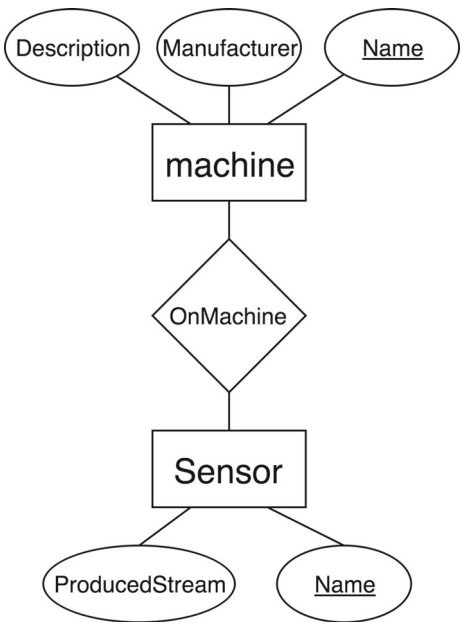

Fig. 3 Simplified equipment meta-data schema

In SVALI's data model entity types are defined as types while relationships and attributes are functions. In Fig. 3 the entity types *Machine* and *Sensor* are defined as following:

```
create type Machine;
create function name(Machine) -> Charstring as stored;
create function description(Machine) -> Chartring as stored;
create function manufacturer(Machine) -> Charstring as stored;
create type Sensor;
create function name(Sensor) -> Charstring as stored;
create function onMachine(Sensor) -> Machine as stored;
```

A query is an expression ***select*** *projection* ***from*** *bindings* ***where*** *condition*, where *bindings* specify the domains of variables, the *condition* is a predicate that specifies restrictions of variable bindings, and the *projection* specifies the result tuples.

For example, the following query returns the names of all sensors installed on a machine "L90F_A":

```
select name(s) from Sensor s, Machine m
where onMachine(s) = m and name(m) = "L90F_A";
```

The definition of a function is an expression ***create function*** *signature* ***as*** *definition*. The *signature* is an expression *fn(argtypes)* → *restypes* specifying the types of the arguments and results of the function named *fn*, while the *definition* specifies what kind of function it is (stored, derived, or foreign) and how its values are computed. For a derived function the definition is a query, while stored and foreign functions are defined by the keywords ***stored*** and ***foreign***, respectively.

For example, to be able run the same query with different machine names, one can define the following derived function:

```
1 create function hasSensor(Charstring machineName)
  -> Bag of Sensor
2 as select s from Sensor s, Machine m
3    where onMachine(s) = m and name(m) = machineName;
```

The function *hasSensor* returns a multi-set (bag) of sensors installed in machine *machineName*. The query can thus be expressed as *hasSensor("L90F_A");*

Here the signature *hasSensors(Charstring machineName)* → *Bag of Sensor* specifies that the function has an argument of type *Charstring* while the result is a bag of objects representing sensors. The *implementation* of the function on lines 2-3 specifies the result of the function for given parameters as a query.

All functions modeling attributes of object are stored functions. Streams can also be stored, for example:

```
create function producedStream(Sensor)
-> Stream of Vector of Number as stored;
```

The function *producedStream* returns a *stream of vector of numbers*, i.e. it is a stored stream function. Here, what is stored is not the stream elements themselves, but code that

generates the elements of the stream, i.e. by receiving them through the CANBUS-wrapper. Queries can be defined on streams, for example,

```
producedStream(hasSensor("L90F_A"));
```

The elements are retrieved as soon as the system can compute them. For example, the elements of a raw data stream of the CANBUS are delivered at the same as the CANBUS stream wrapper emits them. However, buffering, communication, and windowing may distort the pace and cause bursty result delivery, so SVALI does not guarantee that the measurements are returned in real-time at the same pace as the sources produce them.

To play back a stream according to the pace specified by their time stamps, use:

```
playback(producedStream(hasSensor("L90F_A")));
```

In this case the system uses the difference in time between the time stamps to determine when to deliver an emitted stream element to the user.

It is possible to make derived functions that return streams, for example:

```
create function machineStream(Charstring machineName)
                              -> Stream of Vector of Number
 as producedStream(hasSensor(machineName));
```

Here *machineStream()* is a derived stream function that returns a stream of vectors of numbers from a sensor installed on the named machine. The implementation function calls the derived function *hasSensor()* and the stored stream function *producedStream()*. Executing *machineStream("L90F_A")* is another example of a CQ.

One important data type in SVALI is called *stream windows*. Stream windows are motivated by the idea that only the most recent stream elements are of interest, e.g. only the most recent 100 elements (count windows) or the stream elements during the last second (time windows). In SVALI, functions that take data streams as input and return streams of windows as output are called *window functions*. There are several window functions in SVALI that form different kinds of stream windows including the most common ones such as count windows and time windows. New kinds of windows are also supported by SVALI, e.g. predicate windows (Xu et al. 2013) and *partition windows* explained below. For example, count windows are formed by the function *cwindowize(Stream s, Integer size, Integer slide)* → *Stream of Window*, where *s* is the input stream, *size* is the number of stream element in the window, and *slide* defines how many elements will be expired when a new window is formed. The following CQ creates a stream of count windows with size 4 and slide 2.

```
cwindowize(siota(1, 10), 4, 2);
```

Here *siota(1, 10)* is a built-in stream function that generates a stream of integers from 1 to 10.

3.2 Validation functionality

In order to detect unexpected equipment behavior, a *validation model* defines the correctness of a type of equipment as a set of *validation functions*, which for each validated stream from the equipment produces a *validation stream* describing the difference between measured and expected behavior. The validation model is stored as meta-data in the local database. Each tuple in a validation stream has the format *(ts, mv, x, ...)* where *ts* is the time of the

measurement, *m* is the measured value, and *x* is the expected value. In addition, application dependent values describing an anomaly are included in each validation stream element.

For example, a CANBUS stream contains measurements of different kinds, so the validation stream elements *(ts, mv, x, s, ...)* include an identifier of the anomaly *s*, called a *signal* identifier. The validation models can also produce *alert streams*, whose elements *(ts, mv, x, msg, ...)* are time stamped error messages *msg* describing the detected anomalies. Empty strings indicate normal behavior.

The validation functions can be executed per received element to test for anomalies. This kind of validation is called *instant validation*. A simple example of this kind is, "the temperature of functioning equipment should not exceed 90 °C".

Some monitoring is based on stream windows rather than individual stream elements. In SVALI this is naturally handled since the result of a window function is a stream of windows. For example, manufacturing often is cyclic since the same behavior is repeated for each manufactured item. Monitoring manufacturing cycles often is more meaningful than instant validations of the measurements during the cycle. This kind of validation requires the validation models be built based on stream windows and is called *window validation*. For example, instead of validating the temperature of the equipment within each time interval, the moving average of the temperature during each manufacturing cycle is checked.

