Volltextsuchmaschine für digitalisierte Bernensia

Proof of Concept zur Implementierung eines quellenübergreifenden Retrievals in DigiBern.ch

Projektarbeit im CAS Big Data 2021

Studiengang:	CAS Big Data
Autorin:	Kathi Woitas
Betreuerin:	Dr. Ursula Deriu
Auftraggeber:	Marion Prudlo, UB Bern
Datum:	07.10.2021

Management Summary

Mit DigiBern verfügt die Universitätsbibliothek Bern über ein Webverzeichnis für das digitalisierte kulturelle Erbe von Stadt und Kanton Bern. DigiBern ist als Web Content Management System (WCMS) implementiert und verfügt nur über unzureichende Suchmöglichkeiten. Die Volltextsuche, die mittels Google implementiert ist, indexiert nur Inhalte, die direkt auf DigiBern vorhanden sind: Die Pages von des WCMS und in das WCMS eingebundene Digitalisate.
Ein grosser Teil der verzeichneten Dokumente befinden sich als Digitalisate auf den beiden nationalen Plattformen e-rara und e-periodica. Diese werden in DigiBern nur auf höchster Ebene (Titel/Gesamttitel bzw. Zeitschriftentitel) erfasst.

Dieser Proof of Concept soll die Implementierung einer Suchmaschine prüfen, die die verteilt vorliegenden Digitalisate erfasst. Hierzu müssen

- Metadaten und Volltexte der Plattformen geharvestet bzw. gescrapt werden
- Text aus den PDFs extrahiert werden
- Metadaten und Volltexte integriert und vorverarbeitet werden
- und in einen gemeinsamen Such-Server geladen werden.

Als Technology Stack wird eine Multicontainer-Anwendung mit den Komponenten Apache Spark, Apache Tika, Elasticsearch und Kibana auf einem virtuellen Server sowie ein Object Storage benutzt.

Das oben genannte Vorgehen konnte erfolgreich durchgeführt werden und der Suchindex per Web/API bzw. in Kibana angefragt werden. Eine Weiterverfolgung des Ansatzes bis zur produktiven Implementierung in DigiBern wird angestrebt.

Inhaltsverzeichnis

1 Einleitung

DigiBern[1] ist eine Plattform für das kulturelle Erbe des Kantons Bern („Bernensia“), die von der Universitätsbibliothek Bern betrieben wird. Auf der Basis der Web Content Management Systems (WCMS) Drupal werden digitalisierte Dokumente und spezifischere Online-Angebote in einem Portal zusammengefasst und mittels grober Kategorien in Form eines Webverzeichnisses erschlossen (Epochen, Regionen und Orte, Personen, Themen).

Für die digitalisierten Dokumente existiert keine übergreifende Suche, da diese Dokumente zum grossen Teil auf den nationalen Plattformen **e-rara**[2] und **e-periodica**[3] liegen. Für e-rara-Bestände ist zumindest eine Suche nach den Metadaten der Publikationen (zumeist Bücher) in Drupal möglich, für eine Volltextsuche muss auf die e-rara-Plattform gewechselt werden.
Für e-periodica-Bestände (Zeitschriftenaufsätze) ist nur der Zeitschriftentitel recherchierbar, und eine Suche auf Artikel-Ebene ist erst auf der e-periodica-Plattform möglich. Durch diese Brüche und das potentielle «Untergehen» von Bernensia-Publikationen in der Fülle der allgemeinen e-periodica-Plattform ist ein Retrieval nach bernischen Inhalten in Digitalisaten nur sehr zersplittert und unbefriedigend möglich.

Zielsetzung ist der **Proof-of-Concept einer Suchmaschine** für die DigiBern-Plattform, die eine Volltext-Suche (und generell eine Suche auf Artikelebene für e-periodica) **über die verteilt vorhandenen Digitalisate** ermöglicht. Hierzu müssen

- Metadaten und Volltexte der Plattformen geharvestet bzw. gescrapt werden
- Text aus den PDFs extrahiert werden
- Metadaten und Volltexte integriert und vorverarbeitet werden
- und in einen gemeinsamen Such-Server geladen werden.

Für die Realisierung dieses Vorgehens sind verschiedene Komponenten nötig. Während Harvesting und Scraping der Inhalte mit einfachen Skripten möglich ist, benötigen die weiteren Schritte spezifische Applikationen. In dem Proof-of-Concept werden die folgenden State-of-the-art-Komponenten verwendet:

Apache Tika[4] ist ein quelloffenes, «reifes» Java-Framework zur Extraktion von Text aus diversen Formaten wie PDF und Dateien aus Office-Anwendungen. Durch die Integration von Tesseract[5] ist ebenfalls Optical Character Recognition (OCR) mit Tika möglich, also die Schrifterkennung aus verschiedenen Bildformaten. Tika steht als Web-Server zur Verfügung, der über HTTP angesprochen werden kann. Eine öffentlich zugängliche bzw. frei nutzbare Instanz bietet zum Beispiel bei der Open Knowledge Foundation (OKFN) an.

Apache Spark[6] ist ein quelloffenes Scala-Framework für die verteilte Verarbeitung von Big Data für verschiedenste Anwendungen. Aufbauend auf und gewissermassen als Nachfolger von Hadoop[7] bietet es enorme Vorteile (Siehe umfassend: Karau):

- Spark automatisiert das parallelisierte Rechnen auf hoher Abstraktionsebene (im Unterschied zu Hadoop MapReduce sind genuin viele Operationen möglich, die verkettet werden können), optimiert die notwendigen Teilrechenoperationen und bietet hohe Fehlertoleranz.
- Dadurch dass Spark die Daten im Memory behält (und dies auch durch Komprimierung und Indexierung optimiert), ist Spark im Vergleich zu Hadoop MapReduce (Zwischenspeicherungen auf Disk) vielfach schneller.
- Neben dem Hadoop Distributed File System (HDFS) können auch andere Datenspeicher angebunden werden wie Object Storages und NoSQL-Datenbanken.
- Durch die Trennung von Datenverarbeitung und Datenspeicher können Rechenressourcen flexibel, d.h. nur wenn gerade nötig, aufgebaut werden.

[1] DigiBern, https://www.digibern.ch/
[2] E-rara, https://www.e-rara.ch/
[3] E-periodica, https://www.e-periodica.ch/
[4] Apache Tika, https://tika.apache.org/
[5] Tesseract, https://github.com/tesseract-ocr/tesseract
[6] Apache Spark, https://spark.apache.org/
[7] Apache Hadoop, https://hadoop.apache.org/

- Spark ist ebenfalls flexibel im Set-up: Es kann mit verschiedenen Cluster-Managern, sowohl in der Cloud als auch als stand-alone auf dem lokalen Rechner betrieben werden und unterstützt mehrere Programmiersprachen, darunter Python (PySpark).
- Multifunktionalität: Spark beherrscht Batch- und Streamprocessing, und eignet sich mittels spezifischer Libraries sehr für Machine-Learning-Anwendungen, SQL-ähnliche Abfragen oder auch die Verarbeitung von Graph-Daten.
- Durch die Definition von «User Defined Functions» sind komplexe Operationen und die Einbindung von diversen anderen Libraries möglich.

Elasticsearch[8] ist ein frei verfügbarer Such-Server in Java, auf der Basis des Suchindex von Apache Lucene[9]. Elasticsearch ist für verteilte Systeme mit hoher Verfügbarkeit ausgelegt, indem es etwa automatisch Shards und Replikationen anlegt. Die Datenhaltung von Elasticsearch erfolgt in einem JSON Document Store, der sich sehr für die Verwaltung von (nicht übermässig grossen) Textdaten eignet. Mit Elasticsearch ist ein einfacher Ingest und Indexierung von Daten möglich, auch ohne eine vorgängige Definition von Feldern («schemaless»).
Elasticsearch bietet zudem vielfältige Möglichkeiten zur Generierung der invertierten Indizes, indem Feldinhalte mit verschiedenen Analyzern (z.B. Tokenisierung, Stemming, N-gram-Erstellung, Filterung von Stoppwörtern, Sprachspezifik) verarbeitet werden können. Elasticsearch stellt einen Web-Server zu Verfügung, und sämtliche Transaktionen sind direkt über HTTP bzw. über APIs in verschiedenen Programmiersprachen möglich. Mit **Kibana**[10] existiert zudem eine komfortable Web-Anwendung zur direkten Analyse und Visualisierung der Elasticsearch-Indizes.

