Spark-Analysen in einem Hadoop- und Storm basierten Betrugserkennungssystem

Rick P.C. Moritz

8. Januar 2016

Die elektronischen Kundenkanäle von Banken sind beliebtes Ziel von Betrügern. Diese erspähen oder stehlen Legitimationsschlüssel und initiieren Überweisungen von den Konten ihrer Opfer aus. Um diesen finanziellen Schaden abzuwenden – und regulatorischen Anforderungen zu entsprechen – ist ein intelligentes Betrugserkennungssystem unabdingbar. Wir beschreiben, wie ein reines Transaktionsbewertungssystem um ein integriertes Analyseumfeld erweitert werden kann, um Betrugsfälle besser modellieren zu können, und die Transaktionsbewertung zu optimieren. Wir verwenden skalierbare, Open-Source Big-Data-Software aus dem Hadoop-Umfeld. Anhand einer Fallstudie zeigen wir, wie man Storm und HBase für die Bewertung, Spark und Hive für die Analyse einsetzen kann. Zusätzlich zeigen wir, wie man traditionelle Statistik-Software mit wenig Aufwand anbinden kann, am Beispiel von einer Integration mit R. Dabei beachten wir die besonderen Anforderungen, die die Integration in den Zahlungsverkehr mit sich bringt.

1 Einleitung

Das Online-Banking ist für den Endkundenmarkt vieler deutschen Banken zum Hauptgeschäftsweg geworden. Dieser ist allerdings auch zum beliebten Ziel für Angreifer geworden: Indem man die Identität eines legitimen Kunden annimmt, führt man schadhafte Überweisungen in dessen Namen aus. Dieses Problem wurde von der Bundesanstalt für Finanzdienstleistungen(BaFin) als hinreichend ernsthaft eingestuft, um ein Rundschreiben "Mindestanforderungen an die Sicherheit von Internetzahlungenßu veröffentlichen. In diesem wird den Finanzdienstleistern auferlegt ein Konzept zur Verhütung von betrügerisch initiierten Transaktionen zu entwickeln und umzusetzen. Darüber hinaus gibt es natürlich auch betriebswirtschaftliche Motivation Betrugsfälle rechtzeitig zu erkennen: Laut dem Bundeslagebild 2014 des Bundeskriminalamt entstand im vergangenen Jahr ein Schaden in Höhe von fast 28 Mio Euro.

Die Umsetzung der BaFin-Anforderungen kann verschiedene Formen annehmen. Manche Finanzinsitute setzen auf die manuelle Überprüfung von Überweisungen in das Aus-

land, andere entwickeln Experten-getriebene Regellösungen. Diese Methoden sind jedoch sehr arbeitsaufwändig. Aufgrund der Kritikalität des Zahlungsverkehr, ist eine rein maschinelle Ablehnung von verdächtigen Transaktionen zwar nicht vertretbar, jedoch empfiehlt sich eine automatische Verdachtserhebung, die dann einem Fachmann zur Prüfung vorgelegt wird. Diese Lösung kann die Effizienz der menschlichen Komponente im System stark erhöhen.

Die automatische Bewertung von Transaktionen benötigt eine große Menge an Daten, um ein ausgewogenes Bewertungsmodell zu erlernen. Ebenso gibt es im Online-Banking während der Spitzenzeiten eine relativ große Menge von eingehenden Zahlungsaufträgen, die aufgrund der teilweise synchronen Rückmeldung an den Kunden des Bewertungsergebnisses in kürzester Zeit erfolgen muss. Deshalb werden häufig horizontal skalierende Big-Data-Lösungen eingesetzt. Im folgenden beschreiben wir eine Infrastruktur, die sich aus einer Complex-Event-Processing-Engine, einem verteilten Dateisystem und einer NoSQL-Datenbank mit geringer Antwortzeit zusammen setzt. Das System wird von einem Clustermanagementsystem verwaltet, und ist intern wie extern durch eine Message-Queue lose verknüpft. Dieses System ergänzen wir im Folgenden um eine analytische Datenbank und ein performantes Batchverarbeitungsystem, sowie eine Anbindung an einen Analysearbeitsplatz. Die Datenflüsse in einem Grundbewertungssystem sind in Abbildung 1 beschrieben.

Als analytische Batchplattform in diesem Ökosystem setzen wir Apache Spark ein. Es erlaubt eine schnelle – fast interaktive – Verarbeitung von Daten im Cluster und bietet gleichzeitig eine schlankere und elegantere Programmierschnittstelle als Hadoop's Java-Map-Reduce an. Die Geschwindigkeit ermöglicht eine interaktive Verwendung von Spark in sogenannten Notebooks. Diese bieten eine Ein-/Ausgabeschicht über ein Webinterface an, was reproduzierbare Experimente und Visualisierungen ermöglicht. Dabei stellt sich die Herausforderung der Integration von Spark mit dem Bewertungssystem und in bankfachliche Prozesse. Gleichzeitig erlaubt Spark hier eine weitaus bessere Skalierung, als man mit klassischen Analysetools wie R erreichen würde, und hat dank in-Memory-Caching einen deutlichen Leistungsvorteil gegenüber Map-Reduce-basierten Lösungen.

