Integration von Lucene für Volltextanfragen und Facettierung in XML-Datenbanken

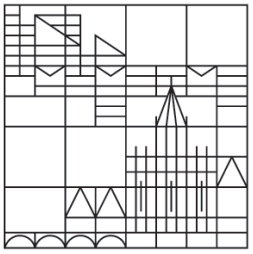
Bachelorarbeit

vorgelegt von

Schuler, Stephan
Matrikel-Nr. 01/789603

an der

Universität Konstanz



Mathematisch-Naturwissenschaftliche Sektion

Fachbereich Informatik und Informationswissenschaft

1. Gutachter: Prof. Dr. Marc H. Scholl

2. Gutachter: Prof. Dr. Michael Grossniklaus

Konstanz, 2015

Inhaltsverzeichnis

1 Einführung

1.1 Danksagung

Mein besonderer Dank für die Betreuung dieser Bachelorarbeit gilt Prof. Dr. Marc H. Scholl und Prof. Dr. Michael Grossniklaus.
Ebenfalls bedanken möchte ich mich bei Dr. Christian Grün für dessen engagierte Unterstützung während der Anfertigung dieser Arbeit.

1.2 Motivation

Die Extensible Markup Language (XML) ist ein einfaches, sehr flexibles Textformat, welches eine zunehmende Rolle im Austausch einer großen Variation von Daten [TW08] spielt. So gibt es auch Datenbanken, speziell für dieses Datenformat. Da XML-Dateien sehr groß sein können, ist eine effiziente Suche essentiell.
Aufgrund der Natur der XML-Dateien, welche baumartig strukturiert sind, bietet sich dafür besonders die Facettensuche an. Aus dieser Struktur lassen sich sehr gut die benötigten Facetten ableiten. Des Weiteren bietet diese Art der Suche auch einige große Vorteile gegenüber anderen Suchverfahren wie beispielsweise der hierarchischen Suche und erfreut sich vor allem im Online-Handel großer Beliebtheit.
In dieser Arbeit wurde eine Facettensuche in eine XML-Datenbank integriert. Für die Implementierung der Suche wurde die Suchbibliothek Lucene genutzt. Als XML-Datenbank für die Integration wurde BaseX gewählt. Im Gegensatz zu anderen XML-Datenbanken setzt BaseX kein Schema voraus, sondern lässt beliebige XML-Dateien zu [Grü15]. Dies stellt für die Facettensuche ein Hindernis dar, da die Facetten zur Indexierungszeit bekannt sein müssen. Dennoch ist es gerade auch bei solch beliebigen XML-Dateien für die Nutzer von Interesse bei der Suche einen besseren Überblick über den Ergebnisraum zu bekommen.
Deshalb beschäftigt sich diese Arbeit zum einen mit dem Thema, wie eine Facettensuche trotz beliebiger XML-Dateien realisiert werden kann. Zum anderen bietet BaseX eine XPath/XQuery-Implementierung [Grü15]. Dadurch konnte die Suche am Beispiel von

BaseX in einer Weise implementiert werden, so dass die Ergebnisse der Facettensuche weiterhin in BaseX mit XPath und XQuery bearbeiten werden können.

2 Vorentscheidung

Das Ziel ist es eine Facettensuche in BaseX zu implementieren. Dabei stellt sich zunächst die Frage, wie diese in BaseX implementiert werden kann und womit die Daten indexiert werden sollen.
Beim Wie fiel die Entscheidung auf ein XQuery-Modul, beim Womit auf Lucene.
Was diese Entscheidungen bedeuten und weshalb sie getroffen wurden, wird in den nächsten Kapiteln betrachtet. Zuvor jedoch ein kurzer Einschub zur Facettensuche.

2.1 Begriffserklärung

Facettenwerte sind die Inhalte von Facetten. Beispielsweise könnte die Facette Genre, die Facettenwerte Drama, Komödie oder Thriller enthalten.

Facettenzahl ist die Anzahl an Ergebnissen einer Suche, welche einen bestimmten Facettenwert enthält. Für jeden Facettenwert gibt es dementsprechend eine Facettenzahl.

Main-Element ist die Entität auf welcher die Facettensuche definiert wird. Wird eine Anfrage durchgeführt, erhält der Nutzer diese Entitäten als Ergebnisse zurück. Beispielsweise könnten dies Bücher, oder andere Waren sein.

drill-down ist die Bezeichnung dafür, wenn der Suchraum auf einen bestimmten Wert der Facette eingeschränkt wird, beispielsweise auf Bücher des Genres Drama.

drill-sideways ist die Bezeichnung dafür, wenn der Suchraum nicht nur auf einen Wert der Facette eingeschränkt wird, sondern auf mehrere Werte einer Facette zur selben Zeit. Beispielsweise auf Bücher des Genres Drama und Thriller.

Base-Query ist die Volltextanfrage, welche einer drill-down und drill-sideways Suche zugrunde liegt.

2.2 Facettensuche

Die Facettensuche definiert im Gegensatz zu einer hierarchischen Klassifikation keine Hierarchie sondern Facetten. Siehe Abbildung 2.1 und 2.2.
Daraus resultieren für den Nutzer einige entscheidende Vorteile.

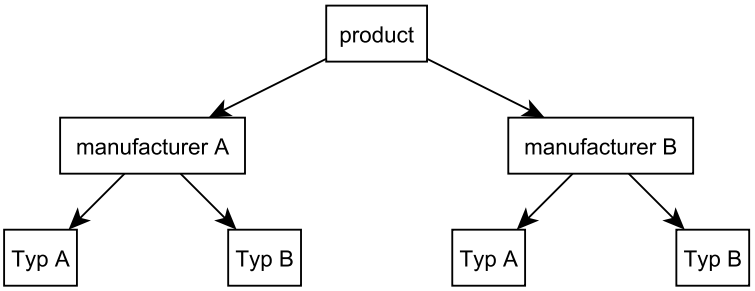


Abbildung 2.1: Hierarchische Suche

So kann der Nutzer in einer hierarchischen Suche nur innerhalb der Hierarchie suchen. Im Beispiel aus Abbildung 2.1 ist es zuerst lediglich möglich nach dem Hersteller und erst dann nach dem Typ des Produktes zu suchen. Auch kann nur nach einem Typ gleichzeitig gesucht werden. Es ist also nur eine drill-down Suche möglich.
Des Weiteren endet der Nutzer bei dieser Methode leicht in einer Sackgasse, da während der Suche keine Informationen über die Treffer unter den möglichen Klassifikationen vorliegen.

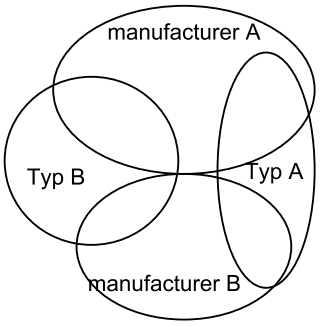


Abbildung 2.2: Facettensuche

Bei der Facettensuche hingegen ist der Suchraum nicht hierarchisch gegliedert, sondern in den namensgebenden Facetten, wie in Abbildung 2.2 zu sehen ist.
Dadurch gibt es keine Reihenfolge die bei einer drill-down Suche eingehalten werden muss. Zudem lässt dies auch zu, dass früher getroffene Einschränkungen wieder aufgehoben werden können, ohne dass spätere Einschränkungen auch betroffen sind. Auf diese Weise ist es möglich bei der Facettensuche inkrementell eine Boolesche Anfrage aufzubauen.
Ein weiterer Vorteil sind drill-sideways Suchen. Hiermit können drill-down Suchen kombiniert werden. Somit kann gleichzeitig nach mehreren Werten der selben Facette gesucht werden.
Auch das angesprochene Sackgassen-Problem der hierarchischen Suche ist bei der Facettensuche nicht vorhanden. Dies ist der Fall, da bei der Facettensuche nicht nur die Suchergebnisse angezeigt werden, sondern auch die Facettenergebnisse. Diese Facettenergebnisse enthalten Informationen über die möglichen Suchkriterien einer Facette sowie die Anzahl der Suchergebnisse, die diese Suchkriterien enthalten. Dadurch kann der Nutzer schon vor der Suche sehen, unter welchen Kriterien er Ergebnisse bekommen würde. Die Facettensuche erhält durch diese Eigenschaften einen sehr explorativen Charakter.

Wie eine Facettensuche in der Praxis aussieht ist am Beispiel der Buchsuche von amazon.com in Abbildung 2.3 zu sehen.
Hier sind auf der linken Seite die Facetten, wie etwa der Erscheinungszeitraum oder die Art des Buches, sowie die zugehörigen Facettenwerte und Facettenzahlen zu sehen. Auf der rechten Seite sind die Suchergebnisse zu den auf der linken Seite getroffenen Suchkriterien zu sehen.
Zusätzlich kann bei dieser Art der Suche eine Volltextsuche angeben werden. Auch diese ist im Amazon-Beispiel durch das Suchfeld vorhanden.