Schliesslich ist neben der persönlichen Hardware, die für einfache Aufgaben und Tests verwendet wird (Laptop mit Intel Core i7 CPU, 16 GB RAM, Betriebssystem Windows 10, Containervirtualisierung mit Docker[11]) leistungsstärkere Hardware und externe Datenspeicherung nötig. Hierzu werden die folgenden Cloud-Dienste verwendet:

AWS Lightsail[12] ist eine Virtual-Private-Server-Dienst der Firma Amazon, mit dem einfach verschieden konfigurierte virtuelle Server erstellt und betrieben werden können. Dabei sind sowohl Ort der physischen Datenhaltung wählbar, wie verschiedene Grössen (Memory, Disk, vCPUs) und Betriebssysteme (Infrastructure as a Service, IaaS), wie auch ganze Technologie-Stacks, etwa LAMP-Server (Platform as a Service, PaaS) bis hin zu dezidierten Anwendungen wie WCMS (Software as a Service, SaaS).

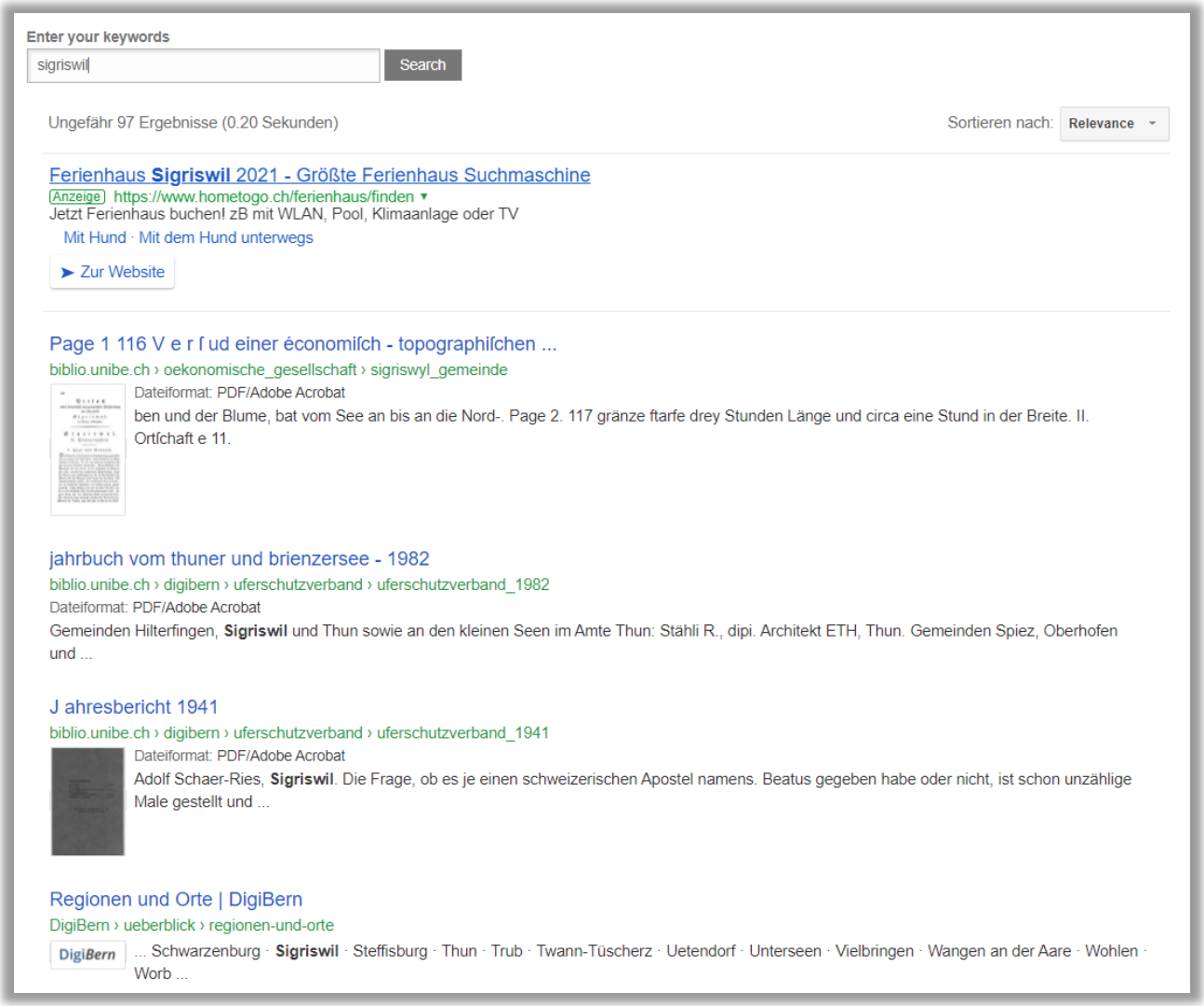
Amazon S3[13] (Simple Storage Service) ist ein Filehosting-Dienst der Firma Amazon, der sich als Object Storage für die persistente Speicherung von grossen Datenmengen eignet. Als verteilte Anwendung sind Datenreplikationen implizit. Der Dienst wird über die Plattform Amazon Web Services (AWS) angeboten und ist über HTTP/HTTPS ansprechbar.

[8] Elasticsearch, https://www.elastic.co/
[9] Apache Lucene, https://lucene.apache.org/
[10] Kibana, https://www.elastic.co/kibana
[11] Docker, https://www.docker.com/
[12] AWS Lightsail, https://aws.amazon.com/aws/lightsail
[13] Aamzon S3, https://aws.amazon.com/de/s3/

Ausgangslage

1.1 DigiBern

Die von der Universitätsbibliothek Bern (UB Bern) betriebene Plattform für das kulturelle Erbe von Stadt und Kanton Bern, **DigiBern**, ist ein Web-Verzeichnis über diverse Quellen zur bernischen Geschichte, Kultur und Sprache (in der bibliothekarischen Terminologie: «Bernensia»). Quellen, die hier aufgeführt und verlinkt sind, sind etwa digitalisierte Bücher, Zeitschriften und Zeitungen, spezifische Online-Portale und Datenbanken. DigiBern wird mit dem quelloffenen WCMS Drupal[14] betrieben und setzt vor allem auf das Browsing über die verzeichneten Quellen mittels verschiedener Kategorien. Es bietet nur eingeschränkte Möglichkeiten für eine Suche durch eine eingebundene Google Search. Erfasst werden von dieser **Volltextsuche nur Inhalte, die direkt auf DigiBern vorhanden** sind: die Pages von Drupal und hier eingebundene Digitalisate.



In Abb. 1 (Screenshot einer Suche in DigiBern mit dem Keyword "sigriswil") ist als unterster Eintrag der Verweis auf eine Drupal-Page («Regionen und Orte») zu sehen, die Einträge darüber zeigen auf digitalisierte Publikationen, die im WCMS hinterlegt sind – diese sind erkennbar an der Server-Adresse «biblio.unibe.ch». Der oberste Eintrag ist eine Anzeige, die durch Google Search eingefügt wird.

[14] Drupal, https://www.drupal.org/

Neben anderen Online-Portalen halten vor allem die nationalen Plattformen **e-rara** und **e-periodica** umfangreiche Bestände an Bernensia vor: E-rara ist hierbei die zentrale Plattform für digitalisierte Monografien, Karten, Druckgrafiken mit Schweizer Bezug und von besonderem Wert, e-periodica für digitalisierte populäre und wissenschaftliche Zeitschriften aus der Schweiz. Die UB Bern benutzt e-rara als Plattform für ihre Digitalisate, soweit sie die Bestimmungen zum Scope der Plattform erfüllen, die sukzessive Befüllung von e-periodica wird zentral durch die Bibliothek der Eidgenössischen Technischen Hochschule Zürich (ETH Library) gesteuert.

