Verarbeitung komplexer XML-basierter Massendaten in BigData-Anwendungen

Max-Emanuel Keller, M.Sc.
Prof. Dr. Peter Mandl
Alexander Döschl, M.Sc.
Dr. Daniel Kailer

Hochschule für angewandte Wissenschaften München
Fakultät für Informatik und Mathematik
Competence Center Wirtschaftsinformatik
Lothstraße 34, 80334 München
E-Mail: {max-emanuel.keller, peter.mandl, alexander.doeschl, daniel.kailer}@hm.edu

Dr. Markus Grimm

IT4IPM – IT for Intellectual Property Management GmbH
Rosenheimer Straße 11, 81667 München
E-Mail: markus.grimm@it4ipm.de

ABSTRACT

XML ist ein semi-strukturiertes Datenbeschreibungsformat, das aufgrund weiter Verbreitung und steigender Datenmengen auch als Eingabeformat für eine BigData-Verarbeitung relevant ist. Der vorliegende Beitrag befasst sich daher mit der Nutzung komplexer XML-basierter Datenstrukturen als Eingabeformat für BigData-Anwendungen. Werden umfangreiche komplexe XML-Datenstrukturen mit verschiedenen XML-Typen in einer zu verarbeitenden XML-Datei beispielsweise mit Apache Hadoop verarbeitet, kann das Einlesen der Daten die Laufzeit einer Anwendung dominieren. Der vorliegende Ansatz befasst sich mit der Optimierung der Eingabephasen, indem Zwischenergebnisse der Verarbeitung im Arbeitsspeicher abgelegt werden. Der Aufwand für die Verarbeitung reduziert sich damit zum Teil erheblich. Anhand einer Fallstudie aus der Musikbranche, in der standardisierte XML-basierte Formate wie das DDEX-Format genutzt werden, wird experimentell gezeigt, dass die Verarbeitung mit dem vorliegenden Ansatz im Vergleich zur klassischen Abarbeitung von Dateiinhalten deutlich effizienter ist.

SCHLÜSSELWÖRTER

XML, Apache, Spark, Hadoop, XmlInputFormat, XML-Parser, DDEX, Musiknutzungsdaten

EINLEITUNG UND MOTIVATION

Für eine effiziente Verarbeitung großer Datenmengen werden häufig Cluster-Computing-Frameworks eingesetzt. Die Verarbeitung wird dabei in einzelnen Teilaufgaben (Tasks) auf eine Vielzahl von Knoten innerhalb eines Clusters verteilt und somit in angemessener Zeit abgeschlossen. Weit verbreitete Frameworks dieser Art sind Apache Hadoop, das unter anderem den Algorithmus MapReduce von Dean und Ghemawat nutzt (Dean und Ghemawat 2004) und Apache Spark (Apache Spark 2017). Beide Frameworks werden häufig mit der Verarbeitung von unstrukturierten Daten, wie z. B. Log-Files, in Verbindung gebracht, jedoch ist auch die Analyse von Dateien mit komplexerer Struktur möglich. Wir bezeichnen in diesem Beitrag Datenstrukturen als komplex, wenn diese Elemente verschiedener Typen enthalten, die zueinander in Beziehung stehen und gemeinsam verarbeitet werden. Im Unterschied zu einfachen, satzbasierten Daten, müssen bei der Verarbeitung komplexerer Datenstrukturen Beziehungen zwischen den Datensätzen erkannt werden, was nicht über eine rein sequenzielle Abarbeitung möglich ist.

Ziel der vorliegenden Forschungsarbeit ist es, die Verarbeitung komplexer Datentypen über Cluster-Computing-Frame-works zu vereinfachen und diese effizient zu ermöglichen. Die Nutzbarkeit des semi-strukturierten Formats XML (Extensible Markup Language) als Eingabeformat für BigData-Anwendungen steht dabei im Fokus der Betrachtung. XML gilt zwar in der Hadoop-Community als nicht sehr gut geeignet für die Massenverarbeitung, allerdings ist es in Unternehmen häufig vorzufinden. Für die Analyse einzelner Informationen aus XML-Dateien gibt es zwar Lösungsansätze aus anderen Projekten, wie etwa aus dem Apache Mahout Projekt (Mahout Projekt 2011) jedoch sind diese Ansätze nur bedingt geeignet, um komplexe XML-Dateien in Apache Hadoop oder Apache Spark zu verarbeiten. Dieser Beitrag betrachtet XML als Eingabeformat insbesondere für Anwendungen, die das Framework Apache Spark nutzen und stellt Programmiermuster vor, mit denen die Verarbeitungszeit effizient reduziert werden kann. Anhand einer konkreten Fallstudie werden die Programmiermuster erprobt und einer Bewertung unterzogen.

STAND DER TECHNIK

MapReduce

Die Grundidee von der Zerlegung einer Aufgabe in Teilaufgaben und der anschließenden Zusammen-

führung der Teilergebnisse ist bereits mehrere Jahrzehnte alt, jedoch erfolgte die Überführung in ein standardisiertes Paradigma erst relativ spät. Cluster-Computing verbreitete sich aufgrund des MapReduce-Algorithmus (Dean und Ghemawat 2004) insbesondere in den letzten Jahren, nicht zuletzt weil Internet-Unternehmen wie Google und Yahoo sehr große Datenmengen verarbeiten mussten und dafür Lösungen suchten. Der MapReduce-Ansatz reduziert Aufgaben mit Hilfe einer gemeinsamen Datenbasis in mehrere Teilaufgaben, die auf den Knoten eines Computer Clusters parallel berechnet werden können. Durch Definition der Aufgaben über abstrakte Operationen (high-level Operations) kontrolliert das System die Verteilung der Arbeit, inklusive der Lastenverteilung und der automatischen Wiederherstellung im Fehlerfall. Hierfür werden zwei Phasen eingesetzt, die als Map und Reduce bezeichnet werden und für die in einer konkreten Anwendung jeweils eine Methode oder Prozedur mit der auszuführenden Aufgabe definiert wird. Die Map-Phase läuft auf allen Knoten parallel ab und löst mit der Map-Methode auf jedem Knoten ein Teilproblem. Die Zusammenführung der Zwischenergebnisse zu einer Gesamtlösung geschieht in der Reduce-Phase, welche üblicherweise mit der Reduce-Methode auf einem einzelnen Knoten abläuft. Häufig lassen sich Probleme nicht nur mit Hilfe einer MapReduce-Abfolge lösen, sodass mehrere MapReduce-Abfolgen hintereinander ausgeführt werden müssen. In jedem Fall muss die Aufgabenstellung für eine Nutzung von MapReduce geeignet sein. Am besten eignet sich eine Problemstellung, die eine große Menge an unabhängigen Datensätzen verarbeiten muss, da diese Datensätze dann problemlos parallel verarbeitet werden können. Das trifft auf viele Aufgabenstellungen zu.

