Rahmenwerk zur Ausreißererkennung in Zeitreihen von Software-Laufzeitdaten

Florian Lautenschlager,[1] Andreas Kumlehn,[2] Josef Adersberger,[1] Michael Philippsen[2]

[1]QAware GmbH
Aschauer Str. 32
81549 München

[2]Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Lehrstuhl für Programmiersysteme
Martensstraße 3, 91058 Erlangen

{florian.lautenschlager|josef.adersberger}@qaware.de
{andreas.kumlehn|michael.philippsen}@fau.de

Abstract: Auch das beste Software-System kann Anomalien im Laufzeitverhalten aufweisen, die nach einiger Zeit in Fehlerzustände münden können. Bekannte Werkzeuge überwachen kontinuierlich, ob Laufzeiten wie z.B. CPU-Last oder Antwortzeiten manuell gesetzte Schwellwerte überschreiten. Das hat zwei Nachteile: (a) Starr vorgegebene Schwellwerte und damit starre Festlegungen, ab wann ein Messwert einen Ausreißer markiert, sind bei saisonalen Schwankungen oder Lastspitzen sowie externer Einflüsse prinzipiell ungeeignet. (b) Oft wird erst nach Auftreten des Fehlerfalls rückblickend im Protokoll der Laufzeitdaten nach einer Anomalie gesucht, die den Fehlerfall erklärt. D.h., es kann erst a posteriori definiert werden, was ein Ausreißer ist, und welche Messwerttypen zu diesem beitragen. Das Papier stellt daher ein Rahmenwerk zur Ausreißererkennung vor, das offline auf einer Vielzahl von protokollierten Laufzeitdaten arbeitet und wegen des Umfangs auf einer neuartigen effizienten Speicherung und Analyse von Zeitreihen basiert. Die Evaluation zeigt die Effizienz der Zeitreihenspeicherung sowie von anspruchsvollen Ausreißererkennern.

1 Einleitung

Software-Systeme können Laufzeitanomalien aufweisen, die es in einen fehlerhaften Zustand versetzen und sich meist in Ausreißern zeigen, bei denen sich Messwerte signifikant von Laufzeitdaten der selben Gattung unterscheiden. Etablierte Ausreißererkenner überwachen kontinuierlich, ob CPU-Last, Antwortzeiten o.ä. manuell gesetzte Schwellwerte überschreiten. Das hat zwei Nachteile: (a) Starr vorgegebene Schwellwerte und damit starre Festlegungen, ab wann ein Messwert einen Ausreißer markiert, sind prinzipiell ungeeignet. Die Unterscheidung zwischen Normalzustand und Ausreißer, hängt vom System, dessen Konfiguration und von äußeren Einflüssen, wie Lastspitzen oder parallel laufenden Aktivitäten, ab. (b) Oft wird erst nach Auftreten des Fehlerfalls rückblickend, meist manuell, im umfangreichen Protokoll der Laufzeitdaten nach einer Anomalie gesucht. D.h., es ist erst a posteriori klar, was ein Ausreißer ist und welche Messwerttypen zu diesem beitragen. Etablierte Werkzeuge protokollieren Laufzeitdaten jedoch meist nur über kurze Zeiträume oder in stark aggregierter Form.

Jeder Ausreißererkennung liegt eine Hypothese zugrunde, die festlegt, wann Laufzeitdaten als Ausreißer zu bewerten sind. Je mehr protokollierte Laufzeitdaten vorliegen, desto zuverlässiger ist ein Ausreißer als solcher erkennbar. Wir stellen ein Rahmenwerk zur Umsetzung zuverlässiger Ausreißererkenner vor, die über Tage, Wochen oder sogar Monate gesammelte, und ohne Aggregation gespeicherte Messwerte effizient analysieren.

Im Folgenden skizzieren wir zunächst Ausreißererkennungsverfahren und deren Zuverlässigkeit. Die Abschnitte 3 und 4 beschreiben das Rahmenwerk zur Ausreißererkennung und die benötigte effiziente Zeitreihenanalyse, ehe Abschnitt 5 unsere Lösung evaluiert. Nach der Diskussion verwandter Arbeiten schließen wir mit einem kurzen Ausblick.

2 Verfahren zur Erkennung und Bewertung von Ausreißern

Prinzipiell gibt es unterschiedliche, nachfolgend skizzierte, Herangehensweisen zur Ausreißererkennung in Laufzeitdaten. Einen Überblick liefern Chandola et al. [CBK07].