Die **Inhalte dieser beiden Plattformen** sind auf DigiBern **nur marginal recherchierbar**, das bedeutet auf der obersten Erschliessungsebene: bei Monografien mit dem Titel bzw. Gesamttitel bei mehrbändigen Werken, bei e-periodica nur mit dem Zeitschriftentitel. Die Suche funktioniert hier lediglich über die Drupal-Pages, die eine grobe Beschreibung der Quellen beinhalten, vgl. Abb. 2 und 3.

Mit dem Proof-of-Concept soll ein Vorgehen zur verbesserten Suche in DigiBern vorgeschlagen werden, indem eine Suchmaschine erstellt wird, die die bernischen Inhalte der beiden Plattformen für das Retrieval auf DigiBern verfügbar macht.

1.2 Inhalte der e-rara-Plattform

Für die bernischen Inhalte der e-rara-Plattform wird die dortige Sammlung «**Bernensia des 18. bis frühen 20. Jahrhunderts**»[15] zu Grunde gelegt. Die zum Download-Zeitpunkt insgesamt **571 Titel** sind mehrheitlich Bücher deutscher Sprache, aber auch andere Publikationstypen und Sprachen sind vertreten.

Für die Dokumente auf e-rara sind Volltexte zum Teil als TXT-Dateien verfügbar, zum Teil nur als PDF. Die bibliografischen Metadaten bietet e-rara per OAI-PMH[16]-Schnittstelle in XML-basierten bibliothekarischen Standards aus. Das OAI-PMH-Protokoll ist auf das Harvesting der Daten auf Grundlage von vordefinierten «Sets» (bzw. einzelne Items oder Gesamtbestand) ausgelegt, eine clientseitige Definition von Subsets wie etwa bei REST-APIs ist nicht möglich.

	Anzahl und Grösse
Volltexte : TXT	297 Dateien, 115 B bis 24 MB
Volltexte : PDF	274 Dateien, 159 KB bis 537 MB
Metadaten : XML (Dublin Core)	571 Dateien, je 2-4 KB

1.3 Inhalte der e-periodica-Plattform

Als Sample für die Inhalte mit Bern-Bezug der e-periodica-Plattform wurde der Komplettbestand der Zeitschrift «Berner Zeitschrift für Geschichte und Heimatkunde»[17] bzw. ihres Nachfolgers «**Berner Zeitschrift für Geschichte**»[18] ausgewählt. Das Sample umfasst damit **918 Zeitschriftenartikel** aus den Jahren 1939 bis 2020, wobei die Hefte des letzten Jahrgangs unter einer Moving Wall aktuell für den Zugriff gesperrt sind.

Volltexte sind bei e-periodica per se nur als PDFs verfügbar. Die PDFs der «Berner Zeitschrift für Geschichte» enthalten den vollständigen Text der Artikel und aufgrund des modernen Schriftbildes sind Erkennungsfehler der OCR eher selten. Die Volltexte können damit einfach z.B. mit Apache Tika aus den PDFs extrahiert werden. Die bibliografischen Metadaten werden von e-periodica ebenfalls per OAI-PMH[19]-Schnittstelle in einem XML-basierten Format ausgeliefert.

	Anzahl und Grösse
Volltexte : PDF	914 Dateien, Total 8.9 GiB Jahrgang 2020 (4 Dateien) unter Sperrfrist – kein PDF verfügbar: Nr. DOI: interner Identifier: 915 10.5169/seals-869568 zgh-002:2020:82::294 916 10.5169/seals-869569 zgh-002:2020:82::295 917 10.5169/seals-869570 zgh-002:2020:82::296 918 10.5169/seals-869571 zgh-002:2020:82::303

[15] Bernensia des 18. bis frühen 20. Jahrhunderts, https://www.e-rara.ch/nav/classification/1395750
[16] Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH), https://www.openarchives.org/pmh/
[17] Berner Zeitschrift für Geschichte und Heimatkunde, https://www.e-periodica.ch/digbib/volumes?UID=zgh-001
[18] Berner Zeitschrift für Geschichte, https://www.e-periodica.ch/digbib/volumes?UID=zgh-002
[19] Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH), https://www.openarchives.org/pmh/

Metadaten : XML (Dublin Core)	918 Dateien, je 3 KB

2 Set Ups

2.1 Set up «Lokal»

Das Set-up «Lokal» diente zum Ausprobieren einzelner Komponenten, insbesondere potentieller Datenverarbeitungsschritte mit Spark und Textextraktion mit Tika.

Tabellenkopf	
Spark, PySpark	Docker Container: jupyter/all-spark-notebook Spark 3.1.2, Python 3.9.6 Local Application, Client Mode
Text Extraktion	Apache Tika als Web-Service: https://okfnlabs.org/projects/tika-server/
Suchindex/Document Store	- (Docker Container Elasticsearch ausprobiert – Ressourcen zu schwach)
Data Storage	Cloud-Dienst: AWS S3

2.2 Set up «Cloud»

Tabellenkopf	
Virtueller Server (IaaS)	AWS Lightsail Instanz, Ubuntu 20.04 LTS (RAM: 16GB, 4 vCPU)
Data Storage	AWS S3
Docker Compose Stack	https://github.com/gschmutz/various-platys-platforms/blob/main/spark-exts3/ mit den folgenden Komponenten:
Spark, PySpark	Spark 3.0.1, Python: 3.8.6 Local Application, Client Mode
Text Extraktion	Apache Tika 2.0.0
Suchindex/Document Store	Elasticsearch 7.10.1
Utilities	wetty, Kibana

3 Vorgehen zum Datenbezug und Preprocessing

3.1 Datenzugang

3.1.1 E-rara-Plattform

Für den Zugang zu Metadaten (OAI-PMH) und Volltexten (Website) der Bernensia-Kollektion von e-rara wurde das entsprechende Jupyter-Skript[20] auf der «Digital Toolbox» der UB Bern genutzt, Urheberin des Notebooks ist die Autorin. Die Daten wurden danach mittels Python-Skript nach S3 geladen.

3.1.2 E-periodica-Plattform

Für den Zugang zu Metadaten e-periodica-Volltexten wurde von der ETH-Library als Betreiberin des Portals eine schwach strukturierte Text-Datei mit Digital Object Identifiern (DOI) der Artikel zur Verfügung gestellt. Dies wurde notwendig, da keine vordefinierten Sets für Zeitschriften vorliegen.

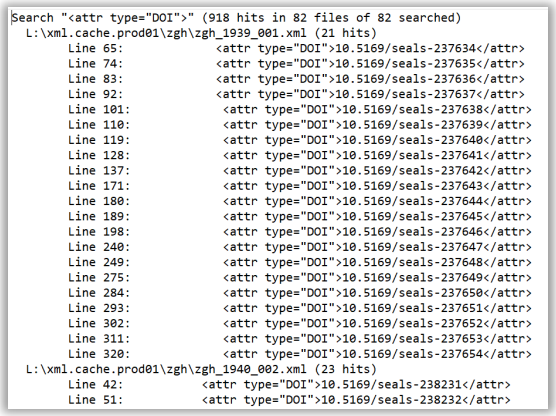


Abbildung 4: Ausschnitt der Identifier-Datei von e-periodica

Mittels OpenRefine[21] wurden DOIs aus dieser Datei ausgelesen, mit diesen DOI-Links (`http://doi.org/<DOI>`) generiert und mit einem Jupyter-Skript aus den entsprechenden Websites die internen Identifier (id_intern) der Zeitschriftenartikel herausgeparst.