With **model-and-validate**, *physical models* are defined as functions that map measured parameters to the expected values of the variables to be validated based on physical properties of the equipment. To detect anomalies, each element of a received stream is checked against the physical model of the equipment stored in the local database. For example, in Xu et al. (2013) a mathematical model is developed estimating the expected normal power usage based on sensor readings in stream elements. The mathematical model is expressed as derived functions and installed in SVALI's local database. The system provides a general function, called *model_n_validate()*, which compares data elements in CQs with the installed physical model and emits a validation stream of significant deviations.

The *model-and-validate()* function has the following signature:

```
model_n_validate(Stream s, Function modelfn, Function validatefn)
                -> Stream of (Number ts, Object m, Object x, ...)
```

The second input parameter, *modelfn(Object r, ...)* → *Object x*, is a function computing expected values *x* of validated physical properties (e.g. expected power consumption) in terms of a received stream element *r* in *s*, where *r* can be, e.g., a number, a vector, or a window. The expected value *x* can be a single value or a complex object specifying several allowed physical properties and their expected values.

The function *validatefn(Object r, Object x, ...)* → *Bag of (Number ts, Charstring mid, Object m)* returns the non-valid measured physical properties in *r*. That is, *validatefn(r, modelfn(r))* extracts from *modelfn(r)* time stamped invalid property value tuples *(ts, mid, m)*, representing the time *ts* when each invalid physical property *mid* had the invalid measured value *m*.

The model function can also be a stored function populated by, e.g., mining historical data. In that case the reference model is first mined offline and the computed parameters explicitly stored in the stored function *modelfn()* passed to *model_n_validate()*. In this paper, the reference model of the wheel loader scenario is learnt offline and then used by the validation function.

With **learn-and-validate**, models are defined that dynamically adapt to received stream elements, for example based on statistical models collecting data from the stream during

learning phases where the behavior of the equipment is guaranteed to be correct. Such kind of model is called a *learn-and-validate* model. To automatically learn a model of correct equipment behavior based on observed streaming data, the system provides the built-in function *learn_n_validate()*. It records the actual behavior of the monitored equipment and builds a statistical model based on the sampled correct behaviors. After the learning phase, the learnt model is used as the reference model with which the streaming data will be compared. As model-and-validate, the system emits a validation stream when significant deviations are detected.

The *learn-n-validate()* function has the following signature:

```
learn_n_validate(Stream s,Function learnfn,Integer n, Function
validatefn) -> Stream of (Number ts, Object m, Object e, ...)
```

The learning function *learnfn(Vector of Object s)* $\rightarrow$ *Object x* returns statistics *x* as a reference model based on *n* sampled stream elements *s*. The advantage with learn-and-validate is that the statistics is more up-to-date than with an offline model such as model-and-validate. Also it does not require defining the physical model. Offline models may be defined based on comparing the online stream with historical data.

3.3 Extensibility

Parts of the data processing will require advanced computations such as numerical and statistical computations made in real-time over the data elements streaming through SVALI. The numerical computations are often provided as algorithms and packages implemented in some conventional programming language such as Java or C. Rather than having to re-implement the algorithms in a new language, it should be possible to call packages implemented in a programming language from CQs without having to change the implementations of the algorithms. To cope with this challenge, SVALI is *extensible* by allowing for calling (dynamically linked) application dependent *foreign functions* implemented in some conventional programming language. The foreign functions can be used in CQs as any other functions. The algorithms themselves can be left unmodified and only simple interface code needs to be developed. There is a large library of system functions implemented as foreign functions in SVALI, e.g. for numerical, statistical, stream, and set operations. Foreign functions provide the basic mechanism for extending the system and to access external systems and data sources.

As an example, to use the built-in Python function *floor(x)* in CQs the following foreign function can be defined:

```
create function pyfloor(Number x) -> Real
  as foreign 'py:math.floor';
```

The prefix *py:* indicates that the foreign language implementing the foreign function *pyfloor()* is Python; the rest of the definition specifies that the function is implemented by the built-in Python function *floor()* in package *math*. It is particularly simple to call foreign functions in Python since it is a very powerful and interpreted, even though slow language. The foreign function interfaces to Java and C require more programming. For maximal performance C should be used, which provides for highest achievable performance, e.g. for FFT over data streams.

3.4 The stream uploader

For security reasons the SVALI server has to run on a computer separated by a firewall from the monitored equipment. The firewall allows client applications to call the SVALI server, not vice versa. This requires the data sources to establish an authorized connection to the SVALI server and then issue CQs and SVALI commands to the server.

Figure 4 illustrates the equipment data stream monitoring architecture. On the remote sites there are embedded *SVALI source clients*, running on-board the wheel loaders, which access local data streams via their CANBUS data stream wrappers. The *STREAM uploader* is a SVALI component executes local CQs that receive streaming data from the equipment. The CQs filter and transform the data streams before emitting them to the SVALI server. To authenticate stream delivery to the SVALI server, the source client has to first issue an authentication request. After authentication the system starts online stream delivery to the SVALI server in real-time. The STREAM uploader logs the uploaded measurements in temporary CSV files on the server, which are simultaneously tailed by the SVALI server when one or several CQs are activated. These CSV files also provide logs of the uploaded data. The logs can either be automatically deleted by the system after some time or uploaded to regular databases for further analyzes.

The uploaded streams are analyzed by application CQs accessing them in terms of stream identifiers managed by the SVALI server. Client applications can access SVALI either through a *CQ query editor* (Bauleo et al. 2014) that allows engineers to graphically specify CQs, or through client applications sending CQs to SVALI for execution.

4 Functional anomaly detection

The theory behind the validation model used for monitoring wheel loaders is based on a general statistical model to determine anomalies in streaming data, presented next.