Apache Hadoop

Das weit verbreitete Framework Apache Hadoop ist die bekannteste Umsetzung des MapReduce-Ansatzes, neben weiteren Ansätzen wie Dryad (Isard et al. 2007) und MapReduce-Merge (Yang et al. 2007). Die genannten Implementierungen stellen die notwendigen Mittel zur Verwaltung der Ressourcen eines Clusters bereit. Hadoop verfügt zusätzlich über ein eigenes verteiltes Dateisystem namens HDFS (Hadoop Distributed Filesystem), sowie über Schnittstellen für beliebige Datenformate und Quellen. Mit zahlreichen Erweiterungen entwickelte sich in den letzten Jahren das Hadoop Ökosystem. Eine Abstraktion zur verteilten Nutzung des Arbeitsspeichers ist jedoch in Apache Hadoop nicht vorhanden, weshalb Aufgaben, die Zwischenergebnisse produzieren, nicht ohne Weiteres umsetzbar sind (Zaharia et al. 2012). So müssen beispielsweise Zwischenergebnisse zwischen den Einzelschritten auf einem verteilten Dateisystem abgelegt und immer wieder eingelesen werden (Abb. 1). Die damit einhergehenden Operationen für den Zugriff auf das Dateisystem (I/O-Operationen), die notwendige Serialisierung der Objekte bei der Zwischenspeicherung sowie die Replikation der Daten können die Ausführungszeit einer Hadoop-Anwendung dominieren. Hadoop und andere klassische MapReduce-Systeme eignen sich damit nur eingeschränkt für Aufgabenstellungen, die Zwischenergebnisse produzieren und über mehrere Schritte verarbeiten. Diese Art von Aufgabenstellungen lassen sich nach (Zaharia et al. 2012) in zwei Kategorien einordnen:

- *Iterative Aufgaben*: Algorithmen wie maschinelle Lernverfahren, die Funktionen iterativ auf ein Zwischenergebnis anwenden, um einen Parameter zu optimieren. Das Zwischenergebnis wird mit jeder Iteration vom externen Speicher eingelesen.
- *Interaktive Auswertungen*: Hierzu gehören Systeme wie Pig (Olston et al. 2008) und Hive (Thusoo et al. 2010), mit denen interaktive Abfragen in SQL-ähnlichen Dialekten formuliert und über Apache Hadoop ausgeführt werden. Das Lesen der Daten vom externen Speicher führt bei derartigen Auswertungen zu Verzögerungen.

Zur Lösung des oben genannten Problems wurden Frameworks entwickelt, die Zwischenergebnisse einer Berechnung im Arbeitsspeicher vorhalten, aber auf bestimmte Muster beschränkt sind. Dazu zählen Pregel (Malewicz et al. 2010) für die iterative Berechnung von Graphen und HaLoop (Bu et al. 2010) für iteratives MapReduce.

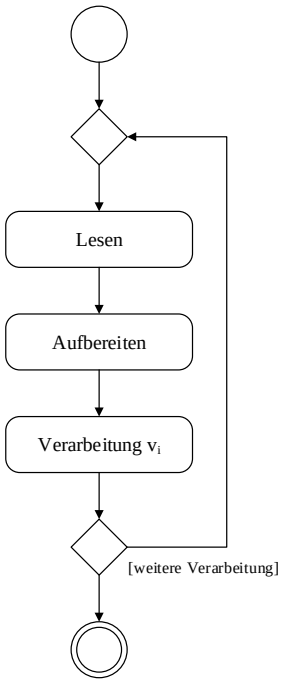

Abbildung 1: MapReduce-Verarbeitung mit wiederholtem Lesen von Zwischenergebnissen

Apache Spark

Apache Spark basiert auf der Abstraktion RDD (Resilient Distributed Dataset). RDDs sind rein lesbare Sammlungen von Objekten, die als Partitionen verteilt über die Knoten des Clusters in den Arbeitsspeichern der beteiligten Knoten abgelegt werden. Ein RDD kann über Operationen aus Daten von einem externen

Speicher oder einem bestehenden RDD erzeugt werden. Für Zwischenergebnisse werden neue RDDs erzeugt, die ebenfalls im Arbeitsspeicher gehalten werden. Das aufwändige Zwischenspeichern und Einlesen über einen externen Speicher kann damit entfallen (Abb. 2) . Davon profitieren insbesondere iterative und interaktive Auswertungen, aber auch andere Aufgabenstellungen, die mehrere Zwischenergebnisse produzieren. Ermöglicht wird dies durch die RDD-Schnittstelle. Diese stellt grobe Transformationen (coarse-grained transformations) wie z.B. *map*, *join* und *filter*, zur Verfügung, die jeweils eine Operation auf jedes einzelne Element des RDD anwenden. Anstelle der Änderungen an den Daten, werden dabei nur die eingesetzten Operationen in Form der Abstammung (lineage) aufgezeichnet.

