Normalization of Unstructured Log Data into Streams of Structured Event Objects

Daniel Tovarňák, Tomáš Pitner

Masaryk University
CSIRT-MU, FI MU

April 11, 2019

Motivation

LOG ANALYSIS VIA COMPLEX EVENT PROCESSING (CEP)

Data stream processing: real-time data processing paradigm

- commonly used to deal with high-velocity data

CEP: detection of complex patterns in streams of data elements

- visions for use in real-time log analysis, especially security monitoring
- as opposed to full-text indexing and column-based indexing of log data

Event objects: actual representation of the elements in the stream

- expected to be properly structured and described via an explicit data schema
- much like in RDBMS

Unstructured log entries ≠ event objects

- semi-structured log entries ≠ event objects

LOGGING AND LOG DATA – 5VS OF BIG DATA

Traditional manifestation – log files with arbitrary text messages

Value: widely-used source of monitoring information

- debugging, troubleshooting, fault detection, security, forensics, compliance

Veracity: poor-quality, unstructured nature, complicated analysis

- `2017-07-23T19:35:45Z [0] ERR!: Jack said he will take care of this!`
- this stems from the way logs are generated – messages in natural language

Variability: pervasive devel. practice spanning SW on all IT layers

- data source and data format heterogeneity

Velocity + Volume: can exceed 100,000 entries/sec, 1 MB/s per node

- HP company – 1×10^{12} entries/day generated, 3×10^{9} entries/day processed

BRIDGING THE GAP BY NORMALIZATION

Data integration perspective: bridge the gap by normalization

- known pattern to improve interoperability
- missing structure is added via transformation and enrichment
- overall heterogeneity is eliminated thanks to a single canonical form

Normalization: unification of data on any of its 4 layers

- data structures
- data types
- data representation
- transport

Thesis Goal:

Improve **the way log data can be represented and accessed** by designing *[algorithms, approaches, concepts]* that would enable the normalization of unstructured log data into streams of event objects in order to allow the log analysis practitioners to analyze them in a unified and interoperable manner.

THESIS GOAL (SIMPLIFIED)

```
Dec 03 2016 10:03:44 [147.251.11.100] --- INFO: User bob logged in
2016-12-03T10:03:45Z 147.251.20.110 sshd[1551]: session closed for user alice
Dec 03 2016 10:03:46 [147.251.10.125] --- WARN: User alice failed to log in
3.12.2016 10:03:47 147.251.19.160 [Super.java]: {service=Billing, status=0x2A}
```

↓ NORMALIZATION: 1 + 3 RESEARCH GOALS ↓

```
    UserLogin() {ts=...424, host="147.251.11.100", success=True, user="bob"}
SessionClosed() {ts=...425, host="147.251.20.110", user="alice", app="sshd"}
    UserLogin() {ts=...426, host="147.251.10.125", success=False, user="alice"}
 ServiceCrash() {ts=...427, host="147.251.19.160", service="Billing", code=42}
```

⇓ `UserLogin` ⇓ ⇓ `SessionClosed` ⇓ ⇓ `ServiceCrash` ⇓

```
CREATE MAP SCHEMA UserLogin(host string, success boolean, user string);

SELECT host, user, count(*) AS attempts
FROM UserLogin.win:time(30 sec)
WHERE attempts > 1000, success=false
GROUP BY host, user
```

Proactive Normalization

RESEARCH GOAL 1 – KEY FINDINGS

It is not overly hard to log semi-structured log messages (JSON)

- we have developed prototype mechanisms for Ruby, Python, C++11, and Java

It is hard to generate explicit data schemes describing them

- static code analysis at compile time had to be used for our 2 Java prototypes

Evaluation and profiling: 66% performance overhead per statement

- caused by data representation
- serialization + appending phase (twice the size ~ twice the time)

We do not expect massive use of structured logging in near future

- how do you convince someone to write *”clean code”*?
- several studies suggest that the developers are unable to properly use even traditional logging mechanisms based on string parameterization

Reactive Normalization

LOG ABSTRACTION (SEPARATION)