	Unnamed: 0	raw	doi	id_intern
0	0	Line 65: <attr type="DOI">10.5169/...	10.5169/seals-237634	zgh-001:1939:1::293
1	1	Line 74: <attr type="DOI">10.5169/...	10.5169/seals-237635	zgh-001:1939:1::294
2	2	Line 83: <attr type="DOI">10.5169/...	10.5169/seals-237636	zgh-001:1939:1::295
3	3	Line 92: <attr type="DOI">10.5169/...	10.5169/seals-237637	zgh-001:1939:1::296
4	4	Line 101: <attr type="DOI">10.5169...	10.5169/seals-237638	zgh-001:1939:1::297
...	...	...	...	...
913	913	Line 201: <attr type="DOI">10.5169...	10.5169/seals-869590	zgh-002:2019:81::470
914	914	Line 32: <attr type="DOI">10.5169/...	10.5169/seals-869568	zgh-002:2020:82::294
915	915	Line 38: <attr type="DOI">10.5169/...	10.5169/seals-869569	zgh-002:2020:82::295
916	916	Line 44: <attr type="DOI">10.5169/...	10.5169/seals-869570	zgh-002:2020:82::296
917	917	Line 73: <attr type="DOI">10.5169/...	10.5169/seals-869571	zgh-002:2020:82::303

918 rows × 4 columns

Abbildung 5: Python Dataframe mit DOIs und geparsten internen Identifiern

[20] E-rara: accessing metadata and fulltext, https://github.com/ub-unibe-ch/ds-pytools/tree/main/web-tools/e-rara-access
[21] OpenRefine, https://openrefine.org/

Der direkte Download der Artikel als PDFs ist mittels dieser internen Identifier möglich. Der Download wurde direkt nach S3 mit dem folgenden Shell-Skript[22] durchgeführt:

downloadpdf.sh

```
#!/bin/bash
# Download PDF from e-Periodica and transfer to AWS S3
download_pdf () {
    curl https://www.e-periodica.ch/cntmng?pid=${1} | aws s3 cp - s3://bgd-test-
content/pdf/${1}.pdf
}
# Invoke function
download_pdf $1
```

Ebenfalls auf Grundlage der internen Identifier wurden mit einem Jupyter-Skript[23] der Autorin die bibliografischen Metadaten im XML-Format von der OAI-Schnittstelle bezogen.

3.2 Datentransformation der Volltexte: PDF -> Text

3.2.1 Textextraktion von e-periodica PDFs

Für die Extraktion der e-periodica PDFs wurden im Set up «Cloud» mittels eines Shell-Skriptes PDFs von S3 geholt, an den Tika Server gesendet und die resultierenden TXT-Dateien direkt nach S3 geladen.

Jupyter-Skriptes
(pdf_access_processing.ipynb)

pdftotext-remote.sh

```
#!/bin/bash
# Get text from PDF and write in TXT files on AWS S3
get_text () {
    IP=$(curl ipinfo.io/ip)
    aws s3 cp s3://bgd-content/eperiodica-pdf/${1}.pdf test.pdf
    curl -T test.pdf http://${IP}:28228/tika --header "Accept: text/plain" | \
                aws s3 cp - s3://bgd-content/eperiodica-txt/${1}.txt
}
# Invoke function
get_text $1
```

Für das Set up «Lokal» wurde ein analoges Skript verwendet, mit dem Unterschied, dass der freie Tika-Web-Service der OKFN genutzt wurde.

[22] Der dem Proof od Concept zu Grunde liegende Code wird auf derm Github-Account der Autorin veröffentlicht, https://github.com/k-woitas
[23] E-periodica: accessing metadata and fulltext, https://github.com/ub-unibe-ch/ds-pytools/tree/main/web-tools/e-periodica-access

3.2.2 OCR von e-rara PDFs

Volltexte von e-rara, die nur als PDF vorliegen, sind zumeist Digitalisate älterer Publikationen bzw. solcher mit besonderem Layout und/oder Schriftbild. Die Qualität der standardmässig eingesetzten OCR-Software reicht hier nicht aus, um eine reliable TXT-Datei zur Verfügung zu stellen. Gleichwohl wurde testweise die eigene Verarbeitung mit Tika vorgenommen[24] – je nach Tika-Konfiguration ist dies direkt möglich, oder über eine eigene vorgängige Umwandlung in Bilddateien[25]. Beide Vorgehensweisen führten jedoch zu keinem befriedigenden Ergebnis, sodass die Volltextanreicherung für diese Publikationen zunächst zurückgestellt wurde.

3.3 Datentransformation der Metadaten (XML -> JSON) und Datenintegration mit Volltexten

Nach dem Datenbezug der Metadaten im XML-Format via OAI-Schnittstellen der beiden Quell-Plattformen wurde mittels des Jupyter-Skriptes `metadata_access_transform.ipynb` die Transformation nach JSON vorgenommen. Das Skript vereinfacht zudem Feldnamen und führt zwei neue Felder ein und befüllt diese: `id_intern` und `fulltext`, wenn eine entsprechende TXT-Datei vorhanden ist. Die Struktur und beispielhaft Inhalte der JSON-Dateien ist in den Abbildungen 6 und 7 ersichtlich.


```
{'title': 'Catalogus codicum mss. Bibliothecae Bernensis, annotationibus criticis illustratus : addita sunt
specimina scripturae ex codicibus variae aetatis, tabulis sculptis exhibita, et praefatio historica',
 'creator': 'Sinner, Jean Rodolphe',
 'description': ['curante I.R. Sinner, bibliothecario',
  'Zusatz zum Sachtitel in tomus 2 und 3: "addita sunt excerpta quamplurima, et praefatio"',
  'Impressum in tomus 2 und 3: "Bernae, ex Officina Typogr. Brunneri & Halleri ..."',
  'Tomus 1 (1760): [2], XXXVI, 636, [16] S., IV Taf. : 4 Ill. (Kupferstiche), mit Handschriften-Beispielen
(Faksimiles)',
  'Tomus 2 (1770): XX, [21]-630, [14] S. (letzte Seite unbedruckt)',
  'Tomus 3 (1772): XII, 565, [1], [116] S.'],
 'publisher': 'ex Officina Typographica illustr. Reipublicae',
 'contributor': 'Sinner, Jean Rodolphe',
 'date': ['1760', '1772'],
 'type': ['Text', 'Book'],
 'format': '3 tomi : Ill. ; 21 cm (8Ã‚Â°)',
 'identifier': ['doi:10.3931/e-rara-15159',
  'https://www.e-rara.ch/bes_1/doi/10.3931/e-rara-15159',
  'system:9911671451860551 1'],
 'relation': 'vignette : https://www.e-rara.ch/bes_1/titlepage/doi/10.3931/e-rara-15159/128',
 'language': 'lat',
 'coverage': 'Bern',
 'rights': 'pdm',
 'id_intern': '4711794'}
```


Abbildung 6: JSON-Datensatz e-rara, ohne Volltext-Feld

[24] Vgl. Apache Tika OCR Documentation, https://cwiki.apache.org/confluence/display/tika/tikaocr
[25] Mithilfe der Python Library pdf2image, https://pypi.org/project/pdf2image/

```
{'title': 'Treueid der Herrschaftsleute von Spiez und Gegeneid des Freiherrn auf das alte Recht
und Herkommen',
 'creator': '[s.n.]',
 'subject': None,
 'description': None,
 'publisher': 'Paul Haupt Bern',
 'contributor': None,
 'date': '1939',
 'type': ['Text', 'Journal Article'],
 'source': ['Berner Zeitschrift fÃ¼r Geschichte und Heimatkunde',
  '280461-x',
  '0005-9420',
  '1',
  '1939',
  '1',
  None,
  '62'],
 'language': None,
 'relation': None,
 'coverage': None,
 'rights': None,
 'format': ['text/html', 'application/pdf', 'text/html'],
 'identifier': ['https://www.e-periodica.ch/digbib/view?pid=zgh-001:1939:1::299',
  'https://www.e-periodica.ch/cntmng?type=pdf&pid=zgh-001:1939:1::299',
  'doi:10.5169/seals-237639'],
 'id_intern': 'zgh-001_1939_1__299'}
```


Abbildung 7: JSON-Datensatz e-periodica, ohne Volltext-Feld

Bezüglich der Inhalte der Metadaten- bzw. JSON-Felder bestehen formale und inhaltlich Unterschiede zwischen den beiden Kollektionen, so sind etwa die Felder `subject` und `source` für e-rara nicht vorhanden, mehrere Felder für e-periodica standardmässig leer (null). Zudem es gibt Unterschiede in der Semantik der Felder, etwa bei `format`, das sich bei e-rara auf die physische Ressource bezieht, bei e-periodica auf die elektronischen.

