SPE 13WRM-165379-MS

SDPF: A Framework for Online, Real-time Cleansing of Upstream Operating Data

Farnoush Banaei-Kashani, University of Southern California, Mohammad Asghari, University of Southern California, Mahdi Rahmani Mofrad, University of Southern California, Cyrus Shahabi, University of Southern California, Lisa Brenskelle, Chevron

This paper was prepared for presentation at the SPE Western Regional & AAPG Pacific Section Meeting, 2013 Joint Technical Conference held in Monterey, California, USA, 19–25 April 2013.

Abstract

Since the quality of the raw upstream operating data can be poor in many instances (due to errors, incompleteness, and inconsistency), there is often an urgent need to cleanse the data in real time before using the data for process and business decision-making. In this paper, we present the design and development of an integrated data cleansing framework that can address a variety of upstream operating data quality issues systematically. Our proposed framework, dubbed SDPF (short for Scalable Data Processing Framework), benefits from the following features: *Online/Real-time Data Cleansing*, *Scalability*, *Configurability*, *Reusability*, and *Comprehensiveness*. We have tested and verified the performance of our data cleansing system with both real upstream operating data collected and synthetic data.

1. Introduction

It is well known that the value of any data is always proportional to the accuracy (and timeliness) of the data, and therefore, transforming raw data (including all relevant meta-data) into trustable data with physical meaning is one of the main requirements of building a solid platform for any information system. In particular, many issues are repeatedly impacting the quality of the upstream operating data, causing data problems such as duplicate values, out of synch values (i.e., values from the same source with different timestamps), null/unknown values, values that exceed data range limits (hi/lo), outlier values, propagation of suspect or poor quality data, and time ranges of missing data due to field telemetry failures.

While there exists an extensive set of tools that deal with data quality, there are still many gaps to fill. In this paper, we present the design and development of an integrated data cleansing framework that can address a variety of upstream operating data quality issues systematically. In particular, our proposed framework, dubbed SDPF (short for Scalable Data Processing Framework), benefits from the following features:

- *Online/Real-time Data Cleansing*: To allow for real-time monitoring and decision-making with time-critical applications, SDPF supports on-the-fly cleansing of the data while the data is being collected.
- *Scalability*: SDPF gracefully scales up to cleanse numerous (i.e. hundreds of thousands) high-rate data streams simultaneously.
- *Configurability*: SDPF can be easily configured/tuned by non-expert users to accommodate specific characteristics of the data source and/or application of interest per use-case.
- *Reusability*: Once accessorized with a specific data cleansing operator, SDPF allows reuse of the capability in various contexts and applications.
- *Comprehensiveness*: The SDPF design is such that it allows accommodating all major data cleansing problems by plugging in the corresponding data cleansing operators.

2. Related Work

There is extensive prior work on data cleaning in the context of data warehouses and data integration. However, traditional data cleaning tends to focus on a small set of well-defined tasks, including transformations, matchings, and duplicate

elimination [1]. SDPF provides a framework to utilize these techniques in an online, streaming manner, extending them to take advantage of the characteristics of physical sensor data.

The AJAX tool [2] proposes an extensible, declarative means of specifying cleaning operations in a data warehouse. This is similar in spirit to SDPF in that we also define an easily deployable and configurable architecture that utilizes declarative query processing. AJAX, however, does not consider streaming data or windowed processing.

Potter's wheel [3] is an interactive tool for assisting in data cleaning operations. However, its interactive nature restricts its ability to handle the streaming data.

IntelliClean [4] propose a rule-based approach to express matching, clustering and merging operations, which is implemented using the Jave Expert System Shell. However, this framework clearly does not scale up for very large datasets due to the use of an expert system shell.

Finally, many commercial efforts (e.g., [5, 6, 7]) have addressed data cleaning issues related to enterprise data integration. These solutions provide tools for cleaning and translating data, but do not address temporal or spatial aspects of physical sensor data.

3. SDPF Design

At the conceptual level, we define SDPF with an architecture consisting of five different data cleansing "modules", namely, Individual Analytics (IA), Temporal Group Analytics (TGA), Spatial Group Analytics (SGA), Arbitration Analytics (AA), and Field Analytics (FA), all serialized in a pipeline (see Figure 1). The SDPF data cleansing modules operate on the data in sequence, with disjoint and covering functionality; i.e., they each focus on a specific set of data cleansing problems and they are complementary. From bottom to top, modules focus on finest data resolution (single readings) to coarsest data resolution (multiple sensors and various modalities). In turn, each SDPF module implements one or more data cleansing "operators" (not shown in the figure), all focusing on the type of functionality supported by the corresponding module.