Log Messages	⇒ Message Types ⇒	Regular Expressions
`User Jack logged in`		
`User John logged out`		
`Service sshd started`	`User * logged * : [$1, $2]`	`User (\w+) logged (\w+)`
`User Bob logged in`	`Service * started : [$1]`	`Service (\w+) started`
`Service httpd started`		
`User Ruth logged out`		

```
LOG.info("User {} logged {}", user, action);
↓
Dec 03 2016 10:03:44 -- INFO: User bob logged in
⤳
User (?<user>\w+) logged (?<action>\w+)
```

Log abstraction is a two-tier procedure:

- message type discovery
- pattern-matching via regular expressions

RESEARCH GOAL 2 – MESSAGE TYPE DISCOVERY

Manual discovery: tiresome process, which leads to errors

- automated approaches are necessary

Static code analysis: perfectly possible

- we were able to discover approx. **4500 message types** in Hadoop source code
- source code is not always available (e.g. for network devices)

Data mining: use already generated log messages (historical data)

- 9 existing approaches were studied, e.g. *SLCT, IPLoM, logSig, N-V, ...*

Existing approaches: accuracy and usability issues

- e.g. message types overlap, hard fine-tuning, tokenization by single character

Our goal:
Design a *message type discovery algorithm* addressing these issues.

EXTENDED NAGAPPAN-VOUK ALGORITHM

```
   Service sshd started  | [4,2,4]
   Service httpd started | [4,2,4]
   Service sshd started  | [4,2,4]
   Service httpd started | [4,2,4]
-----------------------------------
   Service   *   started
```

	1	**2**	**3**
Service	4	0	0
httpd	0	2	0
sshd	0	2	0
started	0	0	4

Method of *n*-th percentile: frequency table + percentile threshold

- `[4,2,4]` in example is log message *score*
- word is a variable, if it has a frequency lower than *n*-th percentile of *score*

Post-processing to improve accuracy and usability

1. eliminate overlapping message types by merging
2. identify multi-word variable positions

DISCOVERED PATTERN-SET EXAMPLE

```
Start processing (xor) Jen=user          Service sshd:22 started
User John logged out                     Start processing (xor) Daniel=user
User Bob logged in                       User Ruth logged out
Start processing (xor) Thomas=user       Start processing (xor) Tom Sawyer=user
Service httpd:8080 started               Start processing (nor) Root=user
```

⇓ `percentile=60, delimiters=' :=\(\)'` ⇓

```
regexes: # regex tokens
  INT:         [integer, "[0-9]+"]
  BOOL:        [boolean, "\btrue\b|\bfalse\b"]
  WORD:        [string,  "[0-9a-zA-Z]+"]
  ARBITRARY:   [string,  "[^ \n\r]+"]
  MWRD_1_2:    [string,  "[^ \n\r]+([\\s][^ \n\r]+){0,1}"]

patterns: # patterns describing the message types
  grp0:
    mt1: 'User %{WORD:var1} logged %{WORD:var2}'
    mt2: 'Start processing (%{WORD:var1}) %{MWRD_1_2:var2}=%{WORD:var3}'
    mt3: 'Service %{WORD:var1}:%{INT:var2} started'
```

EVALUATION, RESULTS AND FINDINGS

Discovered message types partition the log messages into groups

F-measure: common accuracy metric in IR, higher is better

- $F = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$ – how "close" our grouping is to the ground truth

Ground truth: 5 real-life data-sets, MTs manually discovered

- P. He, et al. *An Evaluation Study on Log Parsing and Its Use in Log Mining*
- best average F-measure (*IPLoM*) – ***0.892***

	BGL	**HPC**	**HDFS**	**Zook.**	**Proxif.**	***AVG***
$n = 50$, $d =$ *space*	0.8556	0.8778	1.0000	0.7882	0.8162	*0.86756*
$n = 50$, $d =$ *default*	0.9251	0.9861	1.0000	0.9999	0.8547	***0.95316***
$n = 15$, $d =$ *default*	0.9191	0.9861	0.6965	0.9182	0.8220	*0.86838*
$n = 85$, $d =$ *default*	0.4949	0.9856	1.0000	0.9979	0.8547	*0.86662*
$n = 50$, $d =$ *best**	0.9985	0.9861	1.0000	0.9999	1.0000	***0.99690***

RESEARCH GOAL 3 – MULTI-PATTERN MATCHING

Appropriate pattern must be used for each log message

- variable positions must be captured
- an appropriate structure must be returned

Naïve iteration is extremely slow (yet it is still widely-used!)