	e-periodica	e-rara
title	Titel mit Untertiel	Titel mit Untertitel
creator	Urheber	Urheber, Bsp. [s.n.], ‘Jahn, Albert’
contributor	Beitragende	Beitragende
subject	‘None’ = null	-
description	‘None’ = null	Gesamte Titel- und Verfassserangabe, ergänzende Beschreibungen, Bsp. [‘…’, ‘…’], ‘[Museumgesellschaft in Bern]’
publisher	Verlagsangabe	Verlagsangabe, Bsp. [‘Verlag nicht ermittelbar’], ‘C.A. Jenni’
date	Erscheinungsjahr(e)	Erscheinungsjahr(e), Bsp. [‘1760’, ‘1772’]
language	‘None’ = null	Sprachcode gemäss ISO 639-2, Bsp. ‘ger’
rights	‘None’ = null	Lizenz: ‘pdm’
format	Dateiformate: [‘text/html’, ‘application/pdf, ‘text/html’]	Physische Beschreibung, Bsp. ‘28 Seiten ; 17 cm’
type	Publikationstypen: [‘Text’, ‘Journal Article’]	Publikationstypen, Bsp. [‘Text’, ‘Book’]
coverage	‘None’ = null	Sacherschliessung, Bsp. ‘Bern’, ‘949.4’
source	Angabe zu Journal (Titel, ZDB-ID, ISSN, Jahrgang, Jahr, Band, ‘None’, Seitenzahl)	-
relation	‘None’ = null	Link zu Cover-Thumbnail: ‘vignette : link’

identifier	Drei: [‘…’, ‘…’, ‘…’] • Link e-periodica-Website • Link PDF • DOI	Drei: [‘…’, ‘…’, ‘…’] • DOI • Link e-rara-Website • System-Nummer
Id_intern	interner Identifier, Bsp. ‘zgh-002_2015_77__508’	interner Identifier, Bsp. ‘25503173’
fulltext	Volltext	Volltext (wenn vorhanden)

4 Vorgehen zur Datenverarbeitung mit Spark

Zur Vorbereitung des Ingests in Elasticsearch wurden die JSON-Dateien mithilfe von Spark weiter aufbereitet und angereichert. Auf Spark wurde über PySpark und die DataFrame API zugegriffen, der Python-Code in einem Jupyter Notebook ausgeführt.
Zunächst wurden die JSON-Daten je Kollektion in Spark DataFrames geladen und einige Bereinigungs- und Transformationsschritte ausgeführt. Danach wurden auf die Volltext-Felder Algorithmen aus dem Natural Language Processing (NLP) angewendet, mit dem Ziel, die Publikationen zur besseren Recherche mit «Named Entities» anzureichern.

4.1 Datenbereinigung und Transformationen

Der genaue Ablauf der Datentransformationen ist in den Jupyter-Skripten `text_transform_ingest_elasticsearch_erara.ipynb` und `text_transform_ingest_elasticsearch_eperiodica.ipynb` ersichtlich. Ein Auszug daraus wird in Abb. 8 gegeben.

```
df.select("source").show(5, truncate=False)
```

```
+----------------------------------------------------------------------------------------+
|source                                                                                  |
+----------------------------------------------------------------------------------------+
|[Berner Zeitschrift für Geschichte, 2496177-2, 1663-7941, 76, 2014, 3,, 3]              |
|[Berner Zeitschrift für Geschichte, 2496177-2, 1663-7941, 79, 2017, 2,, 57]             |
|[Berner Zeitschrift für Geschichte und Heimatkunde, 280461-x, 0005-9420, 23, 1961,,, 1] |
|[Berner Zeitschrift für Geschichte, 2496177-2, 1663-7941, 77, 2015, 4,, 46]             |
|[Berner Zeitschrift für Geschichte und Heimatkunde, 280461-x, 0005-9420, 5, 1943,,, 161]|
+----------------------------------------------------------------------------------------+
only showing top 5 rows
```

Abbildung 8: Array-Feld 'source' mit einzelne auszulesenden Informationen

```
# Read out 'source' field into several ones
df_journal = df.withColumn('journal',df['source'][0].cast('String'))
df_volume = df_journal.withColumn('volume', df['source'][3].cast('Integer'))
df_issue = df_volume.withColumn('journal', df['source'][5].cast('Integer'))
df_startpage = df_issue.withColumn('start_page', df['source'][7].cast('Integer'))
df_issn = df_startpage.withColumn('issn', df['source'][2].cast('String'))
df_zdb_id = df_issn.withColumn('issn', df['source'][1].cast('String')) \
            .drop(df_issn['source'])

# Add field for source collection
df_coll = df_zdb_id.withColumn("source_collection", lit('E-Periodica'))

# Clean a OCR failure in field 'publisher'
df_publisher = df_coll.withColumn('publisher_new', \
                        when(col('publisher') == 'Paut Haupt Bern', \
                             lit('Paul Haupt Bern')) \
                             .otherwise(df_issn['publisher'].cast('String'))) \
                                .drop(df_issn['publisher'])
df_publisher = df_publisher.withColumnRenamed('publisher_new', 'publisher')

# Make a new format field which cleans duplicate information
df_form_1 = df_publisher.withColumn('format_new_1', df_publisher['format'][0].cast('String'))
df_form_2 = df_form_1.withColumn('format_new_2', df_form_1['format'][1].cast('String'))
df_form_3 = df_form_2.withColumn('format_new', concat(lit('['), 'format_new_1', lit(', '), 'format_new_2', lit(']'))) \
               .drop(df_form_2['format']).drop(df_form_2['format_new_1']).drop(df_form_2['format_new_2'])
df_form_4 = df_form_3.withColumnRenamed('format_new', 'format')

# Rename 'type' to prevent confusion in Elasticsearch use
df_cleaned = df_form_4.withColumnRenamed('type', 'document_type')

df_cleaned.persist().printSchema()
```

Abbildung 9: Auszug aus der Datenverarbeitung der e-periodica mit Spark

4.2 Natural Language Processing (NLP)

Zunächst wurde für die e-periodica Publikationen eine **Spracherkennung** durchgeführt, da das Feld `language` keine Informationen enthält. Hierfür wurde die Python Library `langdetect`[26] benutzt, die out-of-the-box 55 Sprachen zuordnen kann. Um diverse Spark-fremde Operationen auch auf den Spark-Nodes ausführen zu können, bietet PySpark über die SQL-API benutzerdefinierte Funktionen («User Defined Functions», UDF) an. Diese funktionieren ähnlich wie in relationalen Datenbanksystemen, wo UDF nach einer Registrierung in der Datenbank Library in SQL als reguläre Funktionen eingesetzt werden können. In PySpark wird die Python-Funktion als UDF registriert, und die resultierende Spark-UDF-Funktion kann dann auf den Spark DataFrame angewendet werden.