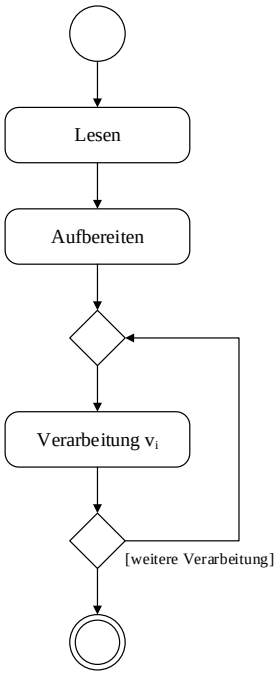

Abbildung 2: Verarbeitung mit einmaligem Einlesen

Gehen aufgrund eines Fehlers Teile eines RDD in Form einzelner Partitionen verloren, können diese auf der Basis der Abstammungsinformationen neu berechnet werden, um Fehlertoleranz und Skalierbarkeit wie in MapReduce zu erreichen. So treten zusätzliche Kosten nur im Fehlerfall auf. Bestehende Ansätze zur Verwaltung von Daten im Arbeitsspeicher wie DSM (distributed shared memory) erreichen die Fehlertoleranz dagegen durch die Replikation der Daten, durch Checkpoints oder durch die Aufzeichnung der Änderungshistorie (Nitzberg und Lo 1991). Durch Aufzeichnung der Abstammung können RDDs zudem verzögert ausgewertet werden. Die Berechnung findet mit der ersten tatsächlichen Nutzung statt. RDDs sind damit für ein breites Spektrum von Einsatzgebieten geeignet.

Die Operationen der API lassen sich im Wesentlichen in zwei Gruppen gliedern. Die Gruppe der Transformationen (Transformations) dient der Überführung von RDDs in neue RDDs. Jedes einzelne Element eines RDD wird über eine Funktion bearbeitet und das Ergebnis wird in ein neues RDD überführt. Die Gruppe der *Aktionen (Actions)* dient der Ausführung von Berechnungen, die direkt ein Ergebnis zurückliefern oder es in ein verteiltes Dateisystem schreiben. *Transformationen (Transformations)* hingegen werden verzögert (lazy) ausgeführt, also erst dann, wenn die Ergebnisse von Aktionen angefragt werden. Drei häufig verwendete Transformationen sollen kurz erläutert werden:

1. Die Operation *map* transformiert jedes Element des Typs T in RDD[T] durch Anwenden einer angegebenen Funktion f zu einem Element des Typs U. Aus RDD[T] entsteht RDD[U]:
$$map\ (f: T \Rightarrow U) : RDD[T] \Rightarrow RDD[U]$$

2. Die Operation *filter* liefert für jedes Element aus RDD[T] unter Anwendung einer angegebenen Funktion f den booleschen Wert *true* oder *false* zurück. Das Ergebnis ist ein RDD[T], das sämtliche Elemente der Quelle enthält, für die f *true* ist:
$$filter\ (f: T \Rightarrow Bool) : RDD[T] \Rightarrow RDD[T]$$

3. Durch Aufruf der Operation *join* auf RDD[(K,V)] und RDD[(K,W)] entsteht RDD[(K,(V,W))]. Das Ergebnis-RDD enthält Paare von Elementen (V,W) mit identischem Schlüssel K:
$$join() : (RDD[K,V], RDD[K,W]) \Rightarrow RDD[(K,(V,W))]$$

Die verteilte Spark-Architektur ist in Abb. 3 skizziert. Für jede Spark-Anwendung wird eine eigene Spark-Instanz zum Ablauf gebracht. Auf einem Masterknoten wird ein sog. *Cluster-Manager* ausgeführt, der die Ressourcenverwaltung übernimmt. Auf einem Clientknoten wird ein *Treiberprogramm (Spark-Anwendung)* gestartet, das über einen *Spark-Kontext* die Bearbeitung initiiert und die Ressourcen-Anforderungen an den Cluster-Manager übergibt. Die Steuerung wird über den Cluster-Manager ausgeführt, der die Aufgaben an einzelne *Arbeiterknoten (worker nodes)* delegiert. Die konfigurierten Arbeiterknoten melden sich beim Cluster-Manager an und geben ihre Ressourcen durch, damit der Cluster-Manager weiß, wie er seine Aufgaben verteilen kann. In den Arbeiterknoten werden Prozesse gestartet, die in sog. *Executors* die vom Treiberprogramm angeforderten Aktionen und Transformationen ausführen.

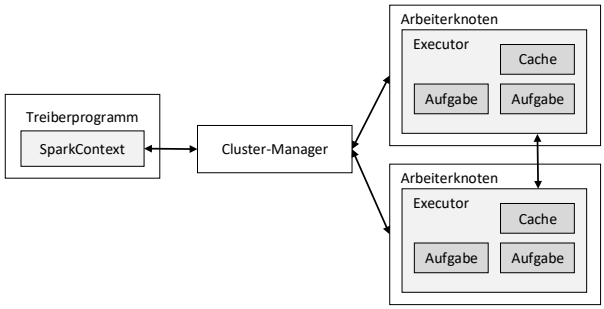

Abbildung 3: Zusammenwirken von Komponenten der Spark-Architektur. In Anlehnung an: Apache Spark (2015)

KONZEPTIONELLE ÜBERLEGUNGEN

Das vorliegende Konzept sieht vor, das Einlesen von komplexen XML-Dateien durch eine Erweiterung des vorhandenen Mahout-Ansatzes zu verbessern. Zudem soll nach einer Speicherung von Zwischenergebnissen beim Wiedereinlesen auf ein erneutes Parsing der XML-Dateien verzichtet werden. Die Nutzung von Apache Spark ermöglicht zudem, die Ein-/Ausgabe deutlich zu reduzieren, indem sämtliche aus den XML-Dateien benötigten Informationen gemeinsam gelesen und zwischengespeichert werden. Der Ansatz soll im Folgenden skizziert werden.