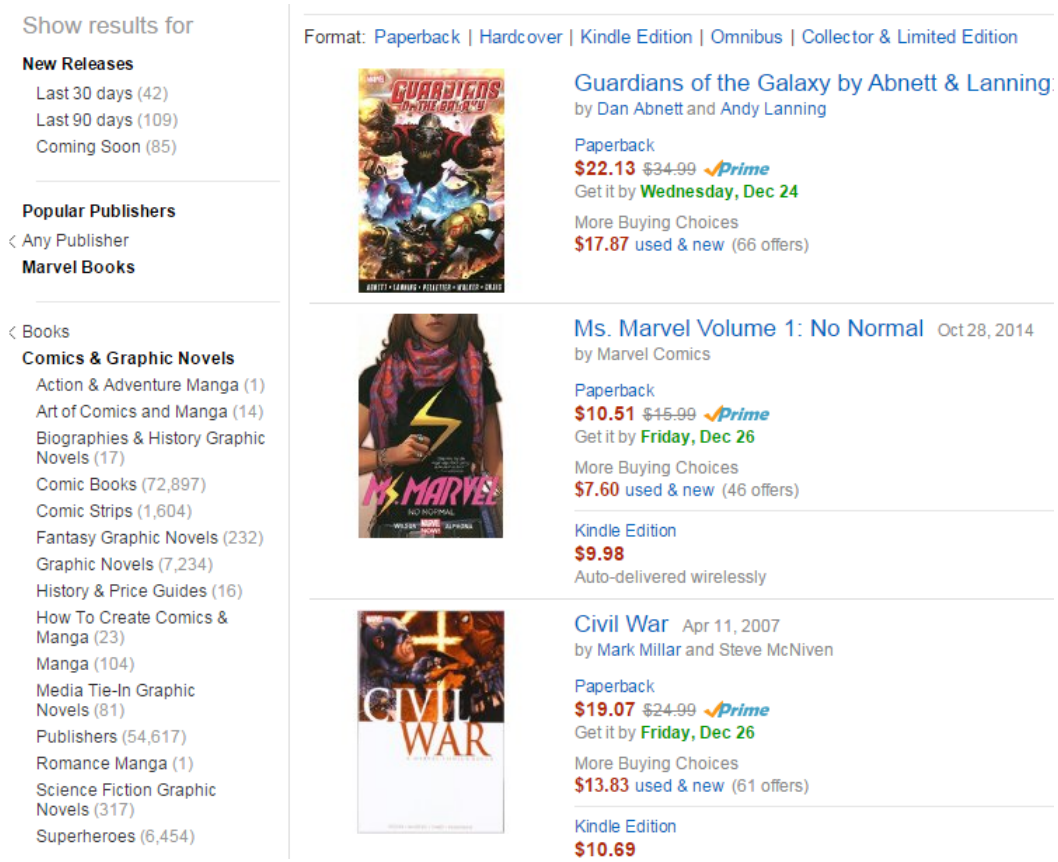


Abbildung 2.3: Buch Suche auf amazon.com

2.3 Lucene

Lucene ist eine Open-Source Suchbibliothek. In den letzten Jahren ist Lucene sehr beliebt geworden und ist eine der am meist genutzten information-retrieval-Bibliotheken [MHG10]. Sie ist für die Suche auf vielen Webseiten und Applikationen verantwortlich, so zum Beispiel auf Wikipedia und Twitter. Des Weiteren ist Lucene wie auch BaseX in Java geschrieben. Da Lucene, wie oben beschrieben, Open-Source mit einer sehr aktiven Community ist, ändern sich die API und die Features von Lucene häufiger. Durch den Grundsatz der Rückwärts-Kompatibilität stellt dies allerdings kein Problem dar.

Die Lucene-Core-JAR ist mit 2MB sehr klein und bringt die Möglichkeit zur Indexerstellung sowie für eine Volltextsuche. Diese Funktionalität lässt sich über zusätzliche JAR-Dateien stark erweitern. So gibt es eine Facettensuche oder auch verschiedene Analyzer. Insgesamt sind für diese Arbeit 4 MB an Bibliotheken nötig.

Für die Indexerstellung und die Suche wird nur die Lucene-API benötigt. Ein Einblick unter die Haube ist nicht von Nöten [MHG10]. Der Fokus der Arbeit liegt also fast vollständig darauf, wie die Daten aufbereitet und an Lucene übergeben werden können.

Die Übergabe der Daten an einen Lucene-Index, und wie diese dann durchsucht werden können, wird im Folgenden betrachtet.

2.3.1 Suche

Die Suche wird in Lucene wie folgt aufgebaut:

- Zunächst wird eine Volltext Base-Query erstellt. Hierfür kann dem Lucene-Query-Parser ein String übergeben werden:

  queryParser.parse(query)

  Diese Volltextsuche wird auf dem Default-Feld des Index durchgeführt. Die Anfrage selbst unterstützt dabei unter anderem folgende Anfragetypen [MHG10]:

  - Wildcard Anfragen
  - Fuzzy Anfragen
  - Range Anfragen

- Diese Base-Query kann im Folgenden mit drill-down Suchen erweitert werden.

  query.add(dimension, subQuery)

- Mittels Kombination mehrerer drill-down Suchen kann eine drill-sideways Suche erstellt werden.

- Ist die Anfrage definiert, kann die Suche durchgeführt und die Ergebnisse ausgegeben werden.
  Dabei wird die Anfrage vom Lucene Searcher ausgeführt. Als Ergebnis erhalten wir Lucene-Dokumente. Diese sind anhand einer Punktezahl in absteigender Reihenfolge geordnet, wobei gilt, dass Dokumente, die der Anfrage am genauesten entsprechen, zuerst ausgegeben werden.

2.3.2 Indexierung

Für die Indexierung müssen Lucene-Dokumente erzeugt werden. Diese Dokumente enthalten Felder, welche einen Namen sowie einen Inhalt besitzen. Dabei gibt es verschiedene Felder: Normale Felder, bspw. IntField oder TextField, als auch Facetten Felder, bspw. FacetField. So kann bestimmt werden, auf welchen Feldern nach welchem Inhalt gesucht werden kann, sowie welche Facetten erstellt werden sollen, und welche Werte sie besitzen können.
Zusätzlich wird während der Indexierung eines der Felder als Default-Feld für die Suche definiert. Dadurch muss der Nutzer für eine Volltextsuche keinen Feldnamen angeben.

Die aus den vorliegenden Daten gebildeten Dokumente können dann zur Indexierung an Lucene übergeben werden. Für die Speicherung des Index bietet Lucene mehrere Indexstrukturen an:

- SimpleFSDirectory
- NIOFSDirectory
- MMapDirectory

Diese haben alle verschiedene Vor- und Nachteile. So ist das NIOFSDirectory wegen eines Sun JRE Bugs eine schlechte Wahl für Windows Systeme. Das MMapDirectory ist gut, falls in Relation zur Indexgröße viel virtueller Speicher vorhanden ist, wie beispielsweise bei einer 64-bit JRE. Dies führt dazu, dass keine einzig beste Implementierung vorhanden ist. Um diesem entgegen zu wirken, bietet Lucene eine Funktion an, die Rücksicht auf die Besonderheiten des vorliegenden Systems nimmt. Dabei wählt sie anhand einiger Kriterien die Beste der drei Implementationen aus [MHG10].
Für eine Facettensuche erstellt Lucene zwei Indexe; einen Volltextindex, sowie einen Taxonomieindex. Um Inkonsistenzen zu vermeiden, muss darauf geachtet werden, dass der Taxonomieindex immer vor dem Hauptindex committet wird.

2.4 BaseX

BaseX ist ein plattformunabhängiges XML-Datenbankmanagementsystem. Es beinhaltet einen XPath/XQuery 3.1 Prozessor und wird zur Speicherung, Visualisierung und Anfrage beliebiger XML-Dateien verwendet[Grü15].
Die BaseX zugrundeliegende Datenstruktur, wird im folgenden Kapitel genauer beschrieben.

2.4.1 Datenstruktur

Wird ein XML-Dokument in BaseX gespeichert, so wird dieses in Elemente, Attribute und Text-Knoten zerlegt und so abgespeichert, dass es aus diesen wieder rekonstruiert werden kann [Grü10].
Abbildung 2.4 zeigt die INFO STORAGE aus BaseX, welche diese Zerlegung schematisch darstellt.

PRE	DIS	SIZ	ATS	ID	NS	KIND	CONTENT
0	1	31	1	0	0	DOC	artists_test.xml
1	1	30	1	1	0	ELEM	artists
2	1	21	1	2	0	ELEM	artist
3	1	3	2	3	0	ELEM	name
4	1	1	1	4	0	ATTR	blup="1"
5	2	1	1	5	0	TEXT	Persuader, The
6	4	2	1	6	0	ELEM	realname
7	1	1	1	7	0	TEXT	Jesper DahlbÃ¤ck
8	6	15	1	8	0	ELEM	aliases
9	1	2	1	9	0	ELEM	name
10	1	1	1	10	0	TEXT	Dick Track

Abbildung 2.4: Info Storage

- PRE beinhaltet den Pre-Wert. Mit diesem kann der Knoten eindeutig identifiziert werden. Dieser Wert kann sich ändern, wenn dass XML-Dokument verändert wird und wird nicht gespeichert, sondern ergibt sich direkt aus der Tabellenposition.

- DIS gibt die Distanz zum Elternknoten an.

- SIZ gibt an, wie viele Knoten sich unterhalb dieses Elements befinden.

- ATS gibt an, wie viele Attribute das Element enthält. Zu beachten ist hier, dass der Wert bei keinem Attribut 1 ist.

- ID ähnlich wie Pre-Wert, allerdings bleibt dieser Wert für jedes Element immer gleich, auch wenn das Dokument verändert wird. Es existiert ein ID-PRE-Mapping.

- KIND gibt an, um was für einen Knoten es sich handelt, ELEM, ATTR, TEXT oder PI.

- CONTENT ist der Inhalt des Knotens.

Die Speicherung der XML-Elemente erfolgt in der Reihenfolge ihres Vorkommens im Dokument. Es handelt sich somit um eine preorder Baumstruktur.

3 Konzept

3.1 Ansätze

Bei der Implementation der Facettensuche in XML-Datenbanken sind mehrere Ansätze möglich. So gibt es statische als auch dynamische Methoden [BL11].

Bei statischen Methoden wird ein Schema während der Indexierung vorausgesetzt. Dies bedeutet, dass alle Facetten vorher bekannt sein müssen, die zu indexierende XML-Datenbank muss dementsprechend auch in diesem Schema vorliegen. Eine solche Methode findet in Solr[1] Anwendung.

Bei dynamischen Methoden hingegen sind die Facetten unabhängig von der Struktur, und werden durch den Nutzer selbst bestimmt. In einem solchen Konzept ist es dem Nutzer möglich selbst zu bestimmen, welche Facetten gebildet werden sollen. Zusätzlich kann der Nutzer auch die Werte der Facetten festlegen. So ist es möglich mehrere Werte zusammenzufassen.

In diesem Konzept wird ein Mittelweg gewählt. Ziel ist es, dem Nutzer beim Erstellen des Index einige Definitionsmöglichkeiten einzuräumen, den Großteil der Facettierung automatisch durchzuführen und anhand der Struktur der XML-Datenbank abzuleiten. Im Rest dieses Kapitels wird betrachtet, wie dieses Vorhaben in diesem Konzept umgesetzt wurde.

3.2 Speicherung

Wie im Kapitel Lucene bereits erwähnt, baut die Indexierung von Daten in Lucene auf der Konstruktion von Lucene-Dokumenten auf.

Für die Facettensuche braucht man ein Medium, auf welchem die Suche durchgeführt

[1] http://lucene.apache.org/solr/

werden soll. Wie etwa die Bücher im Amazon-Beispiel aus Kapitel 2, bei welchen deren Attribute wie etwa der Autor die Facetten bilden. In diesem Konzept wird dazu zunächst zur Indexierungszeit vom Nutzer bestimmt, auf welchem Element dieses Main-Element erstellt werden soll. Zu jedem dieser Elemente wird dann ein Lucene-Dokument erzeugt. Somit bekommt man dieses Element während der Suche auch wieder als Ergebnis zurück.