With SDPF, while the order/sequence of applying the modules is fixed (selected by design), the combination of modules applied to the specific data stream is configurable. Moreover, the operators applied within each module are also configurable/programmable. Finally, the SDPF operators can be implemented in two ways (with increasing level of functionality, and decreasing level of flexibility): declarative continuous queries, and user-defined functions or aggregates.

Next, we briefly explain the functionality of the SDPF modules.

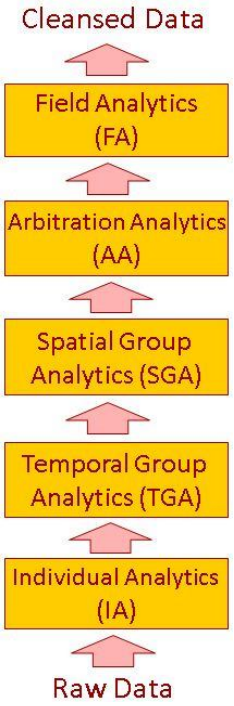

Figure 1: SDPF Conceptual Design

3.1 Individual Analytics (IA) Module

The IA operators operate on single data values in input data streams. Their purpose is to clean and or filter individual data items only based on the value of the item itself. Example IA operators are as follows:

- Simple outliers detection (e.g., exceeding thresholds)
- Raw data conversion (e.g., heat sensors output data into voltages, which must be converted to temperature by considering calibration of that sensor)

3.2 Temporal Group Analytics (TGA) Module

The TGA operators operate on data segments in input data streams. Their purpose is to clean individual data values as part of a temporal group of values by considering their temporal correlation. The TGA operators are implemented using window-based queries. Example TGA operators are as follows:

- Generic temporal outlier detection
- Temporal interpolation for data reconstruction

3.3 Spatial Group Analytics (SGA) Module

The SGA operators operate on data values from multiple data streams. Their purpose is to clean individual data values as part of a spatial group of values by considering their spatial correlation. The SGA operators are implemented using window join queries. An example SGA operator is:

- Generic spatial outlier detection (e.g., within a spatial granule this operator can compute the average of the readings from different sensors and omit individual readings that are outside of two deviations from the mean)

3.4 Arbitration Analytics (AA) Module

The AA operators operate on data values from multiple spatial granules to arbitrate the conflicting cleansing decisions. Example AA operators are as follows:

- Conflict resolution
- De-duplication

3.5 Field Analytics (FA) Module

The FA operators operate on data values from multiple stream sources of different modalities (e.g., heat and pressure). Their purpose is to consider correlation between data values of distinct modality and leverage this correlation to enhance data cleansing results. An example FA operator is:

- Outlier detection by cross-correlation

4. SDPF Implementation

To develop a specific implementation of SDFP, we mapped the conceptual design of SDPF to a more detailed implementation-level design that considers the specific requirements and capabilities of our implementation platform, i.e., the Logica Real-time Data Management Framework which is developed on top of the Microsoft StreamInsight stream data processing engine. Figure 2 shows the high-level workflow of the implementation-level design for SDPF. As illustrated, with SDPF each cleansing task is implemented in 4 steps (shown on the left, with more detailed breakdown of each step on the right):

1. Planning: At this step, SDPF guides the user to interactively plan the data cleansing task by configuring the operators and modules to be applied to the raw data (see Figure 1). The outcome of this step is a directed acyclic graph of operators (with tuned parameters) that defines the flow of the raw data among the operators.
2. Optimization: At this step, SDPF takes the operator-graph generated at the Planning Step as input, and reconfigures the graph such that the functionality of the graph stays invariant while the performance of the graph is optimized in terms of scalability (i.e., number of data streams and rate of the data in each stream, given limited computing resources)
3. Execution: At this step, the optimized plan is enacted by binding the corresponding operators (implemented on top of the Logica Framework) according to the operator-graph.
4. Management: Finally, SDPF allows users to manage the executed tasks, i.e., reconfigure the plan and re-execute the plan, or delete the task entirely.