Ausgangssituation

Um Dateien in unterschiedlichen Formaten aus verschiedenen Quellen verarbeiten zu können, stellen Apache Hadoop und Apache Spark vordefinierte Schnittstellen für die Ein- und Ausgabe (Input-Format und Output-Format) bereit. Neben Standardformaten werden in anderen Projekten spezielle Implementierungen der Schnittstelle *InputFormat* bereitgestellt. Darunter ist mit dem Format *XmlInputFormat* des Mahout-Projekts auch ein Input-Format für XML-Dateien (Mahout Projekt 2011). XmlInputFormat liest Elemente einer XML-Datei als Datensätze (Records) aus, die durch Apache Hadoop verarbeitet werden können. Dazu werden Beginn- und Endauszeichner, durch welche die Elemente begrenzt werden, per Konfiguration festgelegt. Die Dateien werden sequentiell gelesen, um die Beginn- und Endauszeichner zu finden und dazwischenliegende Inhalte einzulesen. Das Input-Format kann damit für Daten beliebigen Inhaltes verwendet werden, jedoch ist pro Durchlauf einer Datei nur ein einzelner XML-Typ einlesbar. Um verschiedene Elementtypen gemeinsam einzulesen, ist das Input-Format in mehreren Instanzen verwendbar, von denen jede einen der Elementtypen einliest. Anschließend werden die Instanzen des Input-Formats über sog. *MultipleInputs* verbunden, um die Ergebnisse zusammenzufassen (White 2012, S. 250f). Der Aufwand für das Einlesen erhöht sich damit bei einer Verarbeitung von n Elementtypen linear auf n, weil für jeden Typ die gleichen Dateien einzulesen sind.

Konzeptidee

Um das wiederholte Einlesen der gleichen Dateien zu verhindern, wurde das Format XmlInputFormat des Mahout-Projekts angepasst. Das zu diesem Zweck erweiterte Format *XmlInputFormatMulti* liest sämtliche Typen, die zur Bearbeitung einer Aufgabe durch Hadoop oder Spark benötigt werden, in einem Lesevorgang aus den Dateien ein. Die Kosten für das Lesen reduzieren sich somit bei n Elementtypen und Einzelkosten je Lesevorgang von k von nk um $(n-1)k$ auf k.
Wird das optimierte Input-Format mit Apache Hadoop verwendet, reduziert sich der Aufwand für die I/O-Operationen bereits erheblich. Um die Leistung weiter zu steigern, kann der I/O-Aufwand durch den Einsatz von Apache Spark weiter reduziert werden, da hier eine Zwischenspeicherung der gelesenen Daten im Arbeitsspeicher der verteilten Cluster-Knoten genutzt werden kann. Im optimalen Fall müssen komplexe XML-Dateien nur ein einziges Mal eingelesen werden und verbleiben bis zum Ende der Verarbeitung in den verteilten Arbeitsspeichern. Dies gelingt, indem sämtliche zu bearbeitenden Daten gemeinsam gelesen, geparst und im Arbeitsspeicher abgelegt werden. Vor jeder Teilaufgabe sind die zwischengespeicherten Datentypen lediglich zu filtern, um nur die für den Arbeitsschritt erforderlichen Bestandteile zu lesen. Für das gleichzeitige Einlesen mehrerer Elementtypen wurde das vorhandene Input-Format in unserer Implementierung angepasst, um den Vorgang zu optimieren. Unsere Logik zur Verarbeitung von XML-Dateien besteht aus drei Phasen:

1. Einlesen und Parsen der XML-Elemente
2. Überführung der XML-Elemente in Java-Objekte
3. Auswertung der Daten

Zur Verarbeitung in Apache Spark werden die XML-Elemente der benötigten Typen in einem RDD gesammelt (Operation *newAPIHadoopFile()*). Dabei wird ein Input-Format verwendet, um Schlüssel-Wert-Paare aus beliebigen Quellen zu lesen. Hier kommt das optimierte Input-Format für XML (XmlInputFormatMulti) zum Einsatz. Für jedes gelesene XML-Element wird ein Java-Objekt erstellt, dessen Attribute aus dem XML-Element geparst werden. Dazu wird das RDD zu einem weiteren RDD transformiert. Die Elemente der gelesenen Typen können nun gemeinsam oder nacheinander verarbeitet werden. Damit sämtliche relevanten Elementtypen in einem Schritt gelesen werden können, müssen alle Elementtypen in den gleichen einzulesenden Dateien vorliegen und der Arbeitsspeicher muss ausreichend groß dimensioniert sein, um die gelesenen Elemente vorzuhalten.
Reicht der vorhandene Arbeitsspeicher nicht aus, um die Daten vorzuhalten, kann trotzdem eine weitere Optimierung vorgenommen werden. Ein Teil der Daten muss erneut eingelesen und geparst werden. Insbesondere das Parsen der Inhalte ist rechenintensiv, weshalb ein nicht unbeträchtlicher zusätzlicher Aufwand entsteht. Die Leistung des Prozessors bildet dabei in der Regel den Flaschenhals. Damit das Parsen in diesen Fällen entfallen kann, speichern wir in unserem Verfahren die Inhalte in einem vorgelagerten Schritt bereits vorab geparst als serialisierte Java-Objekte auf das verteilte Dateisystem HDFS. Anstelle der XML-Dateien werden dann in Folgeschritten die serialisierten Java-Objekte gelesen und nur noch deserialisiert. Da das aufwändige Parsen von XML-Daten beim erneuten Einlesen entfällt, sollte dies zu einer deutlichen Leistungssteigerung führen.