Zunächst werden wir die genauen Anforderungen im Detail erörtern. Dabei beginnen wir mit den üblichen fachlichen Anforderungen an ein solches System, und den daraus abgeleiteten technischen Anforderungen. Es folgt eine kurze Einleitung in Spark, und wie es sich in ein bestehendes Big-Data-System einbetten lässt. Notebooks und deren Verwendungsmöglichkeiten werden in Abschnitt 4 erläutert. Darauf folgt eine Beschreibung der kompletten Integration der analytischen Komponente in das operative System. Schließlich sind die gewonnen Erkenntnisse im letzten Abschnitt noch einmal zusammengefasst.

2 Anforderungen

Die Betrugserkennung im Online-Banking ist Teil der streng regulierten Buchungsprozesse einer Bank. Das heißt, dass eine solche Anwendung den Vertraulichkeit- und Zu-

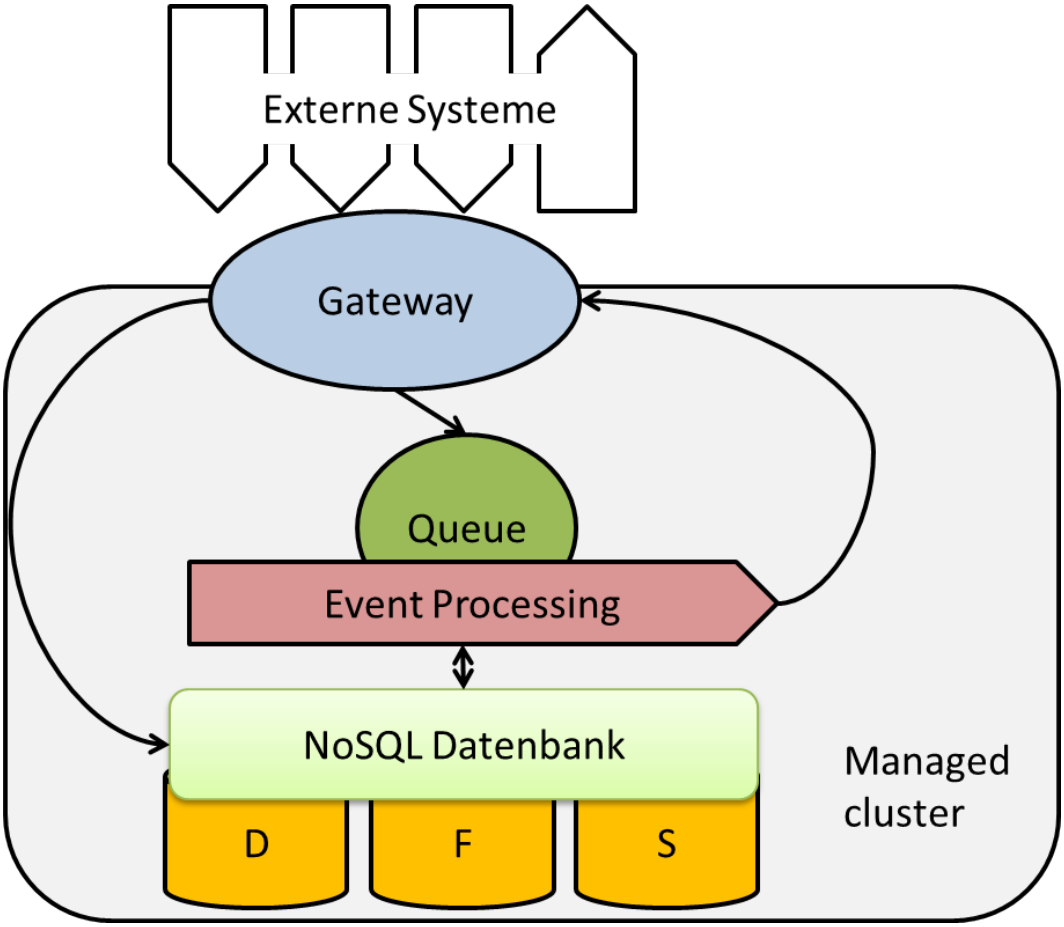

Abbildung 1: Ein über ein Gateway und Message Queue lose an das Bankumfeld gekoppeltes System, dass auf verteilte Verarbeitung und verteilte Datenhaltung (verteiltes Dateisystem und verteilte NoSQL-Datenbank) setzt. Asynchrone Datenlieferungen werden vom Gateway direkt in die Datenbank transferiert.

verlässigkeitsanforderungen an die Finanzdienstleistungsbranche gerecht werden muss. Andererseits werden Veränderungen am bestehenden Buchungssystem um eine Betrugserkennung durchzuführen aus Kostengründen gerne vermieden. Hier werden wir erst die fachlichen Anforderungen diskutieren, und danach diese in technische Anforderungen transformieren, und mit den üblichen technischen Gegebenheiten bei Finanzdienstleistern abstimmen.