Der Vergleich eines gemessenen Werts mit einem **festen Schwellwert** ist eine einfache Ausreißererkennung, die für normierte Werte und Ressourcengrenzen geeignet ist und von Werkzeugen zur kontinuierlichen Überwachung eingesetzt wird [Nag]. **Dynamische Schwellwerte** basieren auf protokollierten Laufzeitdaten, z.B. als Durchschnitt, Median oder Standardabweichung. Im Unterschied zu festen Schwellwerten bestimmen historische Laufzeitdaten den Grenzwert [App, VHK14]. **Kombinierte Verfahren** steigern die Zuverlässigkeit der Ausreißererkennung durch mehrschrittige Bewertung. Beispielsweise wird ein Messwert erst zum Ausreißer, wenn er einen dynamischen Schwellwert öfter überschreitet, als dies eine fest vorgegebene Häufigkeitsschwelle erlaubt [DF14]. **Machine Learning** wie distanz- und dichtebasierte Verfahren, Clustering, Klassifikation etc. werden bislang selten zur a posteriori Ausreißererkennung benutzt [ABA11].

Einfache Erkennungsverfahren (Schwellwerte) sind also nur in manchen Fällen hinsichtlich Precision und Recall zuverlässig genug, jedoch sind dann die Anforderungen an die Laufzeitdaten gering. Aufwändigere Ausreißererkenner (kombinierte Verfahren, Machine Learning) sind deutlich zuverlässiger, stellen aber weitaus höhere Anforderungen an die Laufzeitdaten und kommen deshalb in der Praxis bislang kaum zum Einsatz.

3 Rahmenwerk zur Ausreißererkennung

Unser Rahmenwerk ist eine Lösung für dieses Spannungsfeld. Wie Abb. 1 darstellt, kann der Nutzer einfache Ausreißererkenner benutzen, falls diese zur Analyse ausreichen. Ebenso können aufwändigere oder problemspezifische Verfahren realisiert und in das Rahmenwerk integriert werden. Das ist erstens flexibler als bekannte Werkzeuge, weil je nach vorliegenden Umständen die am besten geeigneten Erkenner verwendbar sind, anstatt sich auf ein bestimmtes Verfahren beschränken zu müssen. Zweitens benutzen die Erkennungsverfahren gemeinsame Basisfunktionen zur Analyse der Laufzeitdaten. Dazu

gehören Filter zur Abbildung fester Schwellwerte, statistische Grundfunktionen (Median, Standardabweichung, Perzentile, Ankunftsraten) sowie Distanzmaße (z.B. Manhatten-Distanz). Aufwändige Erkenner normalisieren Zeitreihen, ändern Zeitachsen durch Stauchen, Dehnen, Verschieben etc. Der zweite Vorteil unseres Rahmenwerks ist daher, dass wir die gemeinsamen Funktionen in eine effizient implementierte Bibliothek auslagern und nur einmal für alle Verfahren realisieren. Der dritte Vorteil besteht darin, dass wir auch den Zugriff auf die Zeitreihen von Laufzeitdaten herauslösen und optimieren, siehe Abschnitt 4. Nur dadurch wird es überhaupt praktikabel, die anspruchsvollen und zuverlässigen Verfahren einzusetzen. Ansonsten wäre der Zugriff auf die Laufzeitdaten zu langsam für eine interaktive Anomalie- und Fehlerdetektion.