TESTANWENDUNG

Fallstudie

Um die Funktionsweise des vorgestellten Optimierungsansatzes nachzuweisen, wurde eine Implementierung anhand einer konkreten Fallstudie aus dem Umfeld der Musikbranche durchgeführt. In der Fallstudie wurden Mediennutzungsmeldungen verarbeitet, die von Lizenznehmern (Apple Music, Spotify usw.) periodisch an eine Verwertungsgesellschaft wie die GEMA (Gesellschaft für musikalische Aufführungs- und mechanische Vervielfältigungsrechte) gesendet werden, um eine Abrechnung der Nutzungsentgelte mit den Künstlern (Musiker, Texter) durchzuführen (Mandl et al. 2016). Die Mediennutzungsmeldungen werden oft in Form von XML-Dateien ausgetauscht. Die Dateien basieren dabei auf dem Verfahren DSR (Digital Sales Reporting) (Digital Data Exchange, LLC 2014) des Standards DDEX (Digital Data Exchange). Sie enthalten neben allgemeinen Angaben (Sender, Empfänger, Zeitpunkt der Erstellung und Meldezeitraum) die eigentlichen Nutzungsdaten. Hierzu gehören Angaben zu den *Musikaufnahmen (SoundRecording)*, zur *Veröffentlichung der Werke (Release)* und zu den *Transaktionen bzw. Umsätzen (ReleaseTransactions)*. Die Meldedaten beinhalten wiederum Unterelemente zu Identifikatoren, Künstler- und Titelinformationen, Absatzkanälen und Lizenzierungszahlen. DDEX-Dateien stellen durch die große Zahl an Elementtypen, die zueinander in Beziehung stehen, ein Beispiel für komplexe XML-Dateien dar, wie im Datenmodellausschnitt (Abb. 4) zu sehen ist. Diese Daten können also nicht rein sequenziell verarbeitet werden.

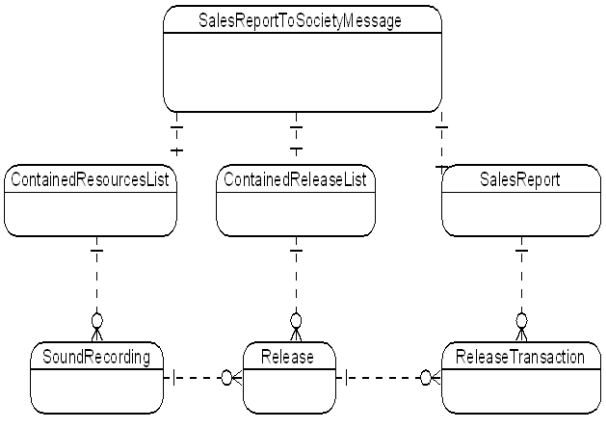

Abbildung 4: Ausschnitt des DDEX-Datenmodells

Für die Fallstudie wurde eine Spark-Anwendung entwickelt, die vor allem Unregelmäßigkeiten in großen DDEX-Dateien auffinden sollte. Konkret wurde nach Duplikaten, unvollständigen oder falschen Daten und auch Ausreißern (Musikwerke mit besonders vielen oder wenigen Nutzungen) gesucht. In diesem Beitrag wird nur die Erkennung von unvollständigen Daten in den Eingabedateien betrachtet. Jede DDEX-Eingabedatei enthält dreizehn verschiedene für die Auswertung relevante Datentypen. Unvollständige Dateien sind zum Beispiel solche, in denen nicht mindestens ein Element jedes Typs enthalten ist. Zur Auswertung standen drei Tebibyte (TiB) an DDEX-Dateien eines ausgewählten Jahres zur Verfügung.

Implementierungsaspekte

Um die Fragestellungen der Fallstudie zu beantworten, wurden zunächst die Informationen aus den XML-Dateien benötigt. Hierzu fand die bereits beschriebene Transformation newAPIHadoopFile sowie das dabei genutzte Input-Format XmlInputFormatMulti Verwendung, um Schlüssel-Wert-Paare einzulesen und in Elemente eines RDD zu transformieren. Die Beginn- und Endauszeichner der einzulesenden Typen wurden als Eigenschaften an die Spark-Konfiguration übergeben. Da es für die gleichzeitige Validierung mehrerer Dateien unerlässlich ist, jeden extrahierten Datensatz seiner Ursprungsdatei zuordnen zu können, galt es den jeweiligen Dateinamen in die Schlüssel-Wert-Paare mit aufzunehmen. Im verwendeten XmlInputFormatMulti wurde der Dateiname dazu als Schlüssel definiert.
Nach dem Einlesen der XML-Daten lag ein RDD[(Text, Text)] vor, dessen Paare aus Schlüssel (Dateiname) und Wert (XML- bzw. DDEX-Element) bestanden und in ihrer Gesamtheit alle Informationen der Input-Dateien hielten. Das RDD wurde zu einem weiteren RDD transformatiert, das durch die selbst definierte abstrakte Klasse *DdexContent* typisiert war (RDD[(String, DdexContent)]). DdexContent definiert eine Reihe von Operationen zum Objektvergleich sowie zur Prüfung von Integrität und Subtyp. Bei der Transformation wurde aus jedem XML-Element ein Objekt des passenden konkreten Subtyps von DdexContent instanziiert. Die Attribute wurden durch Parsen des XML-Elements gewonnen. Um die Typen zu erhalten, die für die einzelnen Auswertungen benötigt werden, konnte die Transformation *filter* eingesetzt werden, die für jedes Element aufgerufen wurde.
Die Fallstudie umfasste unter anderem die Überprüfung, ob jeder relevante Typ mindestens einmal vorhanden war. Um diese Frage zu beantworten, wurden, im Gegensatz zu den übrigen Fragen, alle Typen benötigt. Deshalb wurde das RDD[(String, DdexContent)] ohne Filtern verwendet. Auf dem RDD wird nun die Transformation *aggregateByKey* ausgeführt, die sämtliche Werte (DdexContent) mit identischem Schlüssel (Dateinamen) zu einem neuen Wert zusammenfasst. Das Ergebnis ist ein RDD[(String, Integer)], das ein Element für jede Datei enthält. Innerhalb des Werts (Integer) werden die dreizehn Typen einer Datei durch die dreizehn niederwertigsten Bits repräsentiert. Um die unvollständigen Dateien zu bestimmen, wird die Transformation *filter* eingesetzt. Diese empfängt eine Methode, die *true* zurückliefert, wenn mindestens eines der dreizehn niederwertigsten Bits nicht gesetzt ist und die Datei somit unvollständig ist. Das dabei entstehende RDD[(String, Integer)] enthält ein Element für jede unvollständige Datei, mit einem Wert, aus dem hervorgeht, welche Typen fehlen.