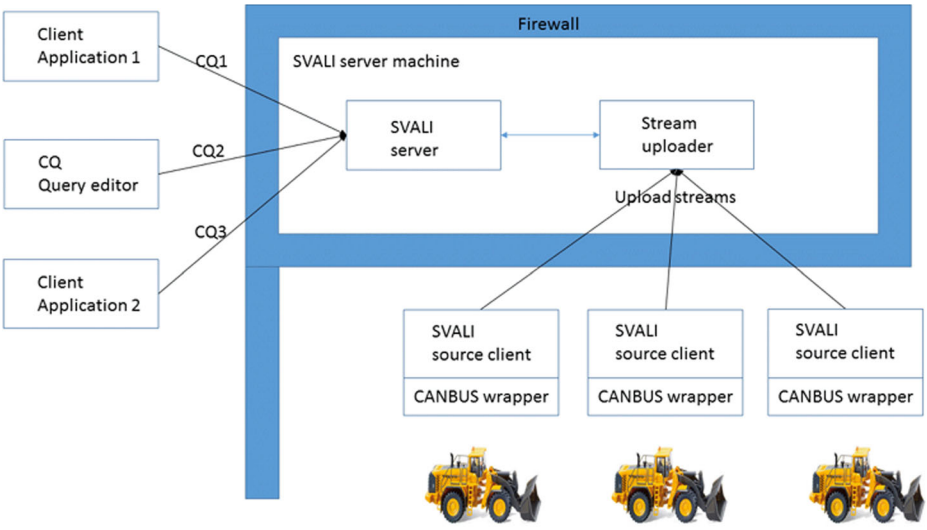

Fig. 4 Equipment data stream monitoring architecture

4.1 Higher order cumulant

Higher-order cumulants are useful in diverse applications for many years for their ability to handle non-Gaussian processes (Olsson et al. 2014). Cumulants above the third-order are regarded as higher order cumulants while lower order cumulants are from the third-order and below (Cumulant, http://en.wikipedia.org/wiki/Cumulant).

Higher-order cumulants are preferred instead of second-order for signals corrupted with Gaussian measurement noise since they are blind to Gaussian processes (Mendel 1991).

Cumulants and Moments are different terms (Dodge 1999). The moment of an ergodic random process is given as Bendat and Piersol (1980)

$$\varphi_k = E[x^k] = \int_{-\infty}^{\infty} x^k P(x) dx \quad k = 1, 2, \ldots$$

where $P(x)$ is the probability density function.

Moments defined about the mean are referred to as central moments (Bendat and Piersol 1980). The central moment of an ergodic random process is defined as

$$\varphi_k = E[(x - \acute{x})] = \int_{-\infty}^{\infty} (x - \acute{x})^k P(x) dx \quad k = 1, 2, \ldots$$

where $\acute{x}$ is the mean and $P(x)$ is the probability density function (Bendat and Piersol 1980).

The first central moment is always zero, the second central moment is the variance, and the third central moment is the skewness (Dodge 1999; Mendel 1991). The first, second and third order cumulants happens to be equal to the first, second and third central moments, but the fourth order cumulant is not equal to the fourth central moment but rather a complicated polynomial function of the central moment (Dodge 1999; Cumulant http://en.wikipedia.org/wiki/Cumulant; Olson et al. 2014).

Cumulants higher than the fourth order result in even much more mathematical complications (Dodge 1999).

The first, second, third and fourth order cumulants are defined as Mendel (1991):

$$C_{1,x} = E[x(n)]$$

$$C_{2,x}(k) = E[(x(n) - E[x(n)])(x(n+k) - E[x(n+k)])]$$

$$C_{3,x}(k_1, k_2) = E[(x(t) - E[x(n)])(x(t+k_1) - E[x(n+k_1)])(x(t+k_2) - E[x(n+k_2)])]$$

$$\begin{aligned} C_{4,x}(k_1, k_2, k_3) = {} & E[(x(n) - E[x(n)])(x(n+k_1) - E[x(n+k_1)]) \\ & (x(n+k_2) - E[x(n+k_2)])(x(n+k_3) - E[x(n+k_3)])] \end{aligned}$$

where the kth-order cumulants is a function of $k-1$ lags (Mendel 1991).

$C_{4,x}(k_1, k_2, k_3)$ is a higher order cumulant (Cumulant, http://en.wikipedia.org/wiki/Cumulant).

Kurtosis is based on the fourth order cumulant and thus a higher order cumulant (Decarlo 1997). The Kurtosis is the normalized fourth order cumulant about the mean and it is given by

$$Kurtosis = \frac{E[(x(t) - E[x(t)])^4]}{(E[(x(t) - E[x(t)])^2])^2} = \frac{\mu^4}{\sigma^4}$$

Where μ^4 is the fourth order cumulant and σ is the standard deviation (Decarlo 1997).

The Kurtosis gives an indication of the "peakedness" of a signal and the tailedness of a probability density function. For a normal distribution the Kurtosis value is 3 but $Kurtosis - 3 = 0$ is often used (Decarlo 1997).

4.2 Implementation of Kurtosis in SVALI

The data is streamed to SVALI through in the format of vectors of numbers called frames. Each frame is a tuple with the following format:

$$(ts, frame_{id}, v_1, v_2, ..., v_i)$$

where ts is the time of the measurement. Each frame of type $frame_{id}$ measures a set of signals, $signals(frame_{id}) = sig_1, sig_2, ..., sig_i$, which are stored as meta-data in SVALI. v_i is the sensor reading of sig_i in the frame. In the application there are five types of frames, as in Table 1.

A *value set* $vs(sig, w)$ is a set of values for a signal *sig* in a window w. In order to analyze statistics about a set of observed signals named $sig_i \in SIG$ in CQs, SVALI provides a function *valueSets(SIG, w)* that computes the values sets of the signals named o_i in window w, $vs(sig_i, w)$. On the value sets different kinds of statistical aggregate functions can be applied, e.g to determine anomalies in the values sets of *SIG* by using Kurtosis.

In the application, the aggregate function *kurtosis(V)* computes a measure of the peakedness of the probability distribution of the values in a value set V. To determine anomalies of signals *SIG* detected in window w, the Kurtosis of $vs(sig_i, w)$ is compared with the expected maximum Kurtosis $emk(sig_i)$ for each signals sig_i stored as meta-data. An anomaly is detected when $kursosis(vs(sig_i, w)) > emk(sig_i)$ for some signal sig_i measured by some frame in window w.

Partition windows In the Volvo wheel loader scenario, one important signal is the *gear* sensor reading, which specifies the current gear of the wheel loader. All the frames read from sensors when the current gear does not change are called one *gear cycle* and is defined as a *partition window*, where a new window is started when the gear changes. In general, partition windows in SVALI are defined based on the value changes of one or several *partition attributes* of the stream elements. In the example the partition attribute is *gear*, which identifies the current gear. When partition attribute values change, the previous window is emitted and new one is started. Partition windows are defined by the function *partwindowize(Stream s, Function partitionBy)* → *Stream of Window*. The partition function *partitionBy(Object o)* → *Object p* maps a received stream element o to p, where the value change of p is used to partition the stream s to form stream windows.