[26] Langdetect, https://pypi.org/project/langdetect/

```python
# Define UDF for detecting languages

def detect_language(text):
    lang = detect(text)
    if lang == 'de':
        language = 'German'
    elif lang == 'fr':
        language = 'French'
    elif lang == 'en':
        language = 'English'
    elif lang == 'la':
        language = 'Latin'
    else: language = 'Other'
    return language

udf_detect_language = udf(detect_language, StringType())
```

```python
# Apply UDF for detecting languages

df_lang = df_cleaned.drop('language')
df_language = df_lang.withColumn("language", udf_detect_language(col("title")))
```

```python
df_language.groupBy("language").count().orderBy('count', ascending=False).show(truncate=False)
```

```
+--------+-----+
|language|count|
+--------+-----+
|German  |872  |
|Other   |22   |
|French  |21   |
|English |1    |
+--------+-----+
```

Abbildung 10: Definition und Anwendung der UDF für die Spracherkennung

Bei den Publikationen mit Bern-Bezug spielt die lokale und regionale Verortung der Inhalte der Dokumente eine entscheidende Rolle: Sprache, Kultur und Geschichte werden – neben der zeitlichen Zuordnung – in der Regel vor diesem Hintergrund gedacht und untersucht. Für einen weiteren Mehrwert bietet sich daher das Verfahren der **Named Entity Recognition (NER)** an, die Eigennamen und Entitäten – hier räumlicher Art – aus den Texten herausfiltern kann.
Entitäten als bestimmte Terme und Phrasen können sich dabei auf reale Entitäten beziehen wie auf konkrete Orte (z.B. Eigernordwand), können aber auch imaginäre Entitäten oder Konzepte (z.B. Raumschiff) und allgemeine Begriffe (z.B. Marktplatz) sein. Weitere Möglichkeiten, die auf NER aufbauen sind Named Entity Disambiguation (NED), d.h. die Individualisierung von erkannten Entitäten und Named Entity Linking (NEL), d.h. Referenzierung der Entität zu bestimmten Systemen, etwa Wikidata.

Für die NER-Verarbeitung wurden zwei Standard-Anwendungen eingesetzt, das Natural Language Toolkit[27] (NLTK) und spaCy[28]. NLTK bietet nicht nur eine «reife» Python-Standardverarbeitung von Text in diversen Sprachen, sondern verfügt auch über einfach zu nutzende Zugänge zu lexikalischen und syntaktischen Ressourcen. Mit spaCy ist ebenfalls Standardverarbeitung von Text möglich, die wahre Stärke liegt jedoch in Training und Anwendung von Machine Learning und Deep Learning Methoden für Sprache. SpaCy bietet zahlreiche vortrainierte Sprachmodelle für 19 Sprachen an. Die sogenannten Pipelines beherrschen Wort- und Satztokenisierung, Part-of-Speech-Tagging, syntaktisches Parsing, Lemmatisierung sowie Vektorisierung mit verschiedenen Embedding-Methoden. Für die Anwendung in der Arbeit ist das ebenfalls vorhandene NER-Tagging entscheidend. Hierbei wurde das größte verfügbare deutschsprachige Modell `de_core_news_lg` gewählt. Dieses Modell wurde mit modernen Texten (TIGER- und WikiNER-Korpus) aus einer deutschen Zeitung und Wikipedia-Artikeln trainiert.

Die Volltexte aus e-periodica und e-rara wurden zunächst mit NLTK-Methoden bereinigt und in Sätze gesplittet, und danach das spaCy-Modell zur Erkennung von localities (räumliche named entities) auf ihnen angewandt. Wie bei der Spracherkennung wurden User Defined Functions hierfür eingesetzt.

[27] NLTK, https://www.nltk.org/
[28] spaCy, https://spacy.io/

Schliesslich wurde der Output der localities noch von überschüssigen Klammern (durch die Satztokenisierung) bereinigt.

```
# Define UDF for cleaning the raw text (remove special chars and words < 3) with NLTK

# Install NLTK
!pip install nltk
import nltk
from nltk import word_tokenize

def preprocess_nltk(text):
    wordlist = nltk.word_tokenize(str(text), language='german')
    # punctuation: all special characters, but not sentence endings
    punctuation = [',', ';', ':', '(', ')', '[', ']', '{', '}', '\"', '\'','\'\'', '\`', '\`\`', \
                   '\-', '«', '»', '£', '\^', '~', '*', '®', '•', '■', '◆', '§']
    wordlist_stripped = [w for w in wordlist if w not in punctuation]
    wordlist_stripped = [w for w in wordlist if len(w) > 2]
    wordlist = ' '.join(wordlist_stripped)
    return wordlist

udf_nltk_tokenize = udf(preprocess_nltk, StringType())
```

Abbildung 11: Definition der UDF zur Vorverarbeitung mit NLTK

```
# Define UDF for applying large Spacy german model on fulltext to recognize named entites of localities and persons

nltk.download('punkt')                # "punkt" = standard classifier for sentence segmentation
from nltk import sent_tokenize

def spacy_ner_loc(text):
    sents = nltk.sent_tokenize(text)
    ner_loc = []
    for s in sents:
        doc = nlp(s)
        ner_loc.append([ent.lemma_ for ent in doc.ents if ent.label_ == 'LOC'])
    return ner_loc

def spacy_ner_per(text):
    sents = nltk.sent_tokenize(text)
    ner_loc = []
    for s in sents:
        doc = nlp(s)
        ner_loc.append([ent.lemma_ for ent in doc.ents if ent.label_ == 'PER'])
    return ner_loc

udf_spacy_ner_loc = udf(spacy_ner_loc, StringType())
udf_spacy_ner_per = udf(spacy_ner_per, StringType())
```

Abbildung 12: Definition der UDF für NER-Extraktion mit spaCy

```
# Apply text cleaning and NER

df_nltk = df_language.withColumn("nltk", udf_nltk_tokenize(col("fulltext")))

# Extract LOC NER with SpaCy
df_spacy = df_nltk.withColumn("locality", udf_spacy_ner(col("nltk"))).drop("nltk")

# tried -> not sufficient
#df_spacy = df_spacy_loc.withColumn("person", udf_spacy_ner(col("nltk"))).drop("nltk")
```

```
DataFrame[locality: string]
```

```
def clean_locality(text):
    return re.sub(r'\[+|\]+', '', text)

udf_clean_locality = udf(clean_locality, StringType())

# Clean the 'locality' from duplicate '[' and ']'
df_loc_clean = df_spacy.withColumn("locality_clean", udf_clean_locality(col("locality"))) \
                                    .drop(df_spacy['locality'])
df_final = df_loc_clean.withColumn("locality", concat(lit('['), 'locality_clean', \
                                        lit(']'))).drop(df_loc_clean['locality_clean'])

df_final.persist()
```

Abbildung 13: Anwendung der NLTK- und spaCy-basierten UDF sowie UDF zum Bereinigen der localities

Testweise wurde ebenfalls eine Extraktion von Personen-Entitäten versucht, wie im Code in Abb. 12 und 13 ersichtlich. Die Ergebnisse hieraus wurden jedoch als unzureichend verworfen.

5 Ingest in Elasticsearch

Elasticsearch bietet verschiedene Möglichkeiten für den Dateningest an. Grundlegend funktioniert der Upload wie die Konfiguration über die HTTP-Methoden PUT und POST, die Abfragen mit GET. Diese Methoden werden durch zwei Python-Wrapper vereinfacht, den offiziellen `elasticsearch`[29] Python Client und die `elasticsearch-dsl`[30] Library, die auf dem Basis-Client aufsetzt und auf Abfragen spezialisiert ist. Besonders interessant im Zusammenspiel mit Spark ist jedoch die Support Library «Elasticsearch for Apache Hadoop»[31]. Dieser Konnektor erlaubt eine bidirektionale Anbindung von Hadoop und darauf aufbauenden Systeme wie Apache Pig, Apache Hive, und Spark. Für Spark im Besonderen sind Interaktionen auf Basis der Resilient Distributed Datasets (RDD) in Scala und Java beschrieben[32]. Schliesslich existiert aber auch eine Portierung für PySpark[33], mit welcher die Interaktion über den Spark SQL Context möglich ist – und damit die Direktanbindung zu Spark DataFrames.

Für den Daten-Ingest wurde diese letzte Möglichkeit gewählt, da die Autorin bisher nur über Python-Kenntnisse verfügt. Die Inbetriebnahme des Elasticsearch-Konnektors ist einfach über die Konfiguration der Spark Session möglich. Hierzu wird das «elasticsearch-spark» Java Archive eingebunden und aktiviert («es.index.auto.create» = true). Dabei ist darauf zu achten, die passende Version des Packages zu wählen, je nach Elasticsearch- und Spark-Version.