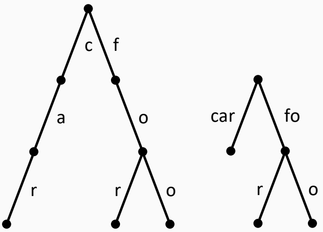
- there can be thousands of patterns

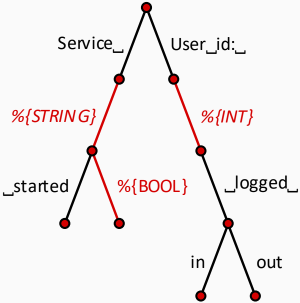
Multi-regex matching is not supported in common libraries

- *set of regexes* ⟶ *NFA* ⟶ *simulate input*
- advanced features are hard to implement, e.g. sub-match capturing
- Google's RE2 – 30k lines of C++
- possible limits in terms of memory

Our goal:
Design an alternative *multi-pattern matching approach* that scales with respect to the number of patterns.

REGEX TRIE





Search: depth-first traversal w.r.t. input log message

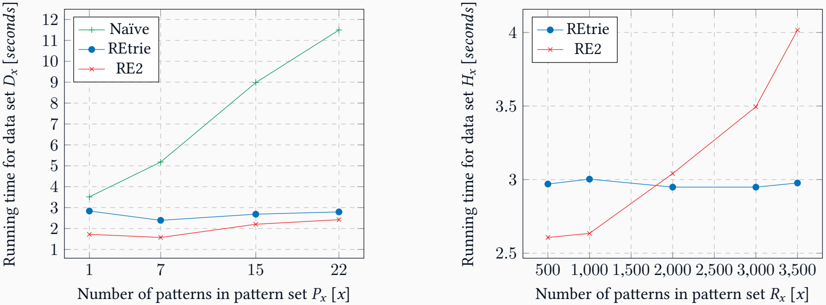
- variables are captured during traversal
- **match** when leaf is reached and no input is left
- **non-match** when traversal cannot continue

EVALUATION, RESULTS AND FINDINGS

Scalability tests: matching of 1.1M entries on single core

- Naïve + REtrie in Erlang, RE2 in C++ as control
- real-world log entries and pattern-sets

1.9 million matches/sec on 8 cores



E2E LOG DATA NORMALIZATION

Log abstraction is crucial, but not the only task of normalization:

- Input adaptation – *TCP, HTTP, UDP*
- Deserialization – *text, JSON, CSV, XML*
- Parsing – *regex parsing, cleansing*
- Transformation – *string manipulation, strucutre manipulation*
- Enrichment – *adding structure, dictionaries*
- Serialization – *JSON, Avro*
- Output adaptation – *messaging systems*

These tasks must be logically combined to achieve the desired results

RESEARCH GOAL 4A,4B – LOG DATA NORMALIZATION

How do we describe and execute some desired normalization logic?

- log data normalization is a specialized domain with highly-specific tasks
- domain-specific languages are believed to fit such scenarios

Domain-specific language (DSL)

