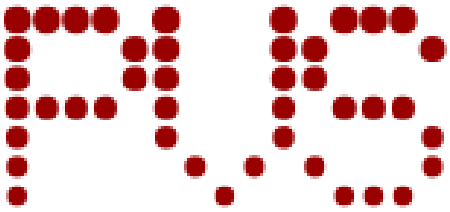


Dokumentation zu "Spark4KNIME"-Projekt

Entwicklung eines KNIME-Plugin, um Spark-"Programme" via KNIME zu erstellen und auszuführen

Dozenten: Prof. Dr. Artur Andrzejak
Mohammadreza Ghanavati
Entwickelt von: Oleg Pavlov

im Rahmen des Softwarepraktikum "Parallele Datenverarbeitung",
Sommersemester 2015, Ruprecht-Karls-Universität Heidelberg

9. Oktober 2015

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Dokumentation dient zur Einsicht in die Entwicklung sowie zu einer besseren Vorstellung der Anwendung des Projektes "Spark4KNIME". Als erstes werden in den Kapiteln 2 und 3 die entsprechenden Frameworks kurz vorgestellt. Kapitel 4 beschreibt die technische Seite der Entwicklung. Zum Schluss werden die Ergebnisse des Projektes beschrieben sowie ein Anwendungsbeispiel gezeigt.

Die Dokumentation ist mit der Softwareverison 1.0.0 verknüpft.

1.1 Motivation und Zielsetzung

Seit langer Zeit beschäftigt man sich mit dem Thema Big Data. Mit dem Zuwachs an Datenmengen, die für unterschiedliche Zwecke verarbeitet werden, sind auch die Anforderungen an die entsprechenden Frameworks gestiegen. So werden z.B. neue Ansätze entwickelt und somit entstehen auch neue Frameworks. Aber was tut man, wenn zur Datenverarbeitung gleich mehrere Software eingeschaltet werden müssen, man möchte die Vorteile mehrerer Frameworks nutzen. Das kann unter Umständen sehr schwierig sein, weil man viele Faktoren wie Programmiersprache, Datenformat und weitere Abhängigkeiten beachten soll.

Das Projekt "Spark4KNIME" hat die Vereinigung zweier Frameworks für Datenanalyse zum Ziel. Genauer soll ein Plugin für KNIME implementiert werden, welcher eine Anbindung zum Spark ermöglicht. Das Erstellen und Ausführen von Spark-"Programmen" soll, wie in KNIME üblich ist, schnell und einfach durch das Zusammensetzen von notwendigen Modulen funktionieren.

2 Apache Spark

Spark entstand im Rahmen eines Forschungsprojekts am AMPLab der University of California in Berkeley und wird aktuell von der Apache Software Foundation weitergeführt. Es ist also ein Open-Source Framework zur verteilten Verarbeitung und Analyse großer Datenmengen. Es bietet High-Level-API's in Java, Scala, Python und R und unterstützt eine breite Palette an Tools für Verarbeitung von SQL und strukturierten Daten, Graphen, Datenströmen sowie maschinelles Lernen.

Der Ansatz hinter Spark bringt viele Vorteile gegenüber dem *Map-Reduce* Programmierparadigma z.B. bei dem "großen Bruder" *Apache Hadoop*. Spark speichert die verteilten Daten in einem Resilient Distributed Dataset (RDD) Objekt. Alle Operationen auf RDD unterteilen sich in zwei Gruppen: Transformationen und Aktionen. Transformationen, wie der Name bereits verrät, wandeln einen RDD je nach Anweisung in einen anderen RDD um. Dabei wird in der Operation nur die Referenz auf Datensatz und Anweisung vermerkt, die auf den Datensatz ausgeführt werden soll. An dieser Stelle wird erstmal kein Resultat berechnet. Wenn eine Aktion auf das RDD aufgerufen wird, bildet Spark

intern einen gerichteten azyklischen Graphen (DAG) aus den Anweisungen auf den Datensatz und führt diesen Graph aus. Das ist der grundlegende Unterschied zwischen den beiden Gruppen.