[29] Python Elasticsearch Client, https://elasticsearch-py.readthedocs.io
[30] Elasticsearch DSL, https://elasticsearch-dsl.readthedocs.io
[31] Elasticsearch for Apache Hadoop, https://www.elastic.co/guide/en/elasticsearch/hadoop/current/reference.html
[32] Apache Spark Support, https://www.elastic.co/guide/en/elasticsearch/hadoop/current/spark.html
[33] Apache Spark Support, Using the connector from PySpark, https://www.elastic.co/guide/en/elasticsearch/hadoop/current/spark.html#spark-python

```
# Start Spark Session

import pyspark
conf = pyspark.SparkConf()

# Session configuration
#conf.setMaster("spark://spark-master:7077")
conf.setMaster("local[2]")

conf.set("spark.executor.memory", "8g")
conf.set("spark.executor.cores", "1")
conf.set("spark.driver.memory", "2g")
conf.set("spark.core.connection.ack.wait.timeout", "1200")

# Elasticsearch
conf.set("spark.executor.extraClassPath", "elasticsearch-hadoop-7.12.0/dist/elasticsearch-hadoop-20_2.11-7.12.0.jar, \
            elasticsearch-hadoop-7.12.0/dist/elasticsearch-spark-20_2.11-7.12.0.jar")
conf.set("spark.jars.packages", "org.elasticsearch:elasticsearch-spark-30_2.12:7.12.0")
conf.set("es.index.auto.create", "true")

# Initialize a Spark session with configuration
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('elastic') \
        .config(conf=conf) \
        .getOrCreate()

# Instantiate a Spark Context
sc = spark.sparkContext
```

Abbildung 14: Konfiguration des Spark Session mit Elasticsearch-Spark Java Package

Schliesslich muss der Ingest selbst konfiguriert werden. Hierfür bietet Elasticsearch viele Parameter an[34]. In der Minimalvariante müssen zumindest die entsprechenden Elasticsearch-Instanzen angegeben werden («es.nodes») die «es.resource», also Index und Document-Type, in der die Dokumente geschrieben werden sollen, und die Art des Ingests («es.write.operation»), deren Default-Wert «index» ist. Ein Identifier für Dokumente wird von Elasticsearch automatisch kreiert, wenn kein eigener («es.mapping.id») übergeben wird. Die Nutzung eines eigener Identifiers ist für das Updating von Dokumenten nötig (sofern man den von Elasticsearch kreierten Identifier nicht in die Ursprungsdaten zurückschreibt). Mit dem Daten-Ingest startet Elasticsearch auch automatisch die Indizierung.

```
# Easy ingest to Elasticsearch as JSON
from pyspark.sql import SQLContext

df_final.write.format("org.elasticsearch.spark.sql") \
    .option("es.resource", "index/eperiodica") \
    .option("es.mapping.id", "id_intern") \
    .option("es.write.operation", "index") \
    .option("es.nodes", "elasticsearch-1").save()
```

Abbildung 15: Minimalkonfiguration eines Elasticsearch-Ingest aus dem Spark SQL Context

5.1 Mappings und Analyzing

Mittels Mappings wird in Elasticsearch gesteuert, ob und wie Daten einzelner Felder in den Index geschrieben werden (Hopf 23–27, 142–46). Hierfür bestehen vielfältige Optionen, und diese haben unmittelbare Wirkung auf die Funktionalität und Leistungsfähigkeit der Suche. Wird kein Mapping definiert, erstellt Elasticsearch automatisch ein eigenes auf Grundlage der gelieferten Daten («dynamic mapping» im Gegensatz zum «explicit mapping»[35]).

[34] Elasticsearch for Apache Hadoop, Configuration, https://www.elastic.co/guide/en/elasticsearch/hadoop/current/configuration.html
[35] Elasticsearch, Explicit Mapping, https://www.elastic.co/guide/en/elasticsearch/reference/7.x/explicit-mapping.html

Die spezifische «Analyzing» der übergebenen Daten, die im Mapping definiert wird, macht erst die für die eine Suchmaschine elaborierten Funktionalitäten wie eine Suche mit Wortteilen, abweichenden Schreibweisen, Autovervollständigung usw. möglich. So werden nicht (nur) der Text, wie er im Feld steht in den invertierten Index geschrieben, sondern diverse Derivate davon: einzelne Worte durch Tokenisierung, Wortstämme durch Stemming, Wortgrundformen durch Lemmatisierung etc. Gleichfalls ist es auch möglich, dass Wörter von der Aufnahme in den Index ausgeschlossen werden, etwas solche, die häufig vorkommen und wenig eigenen Sinn, und somit wenig zu einer potentiellen Suche beitragen (Stoppwörter).

Elasticsearch hält für diese Zwecke verschiedene Analyzer vor, die wiederum aus einzelnen Komponenten wie Char-Filtern, Tokenizern und Token-Filtern bestehen. So werden etwa im Standard Analyzer mögliche Auszeichnungselemente herausgefiltert, der Text auf Wortebene aufgesplittet, und die resultierenden Tokens in Kleinbuchstaben umgewandelt (lowercasing). Analyzer können auch entsprechend selbst definiert werden. Werden zusätzliche Analyzer zu dem (selbst erstellten oder durch Elasticsearch vorgegebenen) Standardmässigen eingesetzt werden diese auf Subfeldern angewendet.

```
%%bash
# Test standard analyzer
curl -X POST "http://172.26.6.171:9200/_analyze?pretty" -H 'Content-Type: application/json' -d'
{
  "analyzer": "standard",
  "text": "D\u00fcrfte von Anfang. Ein anderer Hinweis als\nselbstverst\u00e4ndlich."
}
'
```

```
{
  "tokens" : [
    {
      "token" : "dürfte",
      "start_offset" : 0,
      "end_offset" : 6,
      "type" : "<ALPHANUM>",
      "position" : 0
    },
    {
      "token" : "von",
      "start_offset" : 7,
      "end_offset" : 10,
      "type" : "<ALPHANUM>",
      "position" : 1
    },
    {
      "token" : "anfang",
      "start_offset" : 11,
      "end_offset" : 17,
      "type" : "<ALPHANUM>",
      "position" : 2
    },
```

Abbildung 16: Test des Standard Analyzers von Elasticsearch

Für den Ingest der E-periodica-Publikationen wurde manuell ein explizites Mapping erstellt, dass für bestimmte Felder ein Keyword-Subfeld beinhaltet, und für bestimmte ein zusätzliches Feld, dass ….

Dieses Mapping wurde per Commandline-Befehl an Elasticsearch übergeben (und damit ein Index kreiert), da anscheinend keine Mapping-Konfiguration über den Spark-SQL-Konnektor möglich sind. Bei dem nachgelagerten Ingest der Daten führt das zu einem Abbruch, da Elasticsearch automatisch versucht auf die Daten hin das Mapping anzupassen. Dies wiederum, ein Mapping im Nachgang anzupassen ist – bis auf das Hinzufügen neuer Felder – grundsätzlich nicht möglich (da es vorhandene Daten invalide machen würde). Dieses Problem konnte in der Bearbeitungszeit nicht gelöst werden und muss weiter auf Lösungsmöglichkeiten untersucht werden.

Der Ingest aus dem SparkDataFrame ohne vorheriges Mapping funktioniert reibungslos, und die Daten können nach Erstellung eines Index-Patterns in Kibana durchsucht und analysiert werden.