2.1 Fachliche Anforderungen

Die fachlichen Anforderungen an ein Betrugserkennungssystem sind natürlich Kosten-Nutzen-getrieben. Die Existenz einer Betrugserkennungslösung wird zwar von der BaFin vorgeschrieben, aber die Ausprägung steht frei.

Je nach Situation des Finanzdienstleisters, sind verschiedene Kosten bemerkbar. Wenn Betrug ein großes Problem für eine Bank ist, dann sind oft die betriebswirtschaftlichen Verluste durch Betrug ein signifikanter Kostenpunkt, den man durch Investition in ein Betrugserkennungssystem aufwiegen kann. Dazu kommen immaterielle Kosten in Form von Vertrauensverlust in die Bank bei betroffenen Kunden.

Ein weiterer Kostenpunkt ist die Erkennungsgenauigkeit eines solchen Systems. Es sollten möglichst viele Betrugsfälle erkannt werden, wobei aber die Anzahl der manuell zu überprüfenden Aufträge einen festen Wert nicht überschreiten darf – die Sachbearbeiter sind eine teure und beschränkt verfügbare Ressource.

Dazu kommen noch weitere fachliche Anforderungen: Finanzdienstleister sind sehr eingeschränkt darin, welche Arten von automatischer Bewertungslogik sie einsetzen können, um die Nachvollziehbarkeit getroffener Entscheidungen zu gewährleisten. Dies soll zum Beispiel Kunden bzw. die Bank vor Diskriminierung oder dem Vorwurf von Diskriminierung schützen.

Ein weiterer Schlüsselaspekt ist, dass sich das Verhalten von Kunden, wie von Betrügern, über die Zeit ändert. Ein Betrugserkennungssystem muss sich dementsprechend ständig weiterentwickeln, und auch alte Informationen "vergessen" können, um auf Dauer seine Genauigkeit zu bewahren.

Dazu kann es sein, dass eine Betrugsverdachtsbewertung um Dialog mit dem Endkunden getätigt werden muss. Dafür muss diese mit einer geringen Latenz zur Verfügung stehen, um eine interaktive Verwendung anbieten zu können.

2.2 Technische Anforderungen

Die technischen Anforderungen an ein Betrugserkennungssystem leiten sich aus den fachlichen Anforderungen ab, oder werden durch technische Rahmenbedingungen bestimmt:

- Das Bewertungssystem muss lose an den Zahlungsverkehr gekoppelt sein, um Beeinträchtigungen zu vermeiden.
- Es muss eine Antwortzeit für Einzelüberweisungen unterhalb einiger hundert Millisekunden haben.
- Die verwendeten Modelle müssen regelmäßig aktualisiert werden können.

- Die Bedienung und Steuerung in Echtzeit muss gewährleistet werden.
- Das System muss ständig überwacht werden, um kritisches Verhalten rechtzeitig zu entdecken.
- Es muss ein Mindestmaß an Ausfallsicherheit gewährt werden, zumindest um die fortschreitende Datensammlung zu garantieren.

Dazu kommen Anforderungen an die Integration von Analyse und Bewertung:

- Analyse und Bewertung müssen auf der selben Datengrundlage geschehen.
- Die Analyse darf die Bewertungsperformance nicht so weit beeinflussen, dass versprochene Maximallatenzen nicht mehr eingehalten werden können.
- Analyse muss interaktiv und im Batch möglich sein um Datenexploration und Parametertuning gleichermaßen zu betreiben.
- Es muss eine Analyse- und Entwicklungsumgebung mit Zugriff auf Produktivdaten erstellt werden.
- Es muss eine Feedbackfunktion geben um die Vorhersagequalität des Systems bewerten zu können.

Diese Anforderungen werden durch eine Big-Data-Plattform sehr gut abgedeckt. Hadoop etwa bietet Ausfallsicherheit (auf Knotenebene) für Datensammlung, und multimandantefähige Clusterressourcenverwaltung, sowie Scheduling. Storm leistet Realtime-Complex-Event-Processing und mit Anbindung an eine NoSQL-Datenbank kann es die notwendigen Berechnungen in wenigen Millisekunden durchführen. Spark schließlich integriert sich in die meisten Cluster-Ressourcenmanager und bietet sich als Batch- und Explorationsplattform an, in Kombination mit einer entsprechenden Entwicklungsumgebung.