In the Volvo wheel loader scenario, the partition function defines the gear as the partition attribute, which is element 8 in the frame.

```
create function gear(Vector of Number frame)
-> Number g as frame[8];
```

Table 1 Five types of frames

Frame ID	Signals
10	ForwardPressure1, ForwardPressure2, MainPressure, ForwardPressure3
11	PressureR, RearTorque, FrontTorque
2364542723	InputSpeed, TurbTorque, TrmOiltem, Shifting1To2, OffgSlipping, OngSlipping, Gear, DirGear
2364542467	DiffSpeed1, DiffSpeed2, TurbSpeed, OutgSpeed
15	OutputCoolerTemp

The stream s is partitioned into a stream of windows when the gear is changed by the function *partwindowize(s, #'gear')*. To detect anomalies in observed signals during each gear cycle, on each partition window the Kurtosis of each observed signal is calculated as physical property and compared with its maximum allowed Kurtosis value. The validation over a CANBUS stream s is specified by *model_n_validate(s, #'allowedKurtosis', #'anomalies')*.[1] The model function *allowedKurtosis(Window pw) → Bag of (Charstring sigi, Number ai)* returns a set of pairs (sig_i, a_i) representing the allowed Kurtosis a_i of each observed signal sig_i in the partition window pw. The validation function *anomalies(Window pw, Bag of (Charstring sigi,Number a) exp) → Bag of (Number ts, Charstring sigj, Number mj)* returns a set of triples (ts, sig_j, m_j), indicating timestamped invalid measurements m_j of signal sig_j in pw.

The derived function *allowedKurtosis()* is defined as:

```
create function allowedKurtosis(Window pw)
          -> Bag of (Charstring sigi, Number ai)
as select sigi, maxAllowedKurtosis(sigi)
   from Vector of Number vs
   where vs = valueSet(sigi, pw);
```

The stored function *maxAllowedKurtosis(Charstring sig) → Number m* returns the allowed Kurtosis *m* for a signal *sig*.

The function *anomalies()* is defined as:

```
create function anomalies(Window pw,
        Bag of (Charstring sigi, Number ai) exp)
-> Bag of (Number ts, Charstring sig, Number mi)
  as select ts(w), sigi, mi
    where mi = measuredKurtosis(exp)
      and (sigi, ai) in exp and mi > ai;
```

The function returns the anomalies detected in *pw* by selecting the unexpected measured Kurtosis values m_i of signal sig_i that exceeds the maximum allowed value a_i. The function *measuredKurtosis(sigi, pw)* returns the computed Kurtosis for signal sig_i in pw. It is defined as:

```
create function measuredKurtosis(Charstring sigi, Window pw)
-> Number mi as kurtosis(valueSet(sigi, pw));
```

The function *kurtosis(vs)* of a value set *vs* is defined as:

```
create function kurtosis(Bag of Number vs) -> Number
  as cumulant4(vs) / stdv(vs)^4;
```

where *cumulant4()* computes the 4th cumulate of value set *vs*:

```
create function cumulant4(Bag of Number vs) -> Number
  as sum(select (e - avg(vs))^4
         from Number e
        where e in vs);
```

[1] The syntax #'fn' denotes the function named 'fn'.

A CQ that returns a stream containing invalid measurements for the wheel loader named "L90F_A" is defined as:

```
select model_n_validate(gearcycles,#'allowedKurtosis',
                        #'anomalies')
  from Stream of Window gearcycles
 where gearcycles = partwindowize(
                    machineStream("wheelLoaderA"),
                    #'partBy');
```

The function *partwindowize()* produces a stream of windows for each gear cycle on which *model_n_validate()* is applied using the above model.

5 Distributed equipment monitoring

Figure 5 shows a typical configuration of SVALI in a distributed setting where a number of wheel loaders are monitored to produce data streams transmitted to a monitoring center where they are merged. Each wheel loader runs a local SVALI system running the following CQ to produce a stream of gear cycle windows from CANBUS channel 007 uploaded to the monitoring center "M1":

```
upload(partwindowize(CANstream(007), #'gear'), "M1");
```

Each site has an identifier which is sent to the monitoring center and there stored in a function enumerating the monitored sites, *sites(Number id)* → *Charstring Name*. The monitoring center identifies anomalies in any monitored machine by merging and validating the uploaded gear cycle window streams from all the sites with the CQ:

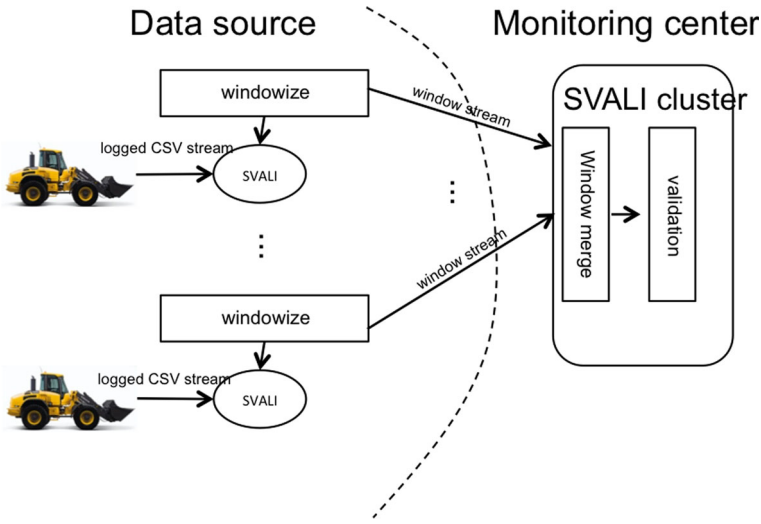

Fig. 5 Central validation

```
select model_n_validate(gearCycles,#'allowedKurtosis',
                        #'anomalies')
  from Stream of Window gearCycles
 where gearCycles = merge(select streamFrom(site(i))
                          from Integer i);
```

The function *streamFrom(Charstring site)* → *Stream* returns the stream uploaded from a given site. The merging is done asynchronously as new tuples arrive from the sites, while local queries produce streams of gear cycle windows in parallel on each wheel loader. This is possible since SVALI systems run both in the monitoring center and on each wheel loader. The execution of local queries on each wheel loader furthermore gives local control on each site what data to send to the monitoring center. The validation is done at the monitoring center. This is called *central validation*.