An dieser Stelle wird nun angegangen, dass die zurückgegebenen Lucene Suchergebnisse kompatibel zur XQuery- und XPath-Funktionalität von BaseX sein sollen. Dazu wird in jedem der erzeugten Lucene-Dokumente ein Int-Feld hinzugefügt, in welchem die BaseX-ID des dazugehörigen Main-Elementes gespeichert wird. Dadurch müssen nach der Lucene-Suche nur die BaseX-IDs der gefunden Dokumente an BaseX übergeben werden und nicht etwa das komplette XML Fragment und es lässt sich auf den originalen Daten aus BaseX weiterarbeiten.

Im nächsten Schritt werden aus den Kind-Elementen der gewählten Main-Elementen die Facetten erzeugt. Hierfür wird bei der Integration des Konzepts ein Schema über die Struktur der XML-Datenbank vorausgesetzt. Dieses Schema ist in Abbildung 3.1 dargestellt.

```
 <[medium]>
   <[facet1]> [Text]
      <[facet1]> [Text] </[facet1]>
   </[facet1]>
   <[facet2]> [Text] </[facet2]>
   <[facet2]> [Text] </[facet2]>
</[medium]>
<[medium]>
   <[facet1]> [Text]
      <[facet1]> [Text] </[facet1]>
   </[facet1]>
   <[facet2]> [Text] </[facet2]>
   <[facet2]> [Text] </[facet2]>
</[medium]>
```

Abbildung 3.1: Schema

Eine XML-Datei, die dieses Schema implementiert, ist in Abbildung 3.2 zu sehen.

```
 <medium>
   <art>Buch
      <art>Comic </art>
   </art>
   <name>Ich bin ein Buch</name>
   <author>Chesterfield McMillen</author>
   <author>ABC D</author>
   <year>1999</year>
   <location>X1</location>
</medium>
<medium>
   <art>CD</art>
   <name>Ich bin eine CD</name>
   <author>Hansi Hammer</author>
   <year>2003</year>
   <location>D3</location>
</medium>
<medium>
   <art>DVD</art>
   <name>Ich bin eine DVD</name>
   <author>Luli La</author>
   <year>1985</year>
   <location>D3</location>
</medium>
```

Abbildung 3.2: Beispiel XML mit Schema

Im Schema können folgende drei Fälle eintreten:

- Element ohne Kind-Elemente
- Element mit Kind-Elementen
- Mehrmalig vorkommendes Element

Im ersten Fall, dem einer flachen Facette, wird für das Element zum Lucene-Dokument ein Text-Feld hinzugefügt, sollte dieses einen Textwert enthalten. Der Facettenname wird dabei aus dem Elementnamen abgeleitet, der Facettenwert aus dem Text des Elements. Im zweiten Fall, dem einer tiefen Facette, wird wie im ersten Fall ein Text-Feld für al-

le gefundenen Textwerte erzeugt. Der Facettenwert wird dieses mal allerdings aus den gesammelten Texten aller Kind-Elemente gebildet. Hierzu werden diese in einem Array gespeichert und an das Facet-Feld übergeben. So ist es zur Suchzeit möglich, hierarchisch innerhalb der Suchkriterien zu suchen.
Im dritten Fall, dem einer multi-valued Facette, wird für jedes der gefundenen Elemente ein eigenes Text-Feld erstellt, wobei diese gleich wie im Fall 1 behandelt werden.
Dieses Schema wurde im Konzept mit Algorithmus 7, siehe Anhang, umgesetzt.

In Zeile 2 bis 5 des Algorithmus wird zunächst ein Verzeichnis für den Index sowie die Taxonomien der Facettensuche innerhalb des BaseX-Verzeichnisses der gegebenen Datenbank erstellt. Sind diese erzeugt, so wird ein Lucene-Index-Directory auf diesen geöffnet und ein neuer Index erstellt und der alte Index, sofern vorhanden, überschrieben.

In Zeile 7 bis 72 werden alle pre-Werte der Datenbank durchlaufen. Trifft man auf einen Element-Knoten, dessen Name gleich des angegebenen mainEle Strings ist, wird dessen Teilbaum durchlaufen und dann alle pre-Werte in der For-Schleife dieses Teilbaums übersprungen.
Für jedes gefundene Element wird ein Lucene-Dokument erstellt und die dazugehörige ID in einem Int-Feld gespeichert.
Am Schluss wird der Index-Writer geschlossen.

In Zeile 23 bis 62 werden die Teilbäume der gefunden Main-Elemente durchlaufen, um anhand des Schemas die Facetten zu erstellen. Dazu wird eine Tiefensuche ausgeführt, welche sich wegen der preorder Baumstruktur der Daten besonders eignet. Bei dieser Tiefensuche wird die Abfolge der Elementnamen und deren Textwerte in einer Liste gesammelt. Gespeichert werden allerdings nur Textwerte, die kürzer als 100 Bytes sind. Diese Einschränkung wird getroffen, da zu lange Facetten kaum von Nutzen sind. Für jeden Pfad, der dabei gefunden wurde, wird ein Facettenfeld erzeugt. Diesem wird als Namen der Wert übergeben, welcher als Erstes in der Liste steht, den des Kind-Elementes des Main-Elements. Als Wert werden dem Feld alle Text-Werte des Pfades übergeben. Zudem wird für jeden Text-Knoten, den wir während der Tiefensuche finden, ein Textfeld für den Volltextindex erstellt. Ist der komplette Teilbaum durchlaufen, wird dem Index-Writer das Lucene-Dokument übergeben.
Um einen Java Heap Space Error zu vermeiden, wenn die Datenbank zu groß ist, wird zusätzlich der Garbage Collector aufgerufen, sollte der belegte Speicher 80% des zur Verfügung stehenden Speichers überschreiten. Eine genauere Betrachtung hierzu wird im

Kapitel 4 Performance durchgeführt.

Der erzeugte Index ist in Segmenten organisiert. Dabei werden je nach Größe und Anzahl der zu indexierenden Dokumente mehrere Segmente erstellt. Dies hat den Vorteil, dass die Indexierungszeit verbessert wird. Allerdings ist die Suchzeit umso länger desto mehr Segmente erstellt werden. Um dies zu kompensieren stellt Lucene eine forceMerge Funktion zur Verfügung, mit welcher Segmente zusammengefügt werden können.
Dem Nutzer steht in diesem Konzept hierfür die optimize Funktion zur Verfügung. Als Argumente nimmt diese den Namen einer Datenbank sowie die Anzahl an gewünschten Segmenten entgegen.
Beim Verwenden dieser Funktion ist wichtig, dass der Vereinigungsprozess umso länger dauert je weniger Segmente gewünscht sind.

Algorithm 1 Optimize algorithm

```
1:  procedure OPTIMIZE(dbname, maxNumSeg)
2:      if maxNumSeg < NUMBER_DEFAULT_SEGMENTS then
3:          index ← FSDirectory.open(indexpath)
4:          writer ← new IndexWriter(index)
5:
6:          writer.forceMerge(maxNumSeg)
7:
8:          close writer
9:      end if
10: end procedure
```

3.3 Suchen

Die Suche in diesem Konzept ist ähnlich aufgebaut, wie in Kapitel 2.3 beschrieben. Es wird zunächst eine Base-Query erstellt, welche dann inkrementell über drill-down Suchen erweitert werden kann.
Damit aber nicht nur eine Suche gleichzeitig möglich ist, muss zunächst eine Suchinstanz erstellt werden, auf welcher die Anfrage bearbeitet werden kann. Dies wird über eine connect Funktion realisiert, die als Argument den Namen der Datenbank entgegen nimmt, auf der die Suche durchgeführt werden soll. Als Rückgabewert besitzt die Funktion einen Integerwert, über welchen auf die Suchinstanz zugegriffen werden kann.

Mit der search Funktion kann eine Base-Query auf der Suchinstanz erstellt werden. Die Funktion nimmt als Argument den Integerwert einer Suchinstanz sowie eine Anfrage als String entgegen. Als Rückgabewert hat sie den Integerwert der Suchinstanz. Die erstellte Basex-Query wird der Query q der Suchinstanz hinzugefügt.

Algorithm 2 Query algorithm

```
1: procedure QUERY(query)
2:     baseQuery ← new Query parsed by QueryParser("text", analyzer)
3:     q ← new DrillDownQuery(fconfig, baseQuery)
4: end procedure
```

Inkrementell erweitert werden kann die Suchanfrage mit der drillDown Funktion. Diese Funktion nimmt als Argument den Integerwert einer Suchinstanz, eine Facette auf der die drill-down Suche durchgeführt werden soll, sowie einen Drill-Down-Term entgegen. Diese werden dann zur Query q der Suchinstanz hinzugefügt. Als Rückgabewert hat die Funktion wieder den Integerwert der Suchinstanz.

Algorithm 3 DrillDown algorithm

```
1: procedure DRILLDOWN(dimension, drillDownTerm)
2:     q.add(dimension, drillDownField)
3: end procedure
```

Ist die Suchanfrage vollständig definiert, kann die Suche mittels der Funktionen result und facetResult ausgeführt werden, die als Argument den Integerwert einer Suchinstanz haben. Result gibt die Ergebnisse der Suche und facetResult die Facettenergebnisse der

Suche aus.

Algorithm 4 Result algorithm

```
1:  procedure RESULT
2:      reader ← open DirectoryReader on index
3:      searcher ← new IndexSearcher(reader)
4:
5:      topDocs ← searcher.search(q, NUMBER_HITS)
6:
7:      pres ← new ArrayList<Integer>()
8:      for all docId in topDocs do
9:          d ← searcher.doc(docId)
10:         id ← d.getField("pre")
11:         pres.add(dbdata.pre(id))
12:     end for
13:     nodes ← new ANode[pres.size()]
14:     pres.toArray(nodes)
15:     close reader
16:     return nodes
17: end procedure
```

In der result Funktion wird die drill-down Suche durchgeführt. Als Ergebnis werden die dabei gefundenen Lucene-Dokumente zurückgegeben. Wie viele Ergebnisse bestimmt werden hängt von NUMBER_HITS ab. Um nicht die kompletten XML-Fragmente an BaseX übergeben zu müssen, werden nur die gespeicherten BaseX-IDs der Dokumente zurückgegeben. Aus diesen werden dann in BaseX XML-Elemente erzeugt.