Während Map-Reduce nur zwei Funktionen (logischerweise Map und Reduce) unterstützt, ist DAG mehr flexibel mit mehr Funktionen wie map(), filter() sort- oder groupByKey() usw. Die Ausführung vom Graph ist auch schneller als eine vergleichbare Map-Reduce Routine mindestens weil die Zwischenergebnisse aus Transformationen nicht zum Main-Programm geschickt bzw. auf Disk geschrieben werden. Bearbeitung von Daten im Hauptspeicher eines Rechners sorgt ebenfalls für eine kürzere Laufzeit eines Algorithmus. In [MZ12] wurde Resilient Distributed Dataset bestens beschrieben. Für eine bessere Vorstellung über die Arbeitsweise von Spark Framework ist der Artikel nur zu empfehlen.

3 KNIME

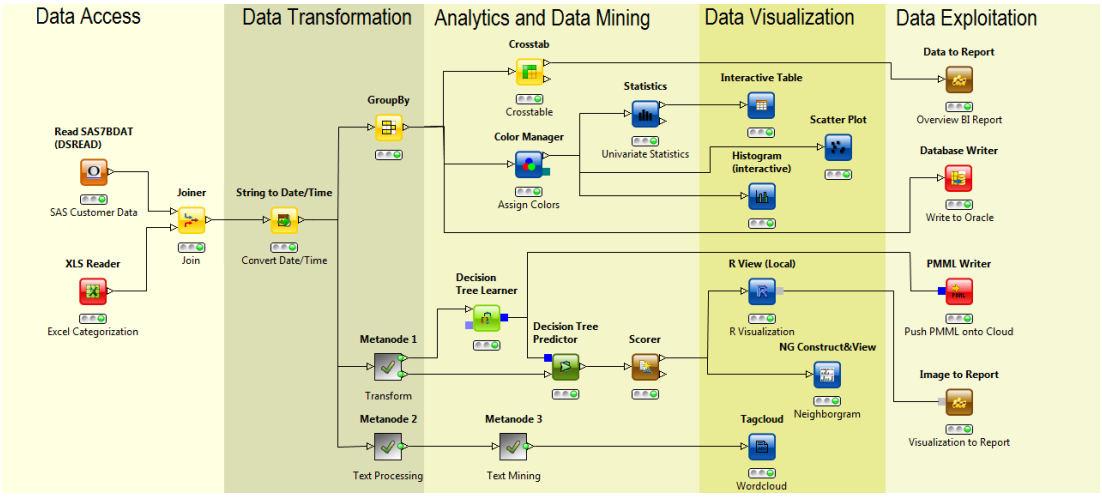
Ein Framework, das ebenso für Datenanalyse eingesetzt wird, ist der Konstanz Information Miner (*KNIME*). An der Universität Konstanz begann im Jahre 2004 die Entwicklung einer Plattform, die in erster Linie nach Baukastenprinzip funktionieren sollte. Die erste öffentliche Version als ein Eclipse Plugin wurde 2006 insbesondere wegen der Einfachheit in Anwendung sehr beliebt.

Das System ermöglicht dank seiner graphischen Oberfläche die Beschreibung ganzer Data-Mining-Prozesse bequem auf modularer Ebene. Aus einzelnen Modulen/Knoten wird durch das Pipeline-Konzept ein Arbeitsablauf "ohne Programmieren" zusammengestellt. Jeder Knoten enthält eigene Anweisungen, wie die Inputdaten verarbeitet werden. Über ein optionales Dialog-Fenster eines Knotens kann Benutzer durch das Setzen eigener Parameter die Routine anpassen. Danach übernimmt KNIME die Ausführung von den Methoden sowie Weiterleitung der Outputdaten an den nächsten Knoten. Abbildung 1 zeigt, wie ein Workflow in KNIME aussehen kann.

4 Entwicklung

Dieses Kapitel richtet sich ehe an die Entwickler, die ein Interesse haben, das im Projekt "Spark4KNIME" entwickelte Plugin weiterzuführen. Das benutzte Software Development Kit (SDK), das entstandene Architektur sowie einige Feinheiten der Anbindung zum Spark werden vorgestellt. Eine Dokumentation zur Anwendung vom Plugin findet man im Kapitel 5.