The local SVALI systems on each wheel loader enable parallel processing of expensive functions. In particular also the expensive *model_n_validate()* can be run in parallel on each wheel loader as illustrated by Fig. 6. This should improve the response and throughput of the validation. This is called *parallel validation*. In this case the following CQ runs on each wheel loader:

```
upload(select model_n_validate(gearCycles,
                        #'allowedKurtosis', #'anomalies')
         from Stream of Window gearCycles
        where gearCycles = partwindowize(CANstream(007),
                                         #'gear'), "M1");
```

The following CQ runs at the monitoring center which just merges the validation stream from each site:

```
merge(select streamFrom(site(i)) from Integer i);
```

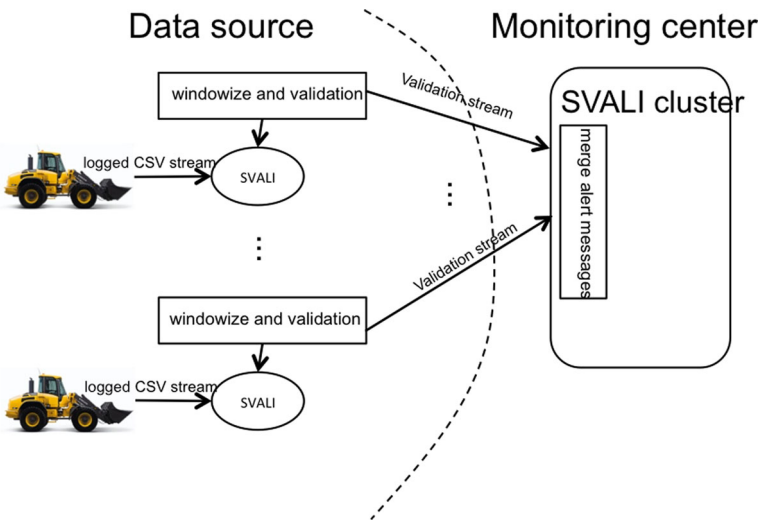

Fig. 6 Parallel validation

6 Evaluation

The two validation strategies are experimentally evaluated to investigate the performance improvement by local SVALI validation on each site. For experimental purposes, we use logged CSV files from Volvo CE wheel loaders to simulate online streams on each site. The number of validated wheel loaders is scaled up to 100 by starting new SVALI instances on separate nodes. The size of each recorded source data stream is around 40 MB (543917 tuples) having more than 500 partition windows. The result stream for validating the recorded source data consists of 1054 tuples, i.e. the data reduction is about 99.81 %. The experiments were made on a Dell PowerEdge R815 which has 4 CPUs with 16 2.3 GHz cores each. Both the processing capacity of SVALI and the response times (delays) were measured for different experimental settings.

6.1 System capacity

The purpose of the first experiment is to investigate the capacity of the system, i.e. how much data can be validated as the number of wheel loaders is increased. In the experiment all of the recorded data was streamed to each site SVALI at disk read speed, which is 201316 tuples/s (20 Mbytes/s per site or 5 μs/tuple), and processed by SVALI with the model above using central and parallel validation. The total through-put of processing the entire recorded streams at full speed was measured in Fig. 7. The throughput of the central validation on one core was around 3.5 Mbps. The throughput of parallel validation reached a maximum of 110 Mbps when the number of machines was more than the available number of cores, 64, since more than one SVALI instance then have to run on the same core.

6.2 Response time

First the average and maximum response times with central and parallel validation were measured. Each wheel loader $WL_n, n = 1...N$ has a recorded data stream S_n over which

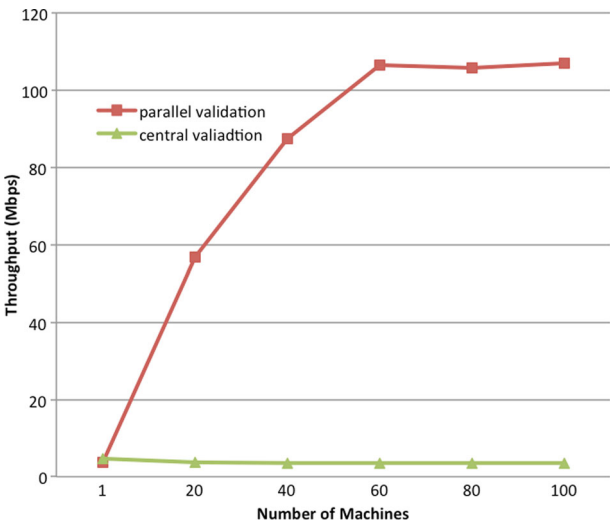

Fig. 7 Full speed streaming throughput

I_n partition windows are created by SVALI during the processing. As in the linear road benchmark (Arasu et al. 2004), the *response time* is defined as the difference between the time $receiveT_i$ when the stream element is received by the DSMS and $emitT_i$ when the DSMS emits the result. Maximum and average response times are calculated as following:

$$avgReponseTime = \frac{\Sigma_{n=1}^{N} \Sigma_{i=1}^{I_n} (emitT_i - receiveT_i)}{\Sigma_{n=1}^{N} I_n}$$

$$maxReponseTime = \max_{1 \le n \le N} (\min_{1 \le i \le I_n} (emitT_i - receiveT_i))$$

6.2.1 Scaling the number of monitored streams

All of the recorded data was streamed to each site SVALI at disk read speed, i.e. 20 Mbytes/s per site or 5 μs/tuple, and validated with the model above using both central and parallel validation. Both the average and maximum validation times were measured in Fig. 8.

Figure 8 shows that parallel validation clearly outperforms the central one by several orders of magnitude. The max response time with central validation was much more slower than the average and therefore not included in the diagram. For parallel validation only, the maximum validation time is compared with the average in Fig. 9. It shows that the average validation time increases with a very small slope, while the maximum time increases faster, in particular when the number of machines exceeds the available number of cores, 64. The figure also shows that the average times are much lower than the maximum one, which means most of the validations are cheap with a few outliers.

6.2.2 Using actual stream rates

The previous experiment was conducted with a very high data rate per site stream. In practice the stream rate is lower. We therefore measured the scalability of the system over the number of machines using the actual stream rates. The streams are time stamped around