Algorithm 5 facetResult algorithm

```
1:  procedure FACETRESULT( )
2:      reader ← open DirectoryReader on index
3:      taxoReader ← open DirectoryTaxonomyReader on taxonomyindex
4:      searcher ← new IndexSearcher(reader);
5:
6:      ds ← new DrillSideways(searcher, fconfig, taxoReader)
7:      dsResult ← ds.search(q, NUMBER_HITS)
8:
9:      ftext ← dsResult.facets
10:
11:     fResult ← ftext.getAllDims(NUMBER_HITS_DIM)
12:
13:     fnodes ← LuceneIndex.elems(fResult, ftext)
14:     nodes ← new ANode[fnodes.size()]
15:     fnodes.toArray(nodes)
16:
17:     close taxoReader
18:     close reader
19:     return nodes
20: end procedure
```

In der facetResult Funktion wird eine drill-sideways Suche durchgeführt. Anstatt die Suchergebnisse zurückzuholen, werden die Facettenzahlen aller Dimensionen bestimmt. Allerdings innerhalb der Dimensionen nur die höchsten 10 Facettenzahlen. Aus den Facettenergebnissen von Lucene erstellen wir mit der elems Funktion eine Ausgabe im XML-Format.

Um eine Übersicht über alle Facetten einer Datenbank zu bekommen, können diese mit der facets Funktion aufgerufen werden. Als Argument nimmt die Funktion den Namen der Datenbank entgegen. Berechnet werden die Werte ähnlich wie in Algorithmus 6, allerdings mit einer Match-All-Documents-Query und einer normalen drill-down Suche anstatt einer drill-sideways Suche.

3.4 In BaseX

In BaseX ist das Konzept als XQuery-Modul implementiert. Dieses Modul kann im XQuery- und XML-Editor verwendet werden.
Wenn das Modul importiert wurde, können in diesem Editor die in den Kapiteln 3.2 und 3.3 beschriebenen Funktionen verwendet werden. Diese können mittels des Array Operators aneinandergereiht werden, wodurch der Integerwert der Suchinstanz direkt an die nächste Funktion weitergegeben wird.

```
import module namespace lucene = ‘‘http://basex.org/modules/Lucene‘‘;

lucene:connect(‘‘schemaxmltief‘‘)
=> lucene:search(‘‘Ich bin‘‘)
=> lucene:drillDown(‘‘location‘‘, ‘‘X1‘‘)
=> lucene:facetResult()
```

Abbildung 3.3: Facetten Suche

Die Suchanfrage aus Abbildung 3.3 ist auf diese Weise aufgebaut. Zunächst wird die Volltextsuche auf den String “Ich bin“ definiert, welche dann um eine drill-down Suche auf das Feld “location“ und den Wert “X1“ erweitert wird. Von diesen drill-down Suchen können beliebig viele hinzugefügt werden. In Abbildung 3.4 wird der Anfrage eine drill-down Suche auf das Feld “art“ und den Wert (“Buch“, “Comic“) hinzugefügt. Dieses Mal wird eine Sammlung übergeben, da auf einer tiefen Facette gesucht wird. Dabei müssen alle Werte, auf die gedrillt werden soll, der Reihe nach angegeben werden.

```
import module namespace lucene = ‘‘http://basex.org/modules/Lucene‘‘;

lucene:connect(‘‘schemaxmltief‘‘)
=> lucene:search(‘‘Ich bin‘‘)
=> lucene:drillDown(‘‘location‘‘, ‘‘X1‘‘)
=> lucene:drillDown(‘‘art‘‘, (‘‘Buch‘‘, ‘‘Comic‘‘))
=> lucene:facetResult()
```

Abbildung 3.4: Facetten Suche auf tiefer Facette

Soll auf multi-valued Facetten gesucht werden, so kann dies durch Kombination mehrere

drill-down Suchen realisiert werden. Bei Kombination mehrerer drill-down Suchen gilt, dass Suchen auf verschiedenen Dimensionen durch einen AND-Operator verbunden werden, Suchen auf derselben Dimension über einen OR-Operator.

```
import module namespace lucene = ‘‘http://basex.org/modules/Lucene‘‘;

lucene:connect(‘‘schemaxmltief‘‘)
=> lucene:search(‘‘Ich bin‘‘)
=> lucene:drillDown(‘‘location‘‘, ‘‘X1‘‘)
=> lucene:drillDown(‘‘author‘‘, ‘‘ABC D‘‘)
=> lucene:drillDown(‘‘author‘‘, ‘‘John Doe‘‘)
=> lucene:facetResult()
```

Abbildung 3.5: Facetten Suche auf multi-valued Facette

Im letzten Schritt der Anfrage werden die Ergebnisse der Suche aus Abbildung 3.4 angezeigt. Zu diesem Zweck wird der Anfrage am Schluss die result Funktion hinzugefügt. Wird das XQuery-Skript ausgeführt, erhält man die Ergebnisse der Suche. Diese bestehen aus den Main-Elementen, die wir bei der Indexierung bestimmt haben.

```
<medium>
  <art>Buch
    <art>Comic</art>
  </art>
  <name> Ich bin ein Buch </name>
  <author>Chesterfield McMillen</author>
  <author>ABC D</author>
  <year>1999</year>
  <location>X1</location>
</medium>
```

Abbildung 3.6: Suchergebnis

Mit der Funktion facetResult() statt result() können an dieser Stelle auch die Facettenergebnisse aus Abbildung 3.7 angezeigt werden.

Die Facettenergebnisse sind so aufgebaut, dass zunächst im Attributname des category

```
<facets>
  <category name=‘‘art‘‘>
    <entry number=‘‘1‘‘>
      <value>Buch</value>
      <entry number=‘‘1‘‘>
        <value>Comic</value>
      </entry>
    </entry>
  </category>
  <category name=‘‘location‘‘>
    <entry number=‘‘1‘‘>
      <value>X1</value>
    </entry>
  </category>
  .
  .
  .
</facets>
```

Abbildung 3.7: Facettenergebnis

Elements die Facette angezeigt wird. Das category Element kann mehrere entry Kind-Elemente haben, welche in sich verschachtelt sein können, falls es sich um eine tiefe Facette handelt, wie etwa in Abbildung 3.7. Die entry Elemente haben ein number Attribut, welches anzeigt, wie viele Treffer wir unter dem im value Element angezeigten Wert haben.
Die Facettenzahlen werden mit einer drill-siedeways Suche gebildet. Es werden also zusätzlich zu den Dimensionen auf denen eine drill-down Suche durchgeführt wurde auch die Facettenzahlen angezeigt, die erreicht werden könnten, sollte die drill-down Suche nicht durchgeführt werden.

4 Performance

Beim Indexieren und Durchsuchen von XML-Datenbanken ist sowohl von großem Interesse in welchem Bereich die Laufzeiten liegen, als auch ob die Anfrage korrekte Ergebnisse liefert.
Dafür werden die Laufzeiten der Suche für normalen Ergebnisse sowie für Facettenergebnisse betrachtet. Der inkrementelle Aufbau der Anfrage wird an dieser Stelle auch mit einbezogen. Deshalb wird inkrementell getestet. Gestartet wird mit einer Volltextsuche, welche dann um mehrere drill-down Suchen erweitert wird. In diesem Kontext werden auch multi-valued und tiefe Facetten auf Laufzeitunterschiede betrachtet.
Darauffolgend wird in diesem Kapitel auch die Facettenübersicht, sowie das Erstellen und Optimieren eines Index begutachtet.

Für die Tests werden verschieden große XML-Dateien genutzt.

- 100 MB artists.xml[1]; Main-Element: artist
  Die XML-Datei stammt von Discogs, ist somit frei verfügbar und enthält Informationen über alle Künstler ,welche ein Lied veröffentlicht haben, beispielsweise den Namen oder Alias des Künstlers. Als Main-Element wird dementsprechend der Künstler, in diesem Fall das artist Element, gewählt.

- 1 GB UniKN-flat.xml; Main-Element: Medium
  Die XML-Datei stammt von der Universität Konstanz. Sie enthält Daten der Universitätsbibliothek, beispielsweise Titel, Erscheinungsjahr, Thema und Genre des Mediums. Als Main-Element werden die Medien, die in der Bibliothek vorhanden sind, in der Datei das Medium Element, gewählt. Den Zusatz flat enthält die Datei, da diese für unsere Testzwecke keine Medium Elemente enthalten darf, die weitere Medium Element als Kinder besitzen.

[1] www.discogs.com/data/discogs_20080309_artists.xml.gz

- 18 GB releases.xml[2]; Main-Element: release
  Die XML-Datei stammt von Discogs, ist somit frei verfügbar und enthält Informationen über alle Veröffentlichungen, beispielsweise den Künstler, das Genre oder der Ort. Als Main-Element werden die Veröffentlichungen, in der Datei das release Element, gewählt.

Diese Dateien wurden gewählt, da sie alle ein geeignetes Main-Element besitzen. Des Weiteren besitzen sie eine Struktur, wie sie das Konzept voraussetzt, mit beliebigen Werten und Namen.

Alle Tests werden mit Version 8.1 von BaseX durchgeführt. Als Testmaschine wurde ein 64-bit Linux Mint 13 Maya System mit 128 GB RAM und einem AMD Opteron 2.6 GHz Prozessor verwendet.

Soweit nicht genauer spezifiziert, werden die Tests mit 10 GB Heap Space und den in Kapitel 3 vorgestellten Algorithmen durchgeführt.

4.1 Suche

Da bei der Indexierung der Daten, abhängig von ihrer Größe, verschieden viele Segmente erzeugt werden, optimieren werden alle Suchindexe vor den Tests auf eine Anzahl von 2 Segmenten optimiert.
Zum Testen der Suche wird für jede Datenbank eine Anfrage, die aus einer Volltextsuche und drei drill-down Suchen besteht genutzt.

```
lucene:connect(‘‘artists‘‘)
=> lucene:search(‘‘Jesper‘‘)
(: 243 Ergebnisse :)
=> lucene:drillDown(‘‘aliases‘‘, ‘‘Persuader, The‘‘)
(: 7 Ergebnisse :)
=> lucene:drillDown(‘‘aliases‘‘, ‘‘Dick Track‘‘)
(: 6 Ergebnisse :)
=> lucene:drillDown(‘‘name‘‘, ‘‘Lenk‘‘)
(: 1 Ergebnis :)
```

Abbildung 4.1: artistq

[2] www.discogs.com/data/discogs_20150301_releases.xml.gz

```
lucene:connect(‘‘UniKN-flat‘‘)
=> lucene:search(‘‘Lebenserinnerungen‘‘)
(: 198 Ergebnisse :)
=> lucene:drillDown(‘‘Year‘‘, ‘‘1902‘‘)
(: 3 Ergebnisse :)
=> lucene:drillDown(‘‘Subject‘‘, ‘‘Geschichte 1815-1890‘‘)
(: 2 Ergebnisse :)
=> lucene:drillDown(‘‘Signature‘‘, ‘‘gsn 86:m69/m64-2‘‘)
(: 1 Ergebnis :)
```