Wie sich dies als Erweiterung der in Abbildung 1 skizzierten Architektur verstehen lässt, wird in Abbildung 2 gezeigt. Im folgenden gehen wir auf zwei mögliche Implementationen dieser Architektur ein: Eine Variante auf Basis einer klassischen Hadoop-Lösung, sowie eine alternative Lösung mit Spark-Cassandra-Integration, ohne Hadoop-Komponenten.

3 Spark im BigData-System

Spark ist eine verteilte, in-memory Rechenplattform, nach dem Map-Reduce-Paradigma. Dabei wird ein gerichteter azyklischer Prozessgraph in Scala, Java oder Python beschrieben. Dieser wird durch Hinzufügen eines Resultat-generierenden Schrittes ausgeführt (zum Beispiel `count`). Diese sogenannte "lazy evaluation" ermöglicht die interaktive Verwendung, da nicht jeder ausgeführte Schritt sofort zu einer langwierigen Berechnung führt.

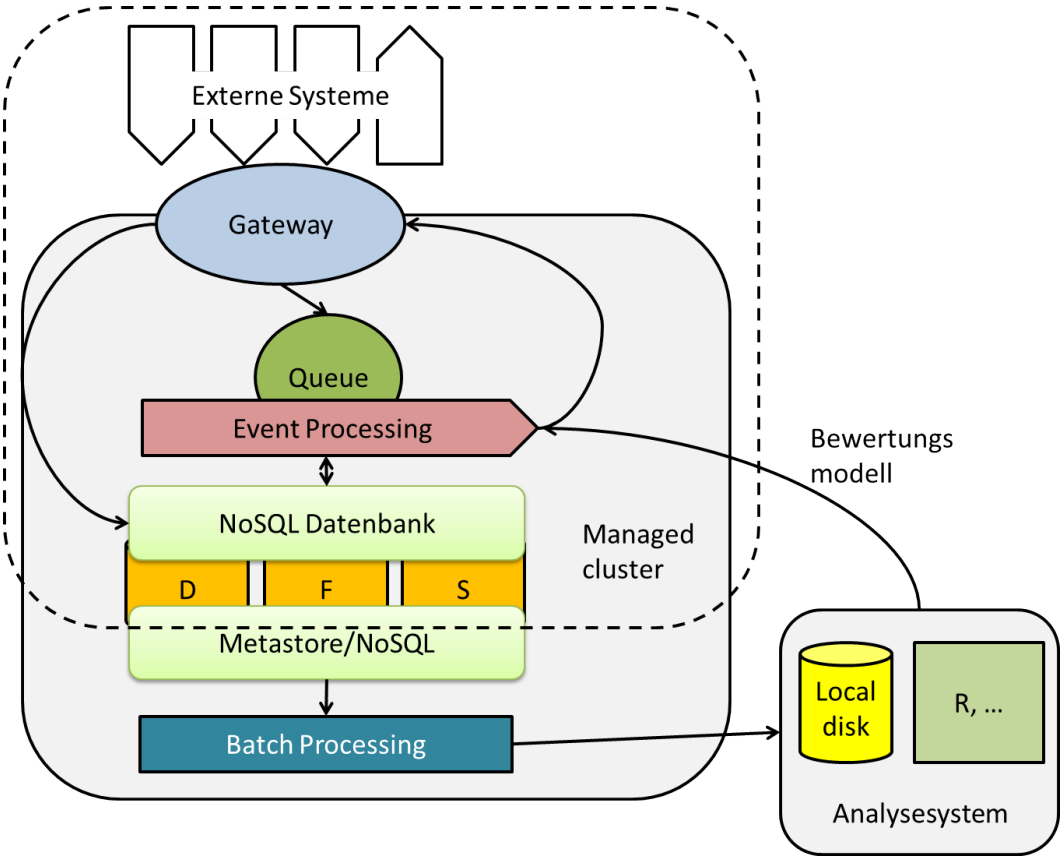

Abbildung 2: Das bestehende System wird durch eine Erweiterung mit einem leistungsfähigen Batch-Verarbeiter (in der Regel Spark) und einer analytischen NoSQL-Lösung um eine initiale Analyselösung erweitert. Eine klassische, R-basierte (oder angepasste) Analyseplattform kann einfach angebunden werden. Gelernte Modelle werden exportiert (beispielsweise im PMML-Standard) und vom Event-Verarbeiter für die Klassifikation eingesetzt.