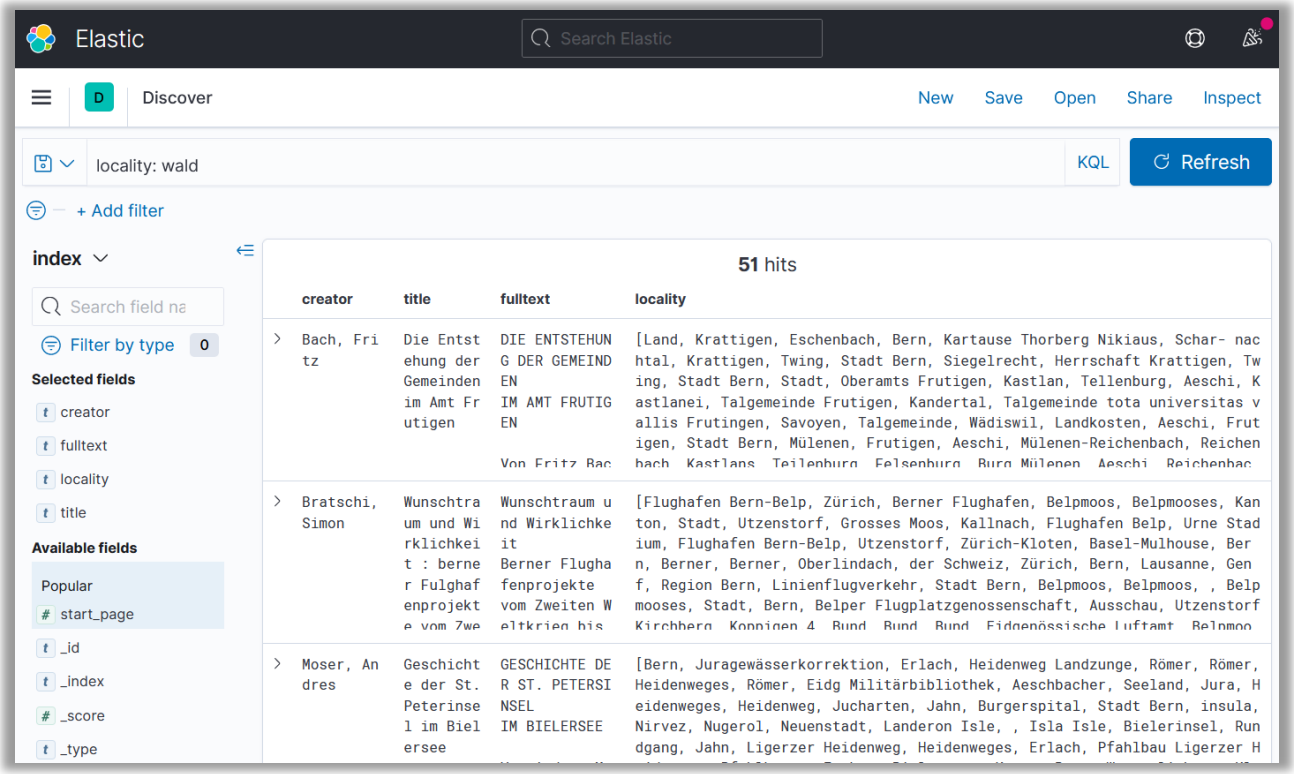


Abbildung 17: Beispielsuche auf dem mit NER erstellten Feld 'locality'

Hervorzuheben ist der zusätzliche Retrievalgewinn durch das Feld `locality`, welches zumindest für die Publikationen aus e-periodica (Jahrgänge 1939-2019) ordentliche, wenn auch nicht fehlerfreie Ergebnisse liefert.

6 Fazit und Ausblick

6.1 Einschränkungen, Optimierungsmöglichkeiten und ungelöste Probleme

Im ersten Abschnitt des Datenbezugs und Preprocessings wurde die Schrifterkennung auf alten Drucken wegen unzureichender Qualität verworfen. Mit dem Einsatz von spezialisierten Anwendungen aus dem Bereich der Handwritten Text Recognition (HTR) – die zum Teil auch eigenes Training von Machine-Learning-Verfahren erlaubt – könnten diese «Problemfälle» gesondert verarbeitet werden.

Weiters wurde die Textextraktion mit einer Standardimplementierung *eines* Tika-Servers vorgenommen. Eine Verarbeitung über Spark mit verteilten Instanzen wäre eine Möglichkeit diesen Schritt zu beschleunigen. Eine mögliche einfache Herangehensweise wäre die Nutzung der Tika-Python Library[36], die im Hintergrund einen Tika-Server implementiert: Mit PySpark besteht die Möglichkeit, diverse Python Libraries auf die Spark Worker Nodes zu importieren.

Im Bereich der Datenverarbeitung mit Spark können insbesondere die NLP-Anwendungen noch passgenauer gewählt werden. So wurde sich hier auf die Verarbeitung auf die grosse Gruppe der deutschsprachigen Publikationen beschränkt – eine Verarbeitung mit Modellen für andere Sprachen ist analog implementierbar. Ebenso können je nach zeitlicher Verortung auch Sprachmodelle Anwendung finden, die auf alte Texte vortrainiert wurden.

Weiter gedacht könnten auch Verfahren der automatischen Sacherschliessung auf den Texten erprobt werden. Hierfür werden Methoden des Supervised Machine Learning zur Multiklassifikation eingesetzt. Mit der Datenverarbeitung in Spark ist hierfür eine ideale Basis geschaffen.

[36] Tika-Python, https://github.com/chrismattmann/tika-python

Ein grosses Manko des Proof-of-Concept bleibt beim Laden der Daten in Elasticsearch, dass die bisher nur ohne eigenes Mapping der Datenfelder gelingt. Die Implementierung eines eigenen Mappings ist für eine Suchmaschine, die sich am state of the art orientiert und vollends überzeugt, notwendig. Hier müssen weitere Abklärungen, gegebenenfalls auch andere Ingest-Wege aus Spark, geprüft werden.

Schliesslich muss betont werden, dass die verwendeten Datenmengen überschaubar sind und weder in den Kriterien «volume» noch «velocity» eine Big-Data-Grösse erreichen. Während dies für das Kriterium «velocity» in diesem Anwendungsfall auch wohl nie der Fall sein wird, ist die parallele Verarbeitung der 10- oder 20-fachen Datenmenge theoretisch durchaus möglich. Hier würden dann Themen des Tunings von Spark und der Betrieb im Distributed Mode virulent werden.

6.2 Fazit und Empfehlung

Zielsetzung der Arbeit ist ein Proof-of-Concept einer Suchmaschine für die DigiBern-Plattform, die eine Volltext-Suche (und generell eine Suche auf Artikelebene für e-periodica) über die verteilt vorhandenen Digitalisate ermöglicht. Mit dem beschriebenen Vorgehen wurde gezeigt, dass eine Aggregation von grossen Quellsystemen wie e-rara und e-periodica und eine performante Verarbeitung der Ausgangsdaten mit Spark funktioniert. Das avisierte Vorgehen

- Metadaten und Volltexte der Plattformen zu harvesten bzw. zu scrapen
- Text aus den PDFs zu extrahieren
- Metadaten und Volltexte zu integrieren und zu verarbeiten
- und diese schliesslich in einen gemeinsamen Such-Server zu laden

konnte erfolgreich durchgeführt werden.

Erwähnenswert ist hinsichtlich der Verarbeitung der Daten, dass diese nicht nur **effektiv bereinigt und angeglichen** werden können, sondern auch **mit NLP-Methoden** wie Named Entity Recognition verarbeitet, und somit **angereichert**. Ein Suchindex kann dann mit dem Ingest aus Spark nach Elasticsearch einfach erstellt werden. Im Hinblick auf die im Moment unbefriedigende Suche im DigiBern-Portal wäre die Bereitstellung eines integrierten Suchindex über e-rara-Quellen und allen Zeitschriften mit Bernbezug der e-periodica-Plattform ein **deutlicher Mehrwert**.

Es wird empfohlen dieses Proof-of-Concept zur Grundlage zu nehmen für eine weitere Exploration des skizzierten Vorgehens. Als dringende nächste Aufgabe muss das Ingesting mit explizitem Mapping realisiert werden. Danach kann eine betriebliche Beratung über die produktive Implementierung der Suchmaschine auf DigiBern begonnen werden.

7 Abbildungsverzeichnis

8 Literaturverzeichnis

Hopf, Florian. *Elasticsearch: ein praktischer Einstieg*. 1. Auflage., dpunktverlag, 2016.

Karau, Holden. *Learning Spark: Lightning-Fast Big Data Analytics*. First edition., O'Reilly, 2015.

9 Selbständigkeitserklärung

Ich bestätige, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel angefertigt habe. Sämtliche Textstellen, die nicht von mir stammen, sind als Zitate gekennzeichnet und mit dem genauen Hinweis auf ihre Herkunft versehen.

Ort, Datum: Winterthur, 07.20.2021

Unterschrift: K. Woitas