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 – $1 \times 10^{12}$ entries/day generated, $3 \times 10^9$ 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.
The Thesis Goal (Simplified)

Dec 03 2016 10:03:44 [147.251.11.100] --- INFO: User bob logged in
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 ↓

<table>
<thead>
<tr>
<th>Function</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserLogin()</td>
<td>{ts=...424, host=&quot;147.251.11.100&quot;, success=True, user=&quot;bob&quot;}</td>
</tr>
<tr>
<td>SessionClosed()</td>
<td>{ts=...425, host=&quot;147.251.20.110&quot;, user=&quot;alice&quot;, app=&quot;sshd&quot;}</td>
</tr>
<tr>
<td>UserLogin()</td>
<td>{ts=...426, host=&quot;147.251.10.125&quot;, success=False, user=&quot;alice&quot;}</td>
</tr>
<tr>
<td>ServiceCrash()</td>
<td>{ts=...427, host=&quot;147.251.19.160&quot;, service=&quot;Billing&quot;, code=42}</td>
</tr>
</tbody>
</table>

DOWN UserLogin DOWN SessionClosed DOWN ServiceCrash DOWN

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)

<table>
<thead>
<tr>
<th>Log Messages</th>
<th>Message Types</th>
<th>Regular Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Jack logged in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User John logged out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service sshd started</td>
<td>User * logged * : ([$1, $2])</td>
<td>User (\w+) logged (\w+)</td>
</tr>
<tr>
<td>User Bob logged in</td>
<td>Service * started : ([$1])</td>
<td>Service (\w+) started</td>
</tr>
<tr>
<td>Service httpd started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Ruth logged out</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>httpd</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>sshd</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>started</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Start processing (xor) Jen=user</th>
<th>Service sshd:22 started</th>
</tr>
</thead>
<tbody>
<tr>
<td>User John logged out</td>
<td>Start processing (xor) Daniel=user</td>
</tr>
<tr>
<td>User Bob logged in</td>
<td>User Ruth logged out</td>
</tr>
<tr>
<td>Start processing (xor) Thomas=user</td>
<td>Start processing (xor) Tom Sawyer=user</td>
</tr>
<tr>
<td>Service httpd:8080 started</td>
<td>Start processing (nor) Root=user</td>
</tr>
</tbody>
</table>

\[ \text{percentile}=60, \text{delimiters}= ' :=\(\(\)\)' \]

**regexes:** # regex tokens

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

**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

\[ F = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \]

– 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

<table>
<thead>
<tr>
<th></th>
<th>BGL</th>
<th>HPC</th>
<th>HDFS</th>
<th>Zook.</th>
<th>Proxif.</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 50, d = \text{space} )</td>
<td>0.8556</td>
<td>0.8778</td>
<td>1.0000</td>
<td>0.7882</td>
<td>0.8162</td>
<td>0.86756</td>
</tr>
<tr>
<td>( n = 50, d = \text{default} )</td>
<td>0.9251</td>
<td>0.9861</td>
<td>1.0000</td>
<td>0.9999</td>
<td>0.8547</td>
<td>0.95316</td>
</tr>
<tr>
<td>( n = 15, d = \text{default} )</td>
<td>0.9191</td>
<td>0.9861</td>
<td>0.6965</td>
<td>0.9182</td>
<td>0.8220</td>
<td>0.86838</td>
</tr>
<tr>
<td>( n = 85, d = \text{default} )</td>
<td>0.4949</td>
<td>0.9856</td>
<td>1.0000</td>
<td>0.9979</td>
<td>0.8547</td>
<td>0.86662</td>
</tr>
<tr>
<td>( n = 50, d = \text{best}^* )</td>
<td>0.9985</td>
<td>0.9861</td>
<td>1.0000</td>
<td>0.9999</td>
<td>1.0000</td>
<td>0.99690</td>
</tr>
</tbody>
</table>
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 $\rightarrow$ NFA $\rightarrow$ 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, structure manipulation
- Enrichment – adding structure, dictionaries
- Serialization – JSON, Avro
- Output adaptation – messaging systems

These tasks must be logically combined to achieve the desired results
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

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}  
      '@out':
          [retrie]
              source: $line
              set: pset1.ps
              group: example
              type: extends

# 'example' + 'service_started' = ExampleServiceStarted

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

type ServiceStarted follows [ExampleServiceStarted]:
  '@in': DEFAULT
  '@bind':
      $svc: {t: string}
  '@do':
      $svc: {uppercase: $svc}
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

Monitoring bus

Producing entities

Input adapter

Structural information

Normalizer

Output adapter

Delivery System (Message Broker)

Consuming entities

Control & configuration

Retarget adapter

Control & configuration

Pub-sub

CEP
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!


- D. Tovarnak and T. Pitner. *Continuous Queries Over Distributed Streams of Heterogeneous Monitoring Data in Cloud Datacenters*. ICSOFT ’14, 2014. [CORE B Ranking, 90%]


ACKNOWLEDGEMENT

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

MINISTRY OF EDUCATION,
YOUTH AND SPORTS