Im Vergleich zum klassischen Hadoop-Map-Reduce, ist Spark in der Lage die Festplatten-I/O-Last von Knoten zu reduzieren, indem zwischen Map- und Reduce-Abfolgen das Zwischenergebnis soweit wie möglich im Speicher gehalten wird, oder Operationen im Prozessgraphen kombiniert werden. Das führt ebenfalls zu einem signifikanten Performancegewinn. Zusätzlich kann man auch als Anwender/Programmierer bestimmte Schritte im Ausführungsgraphen "checkpointen". Dabei wird das Zwischenergebnis an dieser Stelle im Speicher oder auf Festplatte solange gehalten, bis es wieder explizit freigegeben wird. So kann man zum Beispiel auf einem transformierten Zwischenergebnis ad hoc mehrere Analysen und Aggregationen durchführen, ohne je von einer Festplatte lesen zu müssen.

Daten in Spark werden in "Resilient Distributed Datasets"(RDD) organisiert – einer verteilten, replizierten Datenstruktur. Ein RDD ist in mehrere Partitionen unterteilt, die auf die verfügbaren Knoten gleichmäßig verteilt werden. Zusätzlich gibt es mehrere Abstraktionsebenen auf das RDD, zum Beispiel DataFrames, die Schemametadaten enthalten Pair-RDDs, die Key-Value-Operationen unterstützen, sowie einige RDDs, die importierte, externe Datentypen abbilden, zum Beispiel CassandraRDDs.

Spark ist flexibel einsetzbar, dank einer Vielzahl von Verbindungsmodulen auf Datenquellen und -speicher. Außerdem lässt es sich autark auf ein Cluster verteilen, oder in bestehende Mesos- oder YARN-Umgebungen einbinden. Aufgrund der starken Marktposition von Hadoop im Big-Data-Ökosystem betrachten wir diese Kombination detailliert. Eine alternative Integration in ein Mesos/Cassandra-System beschreiben wir ebenfalls.

3.1 Integration mit HDFS und Hive

Die Integration mit HDFS ist bei Spark selbstverständlich, da man sich im Apache-Big-Data-Stack positioniert. Lese- und Schreibkonnektoren für HDFS sind daher im Basisumfang enthalten. Damit lassen sich schemalose Daten einlesen und parsen.

Ebenso ist es möglich Daten mit Schema einlesen, um diese mit SparkSQL zu verarbeiten. Zum Beispiel lassen sich Parquet-Dateien direkt in SparkSQL importieren, und Hive-Tabellen über den Hive-Metastore in einen sogenannten HiveContext laden. Dabei werden DataFrames generiert, die sich leicht in normale RDDs verwandeln lassen, falls man das Schema programmatisch via einer `case class` festlegen möchte. Auf diesem Weg erlangt man Typensicherheit, die SparkSQL nicht garantieren kann. Auch das Schreiben von Schema-behafteten Date in Hive und Parquet-Dateien ist möglich. Dadurch kann man das Hadoop-interne Data-Warehouse (DWH) effizient anbinden und mit SparkSQL abfragen. So kann man Abhängigkeiten zu externen Datentöpfen vermeiden, was der Data-Governance zugute kommt. Eine daraus resultierende mögliche doppelte Datenhaltung ist in der Regel tolerierbar, da die Ressource Hadoop flexibler ist als das klassische DWH.

3.2 Integration mit NoSQL-Datenbanken

HBase ist aktuell noch nicht besonders gut an Spark angeschlossen. HBase-Spark steht erst in der aktuell noch sehr experimentellen HBase Version 2 zur Verfügung. Zwar wird ein Backport auf Version 1 angegangen, allerdings wird es noch einige Zeit dauern, bis

die Hadoop-Plattformanbieter diese Kombination unterstützen.

Darüber hinaus gibt es ein HBase-RDD-Projekt. Es hat allerdings keine offizielle Unterstützung seitens der Big-Data-Distributoren. Cassandra bietet dank Unterstützung durch Datastax wesentlich bessere Konnektoren zu Spark, allerdings treibt Cloudera die Integration von HBase stark voran.

Als Alternative – in Ermangelung einer Standardlösung – bleibt eigenhändig eine Datenbankverbindung zu erstellen. Dies hat allerdings ebenfalls nur begrenzten Nutzen, da bei größeren Datenmengen im Batchbetrieb Bulk-Operationen oft sinnvoll sind, aber zusätzlichen Entwicklungsaufwand bedeuten, um diese performant umzusetzen. Ein Zugriff auf operativ eingesetzte HBase-Regionen ist außerdem nicht günstig, da diese oft den Performance-Engpass für das Latenz-begrenzte Operativsystem darstellt. Zusätzliche Last auf diese Komponente, sowie das Invalidieren von Cache durch große Leseoperationen kann schnell dazu führen, dass Antwortzeitvorgaben nicht mehr eingehalten werden. Aus diesem Grund ist die einfachere Integration von Hive in den meisten Fällen vorzuziehen.