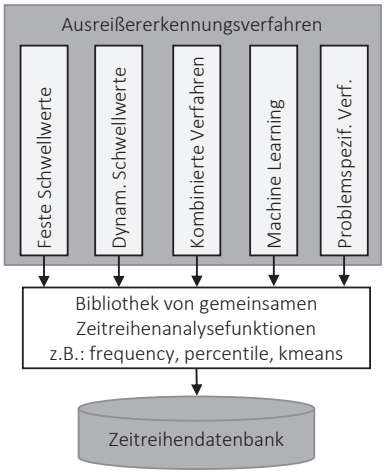

Abbildung 1: Rahmenwerk

4 Effiziente Zeitreihenspeicherung

Das Rahmenwerk erfordert eine effizientere Speicherung und einen schnelleren Zugriff auf Zeitreihen als dies klassische SQL Datenbanken, wie z.B. MySQL [MyS], leisten, die nicht für Zeitreihen optimiert sind und fünf wesentliche Probleme haben: (1) Sie unterstützen die Trennung von massenhaft auftretenden Zeitstempel-Wert-Tupeln und den diese beschreibenden Metadaten nur unzureichend. (2) Sie belegen unnötig viel Platz zur Zeitreihenspeicherung, da sie zeilen- anstatt spaltenorientiert komprimieren und so Redundanzen in den Wert-Tupeln schlechter nutzen. (3) Die Importzeiten durch Indexkonstruktionen sind langsam und somit für die häufige Protokollierung von Zeitreihen ungeeignet. (4) Sie liefern angefragte Tupel in unkomprimierter Form aus, was zu hoher Latenz führt, wenn Ausreißererkenner Datenmassen analysieren. (5) Komplexe Ausreißererkenner erfordern im Allgemeinem mehrere Einzeldatenbankabfragen und damit mehrere latenzbehaftete Ergebnisauslieferungen.

Zeitreihendatenbanken verwenden zwar passende Kompressionen (2) und vermeiden Indexberechnungen (3), indem sie auf Schlüssel-Wert-basierten Datenbanken oder Speichern für Zeichenketten aufsetzen (InfluxDB [Inf] auf LevelDB [Lev], OpenTSDB [Ope] auf HBase [Apab], KairosDB [Kai] auf Cassandra [Apaa], tsdb [DMF12] auf BerkeleyDB [OBS99]). Die übrigen Probleme bestehen weiterhin. Folgender Boxplot-Ausreißererkenner erkennt Werte, die einen dynamischen Schwellwert, berechnet aus dem [25%;75%]-Intervall der Messwerte und einem Faktor 1,5, übersteigen. InfluxDB benötigt hierfür zwei `SELECT`-Anfragen, die unkomprimierte Tupel übertragen, und eine Zwischenrechnung.

```
SELECT percentile(responseTimes,25), percentile(responseTimes,75) FROM metrics
out = (q3 - q1) * 1.5 + q3 //Manuelles Ausrechnen des Boxplot-Schwellwerts
SELECT responseTimes FROM metrics WHERE responseTimes >= out
```

Unsere Zeitreihenspeicherung OTSS (Outlier Time Series Storage) ist für die Analyse von Tupelmassen und für eine Ausreißererkennung auf protokollierten Laufzeitdaten optimiert und adressiert die verbleibenden Nachteile. Dazu bündeln und komprimieren wir Tupelfolgen mit gzip, vglb. einer spaltenorientierten Komprimierung, in sog. Dokumenten, die mit Metadaten angereichert sind (1, 2). Die Länge einer Tupelfolge pro Dokument wird entweder durch ein Zeitintervall oder eine feste Tupelanzahl vorgegeben. Somit gilt für die Beziehung zwischen Dokument und Wert-Tupel-Typ $N : 1$. Die Datenbank Apache Solr [Apac], in der wir die Dokumente speichern, erlaubt eine gezielten Zugriff über die Metadaten (3). Die von diversen Ausreißererkennern angeforderten Messdaten werden komprimiert, also latenzarm (4) zur weiteren Auswertung übertragen, die nur die tatsächlich benötigten Tupel dekomprimieren muss. OTSS benötigt für das Beispiel nur eine `Query` mit der erwähnt geringen Latenz (5). Nach Übertragung der komprimierten Dokumente erfolgt das Dekomprimieren der Tupel implizit, wenn die Ausreißererkennung die Bibliotheksfunktionen benutzt:

```
Analysis analysis = new Analysis(new Query("responseTimes"));
Stream<TimeSeries> outliers = analysis.filter(
  analysis.iqr().multiply(1.5).add(analysis.percentile(0.75)),
  FilterStrategy.GREATER_EQUALS).result();
```

5 Evaluation

Wir evaluieren zuerst isoliert den externen Speicherplatzbedarf und die Dokumentzugriffszeiten von OTSS. Darauf aufbauend vermessen wir dann eine Ausreißererkennung inklusive Zeitreihendatenbankabfrage. Alle Messungen erfolgten auf einem Intel QuadCore i7-4770 @ 3.40 GHz mit 12 GB RAM und einer 120 GB SSD Festplatte unter Linux Mint LMDE Cinnamon Edition 3.11-2-amd64. Der Benchmarkprozess hatte 6 GB Arbeitsspeicher zu Verfügung.