Am Anfang der Entwicklung war es nicht sofort klar, wie man einem KNIME-Benutzer die Möglichkeit geben kann, Aufgaben für Spark zu erstellen und diese dann parallel auf einem Spark Cluster auszuführen. Der erste Gedanke war, dass jeder Knoten in KNIME ein Algorithmus darstellen soll. Die Liste der im Spark Bibliothek implementierten Algorithmen ist sehr groß. Dank der Apache Spark community entstehen immer wieder neue Algorithmen, trotzdem möchte man sich nicht nur auf die vorhandenen Methoden



Quelle: https://www.knime.org/knime

Abbildung 1: Ein möglicher Prozessablauf in KNIME, unterteilt in unterschiedliche Arbeitsbereiche wie Datenin-/output, Data-Mining, Datenvisualisierung

beschränken. Außerdem steht eine Vielzahl von den Methoden bereits in KNIME zur Verfügung.

Eine neue Idee musste her. Nach einer Besprechung wurde festgelegt, dass die KNIME Knoten die einzelne Methoden auf einem RDD repräsentieren sollen. So hat man, nachdem die Eingabedaten in einem RDD gespeichert sind, die größte Freiheit für eine parallele Datenanalyse mit Spark.

4.1 KNIME SDK

Für die Entwicklung eigener Knoten bietet KNIME eigenen Software Development Kit (SDK). Vollständige Entwicklungsumgebung *Eclipse Indigo* mit Java virtueller Maschine und KNIME kann von der Webseite des KNIME Projekts[1] heruntergeladen werden. Zur Vereinfachung beinhaltet SDK einen *New Node Wizard* für das Erstellen eines KNIME Knoten. Natürlich bietet Eclipse weitere Möglichkeiten, die entstehende Knoten zu testen und debuggen oder ein fertiges Projekt als ein KNIME PLugin zu exportieren. Ein Guide für Entwickler ist ebenfalls auf der KNIME Webseite vorhanden und erklärt die Einzelheiten sehr gut. Für das Verständnis weiterer Kapitel wird es empfohlen, diese Anleitung durchzulesen und wenigstens einen eigenen Knoten zu erstellen.

4.2 Projektordner

Durch den New Node Wizard werden alle für einen kleinsten Knoten (z.B. namens MyNode) notwendigen Dateien und Klassen erstellt. Nachfolgend ist eine Liste der Dateien.

[1] https://knime.org/downloads

- plugin.xml
- build.properties
- META-INF/MANIFEST.MF
- org/default/package/MyNodeNodeDialog.java
- org/default/package/MyNodeNodeFactory.java
- org/default/package/MyNodeNodeModel.java
- org/default/package/MyNodeNodeView.java
- org/default/package/MyNodeNodeFactory.xml
- default.png
- package.html

Die ersten drei Dateien sind für das Deployment grundlegend. Sie werden durch Eclipse gesondert behandelt und in einem *Plug-in Manifest Editor* geöffnet. plugin.xml bietet eine Möglichkeit, eigene Kategorien für KNIME zu definieren. Den Kategorien können dann die einzelnen Knoten zugeordnet werden. Weiterhin wurden Erweiterungen erstellt, die für das Referenzieren von Code Template Träger notwendig sind. Die Templates werden in dem Knoten "JavaSnippetForRDD" benutzt. Projektname und Version, Java Version und Klassensuchpfad sowie Plugin Abhängigkeiten werden in MANIFEST.MF definiert. In build.properties kommt die Information über das Kompilieren des Plugins. Dazu gehört ein Pfad zum Source Code sowie eine Liste der Dateien, die mit dem Plugin ausgeliefert wird.

Die externen Bibliotheken, die im Manifest aufgelistet sind, werden vom Eclipse bzw. KNIME bereitgestellt. Die einzige JAR Datei, die für das Projekt unumgänglich ist und auf GitHub aber wegen seiner Größe nicht hochgeladen wird, ist Spark Lib. Diese gehört in den Ordner "lib" und muss manuell in eigener Kopie des Projektes vervollständigt werden. Im Projekt wurde für Hadoop 2.6 vorkompilierte Spark Bibliothek in der Version 1.3.1 benutzt. Die angegebenen Versionen sind nicht zwingend. Durch Benutzung anderer Versionen können manche Fehler behoben werden oder aber auch Inkompatibilitätsprobleme auftauchen.