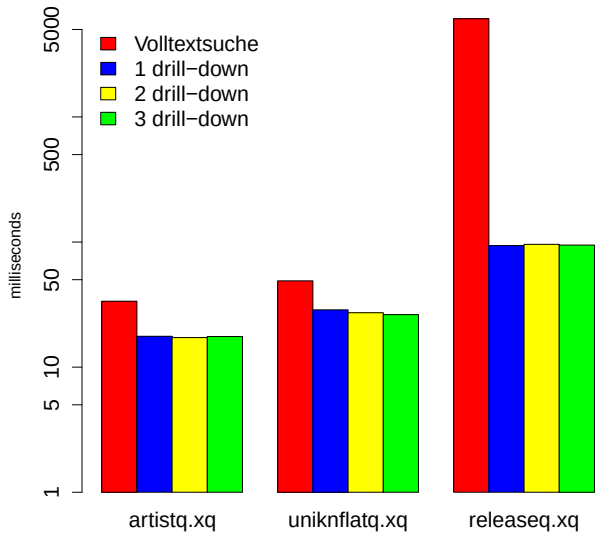
Abbildung 4.2: uniknflatq

```
lucene:connect(‘‘releases‘‘)
=> lucene:search(‘‘Stockholm‘‘)
(: 16098 Ergebnisse :)
=> lucene:drillDown(‘‘artists‘‘, ‘‘Persuader, The‘‘)
(: 12 Ergebnisse :)
=> lucene:drillDown(‘‘extraartists‘‘, ‘‘Jesper Dahlbäck‘‘)
(: 12 Ergebnisse :)
=> lucene:drillDown(‘‘genres‘‘, ‘‘Electronic‘‘)
(: 12 Ergebnisse :)
```

Abbildung 4.3: releaseq

Führt man die Anfrage mit der result Funktion auf der Testmaschine aus, erhält man für die jeweiligen Datenbanken die Ergebnisse aus Abbildung 4.4. Vor allem an der releaseq Anfrage ist deutlich zu erkennen, dass die Laufzeit der Suche stark von der Anzahl an Ergebnissen abhängt. Werden die Ergebnisse durch drill-down Suchen eingeschränkt, so verbessert sie die Laufzeit entsprechend.
Zusätzlich muss in Lucene angegeben werden wie viele Ergebnisse bestimmt werden sollen. Werden beispielsweise nur die 100 wichtigsten Ergebnisse der Anfrage bestimmt, so verbessert sich die Laufzeit von 2088.44 ms auf 98.57 ms. Mit einer Einschränkung auf die 100 wichtigsten Ergebnisse befinden sich die Zeiten alle unter 100 ms und sind somit in der selben Größenordnung wie von Solr Anwendungen, etwa Netflix, dessen Anfragen ebenfalls in unter 100 ms laufen[3].
Durch die inkrementellen Tests ist zudem zu erkennen, dass das Hinzufügen einer drill-down Suche keine Erhöhung oder Verringerung der Suchlaufzeit zur Folge hat. Aus der Anfrage artistq ist zu entnehmen, dass auch die Suche auf multi-valued Facetten keinen

[3] https://wiki.apache.org/solr/SolrPerformanceData

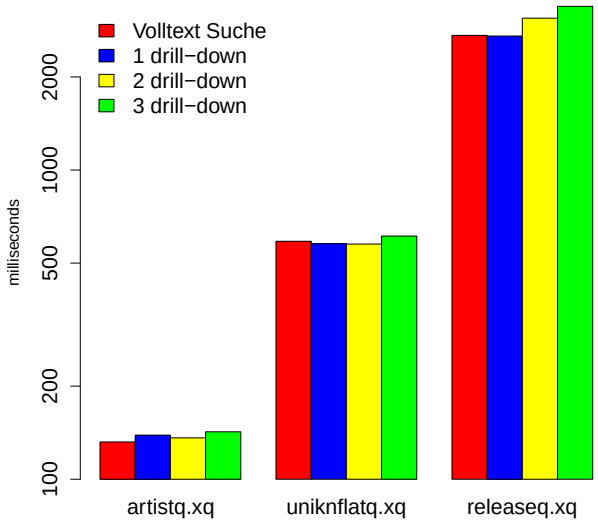


	artistq.xq	uniknflatq.xq	releaseq.xq
Volltextsuche	36.34	56.14	2088.44
1 drill-down	17.65	28.73	93.69
2 drill-down	17.23	27.22	95.85
3 drill-down	17.56	26.28	94.59

Abbildung 4.4: Laufzeit der result Funktion in Millisekunden

Einfluss hat.

Wird die Anfrage mit der facetResult Funktion auf der Testmaschine ausgeführt, so erhält man für die Datenbanken die Ergebnisse aus Abbildung 4.5. Die Zeiten sind langsamer als die der normalen Suchen. Um diese zu verringern kann in Lucene die Einschränkung getroffen werden, für wie viele Facettenwerte pro Facette die Facettenzahlen bestimmt werden sollen. Aber auch wenn diese auf 10 pro Dimension eingeschränkt werden, erhält man keine signifikant besseren Laufzeiten. So läuft die releaseq Anfrage mit 3 drill-down Suchen und einer Einschränkung auf 10 Werte in 3260.75 ms, statt 3383.74 ms.



	artistq.xq	uniknflatq.xq	releaseq.xq
Volltextsuche	132.01	588.31	2725.34
1 drill-down	138.89	577.57	2712.23
2 drill-down	136.10	576.23	3097.96
3 drill-down	142.41	611.72	3383.74

Abbildung 4.5: Laufzeit der facetResult Funktion in Millisekunden

rank	self	accum	count	trace	method
1	16.01%	16.01%	69	301553	org.apache.lucene.facet.taxonomy.IntTaxonomyFacets.getTopChildren
2	14.15%	30.16%	61	301509	org.apache.lucene.facet.taxonomy.directory.TaxonomyIndexArrays.initParents
3	4.87%	35.03%	21	301532	org.apache.lucene.facet.taxonomy.IntTaxonomyFacets.getTopChildren

TRACE 301553:

org.apache.lucene.facet.taxonomy.IntTaxonomyFacets.getTopChildren(IntTaxonomyFacets.java:128)

org.apache.lucene.facet.taxonomy.TaxonomyFacets.getAllDims(TaxonomyFacets.java:93)

org.apache.lucene.facet.MultiFacets.getAllDims(MultiFacets.java:81)

org.basex.modules.LuceneIndex.getFResult(LuceneIndex.java:130)

Wird ein Profiling der facetResult Funktion betrieben, ist zu erkennen, dass der Großteil der Zeit innerhalb Lucene beim Berechnen der Facettenzahlen für alle Dimensionen benötigt wird. Da bei dieser Funktion die exakten Werte der Suche zurückgegeben werden sollen, müssen diese jedoch zwangsweise berechnet werden. Eine weitere Optimierung der Laufzeit ist nicht möglich, ohne die Ergebnisse zu beeinträchtigen.

Ein weiterer Flaschenhals, der für die facetResult Funktion festgestellt werden kann, ist der verfügbare Heap Space. Führt man die releaseq Anfrage mit weniger als 10 GB RAM aus, so erhält man für einige Werte einen Heap Space Error.

Heap Space	10 GB	4 GB	1 GB	512 MB
Runtime	3383.74	3386.61	ERROR	ERROR

Tabelle 4.1: Laufzeit der facetResult Funktion in Millisekunden in Betrachtung der Heap Space Größe

Da die artistq Anfrage auch bei 512 MB Heap Space durchführbar ist, benötigt das Modul abhängig von der Größe der Datenbank einen gewissen Heap Space um die facetResult Funktion durchführen zu können. Wie im Algorithmus FacetResult in Kapitel 3 beschrieben, erfolgt das Zählen der Facetten nach einem Methodenaufruf komplett Luceneintern. Eine Optimierung zur Verringerung des Heap-Space-Gebrauchs ist an dieser Stelle nicht möglich.

Auch bei der facet Funktion können, abhängig von der Größe der Datenbank, stark ansteigende Laufzeiten beobachtet werden.

Datenbank	artists	UniKN-flat	releases
Laufzeit alle Werte	14718.59	111963.62	-
Laufzeit 1000 pro dim	388.9	1664.07	18278.76
Laufzeit 100 pro dim	275.95	1119.62	14996.49
Laufzeit 10 pro dim	255.99	1041.13	13837.04

Tabelle 4.2: Laufzeit der facet Funktion in Millisekunden

Im Gegensatz zur facetResult Funktion gibt es hier keinerlei Einschränkungen der Facettenwerte. Somit fällt die Anzahl der zu zählenden Werte deutlich stärker ins Gewicht. Dies führt soweit, dass der Test ohne weitere Optimierungen bei der releases Datenbank nicht durchführbar ist.

Doch selbst wenn an dieser Stelle wie bei der facetResult Funktion die Anzahl der zu bestimmenden Facettenzahlen pro Dimension auf 10 verringert wird, läuft die Funktion auf der releases Datenbank noch immer über 14 Sekunden. Dies ist darauf zurückzuführen, dass jedes im Index befindliche Dokument betrachtet werden muss, wodurch die Laufzeit der Funktion stark beeinflusst wird.

rank	self	accum	count	trace	method
1	59.72%	59.72%	9187	300503	org.apache.lucene.util.BitSetIterator.advance
2	8.31%	68.04%	1279	300499	org.apache.lucene.facet.taxonomy.FastTaxonomyFacetCounts.count
3	7.18%	75.22%	1105	300576	org.apache.lucene.facet.taxonomy.FastTaxonomyFacetCounts.count

TRACE 300503:

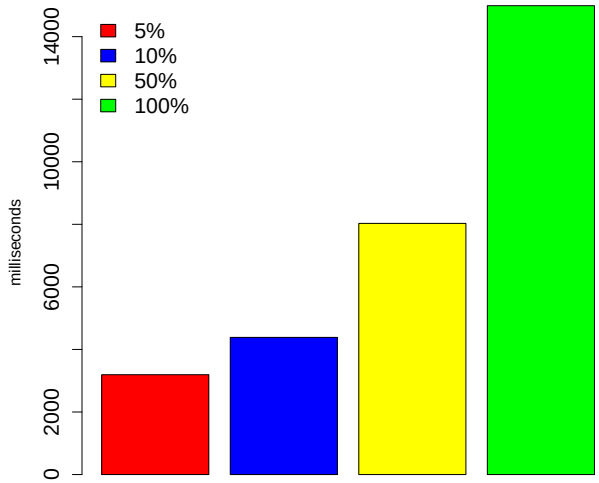
org.apache.lucene.util.BitSetIterator.advance(BitSetIterator.java:77)

org.apache.lucene.util.BitSetIterator.nextDoc(BitSetIterator.java:69)

org.apache.lucene.facet.taxonomy.FastTaxonomyFacetCounts.count(FastTaxonomyFacetCounts.java:62)

org.apache.lucene.facet.taxonomy.FastTaxonomyFacetCounts.<init>(FastTaxonomyFacetCounts.java:49)

Da es sich bei der facet Funktion im Gegensatz zur facetResult Funktion um eine Übersichtsfunktion handelt, bietet es sich an dieser Stelle an, nicht die genauen Facettenzahlen zu bestimmen, sondern eine Statistik zu erstellen. Lucene bietet zu diesem Zweck eine Sampling-Optimierung an, welcher eine sample-size übergeben werden kann.
Die Größe der sample-size hat, wie die Profilingergebnisse bereits vermuten lassen, einen sehr großen Einfluss auf die Laufzeit, siehe Abbildung 4.6. Mit Hilfe dieser Optimierung kann noch einmal einiges an Zeit eingespart werden. An diesem Punkt wird dem Nutzer die Bestimmung der sample-size offen gelassen, da diese um ein gutes Resultat zu erhalten zu stark von den zugrundeliegenden Daten abhängt. Hierzu wird der facets Funktion ein zweites Argument hinzugefügt, mit welchem die gewollte sample-size angeben werden kann.