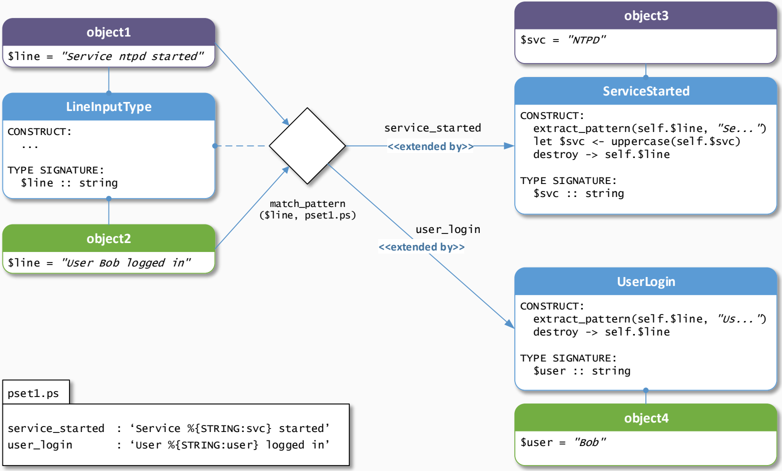
- high-level modeling of domain knowledge
- high expressiveness and rapid development for domain experts

Existing approaches: orientation on untyped transformations

- log management tools – *rsyslog, syslog-ng, Logstash, Fluentd, nxlog*
- the respective DSLs lack typing support
- our goal is to produce structured data – the notion of data types is essential

Our goal:
Design a *log data normalization approach* that is object-oriented and statically-typed. Also, design a *DSL* implementing the approach, and a *normalization engine* able to execute it.

NORMALIZATION VIA PROTOTYPE-BASED INHERITANCE



object1

`$line = "Service ntpd started"`

LineInputType

```
CONSTRUCT:
  ...

TYPE SIGNATURE:
  $line :: string
```

object2

`$line = "User Bob logged in"`

match_pattern ($line, pset1.ps)

service_started <<extended by>>

user_login <<extended by>>

object3

`$svc = "NTPD"`

ServiceStarted

```
CONSTRUCT:
  extract_pattern(self.$line, "Se...")
  let $svc <- uppercase(self.$svc)
  destroy -> self.$line

TYPE SIGNATURE:
  $svc :: string
```

UserLogin

```
CONSTRUCT:
  extract_pattern(self.$line, "Us...")
  destroy -> self.$line

TYPE SIGNATURE:
  $user :: string
```

object4

`$user = "Bob"`

pset1.ps

```
service_started  : 'Service %{STRING:svc} started'
user_login       : 'User %{STRING:user} logged in'
```

New objects are created by reusing existing objects – prototypes

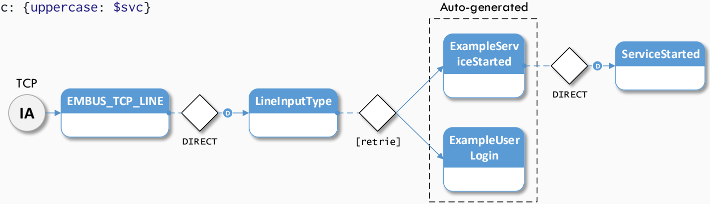
- normalization logic is a series of inheritances

YAML-Based Domain-Specific Language

```
input Syslog5544 produces EMBUS_TCP_LINE:
  '@adapter': {module: embus_tcp_line, args: {port: 5544}}

type LineInputType extends EMBUS_TCP_LINE:
  '@in': DEFAULT
  '@do':                          # # # # # # # # # # # # # # # # # # # # # # # # # # #
    $line: {trim: $line}          # pset1.ps:
  '@out':                         # ...
    [retrie]:                     # example:
      source: $line               #   service_started : ‘Service %{STRING:svc} started’
      set: pset1.ps               #   user_login      : ‘User %{STRING:user} logged in’
      group: example              # # # # # # # # # # # # # # # # # # # # # # # # # # #
      type: extends

# 'example' + 'service_started' = ExampleServiceStarted
type ServiceStarted follows [ExampleServiceStarted]:
  '@in': DEFAULT
  '@bind':
    $svc: {t: string}
  '@do':
    $svc: {uppercase: $svc}
```



DSL + NORMALIZATION ENGINE

The DSL is statically typed and it uses type inference

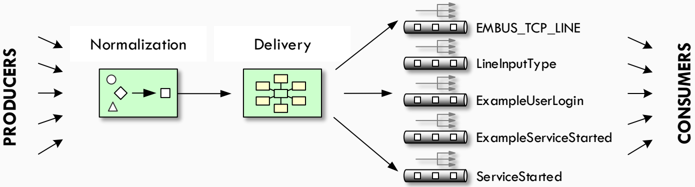
- it can be determined, which object types the normalization logic will produce
- data schemes are generated automatically