Spark braucht für die Ausführung mancher Operationen Hadoop Binaries. Sollte zur Zeit der Anwendung des KNIME Plugins kein Hadoop auf dem Computer installiert sein, so setzt das Plugin die Umgebungsvariable "hadoop.home.dir" auf den Ordner "/hadoop" im Plugin. Der Ordner beinhaltet alle für die Ausführung notwendigen Hilfsprogramme und deren Bibliotheken.

Ordner "img" beinhaltet die Icons für die Spark Kategorien im KNIME. Die Icons für die einzelnen Knoten sind unmittelbar neben der Java Klassen der jeweiligen Knoten gespeichert.

Der "JavaSnippetForRDD" Knoten verfügt über Möglichkeiten die Code Templates zu erstellen, einzulesen und zu speichern. Einige vorgefertigten Templates sind im Ordner "jsnippets" zu finden.

Das Projekt beinhaltet insgesamt 16 Knoten in 4 Kategorien. Die Kategorienzuordnung sieht folgendermaßen aus:

- Input
  - TableToRDD
  - FileToRDD
- Tranformations
  - Union
  - Intersection
  - Sample
  - Distinct
  - JavaSnippetForRDD
  - SortByKey
  - GroupByKey
- Output
  - RDDToTextFile
  - RDDToSequenceFile
- Actions
  - Count
  - CountByKey
  - Take
  - TakeSample
  - Collect

Jede Kategorie davon wird in eigenem Java Package implementiert. Die Ausnahme bildet der Knoten "JavaSnippetForRDD". Kapitel 4.5 erläutert weitere Einzelheiten zu diesem Knoten.

Jeder Knoten im KNIME besteht grundsätzlich aus 4 Java Klassen. *NodeFactory* beschreibt zusammen mit einer XML Datei die Einstellungen für die Vorschau und das Dialogfenster. In die zusätzliche XML Datei gehört die Beschreibung des Knotens. Funktionalität des Knoten, Optionen im Dialogfenster, Vorschaumöglichkeiten sowie die Input und Output Daten können hier beschrieben werden. In KNIME werden diese Texte an verschiedenen Stellen als Hilfe angezeigt. Die Klasse selbst implementiert die Möglichkeit festzulegen, ob ein Knoten ein Dialogfenster oder eine Vorschau besitzt und was ausgeführt werden soll, wenn diese Fenster geöffnet werden. Die Fenster selbst werden in den Klassen *NodeDialog* und *NodeView* behandelt. Für die Ausführung eines Knotens ist die Klasse *NodeModel* zuständig. Hier wird die Anzahl an Inputs und Outputs definiert, die gesetzten Parameter im Dialogfenster verwaltet und die Callbacks für das Konfigurieren, Ausführen und Zurücksetzten eines Knoten implementiert.

Der Java Code zu den Knoten ist sehr gut lesbar und nach einer kurzen Einarbeitung relativ einfach zu verstehen. KNIME Developer Guide und Forum können in Einzelnen Fällen weiter helfen. In den nächsten 3 Kapiteln werden nur für das "Spark4KNIME" Projekt spezifische Themen wie Anbindung zum Spark, Speichern und Weiterleiten der Daten sowie Programmieren in KNIME erläutert.

4.3 Spark in KNIME

Die Anbindung zu einem Spark Cluster erfolgt über sein *master* URL. Ein solcher Pfad kann wie folgt aussehen:

- spark://HOST:PORT für einen entfernten Cluster
- local[*] für einen lokalen Cluster

Wird ein lokaler Cluster benötigt, muss dieser nicht gesondert gestartet werden. Mit dem Master "local[2]" wird automatisch ein Spark Cluster auf dem Rechner aufgestellt, wobei die Angabe in eckigen Klammer die Anzahl der Worker bedeutet. Stern (*) steht für die Anzahl der CPU Kerne des Computers. Für die Benutzung eines entfernten und bereits laufenden Cluster ist die Information über HOST und PORT des Masters nicht ausreichend. Außer dass das Master URL vom lokalen Rechner erreichbar sein muss, muss auch Master eine Verbindung zu dem lokalen Rechner erstellen können. Dies ist notwendig um die Ergebnisse der Berechnung zurück zu KNIME zu schicken.