Accordingly, we have developed a prototype system based on the aforementioned implementation-level design for SDPF. In this prototype, the Planning Step and Management Step are implemented as a graphical user interface (GUI), and a core engine implements the Execution Step of the design. The Optimization Step is a work in progress and the current implementation of SDPF does not include this step. It is important to note that the Optimization Step is mainly used to further enhance the functionality of the system. Below, we describe the GUI as well as the core engine of our prototype system in more detail.

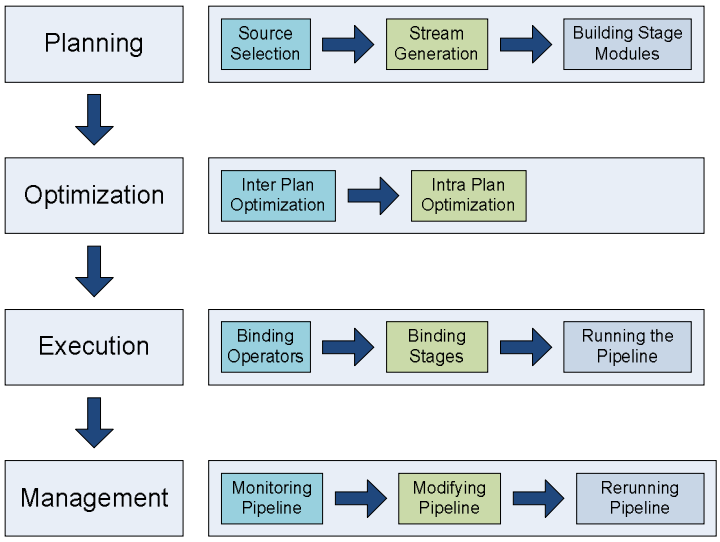

Figure 2: SDPF Implementation-Level Design

4.1 Graphical User Interface (GUI)

The Graphical User Interface (GUI) of our prototype system is implemented as a web-based tool for portability and easy access. It uses PHP (Hypertext Preprocessor) as the server side technology and JS (JavaScript) as the client side technology. Below we describe the functionality of the GUI in more detail.

Figure 3 shows the Plan Management interface of the GUI. Users can use this interface to create new plans and to view, delete, or edit the existing plans running on the core engine of the prototype system.

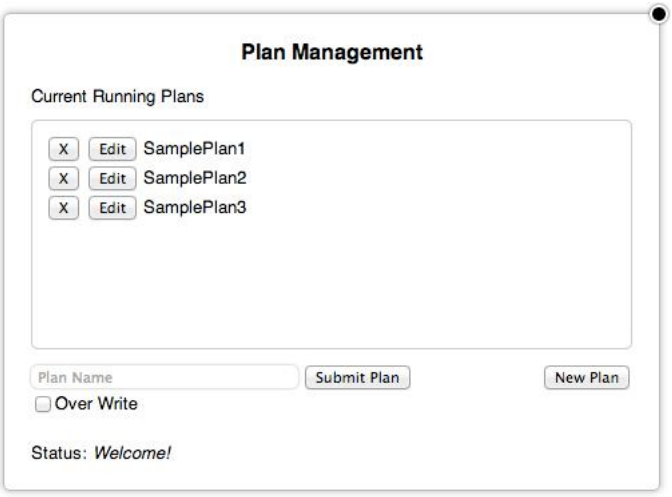

Figure 3: Plan Management Interface

Figure 4 shows the Plan Creation/Editing interface of the GUI, and Figure 5 shows a sample plan developed using this interface. In this interface, the bars on the left and right are the desired input and output tags for the plan. User can select the desired input tags by searching and filtering the tags based on the tag attributes (e.g., location). For each input added to the plan, a corresponding output tag will be automatically added; however, the list of the output tags is editable (tags can be added or deleted by the user on demand). Once the desired lists of inputs and outputs are specified for the plan, user can add as many different operators as needed from any of the five SDPF modules (see Section 3). While an operator is being added to the plan, it can also be configured by setting the operator-specific parameters using the pane shown at the bottom of the interface (see Figure 4). Finally, the input and output sets for the operators can be interconnected by simply clicking on the corresponding operators that feed them or are fed by them, respectively. Once a plan is finalized, user saves the plan and the plan is submitted for execution by the core engine.