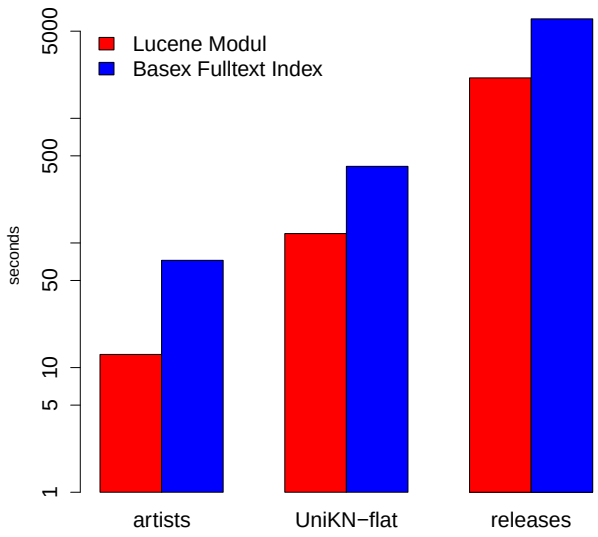


sample size	5%	10%	50%	100%
Laufzeit	3192.62	4384.96	8029.36	14989.80

Abbildung 4.6: Laufzeitvergleich sample size in Millisekunden

4.2 Indexierung

Bei der Erstellung des Index werden wieder die drei genannten XML-Dateien getestet. In Abbildung 4.7 werden die Laufzeiten für die Erstellung eines Lucene Index und die eines BaseX Volltextindex verglichen. Bei diesen Zeiten sollte zudem berücksichtigt werden, dass im Lucene-Modul, im Gegensatz zu BaseX, zusätzlich zum Volltextindex auch ein Taxonomy-Index erstellt wird.
Wie im luceneIndex Algorithmus aus Kapitel 3.2 bereits beschreiben, enthält dieser einen Aufruf des Garbage Collectors, sollte die Speicherbelegung über 80% betragen. Dies hat den Grund, dass bei der Erzeugung eines Index für große Daten die Größe des Heap Space ein Flaschenhals darstellt. Der Laufzeittest wurde bereits mit dem Aufruf des Garbage Collectors und 10 GB RAM durchgeführt. Wird der Test ohne diesen und bei unterschiedlichen RAM Werten durchgeführt, so erhält man bei einigen Paarungen von Datenbanken und RAM-Werten einen Java Heap Space Error. Ein weiterer Effekt, den man auch erkennen kann ist, dass es eine Heap Space Größe gibt, bei der die Laufzeit ein Minimum erreicht. Heap Space Größen über und unterhalb dieses Wertes verschlechtern die Laufzeit. Zurückzuführen ist dies auf große GC Pausen bei ungünstigen Heap Space



Datenbank	artist	UniKN-flat	release
BaseX Volltext	12755.26	118887.78	2110551.54
Lucene-Modul	72635.72	411919.8	6282410.86

Abbildung 4.7: Indexerstellung Laufzeiten in Millisekunden

Größen. Wie viel Speicher dabei optimal gebraucht wird, hängt stark von der Größe der Datenbank, aber auch von der Größe der Lucene-Dokumente ab[4]. So steigt der benötigte Heap Space mit wachsender Größe der Lucene-Facettenfelder enorm an. Im vorgestellten Modul sind diese auf eine Länge von 100 Byte beschränkt. Würde man alle Längen zulassen, so benötigt die Indexerstellung der releases Datenbank über 10 GB statt 3 GB Speicher.

RAM	2 GB	3 GB	5 GB	10 GB
Laufzeit	ERROR	6929005.09	5973568.52	6282410.86

Tabelle 4.3: Indexerstellung Laufzeiten in Millisekunden

[4]https://wiki.apache.org/solr/SolrPerformanceProblems

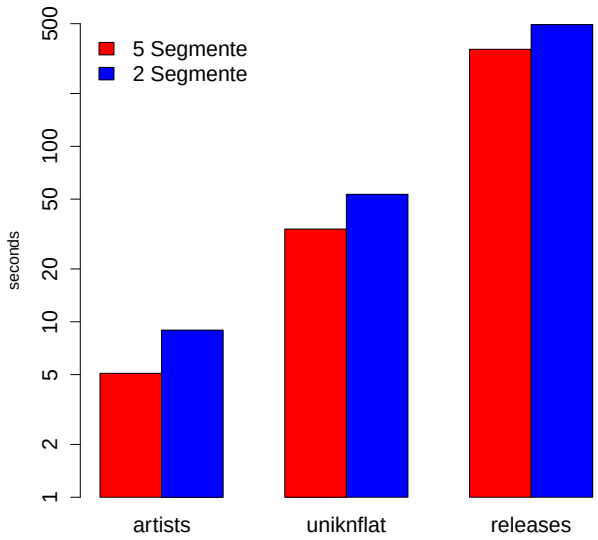
Zur Vermeidung eines Heap Space Errors gibt es mehrere Möglichkeiten[5]:

- Verringerung der Lucene-RAM-Buffer-Size
- Häufigere commit-Aufrufe des Lucene-Writers
- Aufrufe des Garbage Collectors

Da die RAM-Buffer-Size mit 16MB bereits auf einen sehr kleinen Wert eingestellt ist (vgl. Solr default 100 MB), muss die Optimierung in unserem Fall durch den Garbage Collector oder häufigeres commiten erreicht werden.
In beiden Fällen muss ein Zeitpunkt bestimmt werden, zu welchem der Commit oder Garbage Collector aufgerufen wird. Sinnvoll ist hierbei natürlich diesen vom belegten Speicher abhängig zu machen. Damit dabei der volle Speicher nicht erreicht wird, und somit die verbrauchte Zeit im Garbage Collector zu verringern, wählen wird der Zeitpunkt bei 80% des belegten Speichers gewählt.
Doch wieso ein GC-Aufruf satt einem Commit? Ein wichtiger Grund hierfür ist, dass ein zu häufiges commiten zu Performanceproblemen führen kann. Dies tritt auf, wenn ein zweiter Commit aufgerufen wird während der erste noch nicht abgeschlossen ist[5]. Da der Zeitpunkt des Aufrufes von der Menge an belegtem Speicher abhängt, tritt dieses Problem bei zu wenig Heap Space sehr häufig auf.
Bei kleinem Heap Space kann das soweit führen, dass der Index nicht mehr erstellt werden kann. Bei gleichen Einstellungen ist die Indexerstellung mit einem GC Aufruf noch durchführbar.

Die letzte Funktion, die dem Nutzer in der Implementierung noch zur Verfügung steht, ist die optimize Funktion. Die entsprechende Lucene-Funktion wird bereits im Index Algorithmus verwendet. Allerdings werden die beim Indexieren angelegten Segmente nur auf 5 Stück vereint. Der Grund hierfür sind die stark ansteigenden Laufzeiten. Dieser Kompromiss wird eingegangen, um eine sinnvolle Suchzeit zu garantieren ohne die Laufzeit der Index Funktion in die Höhe zu treiben. Sollte der Nutzer den Index der releases Datenbank von 5 auf 2 Segmente verringern, so benötigte diese nicht die vollen 494013.85 ms sondern nur etwa 160000 ms, was der Differenz der Zeiten für 5 und 2 Segmenten entspricht. Es wird somit keine Zeit verschwendet, wenn in der Indexfunktion nur eine Optimierung auf 5 Segmente durchgeführt wird.

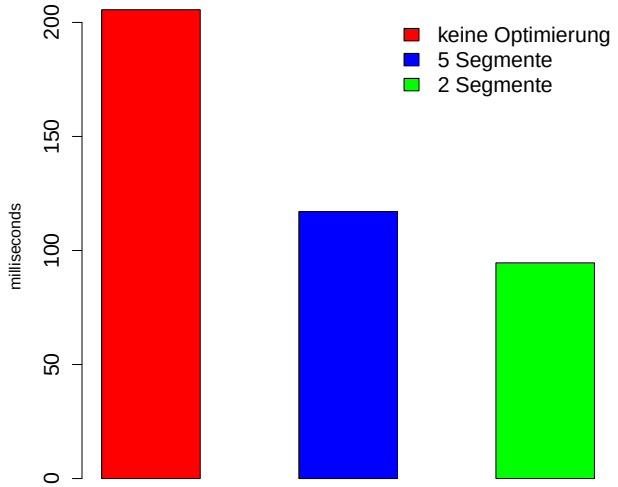
[5] https://wiki.apache.org/solr/SolrPerformanceProblems



Database	artists	uniknflat	realeases
5 Segmente	5085.4	33776.88	357254.44
2 Segmente	8369.0	53280.09	494013.85

Abbildung 4.8: Laufzeitvergleich optimize Funktion in Abhängigkeit von Anzahl der Segmenten in Millisekunden

Ein weiteres Argument für eine Verringerung sind die daraus resultierenden Suchzeiten. Führt man die Tests der releasesq Anfrage mit 3 drill-down Suchen mit unterschiedlicher Anzahl von Segmenten durch, ohne Optimierung, mit 5 Segmenten und mit 2 Segmenten, dann ergeben sich große Unterschiede. Mit einer Optimierung auf 5 Segmente kann die Laufzeit von 205 ms auf 117 ms reduziert werden. Weitere Zusammenschlüsse reduzieren die Laufzeit zwar, relativ zur Einsparung ist der Aufwand jedoch größer. Darum wird der Index beim Erstellen zunächst auf 5 Segmente optimiert.