Eine Angabe des Masters ist in den beiden Knoten in der Kategorie *Input* möglich. Mit diesen Knoten verteilt man die Daten (eine Datei oder Ausgangsdaten eines vorangegangenes Knotens) und beginnt mit der Datenverarbeitung via Spark. Die Klasse SparkContexter ist ein Singleton. Das bedeutet, dass es nur eine Instanz von SparkContext erstellt wird, auch wenn mehrere Inputknoten in einem Workflow in KNIME benutzt werden. Nachdem ein solcher Knoten ausgeführt wurde und die Verbindung zu einem entfernten Spark Cluster besteht bzw. ein lokaler Cluster gestartet ist, werden die Einstellungen der anderen Inputknoten ignoriert.

4.4 RDD in KNIME

Die Weiterleitung der Daten von einem Knoten zu einem anderen erfolgt in KNIME über die Klasse BufferedTable. Dabei werden einzelne Werte in eigenen Zellen der Tabelle serialisiert und gespeichert. Das verleiht dem Benutzer eine Möglichkeit, die Ergebnisse aller Knoten zu speichern und weiter zu benutzen ohne eventuell lange Berechnungen wiederholen zu müssen. Für einen mit Spark verteilten Datensatz ist es aber inakzeptabel, alle Zwischenergebnisse auf den lokalen Rechner zu holen, um diese im nächsten Knoten wieder zu verteilen. Durch einen solchen Overhead würden alle Vorteile der Performance verloren gehen.

Die in Spark verteilten Daten werden durch eine Instanz der Klasse RDD (JavaRDD oder JavaPairRDD) repräsentiert bzw. referenziert. Diese wird dann vom Knoten zum Knoten weitergereicht. Wird ein Spark Knoten aus den Kategorien Input oder Transformation ausgeführt, steht am Ausgangsport die Instanz der RDD Klasse als Eingabe für den nächsten Spark Knoten zur Verfügung. Die Instanz wird als ein RddValue (bzw. PairRddValue) Objekt in eine einzige Zelle RddCell (bzw. PairRddCell) einer BufferedTable gespeichert. So wird nur die Referenz auf ein RDD serialisiert weitergegeben und nicht die gesamten Daten.

4.5 Java in KNIME

Ganz ohne Programmieren kommt man leider mit den Spark Knoten nicht sehr weit. Die wichtigsten Methoden wie *map()* oder *reduce()* brauchen eine Closure, eine Funktion, die als Parameter übergeben wird. Diese Funktion wird dann auf den Datensatz angewendet. Zum Glück gibt es in KNIME bereits einen Knoten "JavaSnippet", welcher das Schreiben eines Programms in Java zur Datenmanipulation ermöglicht. An dieser Stelle möchte man erwähnen, dass der Autor des Knotens Heiko Hofer eine hervorragende Arbeit geleistet hat. Dieser Knoten bzw. der Source Code des Knotens wurde in diesem Projekt als Grundlage benutzt und soweit modifiziert, dass das Arbeiten auch mit Spark RDD's möglich ist.

Als erstes musste man dem Knoten beibringen, die RDD's als Eingabedaten zu erkennen. Dazu wurden die o.g. Klassen RddValue, PairRddValue, RddCell und PairRddCell erstellt. Die Klasse Type aus dem Java-Paket jsnippet.expression wurde um die neuen Datentypen erweitert. Gleichzeitig wurden sie mit einem Icon versehen, damit der Benutzer die verteilten Daten auch visuell unterscheiden kann. Außerdem wurden entsprechende Konverter benötigt um diese Klassen in richtige JavaRDD oder JavaPairRDD Klassen umzuwandeln. TypeProvider aus dem Java-Paket jsnippet.type verwaltet alle Konverter, die sich im Paket jsnippet.type.data befinden. Hier wurden die neuen Konverter JavaToRddCell, JavaToPairRddCell, RddValueToJava und PairRddValueToJava implementiert. Die Provider Klasse wurde entsprechend um die Referenzen auf die neuen Klassen erweitert.

Der Knoten selbst, sprich die Klassen NodeFactory, NodeModel und NodeView, hat eine Vorschau bekommen, welche wie bei den meisten Spark Knoten die ersten 10 Einträge des Datensatzes am Ausgangsport in einer Tabelle anzeigt. Außerdem wird die Spezifikation der resultierenden Tabelle überprüft und eventuell angepasst. So erhält der nächste Knoten die richtige Information über den Inhalt des Inputs.