Da jede der in Kapitel 3.2 beschriebenen Phasen Einfluss auf die Geschwindigkeit der Verarbeitung nimmt, wurden für die einzelnen Verarbeitungsschritte Alternativen untersucht. So wurde zunächst in einem Vergleich der DOM-Parser der Java SE 1.7 Standardbibliothek dem VTD-XML-Parser 2.11 (Vtd Projekt 2012) gegenübergestellt. Es wurde festgestellt, dass sich mit dem Parser VTD-XML die Laufzeit gegenüber DOM halbiert.
Wie bereits erwähnt, erweist sich das Parsen der Inhalte als sehr rechenintensiv, wobei die Leistung des Prozessors in der Regel den Flaschenhals bildet. Es lag folglich der Versuch nahe, diesen Verarbeitungsschritt weitestgehend zu eliminieren und somit die Leistung zu steigern. Hierfür müssen die Inhalte in einem vorgelagerten Schritt einmalig geparst und als serialisierte Objekte auf das Dateisystem HDFS geschrieben werden. Anstelle der XML-Dateien werden dann lediglich die serialisierten Objekte gelesen und deserialisiert – ein erneutes Parsen kann entfallen. Zur Bewertung dieses Vorgehens wurde ein Vergleich durchgeführt. Dazu wurden die Testdaten eines Monats verwendet, die 70 Dateien mit insgesamt 108,1 GiB umfassen. Beim normalen Vorgehen konnten die Daten in 14,7 Minuten gelesen und geparst werden. Die vorgestellte Alternative benötigt zwei Schritte:

- Schritt 1: Einmalige Vorbereitung der Daten, wobei diese gelesen, geparst und serialisiert ins HDFS-Dateisystem geschrieben werden (gemessene Laufzeit: 21,5 Minuten)
- Schritt 2: Lesen der vorbereiteten Daten. Die Messungen ergaben eine Lesezeit von einer Minute, bei der auch gleichzeitig die Deserialisierung erfolgte.

Das vorgelagerte Parsen ist also dem klassischen Parsen beim Einlesen vorzuziehen, wenn die Daten in mindestens zwei Ausführungen verarbeitet werden.

LEISTUNGSMESSUNG

Um die entwickelte Lösung zu erproben, wurde ein Spark-Cluster zur Durchführung von Leistungsmessungen auf Basis der realen Daten der Fallstudie aufgebaut. Ziel der Leistungsmessungen war es, mehrere prototypische Implementierungen zu vergleichen, mit denen die vorgestellte Fragestellung beantwortet werden sollte.

Versuchsaufbau

Für die Messungen wurden zwölf virtuelle Maschinen mit Cloudera Express CDH (Cloudera Distribution Including Hadoop) 5.3, Spark 1.2 sowie je drei Prozessoren (vCPU) und zwischen 15 und 20 GiB RAM verwendet. Als externer Speicher stand insgesamt 1 TiB in einen Storage Area Network (SAN) zur Verfügung. Als Betriebssystem wurde auf allen VMs das Linux-Derivat CentOS 6.5 x86_64 installiert. Die virtuellen Maschinen (VM) waren dabei über vSphere 5.0 Enterprise Edition des Unternehmens VMware gleichmäßig auf drei Systeme mit je zwei Intel Xeon X5650 (sechs Kerne à 2,66 GHz) und 96 GiB an RAM verteilt. Die Verbindung unter den Systemen und zum SAN-Speicher, angebunden über iSCSI, wurde über jeweils zwei Ethernet-Schnittstellen mit einer Datenübertragungsrate von 10 Gbit/s hergestellt (Abb. 5). Auf dem SAN-Speicher waren die virtuellen Festplatten der VM abgelegt. Die Spark-Komponenten wurden auf die VMs des Clusters verteilt. Die drei Systeme des Clusters standen dabei exklusiv für die Ausführung der VMs zu Verfügung, wodurch sämtliche Messungen zu identischen Ausgangsbedingungen durchgeführt werden konnten.

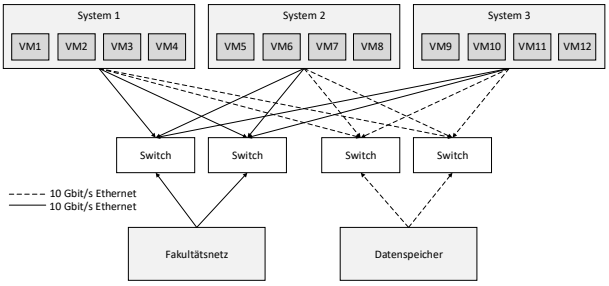

Abbildung 5: Messumgebung

Um die Leistungsfähigkeit und Skalierbarkeit der Implementierung zu ermitteln, wurden Messungen mit drei, sechs, bzw. neun Knoten durchgeführt. Damit keine Daten vom externen Speicher nachgeladen werden mussten, war es notwendig, dass der Arbeitsspeicher der Knoten zur Zwischenspeicherung der Daten ausreicht. Der YARN NodeManager der Testumgebung stellte sieben GiB an RAM je Knoten bereit, von denen sechs GiB für Apache Spark genutzt wurden. Davon wurden jeweils 60 % ohne Anpassungen für die Zwischenspeicherung verwendet. Der verfügbare Arbeitsspeicher war damit begrenzt. Er reichte nicht aus, um die gesamten Testdaten aus dem Jahr 2013 (insgesamt drei TiB) vorzuhalten. Daher wurden erste Messungen mit den Daten des Monats Januar (108 GiB) durchgeführt. Der Speicherbedarf wurde zusätzlich optimiert, indem die Inhalte der RDDs als serialisierte Java-Objekte im Arbeitsspeicher zwischengespeichert wurden. Die Daten wurden in 910 Partitionen à 128 MiB aufgeteilt. Von diesen konnten 743 (82 %) mit neun Knoten, 534 (59 %) mit sechs Knoten und 247 (27 %) mit drei Knoten zwischengespeichert werden. Darüber hinausgehende Daten wurden vom Festspeicher eingelesen. Die Ergebnisse zeigen deutlich, wie sich die Anzahl der Knoten auf die Laufzeit auswirkt (Abb. 6).