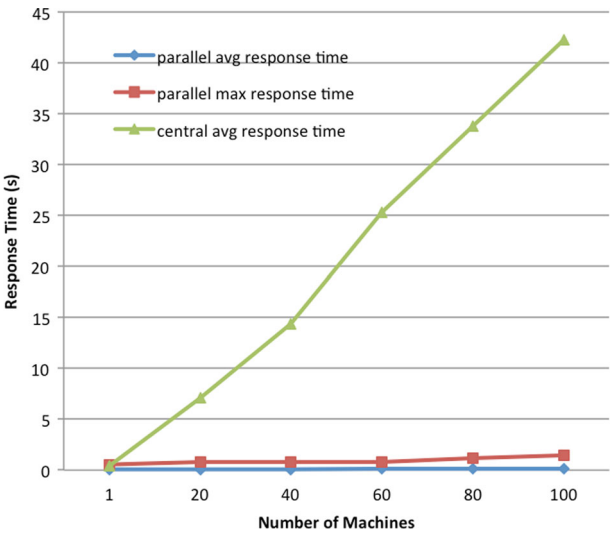

Fig. 8 Full speed streaming response time

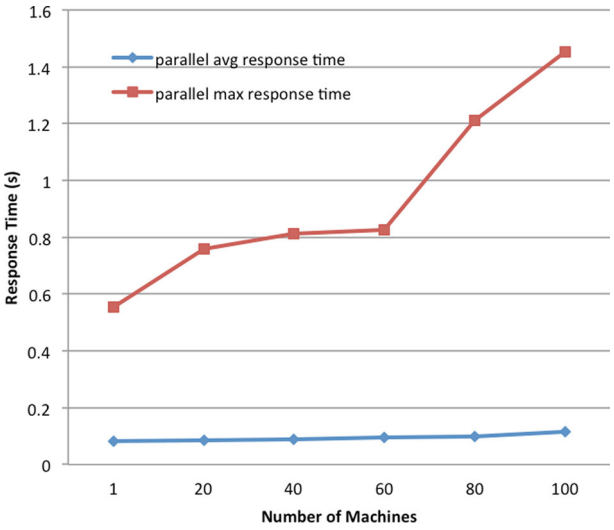

Fig. 9 Full speed streaming parallel response time

each 5 *ms* / tuple and the *playback()* function was used, unlike in the first experiment. Figure 10 shows that in this case parallel validation also outperforms the central one. With parallel validation the average response time stays almost constant while it increases slowly when scaling the number of machines with full speed validation. However, the central validation here performs comparably better, as illustrated by Fig. 11 measuring the improvement ratio of central and parallel validation for the full speed and actual data rates. It shows that the response time of central validation improves a lot with the actual stream rate, which

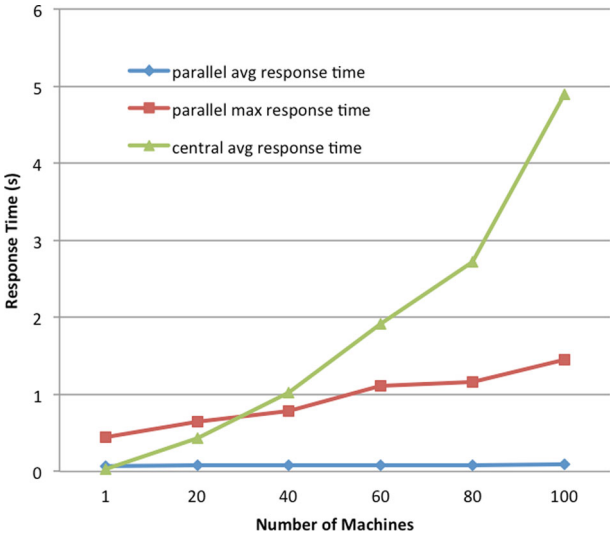

Fig. 10 Playback stream average response time

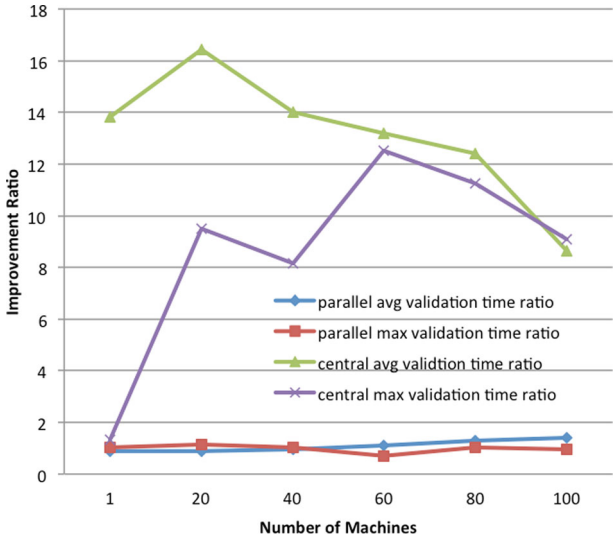

Fig. 11 Improvement ratio between full speed and actual rates

is because the playback function delays the delivery of the data streams and thus gives more room for the central server to do the validation.

6.2.3 Scaling the site stream rates

To investigate how different site stream rates influence the validation scalability, they were scaled from 1 ms to 10 ms per tuple while keeping the number of machines constant at 100. As shown in Figs. 12 and 13, central validation gets saturated when the stream rate is high while for parallel validation both the maximum and average response times are virtually rate independent as long as there are sufficient computational resources.

In conclusion, we show that the parallel validation in SVALI scales very well w.r.t. response time and system throughput when pushing expensive computations as close to the source as possible. In the experiments parallel validation has 0.09 second average response time, which is sufficient for our application. Different from hard real-time systems, for equipment anomaly detection the average response time is much more important than the maximum as long as the overall stream process can keep up with the stream rate.

7 Related work

In the last decades, data stream processing has gained a lot of research interests. Several Data Stream Management Systems (DSMSs), such as Aurora (Abadi et al. 2003), and STREAM (Motwani et al. 2002), have been developed based on modified relational data models where variables in queries are bound to rows. By contrast, SVALI uses a functional data model to express CQs where streams are first class objects in domain calculus queries. Furthermore, SVALI allows calling external libraries as foreign functions so that complex algorithms over data streams can be efficiently implemented and used in CQs.

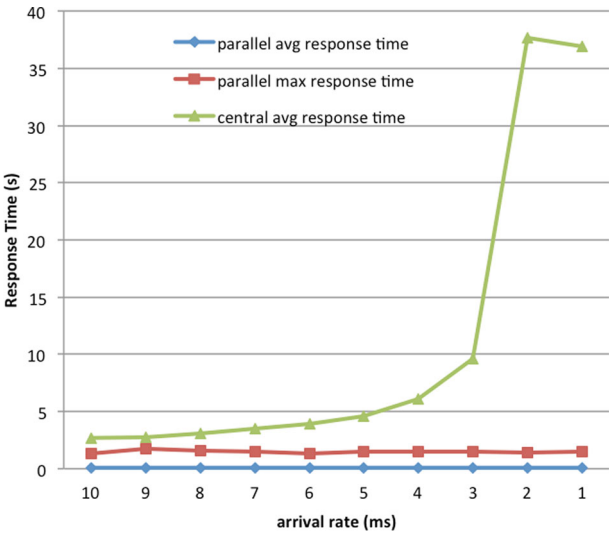

Fig. 12 Validate 100 machines with different arrival rates

Various research issues on outlier detection for data streams are discussed in Sadik and Gruenwald (2014). In our work, unexpected behavior of the equipment can also be seen as outliers from normal behavior. Because data streams are online and dynamic, outlier detection in the stream context becomes fundamentally different than regular outlier detection, which often done in a *store-and-process* fashion. In Sadik and Gruenwald (2014) previous work on stream outlier detection is categorized into four major classes: (i) outlier detection over sliding windows (Angiulli and Fassetti 2010; Cao et al. 2014; Subramaniam et al. 2006; Yang et al. 2009), (ii) auto-regression (Shuai et al. 2008) , (iii) data stream clustering