Die Klasse JSnippet, die der Benutzer selbst schreiben muss und die in diesem Knoten eigentlich ausgeführt wird, taucht vorgefertigt in einem Java Editor auf. Die Spark Bibliothek wird automatisch zum Klassensuchpfad und kann sofort benutzt werden. Nachdem Kompilieren der Klasse wird eine JAR Datei erstellt, die Spark Executor ausführt. Damit aber die Klasse für Spark ausführbar ist, fehlten zwei Kleinigkeiten. Zum einen musste sie die Serializable Klasse implementieren, damit sie serialisiert an den Spark Executor geschickt werden kann. Zum anderen war es notwendig, dass die Klasse einen eindeutigen Namen hat. Würde ein KNIME Benutzer mehrere JSnippet Knoten erstellen, so könnte Spark diese nicht unterscheiden. Eine Lösung für das Problem schaffte eine eindeutige Nummer, die jeder neue JSnippet Knoten zugewiesen bekommt. So heißen die Klassen z.B. *JSnippet13* mit einer eindeutigen Endzahl und Spark kann somit alle Knoten auseinander halten.

Weitere hilfreiche Funktion des Knotens verbirgt sich in dem Reiter "Templates" des Dialogfensters. Funktionen, die man öfter benutzt, können als Vorlage gespeichert werden. Plugin "Spark4KNIME" beinhaltet bereits Templates für solche RDD Funktionen wie *map()*, *reduce()* oder *filter()* im Ordner namens jsnippets. In der Klasse PluginFileTemplateRepositoryProvider wurde der Ordner als ein Ort definiert, wo KNIME nach den vorhandenen Vorlagen sucht.

5 Spark Knoten

Die nachfolgende Dokumentation der einzelnen Knoten im Plugin ist auch zum Teil in den Knotenbeschreibungen in KNIME vorhanden. Die Programmiertechnische Seite des Projektes findet man im Kapitel 4.

5.1 Input

Kategorie *Input* beinhaltet zwei Knoten, mit denen die eigentliche Spark Berechnung anfängt. Beide Knoten erstellen aus vorhandenen Daten einen RDD oder PairRDD. Diese verteilten Datensätze sind Grundlage für parallele Berechnungen. Außerdem sind es die einzigen Knoten, die Spark Master URL festlegen. In gesamten Workspace gilt die URL, die als erste ausgeführt wird. Alle anderen Knoten benutzen danach immer die gleiche Instanz bis KNIME geschlossen wird. Weitere Information über Spark Master sind in Kapitel 4.3 zu finden.

Inputknoten besitzen eine Vorschau. Sie ermöglicht eine kurze Einsicht in die Daten, die eben verteilt wurden. Es werden die ersten 10 Einträge der RDD in einer Tabelle angezeigt.

FileToRDD erstellt einen verteilten Datensatz aus einer Textdatei. Diese wird zeilenweise eingelesen und in einer einfachen RDD gespeichert. Die Einträge des Datensatzes, der am Outputport des Knotens steht, sind somit immer vom Typ *String*. Im Konfigurationsfenster wird zusätzlich zum Spark Master der Ort der zu behandelten Datei festgelegt.

TableToRDD verteilt die Outputdaten eines vorangegangenes Knotens. Die Typen der Einträge in RDD (nachdem der Knoten ausgeführt wurde) sind logischerweise von den Eingabedaten abhängig. Das Konfigurationsfenster gibt die Möglichkeit, die Spalten, die bearbeitet werden müssen, auszuwählen. Ist es eine Spalte, so entsteht am Ende eine einfache RDD bestehend aus den einzelnen Zellen der Spalte. Aus zwei Spalten wird eine PairRDD (Zeile wird zu einem Schlüssel-Wert Paar) erstellt.

5.2 Output

Mit den Output Knoten speichert man die verteilten Daten. Als Speicherort kann entweder lokaler Rechner oder Hadoop Distributed File System (HDFS) genutzt werden. Dabei muss es ein nicht existierender Ordner angegeben werden. KNIME wird zwar deswegen eine Warnung anzeigen, diese kann aber ignoriert werden. Die beiden Knoten aus der Kategorie Output besitzen auch eine Option zum Überschreiben der Daten. Durch das Aktivieren der Option wird es möglich einen Ordner immer wieder für aktuelle Ausgabe zu benutzen. Daten aus der vorherigen Ausführung werden vorher gelöscht.