Alternativ kann man die gute Integration von Cassandra nutzen, und die Performancenachteile abpuffern, indem man einen separaten Hardwarepool verwendet, und eine zusätzliche Cassandra-Instanz. Dadurch hat man einen leichten Performancevorteil gegenüber Hive, durch die bessere Integration von Cassandra und Spark, und man reduziert die Zahl der eingesetzten Technologien. Auch beim Einsatz von Hive ist zudem zu beachten, dass man I/O-Ressourcen des HDFS-Clusters verwendet, auf dem auch die Daten persistiert sind, die HBase verwendet, wenn es eine Anfrage nicht aus dem Cache bedienen kann. Auch dort ist somit der Einsatz von dedizierter Hardware eine Option, wenn Ressourcenkonflikte die analytische Arbeit zu sehr einschränken.

3.3 Integration mit YARN

Spark wurde von Anfang an für den Betrieb unter YARN ausgelegt und ist dort gut unterstützt. Es werden dafür die Modi `yarn-client` und `yarn-cluster` angeboten. Ersterer startet einen Driver-prozess auf der Maschine, von der aus `spark-submit` (dieses Skript bringt Sparkprogramme im Cluster aus) aufgerufen wurde, beim zweiten wird ein Knoten im YARN-Pool die Driver-Rolle zugewiesen. Ersteres ist insbesondere von Nutzen, falls der Prozess Netzwerkverbindungen annehmen soll, oder anderweitig Logik ausführt, die einer bestimmten Maschine zugeordnet sein soll. YARN und Spark ermöglichen eine dynamische Skalierung. Exekutoren können nach Bedarf zugeschaltet oder deaktiviert werden um Speicher wieder freizugeben. Zusätzlich kann man mit YARN Ressourcenpools strikt trennen. Dies erlaubt eine Priorisierung des operativen Systems über die Analyse.

Das diagnostische SparkUI wird vom Yarn-Resource-Manager gehostet, wenn man einen der beiden YARN-Modi verwendet. Dies bedeutet, dass man einfachere Firewall-Regeln umsetzen kann, da die Standardports für das SparkUI nur noch kosmetische Funktion haben, und man direkt per YARN-Job-ID auf die UI zu jedem Spark-Job zugreifen kann, auch wenn mehrere gleichzeitig aktiv sind.

Alternativ bieten sich unter Mesos ähnliche Möglichkeiten an. Insbesondere wurde

Mesos von den Entwicklern von Spark vorgestellt und unterstützt auch den Einsatz von Cassandra. Spark unterstützt identische client- und cluster-Modi auf Mesos wie auch auf Yarn. Damit bietet es eine vollumfängliche Alternative zum Hadoop-Stack.

4 Spark Notebooks

Ein Notebook ist ein Paradigma um Rechenressourcen interaktiv zu nutzen. Das Konzept dabei ist, dass man ein Ergebnis direkt mit dem Code assoziiert, der es generiert hat. Dies stellt sicher, dass Ergebnisse reproduzierbar sind. Reporduzierbarkeit ist insbesondere hilfreich für Datenexploration, das entwickeln von Proof-Of-Concept-Code oder dem Dashboarding. Insbesondere kann man es als eine Alternative zum Testen von Code im vor-Produktionsstadium verstehen, da es die Nachvollziehbarkeit und Wiederverwendbarkeit des Codes sicherstellt. Wie relevant dies für die Datenanalyse mit Spark ist zeigen auch Databricks, die "Erfinder von Spark", die eine eigene Notebooklösung im als Kern ihres Angebots verstehen.

Technisch gesehen steckt hinter einem Notebook eine Web-Anwendung, die durch einen "Kernel"gestützt wird. Server-seitig wird das Spark-REPL-Interface (Read-Eval-Print-Loop) verwendet um Code-Blöcke aus "Absätzenßu kompilieren und auszuführen. Client-seitig – also im Webbrowser – wird eine sehr leichtgewichtige Entwicklungsumgebung angeboten. Einzelne Absätze können in beliebiger Reihenfolge (aber nicht gleichzeitig) gestartet werden. Jeder Absatz besteht aus genau einem Stück Code und dem aus der Ausführung resultierenden Ergebnis.

Es gibt eine Vielzahl von Notebooks, die Spark-kompatibel sind, in verschiedenen Entwicklungsstadien: Das bereits genannte Databricks Cloud Notebook, das von Cloudera im Rahmen des Hue-Projekts entwickelte Hue-Notebook, die Anbindung von Jupyter an Spark, Spark-Notebook und Apache Zeppelin. Die letzteren Beiden sind die am meisten genutzten und am weitesten entwickelten Open-Source-Varianten, die wir im Folgende näher betrachten werden.