keine Optimierung	5 Segmente	2 Segmente
205.59	117.08	94.59

Abbildung 4.9: Laufzeitvergleich der releaseq in Abhängigkeit des Optimierungsgrads in Millisekunden

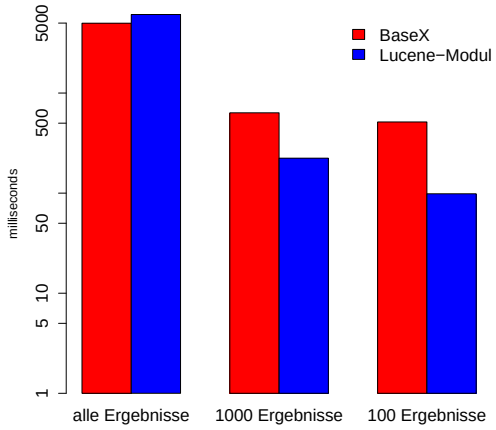
4.3 XQuery

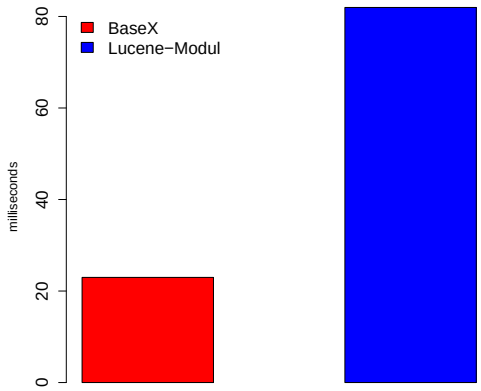
Um die Korrektheit des Moduls zu zeigen, werden die Ergebnisse von Lucene-Anfragen mit einer äquivalenten XQuery-Anfrage verglichen. Für diesen Test wird die artists Datenbank genutzt. Als XQuery-Äquivalent für die artistq Anfrage wird die Anfrage aus Abbildung 4.10 verwendet.

```
for $x score $score in db:open("artists")/artists/artist[.//text() contains text
"Jesper"]
where $x/aliases[.//text() contains text "Persuader, The"]
and $x/aliases[.//text() contains text "Dick Track"]
and $x/name[.//text() contains text "Lenk"]
order by $score descending return $x
```

Abbildung 4.10: XQuery-Äquivalent der artistq Anfrage

Für beide Anfragen erhält man die selben 1/6/7/243 Ergebnisse. Interessant sind neben der Korrektheit des Modells im Vergleich zur XQuery Implementation in BaseX natürlich die Laufzeiten beider Anfragen. Um in diesem Test vergleichbare Ergebnisse zu bekommen, wird mit dem CREATE INDEX fulltext Kommando einen Volltextindex in BaseX gebildet. Verglichen werden dabei allerdings nur die Volltextsuchen von BaseX und dem Lucene-Modul. Zudem werden zwei verschiedene Volltextanfragen getestet. Die erste Anfrage nach “Sol“liefert 24458, die zweite Anfrage nach “Morgon Sol“nur 3 Ergebnisse.





Ergebnisse	BaseX	Lucene-Modul
alle	4987.26	6096.08
1000	634.93	223.52
100	514.35	98.57

Ergebnisse	BaseX	Lucene-Modul
alle	22.98	81.97

Abbildung 4.11: Vergleich der Volltextsuchen, in Abhängigkeit der Anzahl an Ergebnissen, in Millisekunden

Aus den Laufzeiten dieses Tests, zu sehen in Abbildung 4.11, ist zu erkennen, dass das Öffnen des Luceneindex für Anfragen mit kleinen Ergebnismengen einen deutlichen Einfluss hat. Auch bei großen Mengen hat die BaseX Volltextsuche einen Laufzeitvorteil. Dieser kommt Zustande, da in der result Funktion des Lucene-Moduls die gefundenen Dokumente nicht direkt ausgegeben werden können. Diese müssen durchlaufen werden um die gespeicherten BaseX-IDs zu bekommen. Erst mit diesen BaseX-IDs können die Ergebnisse ausgegeben werden, um eine XQuery/XPath-Kompatibilität zu ermöglichen. Wird die Anzahl der zu bestimmenden Ergebnisse auf 1000 oder 100 eingeschränkt, so

ist das Lucene-Modul trotz dieses Nachteils schneller. Da Lucene standardmäßig angeben werden muss, wie viele Ergebnisse bestimmt werden, kann die Einschränkung an dieser Stelle vorgenommen werden. Anzumerken ist an dieser Stelle, dass Lucene immer eine ein Array der angegeben Größe erstellt welches, sollten weniger Ergebnisse vorliegen, mit Dummy-Variablen gefüllt wird. Standardmäßig den maximal Wert zu Übergeben ist also nicht zu empfehlen. Besser ist es einen festen Wert zu bestimmen oder der Nutzer diese Entscheidung durch ein zusätzliches Argument offen zu lassen. Bei der XQuery-Anfrage wird die Ergebnismenge über einen [1 to 100] und [1 to 1000] Block eingeschränkt.

5 Updates

Die Daten einer XML-Datenbank können sich ständig verändern. Bis jetzt konnte in diesem Konzept aber nur ein neuer Index für eine Datenbank erstellt werden. Was soll passieren, wenn sich nur ein Teil der Daten verändert? Selbst für eine kleine Änderung an der Datenbank den Index komplett neu zu erstellen ist für große Datenbanken eine ineffiziente Möglichkeit. Sind viele Änderungen zu erwarten ist eine Neuerstellung des kompletten Index kaum durchführbar.
Man braucht also eine Möglichkeit einzelne Daten im Index zu aktualisieren und hinzuzufügen. Welche Methoden Lucene dazu anbietet wird im Folgenden betrachtet.

5.1 Updates in Lucene

In Lucene können wir zur Aktualisierung des Index existierende Dokumente löschen und Neue hinzufügen. Beim Löschen gibt es mehrere Möglichkeiten anzugeben, welche Dokumente entfernt werden sollen:

- Löschen der Elemente, welche das Ergebnis einer Anfrage sind. Dabei können auch mehrere Anfragen angeben werden, wobei das zu löschende Dokument nur das Ergebnis von einer sein muss:

  deleteDocuments(Query... queries)

- Löschen der Elemente, welche einen Term enthalten. Ein Term enthält dabei ein Feld sowie einen Text. Dabei können auch mehrere Terme angegeben werden, wobei das zu löschende Dokument mindestens einen der angegebenen Terme enthalten muss:

  deleteDocuments(Term... term)

Neue Dokumente einfügen kann beim Updaten auf die gleiche Weise durchgeführt werden wie beim Indexieren einer kompletten Datenbank. Es müssen wieder Lucene-Dokumente

erstellt werden und aus den Daten mit Feldern gefüllt werden. Die genau Abfolge dazu ist in Kapitel 3 nachzulesen. Diese Dokumente können dann direkt mittels

addDocument(Iterable<? extends IndexableField> doc)

an den Index übergeben werden.

Im Fall des Updatens eines Dokuments gibt es in Lucene auch die Möglichkeit das Löschen und Hinzufügen mit einer Funktion auszuführen.

updateDocument(Term term, Iterable<? extends IndexableField> doc)

Dabei werden zunächst die Dokumente gelöscht, welche term enthalten, und danach das übergeben Dokument dem Index hinzugefügt.

5.2 Konzept

Für das Konzept bedeuten Updates, dass nach einer Änderung der Daten in BaseX ein neues Lucene-Dokument erstellt wird, sowie falls vorhanden, das alte Dokument aus dem Index gelöscht werden muss.
Die Erstellung der Lucene-Dokumente kann auf die gleiche Weise durchgeführt werden wie bei der Erstellung eines neuen Index. Allerdings muss für Updates nicht die komplette Datenbank durchlaufen werden, sondern die zu updatenden Elemente können direkt per Tiefensuche durchlaufen werden.
Um Dokumente aus dem Index zu löschen, können die gespeicherten BaseX IDs genutzt werden. Dazu löscht man über die deleteDocuments Funktion Dokumente, welche eine der angegebenen IDs besitzen.
Eine entscheidende Frage die sich dabei stellt, ist der Zeitpunkt der Updates. Auch hierfür gibt es mehrere Möglichkeiten:

- Updates bei jeder durchgeführten Änderung
- Updates in festen Intervallen
- Update wird vom Nutzer aufgerufen

Bevor die Updates jedoch durchführt werden können, benötigt man die Main-Elemente die verändert wurden. Wie diese bestimmt werden könnten, wird im nächsten Abschnitt betrachtet.

5.2.1 Was Updaten

Auch für die Bestimmung der zu updatenden Main-Elemente gibt es mehrere Möglichkeiten. Gut geeignet wäre eine Triggerimplementierung. So könnte für jede Änderung, die in BaseX vorgenommen wird, überprüft werden, ob die Änderung ein Main-Element des Index betrifft.
Für diese Überprüfung muss jedes Eltern-Element angeschaut werden. Da in Facettensuchen meist flache Facetten oder Facetten mit geringer Tiefe vorkommen, kann dies schnell durchgeführt werden. Nutzt man eine XML-Datenbank ohne Triggerimplementierung, wie es mit BaseX der Fall ist, gestaltet sich die Aufgabe schwieriger.

Updatevorschlag ohne Triggerimplementierung:
Sind die Einheiten der Datenbank gleich den Main-Elementen des Luceneindex, können ähnlich der Indexerstellung alle Einträge der Datenbank durchlaufen werden. Diese werden nun durch eine Luceneindex-Flag auf Änderungen überprüft. Die Flag wird auf FALSE gesetzt, sobald eine Änderung auftritt. Wird ein Index erstellt oder ein Update durchgeführt, so wird diese Flag auf TRUE gesetzt. Anhand dieser Luceneindex-Flag kann somit entschieden werden, ob für diesen Eintrag ein Update durchgeführt werden muss oder nicht.
Das Löschen von nicht mehr vorhandener Dokumenten gestaltet sich etwas komplizierter. Um festzustellen, welche Main-Elemente gelöscht wurden, benötigt man Wissen darüber, welche Dokumente sich momentan im Index befinden. Dazu können alle IDs, für die eine Lucene-Dokument dem Index hinzugefügt wird, zusätzlich noch einer Sammlung hinzugefügt werden. Diese Sammlung müsste dann beim Durchlaufen der Datenbank auf Unterschiede mit den gefundenen Main-Elementen untersucht werden und die Differenz der beiden Mengen aus dem Index entfernt werden.