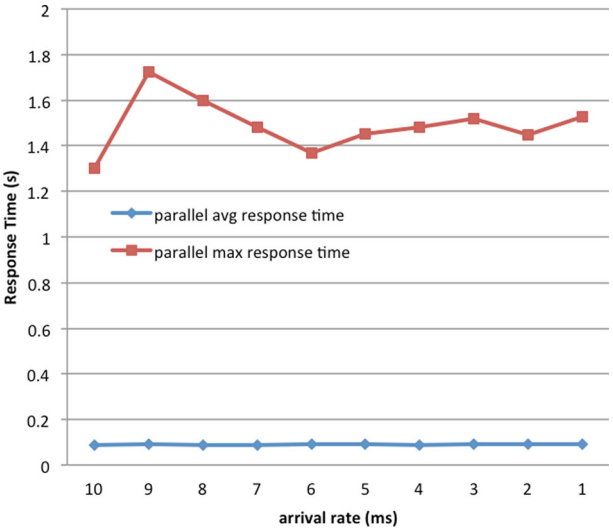

Fig. 13 Validate 100 machines with different arrival rates

(Yang et al. 2009), and (iv) statistical density functions over data stream elements (Huang et al. 2012; Subramaniam et al. 2006; Zhang et al. 2008). Because SVALI makes no difference between regular data types and stream objects, anomaly detection using SVALI's built in validation functions falls into the first category. Our application that validate correct behavior of wheel loaders with a Kurtosis-based statistical model shows that the domain query language of SVALI provides a powerful tool to express statistical and other numerical functions in mathematical models that identify abnormal behavior of the equipment. Hence, our work also belongs to the fourth category.

There are two main parallelization strategies for processing data streams. One is to parallelize continuous query execution plans (Abadi et al. 2005; Jain et al. 2006) where operators are placed on different compute nodes. The other is to partition the input streams into substreams, on which CQs are applied in parallel (Brenna et al. 2009; Zeitler and Risch 2011). In SVALI, the latter strategy is used by parallelizing expensive validations over the equipment sites. The very high reduction in streams data rates for anomaly detection makes parallel validation particularly favorable.

In Xue et al. (2006), the authors describe an approach resembling our Kurtosis model for fault detection in locomotives as an add-on to the CBR (Case Based Reasoning) diagnosis system (Varma and Roddy 1999). Like in our system, the signals are processed individually to detect an anomaly and then fused together using another machine learning algorithm. They use a non-parametric test to detect individual anomalies and a generalized regression neural network to combine the signals to one anomaly indication output. However, they do not describe how they integrate the CBR system and the anomaly detection part, while we show how the functional data model of SVALI enables convenient integration of data streams from distributed equipment.

In other work on anomaly detection from equipment, e.g by Miura et al. (1998), Mäki (2003b), Marklund (2010), Ito et al. (2012), Fatima et al. (2012), Okabe (2009), Berglund (2013), the anomaly detection is made in test-rigs, but not in the actual heavy duty machines. In our application clutch slippage detection and diagnoses are done on-board the equipment where streams of sensors are processed on the machine by SVALI using a CAN bus wrapper. SVALI enables the anomaly detection to be expressed on a very high level as CQs using a functional anomaly detection model.

8 Conclusions and future work

We presented a general system, SVALI, to detect anomalies in data streams. Anomalies in the behavior of heavy duty equipment streams are detected by running SVALI on-board the machines. In SVALI anomaly detection rules are expressed declaratively as continuous queries over mathematical/statistical models that match incoming streamed sensor readings against an on-board database of normal readings.

To enable scalable validation of geographically distributed equipment, SVALI is a distributed system where many SVALI instances can be started and run in parallel on the equipment. Central analyses are made in a monitoring center where streams of detected anomalies are combined and analyzed.

The functional data model of SVALI provides definition of meta-data and validations models in terms of typed functions. Continuous queries are expressed declaratively in terms of a domain calculus where streams are first class objects. Further-more, SVALI is an extensible system where functions can be implemented using external libraries written in C, Java, or Python without any modifications of the original code.

To control the transmission of equipment data streams to the monitoring center, there is a firewall around the monitoring center. Therefore, the data streams from the equipment are transmitted to the monitoring center using a stream uploader, rather than accessing the sensored data in the inverse direction from the monitoring center.

To enable stream validation on a high level, the system provides two system vali-dation functions, *model_n_validate()* and *learn_n_validate()*. *model_n_validate()* allows the user to define mathematical models based on physical properties of the equipment to detect unexpected equipment behavior. The model can also be built using historical data and then stored in the database as reference model. In the scenario from Volvo CE, the maximum allowed Kurtosis is first built off-line and then used to detect clutch slippages of wheel loaders. By contrast, *learn_n_validate()* builds a statistical model by sampling the stream on-line as it flows. The model can also be re-learnt in order to keep it updated, e.g. after some time units or amount of processed stream elements.

Experimental results show that the distributed SVALI architecture enable scalable monitoring and anomaly detection with low response times when the number of monitored machines and their data stream rates increase. The experiments were made using real data recorded in running equipment. The experiments show that parallel validation where expensive computations are done in the local SVALI peers has good response time and throughput.

The monitoring capability presented is furthermore a necessary means for monitoring large numbers, or fleets, of for instance vehicles or production equipment when customers are offered result or availability oriented contracts. Examples of such offers are Industrial Product-Service Systems and Functional Products, where the ability to act in a proactive manner and conduct predictive maintenance based on facts are key (Olsson et al. 2014).

A future work is to combine different kinds of data streams from different equipment exploring more information to refine the model. New scalability challenges may come up w.r.t. stream joins. Another direction is to analyze parallelization strategies when there are shared computations between CQs over the same data stream.

Acknowledgments This work is supported by EU FP7 project Smart Vortex http://www.smartvortex.eu/ and the Swedish Foundation for Strategic Research under contract RIT08-0041.

References

Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., & Zdonik, S. (2003). Aurora: a new model and architecture for data stream management. *The VLDB Journal*, *12*, 120–139.

Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., & Zdonik, S. (2005). The design of the borealis stream processing engine. In *Second biennial conference on innovative data systems research (CIDR 2005), Asilomar, CA*.