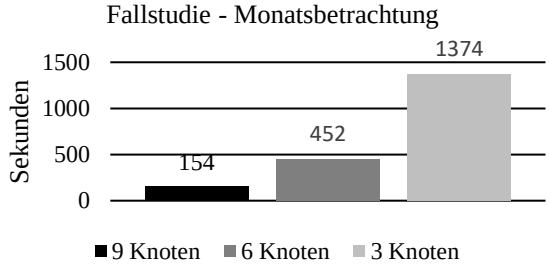

Abbildung 6: Messergebnisse der Monatsbetrachtung

In einem zweiten Schritt sollten die Spotify-Daten des ganzen Jahres verarbeitet werden. Das Verhältnis des Arbeitsspeichers zum Umfang der Daten, das viel geringer war als bei der Auswertung eines einzelnen Monats, machte sich dabei bemerkbar. Mit neun Knoten konnten 82 % der Daten des Monats Januar im Arbeitsspeicher zwischengespeichert werden. Bei der Auswertung des ganzen Jahres reduzierte sich dieses Verhältnis weiter. Ein Großteil der Daten musste erneut vom externen Speicher eingelesen und verarbeitet werden. Durch das Parsen entstanden hohe Kosten. Um die Verarbeitung von zwölf Monaten zu beschleunigen, wurden die Daten in einem vorhergehenden Schritt geparst und als serialisierte Objekte im HDFS-Dateisystem abgelegt. Der Vorgang dauerte 320 Minuten. Bei der Verarbeitung wurden die serialisierten Objekte vom HDFS-Dateisystem gelesen und deserialisiert. Die Kosten der Deserialisierung der Objekte fielen geringer aus als die Kosten für das Parsen. Weil die Daten vor jeder Ausführung nahezu vollständig neu geladen wurden, skalierte die Laufzeit zwischen drei und neun Knoten nahezu linear (Abb. 7).

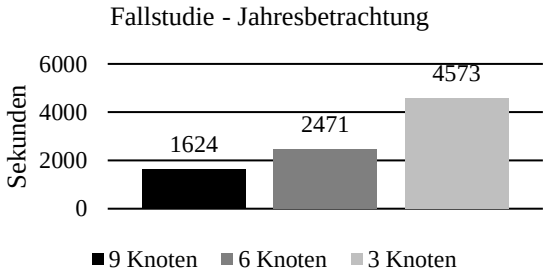

Abbildung 7: Messergebnisse der Jahresbetrachtung

Durch einen Vergleich der InputFormat-Klassen sollte aufgezeigt werden, wie sich die Unterschiede bezüglich des Verarbeitungsaufwands bei Nutzung der verschiedenen InputFormat-Klassen auf die praktisch zu erzielenden Verarbeitungsgeschwindigkeiten auswirken. Dazu wurde ein Szenario erarbeitet, in dem Elemente fünf verschiedener Elementtypen eingelesen wurden. Das Einlesen war mit XmlInputFormatMulti in einem kombinierten Vorgang möglich, wohingegen mit XmlInputFormat fünf Durchläufe notwendig waren und deren Ergebnisse zusammengefasst werden mussten, um das gleiche Endergebnis zu erzielen. Eingelesen wurden Daten mit einer Gesamtgröße von 108 GiB. Zur Verifikation der Ergebnisse wurde die Anzahl der Elemente verglichen, die in beiden Varianten importiert wurden. Der Aufwand für das Einlesen ist vom Aufwand für die Verarbeitung nicht isoliert. Jedoch ist der Aufwand für das Zählen vergleichsweise gering. Für dieses Beispiel wurde festgestellt, dass sich die Laufzeit mit XmlInputFormatMulti auf ungefähr ein Fünftel der Laufzeit von XmlInputFormat reduziert (Abb. 8).

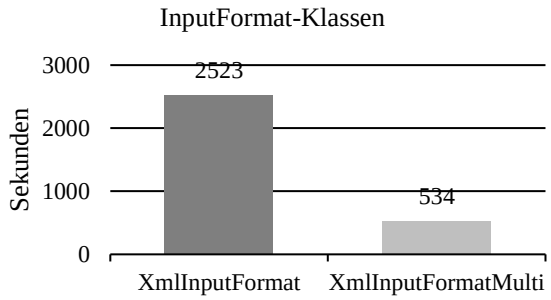

Abbildung 8: Messergebnisse zum Vergleich der InputFormat-Klassen

ZUSAMMENFASSUNG

Ziel der Forschungsarbeit war es, die Verarbeitung komplexer Datenstrukturen in BigData-Anwendungen durch eine effiziente Speicherung von Zwischenergebnissen zu optimieren. Da die Verarbeitung von XML-Dateien nach wie vor auch im BigData-Umfeld weit verbreitet ist, wurden im Besonderen XML-basierte Datenstrukturen betrachtet. In diesem Beitrag wurde eine Lösung vorgestellt, die es erlaubt, komplexe XML-Dateien mit Apache Hadoop und Spark zu verarbeiten. Hierzu wurden bestehende Techniken aus dem Hadoop-Ökosystem verwendet. Das Input-Format des Mahout-Projektes zum Einlesen von Informationen aus XML-Dateien wurde dabei um Möglichkeiten für das parallele Einlesen verschiedener XML-Datentypen erweitert. Der entwickelte Ansatz lässt sich problemlos auf andere komplexe Datenbeschreibungssprachen abbilden, lediglich der Parsing-Teilschritt der Implementierung muss dafür angepasst werden.