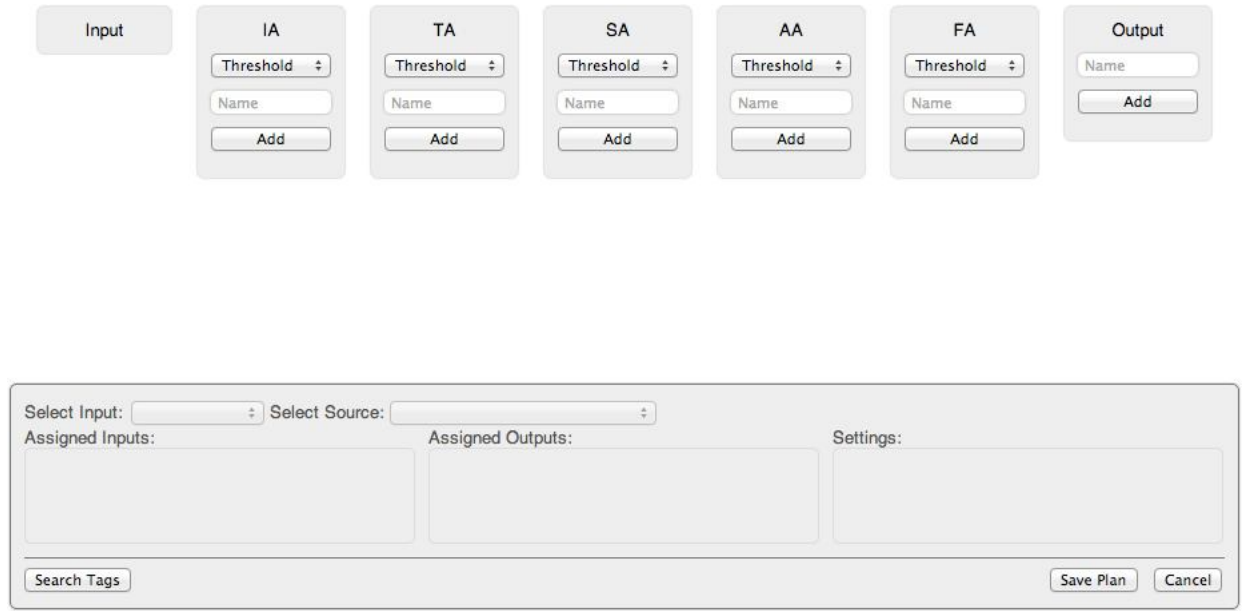

Figure 4: Plan Creation/Editing Interface

Figure 5: An Example Plan Developed on the Plan Creation/Editing Interface

4.2 Core Engine

As mentioned above, the core engine of SPDF ultimately runs on an instance of the Microsoft StreamInsight server. StreamInsight supports two modes of operation. The developer can either embed the StreamInsight server in-process or remotely connect to the server. The current implementation of SPDF supports both modes of operation. Each mode has its own advantages and based on the application, the user can choose which mode to use. When embedding the server in-process, each plan executes in its own process space on the operating system, where there is no naming conflict and each plan can be debugged separately. On the other hand, when the remote server mode is used, multiple plans can share the same resource, which in turn improves the scalability of the framework.

With our SPDF prototype, the input data comes from the OSISoft PI System. At the heart of a PI System, sits a PI Server which is a time-series database. All the data being streamed to the PI System are archived in the PI Server. The data on each stream are stored under a unique tag name. SPDF captures the data from the PI Snapshot *before* they are archived in the PI Server. Thereafter, the cleansing plan is applied to the data, and eventually the cleansed data are stored in the PI Server. Like any other data source/destination, StreamInsight requires input/output adapters to be able to read/write the data. Figure 6 shows the architecture of the core engine implemented in our SDPF prototype, along with a specific data cleansing plan illustrated in the middle box:

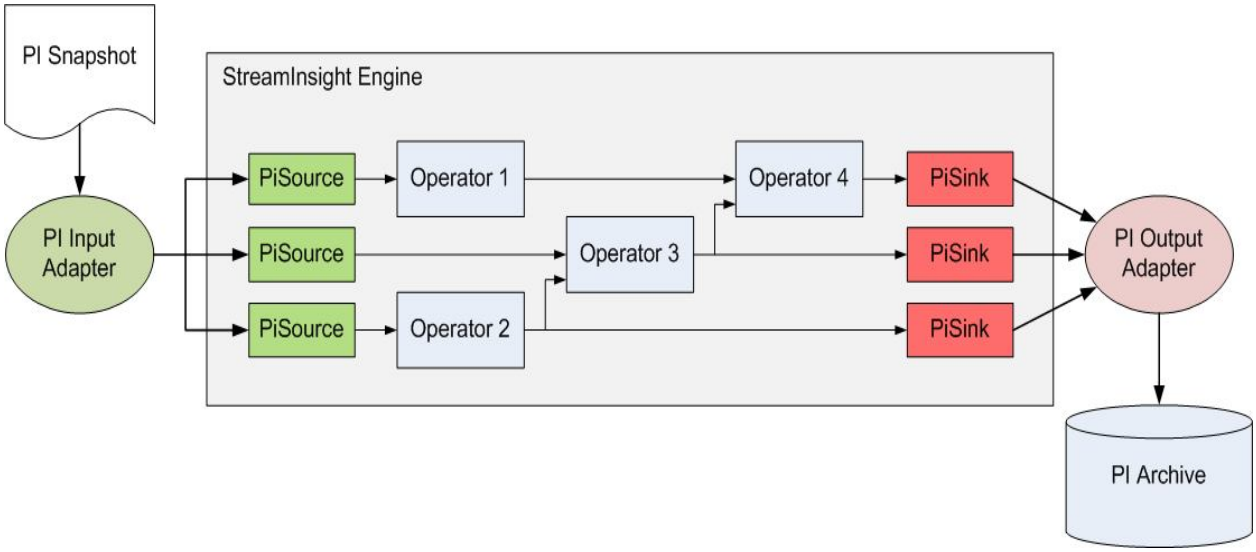

Figure 6: SPDF Core Engine Implementation with Sample Data Cleansing Plan

As shown in Figure 6, for each plan we have only one input adapter. The input adapter reads the data coming in from multiple streams and groups them into one super-stream and feeds it to the engine. The running operators often do not require all the data we are reading from the PI Snapshot. A module called PiSource is responsible for extracting the specific data that the operators need from the super-stream. An alternative is to use multiple input adapters, one per specific data stream its succeeding operators need. Figure 7 shows the latter (7a) and former (7b) options side by side:

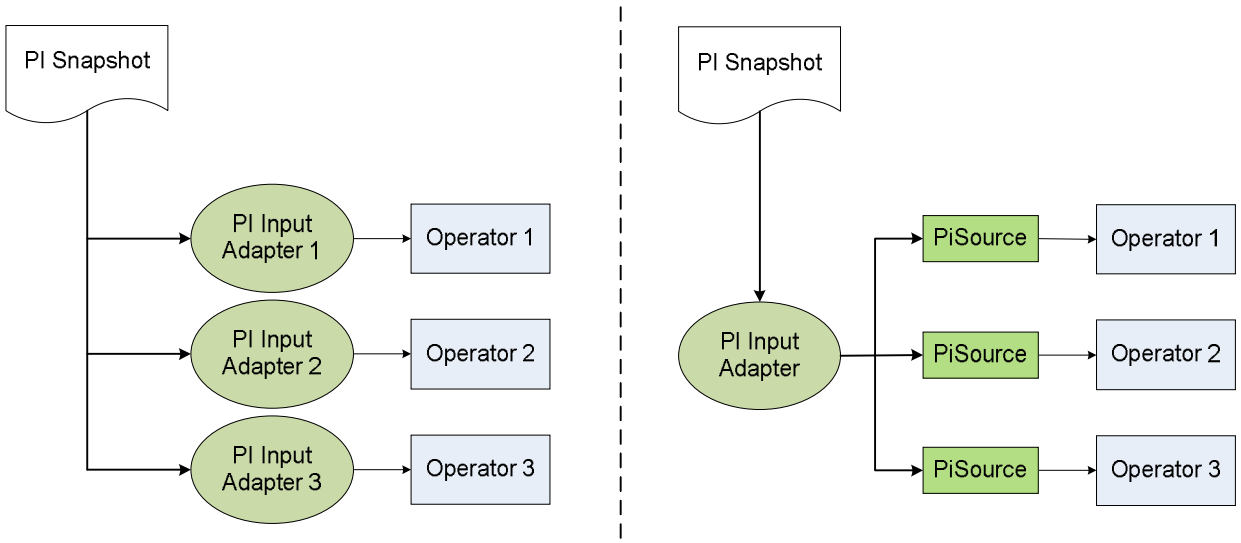

Figure 7: Multiple Input Adapters vs. Single Input Adapter

Adapters demand high system resources. By using only one input adapter to read all the data together and then filtering the data as required, we significantly optimize the use of system resources, which in turn significantly improves the scalability of our SPDF prototype. A similar argument holds for output adapter design.

In addition to extracting the required data from the super-stream (as described above), the PiSource serves as a universal interface. To elaborate, the PiSource module is implemented in the same way the regular operators are implemented. Hence, the output of a PiSource module is compatible with the output of any other operator. As a result, with the current implementation an operator can seamlessly read data either from the output of another operator or from the output of a PiSource module. Therefore, when implementing each operator, one has to only handle one type of stream as the input. On the other hand, as shown in Figure 8, with the alternative (i.e., without the PiSource module) some operators had to read data directly from the output of an input adapter, which is a different type of stream compared to the output of a regular operator. This would have complicated the implementation of the operators, whereas with the current implementation operators are more generic, and hence, more reusable.

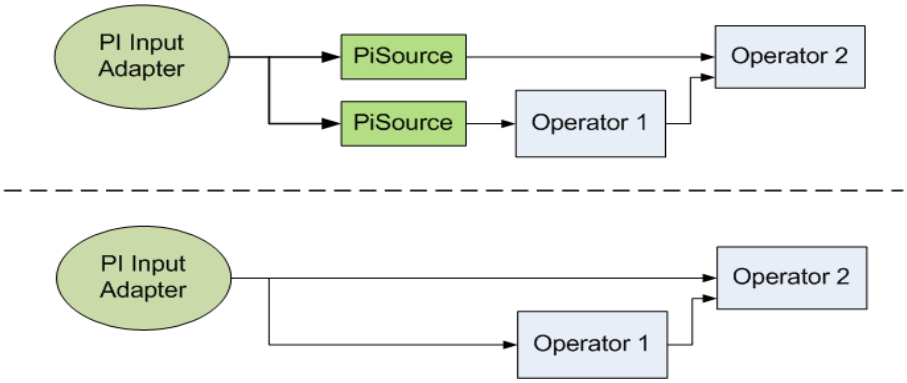

Figure 8: With and Without the PiSource Module

Finally, with our core engine the operators are implemented in two ways. The "simple" operators (i.e., those that can be implemented as typical SQL-like queries) are implemented using LINQ (Language Integrated Queries), which offers higher efficiency. On the other hand, the more complex operators are implemented leveraging the query extensibility features of StreamInsight. These features allow the developer to run more complex algorithms on the streamed data. The query extensibility options of StreamInsight include: User Defined Functions (UDF), User Defined Aggregates (UDA), User Defined Operators (UDO) and User Defined Stream Operator (UDSO). With our prototype implementation, we choose the option that allows for optimal efficiency to implement each operator. Table 1 compares the query extensibility options provided by StreamInsight.

	UDF	UDA	UDO	UDSO
Single Event	Yes	No	No	Yes
Event Window	No	Yes	Yes	No
Statefulness	No	No	No	Yes
Input Parameters	Single input, primitive type	Set of events, any type	Set of events, any type	Single event, any type
Output Parameters	Single result, primitive type	Single result, primitive type	Set of results, any type	Zero to many results, any type
Efficiency	Low	High	High	Very High

Table 1: Query Extensibility Options of StreamInsight

5. Experiments

We conducted two runs of experiments with our SPDF prototype to evaluate the scalability of our framework (other SDPF features, i.e., online processing, configurability, reusability, and comprehensiveness, are qualitative measures). We used randomly generated tags/readings to perform these experiments.