4.1 Spark-Notebook

Spark-Notebook ist streng Scala-orientiert. Das heißt es bietet eine Entwicklungsumgebung mit Code-Vervollständigung, aber auch weniger Vielfalt und Möglichkeiten andere Sprachen einzubinden, als andere Notebooks. Es wird je ein "Kernel"pro Notebook instanziiert. Das bedeutet, dass jedes Notebook, von denen ein Spark-Notebook-Server mehrere gleichzeitig aktiv halten kann, einen eigenen SparkContext erhält. Dadurch gelten Variablen nur in dem Notebook, in dem sie definiert werden. Eine weitere besondere Eigenschaft von Spark-Notebook ist, dass es eine Streaming-Aktualisierung von Visualisierungen unterstützt, und sich so besonders für Dashboarding mit Spark-Streaming eignet.

4.2 Apache Zeppelin (incubating)

Während Spark-Notebook ein Ein-Mann-Projekt ist, ist Zeppelin in der Apache Software Foundation eingebettet, allerdings noch im Inkubationsstadium. Dennoch haben sich bereits Unterstützer unter Anderem in den Reihen von Hortonworks gefunden, die Zeppelin bereits in ihrer Distribution aufgenommen haben.

Im weiteren Gegensatz zu Spark-Notebook wird lediglich ein einziger SparkContext per Serverinstanz geschaffen. Das heißt Variablen sind über Notebooks hinaus gültig, und man kann Datenstrukturen über die Grenzen persönlicher Notebooks hinweg teilen, wenn diese in Spark persistiert sind. Insbesondere kann man auch Python und Scala im gleichen SparkContext gemischt verwenden. Zeppelin ist wesentlich vielfältiger als Spark-Notebook, es gibt mehr eingebaute Interpreter, über Spark hinaus. Dies bedeutet allerdings auch, dass die Oberfläche weniger optimiert für Spark ist, als möglich wäre. Um Zeppelin herum hat sich auch bereits ein eigenes Ökosystem gebildet um Autorisierung und die Installation zu verwalten.

5 Operative Integration

In diesem Abschnitt beschreiben wir, wie sich die Komponenten in die Umgebung einpassen. Technisch integrieren sich die verschiedenen Lösungen gut, wie oben kurz beschrieben. Allerdings müssen Datenprodukte, die der Analyse entspringen – in diesem Fall ein Betrugserkennungsmodell – aus der Analyse in das operative System übertragen werden. Dafür empfiehlt sich entweder die Verwendung von serialisierten Java-Objekten, oder die Verwendung eines Austauschformats wie PMML. Letzteres ist völlig plattformunabhängig, aber wird von Spark (aktuell) nur schlecht unterstützt. Ebenso sind nur sehr grundlegene Entscheidungsbaumalgorithmen in Spark vorhanden. Deshalb wird R als zusätzliche Komponente bis auf Weiteres eine wichtige Rolle spielen. da dieses jedoch lediglich mit Datenmengen funktioniert, die in den lokalen Arbeitsspeicher passen, muss ein Arbeitsablauf beschrieben werden, der die Stärken von R und Spark aufeinander abstimmt.

Darüber hinaus muss das Modell technisch in das Bewertungssystem ausgebracht werden. Dabei muss eine QA-Prüfung durchgeführt werden, um sicherzustellen, dass das Gesamtsystem weiter den Erwartungen entspricht. Beide Prozessabläufe sind in diesem Abschnitt beschrieben.

5.1 Integration mit R

Eine R-Umgebung bietet mehr fertige Algorithmenpakete als Spark, und potenziell bessere Performance für nicht verteilbare Probleme des maschinellen Lernens. Allerdings ist natives R nicht horizontal skalierbar. Im Anwendungsfall Betrugserkennnung spielt dies keine große Rolle. Man verwendet Entscheidungsbäume aufgrund der fachlichen Anforderungen, und diese müssen auf balancierten Datensätzen trainiert werden. Da es nur wenige Betrugsfälle gibt, kann man hier mit einer kleinen Teilmenge der Daten arbeiten, und muss das sogar tun. Balancierte Datensätze passen komfortabel in den Speicher

einer einzelnen Maschine und können aus Spark auf die lokale Platte eines Rechners im Verbund geschrieben werden. Durch iteratives Sampling kann man die Stabilität des Modells überprüfen: Wenn man mit unterschiedlichen Extrakten vergleichbare Modelle trainiert, dann ist jedes herangezogene Extrakt hinreichen repräsentativ. Später kann man in Spark und im Live-Betrieb gegen große Datenmengen verproben.