Apache Spark ist für die Verarbeitung komplexer XML-Dateien im Rahmen der Fallstudie aufgrund der guten Skalierbarkeit geeignet. Dabei wird jedoch die Bedeutung des verfügbaren Arbeitsspeichers ersichtlich. Die Geschwindigkeit hängt maßgeblich von der zwischengespeicherten Datenmenge ab. Neben dem Arbeitsspeicher spielen weitere Faktoren eine wichtige Rolle. In jedem Fall ist auf die Wahl geeigneter Datentypen zu achten. Auch das Parsing der Eingabedaten hat einen erheblichen Einfluss auf die Laufzeit der Verarbeitung. Ebenso ist eine entsprechende Vorbereitung der Daten bei mehrfacher Auswertung, die ein Parsing überflüssig macht, von Vorteil. Es kann festgehalten werden dass, der vorgestellte Ansatz zur Verarbeitung komplexer XML-Dateien problemlos auf andere Anwendungsfälle, in denen komplexe Datenstrukturen bearbeiten werden, übertragbar ist. Auch eine Verarbeitung von XML-Dateien in Pig (Olston et al. 2008; Pig Projekt 2015a; Pig Projekt 2015b) und Hive (Hive Projekt 2015a; Thusoo et al. 2010; Hive Projekt 2015b; Vasilenko und Kurapati 2014; Vasilenko 2015) ist möglich.

DANKSAGUNG

Die Forschungsarbeit wurde im Rahmen des durch die GEMA und durch das CCWI (Competence Center Wirtschaftsinformatik) der Hochschule München initiierten Forschungsprojekts MPI (= Massively parallel Processing of Internet events) durchgeführt. Das Projekt beschäftigt sich u.a. mit der massiv parallelen Verarbeitung von Musiknutzungsdaten. Anonymisierte Testdaten für unsere Fallstudie wurden dankenswerter Weise durch die GEMA bereitgestellt.

LITERATUR

Apache Spark (2015) Cluster Mode Overview. https://spark.apache.org/docs/1.2.0/cluster-overview.html. Zugegriffen: 29.09.2017.

Apache Spark (2017) https://spark.apache.org. Zugegriffen: 01.07.2017.

Armbrust M, Xin R, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, Zaharia M (2015) Spark SQL: Relational Data Processing in Spark.

Bu Y, Howe B, Balazinska M, Ernst MD (2010) HaLoop: Efficient iterative data processing on large clusters. Proceedings of the VLDB Endowment 3:285–296. doi:10.14778/1920841.1920881.

Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6. USENIX Association, San Francisco, CA.

Digital Data Exchange, LLC (2014) Digital Sales Reporting Message Suite Standard. https://kb.ddex.net/download/attachments/3901276/MM-0976%20-%20DSR%204.3.pdf. Zugegriffen: 01.07.2017.

Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I (2014) GraphX: graph processing in a distributed dataflow framework. In: Flinn J, Levy H (Hrsg) Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. USENIX Association, Berkeley California.

Hive Projekt (2015a) Apache Hive TM. https://hive.apache.org/. Zugegriffen: 01.07.2017.

Hive Projekt (2015b) LanguageManual XPathUDF. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+XPathUDF. Zugegriffen: 01.07.2017.

Isard M, Budiu M, Yu Y, Birrell A, Fetterly D (2007) Dryad: distributed data-parallel programs from sequential building blocks the 2nd ACM SIGOPS/EuroSys European Conference.

Mahout Projekt (2011) InputFormat für XML. https://github.com/apache/mahout/blob/ad84344e4055b1e6adff5779339a33fa29e1265d/examples/src/main/java/org/apache/mahout/classifier/bayes/XmlInputFormat.java. Zugegriffen: 01.07.2017.

Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: ACM (Hrsg) Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, Indianapolis, Indiana, USA, S 135–146.

Mandl P, Bauer N., Döschl A. (2016) Die Verwertung von Online-Musiknutzungen – Herausforderungen für die IT. HMD Praxis der Wirtschaftsinformatik, 53(1), pp.126–138.

Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai DB, Amde M, Owen S, Xin D, Xin R, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A (2015) MLlib: Machine Learning in Apache Spark.

Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig Latin: A Not-So-Foreign Language for Data Processing. In: Lakshmanan LVS, Ng RT, Shasha D (Hrsg) the 2008 ACM SIGMOD international conference, S 1099.

Pig Projekt (2015a) Apache Pig. https://pig.apache.org/. Zugegriffen: 01.07.2017.

Pig Projekt (2015b) XMLLoader (Pig 0.15.0 API). http://pig.apache.org/docs/r0.15.0/api/org/apache/pig/piggybank/storage/XMLLoader.html. Zugegriffen: 01.07.2017.

Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Zhang N, Antony S, Liu H, Murthy R (2010) Hive - a petabyte scale data warehouse using Hadoop 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), S 996–1005.

Vasilenko D (2015) Apache Hive XML SerDe. https://github.com/dvasilen/Hive-XML-SerDe/wiki/XML-data-sources. Zugegriffen: 01.07.2017.

Vasilenko D, Kurapati M (2014) Efficient Processing of XML Documents in Hadoop Map Reduce. International Journal on Computer Science and Engineering 6:329–333.

Vtd Projekt (2012) Project Homepage of VTD. http://vtd-xml.sourceforge.net. Zugegriffen: 01.07.2017.

White T (2012) Hadoop; The definitive guide. O'Reilly, Sebastopol, CA.

Yang H, Dasdan A, Hsiao R, Parker DS (2007) Map-reduce-merge: simplified relational data processing on large clusters. ACM.

Zaharia M, Chowdhury M, Das T, Dave A, Ma J, Mccauley M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation.

Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I (2013) Discretized streams: fault-tolerant streaming computation at scale. In: Kaminsky M, Dahlin M (Hrsg) SOSP '13 Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, S 423–438.