Basic transformation functions + automated structure extraction

- string manipulation (`split`, `trim`, `replace`)
- boolean operations (`eq`, `gt`, `lt`)
- structure extraction (`retrie`, `tree-struct`)

Normalization engine executes the DSL

- manages data input, normalization, and output into data streams
- highly-parallel implementation in Erlang



RESULTS AND FINDINGS

E2E throughput evaluation for basic normalization logic

- throughput of the solution as a whole – log abstraction (500 patterns)
- approx. 220,000 normalized log entries/second on an 8-core server
- engine can handle 16k concurrent TCP connections

DSL preliminary applications (described logic)

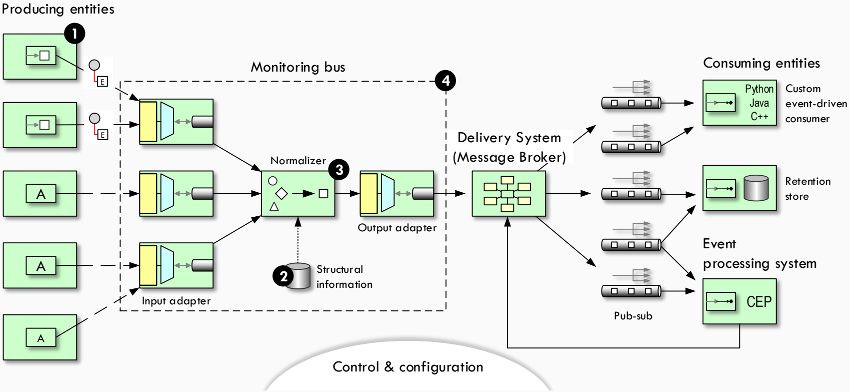
- unstructured Syslog – *Sendmail logs, Debian logs*
- XML log entries – *Windows Event Logs*
- JSON log entries – *BRO intrusion detection, IP flows*

The normalized event objects can be directly consumed as streams!

- retention stores – e.g. *Elasticsearch, HDFS*
- data stream processing solutions – e.g. *Apache Spark, Apache Storm*
- Complex Event Processing solutions – e.g. *Esper, WSO2 CEP*

Summary

DEDMA + FUTURE WORK



FINAL TALLY

9 publications

23 undergraduate students

2+1 research project applications (security)

- *"Security Cloud"* (TA04010062/2014)
- *"KYPO Cyber Range"* (VI20162019014)
- *"CSIRT-MU"* (day-to-day operation)

∞ friends and colleagues

SELECTED PUBLICATIONS & THANK YOU!

- T. Jirsik, M. Cermak, D. Tovarnak, and P. Celeda. *Toward Stream-Based IP Flow Analysis.* IEEE Communications Magazine, 2017.
  [Q1 Journal, IF 10.435, 20%]
- D. Tovarnak. *Practical Multi-Pattern Matching Approach for Fast and Scalable Log Abstraction.* ICSOFT '16, 2016.
  [CORE B Ranking, 100%]
- D. Tovarnak and T. Pitner. *Continuous Queries Over Distributed Streams of Heterogeneous Monitoring Data in Cloud Datacenters.* ICSOFT '14, 2014.
  [CORE B Ranking, 90%]
- D. Tovarnak, A. Vasekova, S. Novak, and T. Pitner. *Structured and Interoperable Logging for the Cloud Computing Era: The Pitfalls and Benefits.* UCC '13, 2013.
  [IEEE/ACM Conference Proceedings, 80%]
- D. Tovarnak and T. Pitner. *Towards Multi-tenant and Interoperable Monitoring of Virtual Machines in Cloud.* SYNASC '12, 2012.
  [CORE C Ranking, 90%]

ACKNOWLEDGEMENT

EUROPEAN UNION
European Structural and Investment Funds
Operational Programme Research,
Development and Education

MŠMT
MINISTRY OF EDUCATION,
YOUTH AND SPORTS