With our first run of experiments, we evaluated the scalability of the framework as the number of input tags grows. Toward this end, we measured the processing time of each data item received from a tag by recording the entry time (i.e., when the data is received from the PI Snapshot) and exit time (i.e., when the data is archived in the PI server) of the data. Clearly, the complexity of the algorithms/operators applied to the data items affects the processing time of the data item. However, throughout this experiment we applied the same data cleansing operators to all data items to isolate the scalability feature as the only standing variable. Figure 9 depicts the average processing time for a data item in plans containing up to 5000 tags.

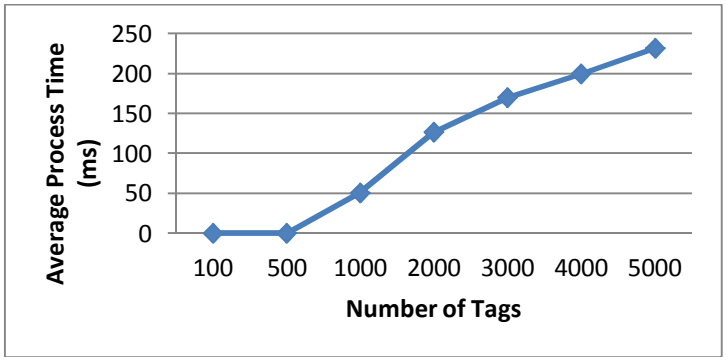

Figure 9: Average Processing Time for a Data Item/Reading as the Number of Tags Grow

With the second run experiments, we evaluate the scalability of the framework as the number of simultaneously running plans grows. Toward this end, we consider a simple plan with 100 input tags, and execute multiple instances of the same plan while measuring the processing time for the input data items. Figure 10 shows the average processing time for a data item when up to 50 simultaneous plans are running. As illustrated, the overhead of adding new plans is negligible.

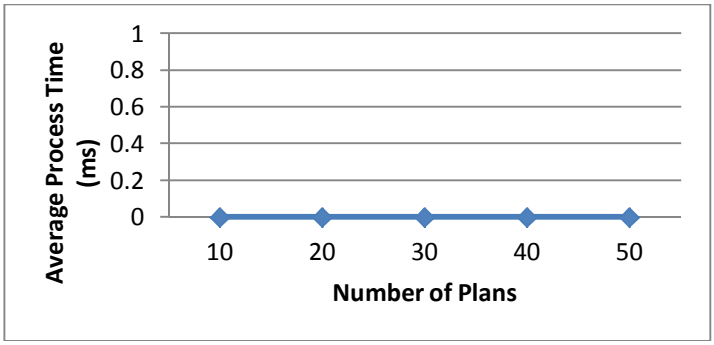

Figure 10: Average Processing Time for a Data Item/Reading as the Number of Plans Grow

6. Conclusion and Future Work

In this paper, we presented a scalable, configurable, reusable, and comprehensive framework for real-time cleansing of upstream operating data. In the future, we plan to extend this work in three ways. First, we will populate SDPF with a variety of operators to study the interaction between operators and the scalability of the system as diversity of the operators increase. Second, currently SDPF can only cleanse the data as it is arrived, and therefore, can only use the past data for cleansing of the current data. We intend to complement the current implementation of SDPF by including capabilities that can retrospectively clean the data arrived in the past. This feature will enhance the quality of data cleansing. Third, we plan to deploy and verify the SDPF capabilities in a real-world scenario.

Acknowledgements

This research was funded by the Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (CiSoft); CiSoft is a joint University of Southern California-Chevron research center. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of University of Southern California and/or Chevron.

References

1. E. Rahm, Hong Hai Do: “Data cleaning: Problems and current approaches”, IEEE Data Eng. Bull., 23(4):3–13, 2000.
2. Helena Galhardas, Daniela Florescu, Dennis Shasha: “Declarative data cleaning: Language, model, and algorithms”, In Proceedings of VLDB, pp. 371–380. 2001.
3. Vijayshankar Raman and Joseph M. Hellerstein: “Potter’s Wheel: An Interactive Data Cleaning System”, In VLDB Journal, pp. 381–390. 2001.
4. M. L. Lee, T. W. Ling, W. L. Low: “A Knowledeg-based Framework for Intelligent Data Cleaning”, Information Systems Journal – Special Issue on Data Extraction and Cleaning, 2001.
5. Ascential. http://www.ascential.com/
6. P. Buonadonna: “Task: Sensor network in a box”, In Proceedings of EWSN. 2005.
7. OSIsoft. http://www.osisoft.com/