Speicherplatzbedarf und Zugriffszeiten. Tab. 1 zeigt das Datenvolumen, das bei der Speicherung von einem Zeitstempel-Messwert-Tupel pro Sekunde entsteht. OTSS schlägt die klassische relationale Datenbank MySQL Community Server 5.6.21 mit dem Client Connector/J 5.1.33 und die neuartige Zeitreihendatenbank InfluxDB 0.8.4 mit dem Client influxdb-java 1.3 sowohl beim Zeit- als auch beim Speicherplatzbedarf, vor allem bei steigender Tupelanzahl. Ausreißerkenner erfordern meist einen Zugriff auf Tupelfolgen, dem die dokumentenorientierte Speicherung von OTSS hilft. Der Extremfall ist der Zugriff auf ein einzelnes Tupel. Hier ist MySQL wegen des Index auf der Zeitstempelspalte besser. OTSS ist aber der Zeitreihendatenbank In-

Tabelle 1: Speicherplatzbedarf und Zugriffszeiten

	1 Tag	7 Tage	1 Monat	6 Monate	1 Jahr
Gesamtdauer der Speicherung / ms					
MySQL	804,0	2789,0	11131,0	102372,0	276126,0
InfluxDB	638,0	3539,0	14746,0	93166,0	186715,0
OTSS	188,0	679,0	1815,0	11715,0	14197,0
Speicherplatzbedarf / MB					
MySQL	11,0	30,0	197,7	1780,0	3080,0
InfluxDB	10,5	25,5	79,2	432,7	764,4
OTSS	0,9	6,4	29,2	152.0	237,8
Zugriff auf 1 Tupel / ms					
MySQL	1,6	0,4	0,2	1,0	1,0
InfluxDB	399,0	2651,2	11212,0	67937,2	135466,0
OTSS	21,0	21,0	19,0	21,0	19,6

fluxDB überlegen, weil es über Metadaten gezielt auf das Dokument mit dem gesuchten Zeitstempel zugreift. Daher muss nur dieses Dokument dekomprimiert werden.

Analysezeiten der Ausreißererkennung. Tab. 2 zeigt die Effizienz unseres Ansatzes exemplarisch am Beispiel mit einem dynamischen Schwellwert aus Abschnitt 4 für Tupelmengen verschiedener Messzeiträume. MySQL wird aufgrund der fehlenden Analysefunktionen Quantil und Perzentil sowie des manuellen Aufwands für komplexe SQL-Abfragen oder einer Berechnung außerhalb der Datenbank nicht berücksichtigt. Der ungefähr zehnfache Geschwindigkeitsvorteil von OTSS gegenüber InfluxDB beruht auf den in Abschnitt 4 skizzierten Gründen und wächst wieder mit steigender Tupelanzahl. Zu große Datenmengen führen bei InfluxDB zu Fehlermeldungen und einem Abbruch der Analyse. Offensichtlich führen InfluxDB und OTSS zur selben Erkennungsgüte (Precision, Recall), da sie denselben Erkenner verwenden.

Tabelle 2: Analysezeiten der Ausreißererkennung

	1 Tag	7 Tage	1 Monat	6 Monate	1 Jahr
Analysezeit / ms					
InfluxDB	638,0	4383,4	18987,2	-	-
OTSS	135,4	421,6	2258,8	14908,4	22242,4

6 Verwandte Arbeiten

Alle in der Literatur untersuchten Ansätze zur Ausreißererkennung in Laufzeitdaten und zur Vorhersage von Fehlerzuständen (siehe Abschnitt 2) sind orthogonal zu dem hier vorgestellten Rahmenwerk, weil sie als Erkenner unter Verwendung der Basisbibliothek realisiert werden könnten. Abschnitt 4 hat OTSS bereits von klassischen und Zeitreihendatenbanken abgegrenzt.

Der gegenwärtige Trend zum Cloud Computing in der Zeitreihenanalyse [WTSR10, COM+08] nutzt verteilte Datenspeicherung (z.B. mit Apache HBase bzw. Cassandra) und verteilte Analysebibliotheken (z.B. Apache Hadoop) zumeist zur effizienten Verarbeitung von Datenbeständen, die aus dem Monitoring von Cloud Netzwerken entstammen. Unser Fokus liegt stattdessen auf der Analyse von Enterprise-Software-Systemen, die auf einem Rechnerverbund mit überschaubarer Knotenzahl laufen und bei denen die Messwerte lokal und interaktiv analysiert werden, anstatt erst latenzbehaftet in die Cloud hochgeladen werden zu müssen. Bei einer sekündlichen Messung von hundert Metriken auf zehn Rechnern mit je fünf Prozessen fällt in zwei Wochen ein Datenvolumen von 145 GB (4.788.000.000 Tupel) in unkomprimierten CSV-Dateien an, das OTSS lokal effizient verarbeiten kann (Tupelzugriff ca. 200 ms, Speicherplatzbedarf ca. 20 GB, Importdauer ca. 25 Min).