Angiulli, F., & Fassetti, F. (2010). Distance-based outlier queries in data streams: the novel task and algorithms. *Data Mining and Knowledge Discovery*, *20*, 290–324.

Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., & Tibbetts, R. (2004). Linear road: a stream data management benchmark. In *Proceedings of the thirtieth international conference on very large data bases - volume 30, VLDB '04, VLDB Endowment* (pp. 480–491).

Bauleo, E., Carnevale, S., Catarci, T., Kimani, S., Leva, M., & Mecella, M. (2014). Design, realization and user evaluation of the smartvortex visual query system for accessing data streams in industrial engineering applications. *Journal of Visual Languages and Computing*, *25*(5), 577–601.

Bendat, J., & Piersol, A. (1980). Engineering applications of correlation and spectral analysis. Wiley.

Berglund, K. (2013). Predicting wet clutch service life performance. PhD thesis.

Brenna, L., Gehrke, J., Hong, M., & Johansen, D. (2009). Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the third ACM international conference on distributed event-based systems, DEBS '09* (pp. 3:1–3:12). New York, NY, USA: ACM.

Canbus. http://en.wikipedia.org/wiki/CAN_bus.

Cao, L., Wang, Q., & Rundensteiner, E.A. (2014). Interactive outlier exploration in big data streams. *Proc. VLDB Endow.*, *7*, 1621–1624.

Cumulant. http://en.wikipedia.org/wiki/Cumulant.

Decarlo, L.T. (1997). On the meaning and use of kurtosis. *Psychological Methods*, 292–307.

Dodge, V.R.Y. (1999). The complications of the fourth central moment. *The American Statistician*, *53*(3), 267–269.

Fatima, N., Marklund, P., & Larsson, R. (2012). *Water contamination effect in wet clutch system*.

Huang, J., Zhou, B., Wu, Q., Wang, X., & Jia, Y. (2012). Contextual correlation based thread detection in short text message streams. *Journal of Intelligent Information System*, *38*, 449–464.

Ito, K., Barker, M., Kubota, K., & Yoshida, S. (2012). Designing paper typewet friction material for high strength and durability. In *SAE International off-highway and powerplant congress and exposition*.

Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., & Venkatramani, C. (2006). Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD international conference on management of data, SIGMOD '06* (pp. 431–442). New York, NY, USA: ACM.

Katchaounov, T., Josifovski, V., & Risch, T. (2003). Scalable view expansion in a peer mediator system. In *Eighth international conference on database systems for advanced applications, 2003. (DASFAA 2003). Proceedings* (pp. 107–116).

Mäki, R. (2003a). Wet clutch tribology: friction characteristics in limited slip differentials. PhD thesis.

Mäki, R. (2003b). Wet clutch tribology; friction characteristics in all-wheel drive differentials. In *Tribologia*, (Vol. 22, no. 3 pp. 5–16).

Marklund, P. (2010). Permeability measurements of sintered and paper based friction materials for wet clutches and brakes 2010.

Mendel, J. (1991). Tutorial on higher-order statistics (spectra) in signal processing and system theory: theoretical results and some applications. *Proceedings of the IEEE*, *79*, 278–305.

Miura, T., Sekine, N., Azegami, T., Murakami, Y., Itonaga, K., & Hasegawa, H. (1998). Study on the dynamic property of a paper-based wet clutch. In *SAE International congress and ex-position*.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., & Varma, R. (2002). Query processing, resource management, and approximation ina data stream management system. Technical Report 2002-41, Stanford InfoLab.

Namburu, S., Wilcutts, M., Chigusa, S., Qiao, L., Choi, K., & Pattipati, K. (2006). Systematic data-driven approach to real-time fault detection and diagnosis in automotive engines. In *Autotestcon, 2006 IEEE* (pp. 59–65).

Okabe, K. (2009). Proposal of field life design method for wet multiple plate clutches of automatic transmission on forklift-trucks.

Olsson, T., Källström, E., Gillblad, D., Funk, P., Lindström, J., Håkansson, L., Lundin, J., Svensson, M., & Larsson, J. (2014). Fault diagnosis of heavy duty machines: Automatic transmission clutches. In *Workshop on synergies between CBR and data mining at 22nd international conference on case-based reasoning*.

Ompusunggu, A.P., Papy, J.-M., Vandenplas, S., Sas, P., & Brussel, H.V. (2013). A novel monitoring method of wet friction clutches based on the post-lockup torsional vibration signal. *Mechanical Systems and Signal Processing*, *35*(1-2), 345–368.

Sadik, S., & Gruenwald, L. (2014). Research issues in outlier detection for data streams. *SIGKDD Explor. Newsl.*, *15*, 33–40.

Shuai, M., Xie, K., Chen, G., Ma, X., & Song, G. (2008). A kalman filter based approach for outlier detection in sensor networks. In *International conference on computer science and software engineering, 2008*, (Vol. 4 pp. 154–157).

Subramaniam, S., Palpanas, T., Papadopoulos, D., Kalogeraki, V., & Gunopulos, D. (2006). Online outlier detection in sensor data using non-parametric models. In *Proceedings of the 32nd international conference on very large data bases, VLDB '06, VLDB Endowment* (pp. 187–198).

Varma, A., & Roddy, N. (1999). Icarus: design and deployment of a case-based reasoning system for locomotive diagnostics. *Engineering Applications of Artificial Intelligence*, *12*(6), 681–690.

Xu, C., Wedlund, D., Helgoson, M., & Risch, T. (2013). Model-based validation of streaming data: (industry article). In *Proceedings of the 7th ACM international conference on distributed event-based systems, DEBS '13* (pp. 107–114). New York, NY, USA: ACM.

Xue, F., Yan, W., Roddy, N., & Varma, A. (2006). Operational data based anomaly detection for locomotive diagnostics., In Arabnia, H.R., Kozerenko, E.B., & Shaumyan, S. (Eds.) *MLMTA* (pp. 236–241): CSREA Press.

Yang, D., Rundensteiner, E.A., & Ward, M.O. (2009). Neighbor-based pattern detection for windows over streaming data. In *Proceedings of the 12th international conference on extending database technology: advances in database technology, EDBT '09* (pp. 529–540). New York, NY, USA: ACM.

Zhang, J., Gao, Q., & Wang, H. (2008). Spot: a system for detecting projected outliers from high-dimensional data streams. In *IEEE 24Th international conference on data engineering, 2008. ICDE 2008* (pp. 1628–1631).

Zeitler, E., & Risch, T. (2011). Massive scale-out of expensive continuous queries. *PVLDB*, *4*(11), 1181–1188.