Sind die zu updatenden Elemente bestimmt, werden diese an den Luceneindex übergeben.

5.2.2 Live Update

Will man den Index synchron mit der Datenbank halten, so muss direkt nach einer Änderung an der Datenbank auch der Index upgedatet werden.
Dazu wird direkt beim Auftreten einer Änderung die updateDocuments bzw. deleteDocuments Funktion aufgerufen.

Eingeschränkt im Bezug auf BaseX wird diese Methode durch das Nichtvorhandenseins eines Triggers. Somit hat man keine direkte Möglichkeit festzustellen, wann eine Änderung auftritt. In diesem Fall wäre es somit sinnvoll, eine Intervall oder Event Methode vorzuziehen.
Ein negativer Aspekt dieser Methode ist es, dass jedes Update einzeln durchgeführt und für jedes Update eine Anfrage durchgeführt werden muss.

5.2.3 Intervall Update

Um nicht für jedes Update eine eigene Anfrage durchzuführen, kann eine Intervall Methode gewählt werden. Wird mit einer Triggerimplementation gearbeitet, können alle IDs der zu aktualisierenden und zu löschenden Elemente in Sammlungen gespeichert werden. Diese können dann zum gewählten Zeitpunkt mit einer deleteDocuments und einer updateDocuments Funktion ausgeführt werden.
Im Gegensatz zu Live Updates kann diese Methode auch ohne Trigger implementiert werden. Hierzu muss zum gewählten Zeitpunkt der in Kapitel 5 “Was Updaten“ vorgeschlagene Ansatz verwendet werden.

5.2.4 Event Update

Bei dieser Methode würde der Nutzer mittels eines Funktionsaufrufs das Update selbst durchführen. Ähnlich zur index Funktion des Moduls würde der Name einer Datenbank übergeben werden, für welche der Index aktualisiert werden soll.
Auch die Event Methode kann mit einem Trigger oder dem in Kapitel 5 “Was Updaten“ vorgeschlagenen Ansatz implementiert werden. Wird ein Trigger realisiert, werden die IDs der zu ändernden oder löschenden Elemente in einer Sammlung gespeichert. Ohne Trigger wird beim update Funktionsaufruf der vorgeschlagene Ansatz ausgeführt.

5.2.5 Fazit

Welche dieser Möglichkeiten gewählt werden sollte, hängt sowohl von den Daten als auch den Kapazitäten und Wünschen des Nutzers ab.
Soll der Index möglichst synchron mit der Datenbank sein, so ist ein Live Update die beste Wahl. Mit einer Intervall Methode wäre abhängig von der Länge der Intervalle der Index für diese Zeit asynchron. Bei einer Event Methode müsste die update Funktion, um den Index einigermaßen aktuell zu halten, sehr häufig aufgerufen werden.

Ein Nachteil ist jedoch, dass für jedes Update eine Anfrage ausgeführt werden muss, sowie der Index geschlossen werden muss.
Treten Änderungen an den Daten sehr häufig auf, kann dies bei einer Live Update Methode zu Problemen führen. Eine Intervall Update Methode wäre an dieser Stelle eher geeignet. Diese hat den Vorteil, dass die Änderungen gesammelt mittels einer Anfrage durchführbar sind, anstatt für jede dieser Änderungen eine eigene Anfrage durchführen zu müssen. Die Website JobHits updated ihren Index beispielsweise in Intervallen von 1min [1].
Ist es unwichtig den Index jederzeit mit der Datenbank synchron zu halten, so stellt die Event Methode eine Alternative dar. Dadurch wird dem Nutzer ermöglicht, selbst zu bestimmen, wann er einen aktualisierten Index erstellen möchte. Auch hier hat man gegenüber der Live Update Methode den Vorteil, die Updates gesammelt durchführen zu können. Im Vergleich zur Intervall Methode muss das Update vom Nutzer selbst aufgerufen werden.

[1] https://wiki.apache.org/solr/SolrPerformanceData

6 Fazit

Im Rahmen dieser Arbeit wurde in BaseX ein Lucene-Modul implementiert, welches Volltextanfragen sowie Facettierung unterstützt. Die Suche ermöglicht viele Anfragetypen wie etwa fuzzy- und wildcard-Anfragen. Die Facettierung bietet die Möglichkeit, aus einer Datenbank mittels Angabe eines Main-Elements einen Taxonomy-Index zu erstellen. Bei diesem können neben flachen Facetten auch tiefe und multi-valued Facetten gebildet werden.
Zwischen Lucene und BaseX hat man eine sehr schmale Schnittstelle. So müssen nur die BaseX-IDs statt ganzen XML-Fragmenten übergeben werden. Dadurch ist die Suche komplett kompatibel mit XQuery sowie XPath. Dies hat den Vorteil, dass die Ergebnisse auf den Originaldaten direkt weiter bearbeitet werden können. Im Vergleich zu XQuery ist es mit dem Lucene-Modul möglich in kürzerer Zeit die selben Ergebnisse zu erzielen. Zusätzlich können zu jeder Anfrage die Facettenergebnisse angezeigt werden.
Liegen die Daten bereits in geeigneter Form zur Facettenbildung vor, so kann ein Index für Volltextanfragen und Facettensuche erstellt werden, ohne dabei weitere Einschränkungen vornehmen zu müssen.
Die Volltextsuche liefert vergleichbare Laufzeiten und Möglichkeiten wie die Volltextsuche von BaseX. Mit Hilfe der Facettierung wurde diese Suche im Lucene-Modul stark erweitert, so konnten durch drill-down Suchen die Laufzeiten der Volltextanfragen stark verbessert werden. Durch Anzeige von Facettenwerten und Facettenzahlen, als Übersicht der kompletten Datenbank oder auch für beliebige Anfragen, gestaltet sich die Suche sehr explorativ.

6.1 Ausblick

Es gibt noch einige Einschränkungen im Modul. Neben den Updates beziehen diese sich vor allem auf die Indexerstellung. Durch die Voraussetzung der Struktur des Moduls, müssen die Daten der Datenbank verändert werden, um eine geeignete Suche definieren zu können. Zwar lässt das Modul die Definition eines Mediums zu, jedoch sind nicht immer alle Kind-Elemente gewünschte Facetten. In diesem Fall müsste der Nutzer diese

aus den Originaldaten löschen. Oft will der Nutzer seine Daten nicht verändern, aber seine Suche beziehungsweise seinen Index, genauer definieren. Ein Möglichkeit der index Funktion neben dem Main-Element auch die gewünschten Facetten zu übergeben, könnte sich als sinnvoll erweisen. Enthält das Main-Element ein Kind-Element, welches die Beschreibung des Mediums beinhaltet, wäre dies meist keine sinnvolle Facette und könnte beim Indexieren ignoriert werden. Zusätzlich kann eine Gruppierung mehrerer Werte zur Facettengenerierung interessant sein. Wenn beispielsweise in einem Element date das Datum mit Tag, Monat und Jahr angeben wird, so würde der Index mit diesem Schema sehr viele Werte für die Facette date erhalten. Dabei ist es für viele Suchen nicht sinnvoll nach einzelnen Tagen zu filtern. Oft wäre in diesem Fall nur das Jahr gefragt. Denkbar in einem solchen Szenario wäre auch, dass die Jahre ab einem Grenzwert zu einem Facettenwert zusammengefasst werden.
Auf diese Weise könnte man das Modul komplett unabhängig vom Schema der XML-Datenbank machen. Einzig die Erstellung der Facetten würde weiterhin auf der XML-Struktur basieren.

Anhang

Algorithmus 7

Algorithm 6 LuceneIndex algorithm

```
1:  procedure LUCENEINDEX(context, data, dbname, mainEle)
2:      index ← FSDirectory.open(indexpath)
3:      taxoIndex ← FSDirectory.open(taxoindexpath)
4:      writer ← new IndexWriter(index, config)
5:      taxoWriter ← new DirectoryTaxonomyWriter(taxoIndex)
6:
7:      for all pre-Values in data do
8:          kind ← data.kind(pre)
9:          if kind == Data.ELEM then
10:             eleName ← data.name(pre, Data.ELEM)
11:             if elemName == mainEle then
12:                 tsize ← data.size(pre, Data.ELEM)
13:                 id ← data.id(pre)
14:                 doc ← new Document()
15:                 doc.add(new IntField(“pre“, id, Field.Store.YES))
16:                 pres ← new IntList()
17:                 names ← new TokenList()
18:                 texts ← new TokenList()
19:
20:                 for i ← pre; i < pre + tsize; i + + do
21:                     ckind ← data.kind(i)
22:                     par ← data.parent(i, ckind)
23:
24:                     if !pres.isEmpty() ∧ pres.peek() ≥ par then
25:                         if !texts.isEmpty() then
26:                             name ← Token.string(names.get(1))
```

```
27:                         doc.add(new FacetField(name, texts))
28:                     end if
29:                 end if
30:
31:                 while !pres.isEmpty() ∧ pres.peek() ≥ par do
32:                     pres.pop()
33:                     names.pop()
34:                     if !texts.isEmpty() then
35:                         texts.pop()
36:                     end if
37:                 end while
38:
39:                 if ckind == Data.ELEM then
40:                     pres.push(par)
41:                     name ← data.name(i, kind)
42:                     names.add(name)
43:                 end if
44:                 if ckind == Data.TEXT then
45:                     text ← data.text(i, true)
46:                     if text.length ≤ 100 then
47:                         texts.add(text)
48:                     end if
49:                     doc.add(new TextField("text", text, Field.Store.YES))
50:                 end if
51:             end for
52:
53:             if !texts.isEmpty() then
54:                 name ← names.get(0)
55:                 doc.add(new FacetField(name, texts))
56:             end if
57:
58:             writer.addDocument(fconfig.build(taxoWriter, doc))
59:
60:             if memory > totalmemory * 0.8 then
61:                 System.gc()
62:             end if
```

```
63:
64:                 pre ← pre + (tsize - 1)
65:             end if
66:         end if
67:     end for
68:     taxoWriter.close()
69:     writer.close()
70: end procedure
```

Literatur

[BL11] Steve Berry and Matt Lease. Design Considerations for Faceted Search: Literature Review and Case Study, 2011.

[Grü10] Christian Grün. Storing and Querying Large XML Instances, 2010.

[Grü15] Christian Grün. BaseX: The XML Database, 2015. http://basex.org.

[MHG10] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action, Second Edition*. Manning Publications, 2010.

[TW08] Bill Trippe and Dale Waldt. Using XML and Databases W3C Standards in Practice, 2008.

Alle in der Arbeit angegebenen Links wurden am 04.05.2015 aufgerufen.

Abbildungsverzeichnis

Tabellenverzeichnis