5.2 Integration mit Storm

Storm ist die Referenz für Open-Source Complex-Event-Processing mit niedriger Verarbeitungslatenz. Modelle aus der Analyseumgebung (aus einer lokalen Lösung oder direkt aus Spark's Machine-Learning-Komponente) müssen in Storm ausgebracht werden. In unserem Fallbeispiel werfen PMML-export und JPMML-Evaluation eingesetzt um das Modell zu serialisieren und deserialisieren. Da Stormbolts verteilt laufen und zur Laufzeit aktualisiert werden soll, werden Zookeeper-Watches eingesetzt um Veränderung am PMML an die Worker zu verteilen. Zusätzlich ist ein Qualitätssicherungsprozess nötig, der gewährleistet, dass trainierte Modelle gewisse Mindestanforderungen erfüllen, zum Beispiel nicht schlechtere Genauigkeitsmetriken, als das aktuell im Einsatz befindliche, auf neuen Daten. Da dieser Prozess allerdings innerhalb des Betriebs der Betrugserkennungsplattform stattfinden sollte, muss die Möglichkeit bestehen klassische Softwareausbringungsmechanismen und offizielle Teststufen zu umgehen. Stattdessen muss jede Veränderung des aktiven PMML-Modells verfälschungssicher auditiert werden, um Rechenschaft über dieses System zu führen. Eine beliebte Alternative zu Storm findet sich in Akka, einem leichtgewichtigeren Framework, welches allerdings dadurch auch teilweise einen höheren Entwicklungsaufwand mitbringt um die Verarbeitungsgarantien die Storm, und insbesondere Storms Microbatchumgebung Trident, umzusetzen. Die Aktualisierung des Modells in einer Akka-Umgebung würde allerdings die gleichen Mechanismen verwenden, wie das auch in Storm der Fall ist.

5.3 Integration in ein Dashboard

Bei einem solchen, den Zahlungsverkehr beeinflussenden Modul, ist es wichtig stets das aktuelle Systemverhalten bewerten zu können. Dies gilt sowohl für technische Metriken (Antwortzeiten, Systemlast), wie auch für fachliche (Anzahl der gemeldeten Transaktionen, aktuelle Schadensfalllage). Dafür eigenen sich Dashboards gut. Mit Spark-Notebook, SparkSQL und Spark-Streaming lassen sich diese schnell erstellen und gut warten. Für Drill-downs kann man dann HBase oder Hive anbinden (wie oben beschrieben) - allerdings sind sehr dynamische Dashboards in einer Notebookumgebung schwer umzusetzen. Falls dies erwünscht ist kann man aber auch viele klassische Dashboarding-Tools per Thrift-Schnittstelle an SparkSQL anbinden, oder eine Hive-Integration nutzen.

6 Zusammenfassung

Ein automatisch lernendes Betrugserkennungssystem als Expertenhilfssystem ist eine optimale Lösung um den Zahlungsverkehr abzusichern und Kundenvertrauen aufzubauen.

Eine Umsetzung auf Basis von Hadoop und Storm ist eine zuverlässige und skalierbare Variante, die dank offenem Quellcode Vendor-Lock-In vermeidet. Die Erweiterung eines solchen Systems um eine Analyseumgebung, die direkten Zugriff auf die Daten im Cluster und dennoch beinah-interaktiv Ergebnisse und Einsichten anbietet, ermöglicht eine fachliche Skalierung: Man kann nun statistisch fundierte Modelle trainieren, auswerten und einsetzen, und nicht mehr rein auf Fachlogik für die Bewertung zurückgreifen. Eine analytische Umgebung aus Spark und R bietet – ebenso Open-Source wie die Plattform selber – eine Lösung für dieses Problem, und lässt sich nahezu nahtlos integrieren. Dabei werden Clusterressourcen besser ausgenutzt, da man ein Datenbanksystem auf Lastspitzen auslegen muss um SLAs einzuhalten, und dann bei Unterauslastung der Rechenressourcen diese mit Yarn's und Spark's dynamischer Skalierung und Priorisierung für Analysezwecke zusätzlich nutzen kann.

Eine alternative Implementierung ohne Hadoop und Storm ist ebenso möglich, wie wir an einigen Stellen kurz angedeutet haben. Dies könnte bei der Erweiterung bestehender Systeme eine günstigere Alternative sein. Ebenso ist Mesos etwas besser für generische Anwendungen ausgelegt als Yarn, was ein wichtiger Aspekt ist, falls man zusätzliche Dienste im gleichen Cluster einsetzen möchte, oder bestehende Lizenzverträge erweitern möchte. In Isolation betrachtet für den speziellen Use-Case ist der All-In-One-Aspekt, den eine Hadoop-Distribution bietet jedoch ein klarer Vorteil gegenüber der Alternativlösung.