7 Zusammenfassung und zukünftige Arbeiten

Wir haben ein neuartiges Rahmenwerk zur Erkennung von Ausreißern beschrieben. Da beim Monitoring eine Vielzahl von Laufzeitdaten anfallen, ist für eine effiziente Ausreißererkennung und Zeitreihenspeicherung erforderlich. Deren Grundzüge haben wir ebenfalls dargestellt. Eine erste Evaluation zeigt erhebliche Effizienzvorteile gegenüber herkömmli-

chen Methoden. In Folgearbeiten erweitern wir die Bibliothek mit Funktionen zur Analyse von Zeitreihen und setzen weitere Ausreißererkenner um. Des Weiteren steht eine Evaluation der vertikalen und horizontalen Skalierung des Rahmenwerks aus. Entwurf und Implementierung einer domänenspezifischen Sprache zur Formulierung von Zeitreihenanalysen sind geplant.

Literatur

[ABA11] J. Alonso, L. Belanche und D.R. Avresky. Predicting Software Anomalies Using Machine Learning Techniques. In *Proc. Intl. Symp. Netw. Comp. and Appl. (NCA '11)*, Seiten 163–170, Cambridge, MA, Aug. 2011.

[Apaa] Apache Cassandra. `http://cassandra.apache.org/`. [Abruf 21. Okt. 2014].

[Apab] Apache HBase. `http://hbase.apache.org/`. [Abruf 21. Okt. 2014].

[Apac] Apache Solr. `http://lucene.apache.org/solr/`. [Abruf 21. Okt. 2014].

[App] AppDynamics. `http://www.appdynamics.com/`. [Abruf 20. Okt. 2014].

[CBK07] V. Chandola, A. Banerjee und V. Kumar. Outlier detection: A survey. Bericht 07-017, University of Minnesota, MN, 2007.

[COM+08] L. Cherkasova, K. Ozonat, N. Mi, J. Symons und E. Smirni. Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change. In *Proc. Intl. Conf. Dependable Syst. and Netw. (DSN '08)*, Seiten 452–461, Anchorage, AK, Juni 2008.

[DF14] T. Dunning und E. Friedman. *Practical Machine Learning: A New Look at Anomaly Detection*. O'Reilly Media, 2014.

[DMF12] L. Deri, S. Mainardi und F. Fusco. tsdb: A Compressed Database for Time Series. In *Proc. Intl. Work. Traffic Monitoring and Analysis (TMA '12)*, Seiten 143–156, Wien, Österreich, März 2012.

[Inf] InfluxDB. `http://influxdb.com`. [Abruf 20. Okt. 2014].

[Kai] KairosDB. `https://github.com/kairosdb`. [Abruf 20. Okt. 2014].

[Lev] LevelDB. `https://github.com/google/leveldb`. [Abruf 20. Okt. 2014].

[MyS] MySQL. `http://www.mysql.com/`. [Abruf 25. Okt. 2014].

[Nag] Nagios. `http://www.nagios.com`. [Abruf 20. Okt. 2014].

[OBS99] M. A. Olson, K. Bostic und M. I. Seltzer. Berkeley DB. In *Proc. USENIX Annual Tech. Conf., FREENIX Track (USENIX '99)*, Seiten 183–191, Monterey, CA, Juni 1999.

[Ope] OpenTSDB. `http://opentsdb.net`. [Abruf 20. Okt. 2014].

[VHK14] O. Vallis, J. Hochenbaum und A. Kejariwal. A Novel Technique for Long-Term Anomaly Detection in the Cloud. In *USENIX Work. on Hot Topics in Cloud Comp. (HotCloud '14)*, Philadelphia, PA, Juni 2014.

[WTSR10] C. Wang, V. Talwar, K. Schwan und P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *Proc. Netw. Operations and Management Symp. (NOMS '10)*, Seiten 96–103, Osaka, Japan, Apr. 2010.