Ausgabeformat der Daten ist mit der Hadoop Map-Reduce Ausgabe identisch. Im angegebenen Ordner erscheinen nach der erfolgreichen Ausführung die partiellen Dateien *part-00000*, *part-00001* usw. sowie die *_SUCCESS* Datei.

Eine Vorschau ist für die Outputknoten deaktiviert.

RDDToSequenceFile ist zum Speichern von Daten für weitere Bearbeitung mit Hadoop Map-Reduce bestens geeignet. Die Daten aus RDD werden als Hadoop SequenceFile gespeichert.

RDDToTextFile speichert den RDD in einfache Textdateien unter Einhaltung des o.g. Ausgabeformats.

5.3 Transformations

Transformationen an RDD's erstellen immer eine neue RDD. Diese Operationen sind *lazy*. Die Transformationen werden erst dann berechnet, wenn eine Aktion das Ergebnis erfordert.

Distinct erstellt einen neuen Datensatz, welcher eindeutige ungleiche Elemente des Inputdatensatzes beinhaltet.

GroupByKey ist nur für Schlüssel-Wert-Paare gedacht. Der resultierende Datensatz ist wieder ein PairRDD, die zu jedem Schlüssel alle Werte gruppiert und als neuer Wert speichert.

Intersection Knoten benutzt man zum Erstellen eines neuen RDD, welcher aus der Schnittmenge der Elemente in den gegebenen Datensätze besteht.

Sample erstellt einen neuen Datensatz aus einem Anteil des Eingabedatensatzes als Stichprobe.

SortByKey

Union gibt einen neuen Datensatz zurück, welcher die Vereinigung von den Elementen in zwei gegebenen Datensätzen besitzt.

JavaSnippetForRDD ist der mächtigste Knoten in der Spark Kategorie. Im Prinzip kann er alle hier aufgelisteten Knoten ersetzen indem man ein Java Programm erstellt, das die notwendige Datenverarbeitung übernimmt. Für Spark ist der Knoten unumgänglich. Die wichtigen Methoden für eine Map-Reduce Operation erwarten eine Funktion als Parameter. Diese Funktion wird auf den Datensatz parallel angewendet. Und die beste Methode, solche Funktion zu erstellen, ist durch das Programmieren in Java gegeben.

5.4 Actions

Während eine Transformation *nur* eine Modifikation eines verteilten Datensatzes, ist liefern Aktionen ein Ergebnis einer Berechnung zum Hauptprogramm. Vor der Ausführung einer Aktion werden alle vorangegangenen Transformationen abgearbeitet, damit die Basisdaten für die Aktion bereit liegen. Das Ergebnis einer Aktion ist kein RDD mehr.

Collect holt alle Daten einer RDD zum Hauptprogramm als eine Java Collection. Wenn man mit großen Daten arbeitet, soll man mit dem Knoten vorsichtig umgehen, da der Hauptspeicher schnell voll sein kann. Der Knoten ist ehe für kleine Datensätze und Endergebnisse gedacht.

CountByKey ist nur für PairRDD verfügbar. Das Ergebnis des Knotens ist eine Hashtabelle von Schlüssel-Anzahl-Paaren für jeden Schlüssel.

Count zählt die Einträge eines RDD's.

Take gibt die ersten N Elemente eines Datensatzes zurück.

TakeSample erstellt eine randomisierte Stichprobe aus N Elementen eines Datensatzes mit oder ohne Zurücklegen.

6 Wordcount Anwendungsbeispiel

Eine der qnd bekanntesten Prozeduren ist "word count". Wenn man einen sehr großen Text vor sich hat, kann es sehr lange dauern, die Wörter im Text zu zählen und z.B. die 10 häufigsten davon zu identifizieren. Mit Spark funktioniert es schneller wegen paralleler Berechnung. Die mögliche Zusammenstellung von Knoten in KNIME für die Lösung des Problems ist auf dem Bild 2 zu sehen.

Als erstes muss ein Text parallelisiert werden. Das übernimmt der erste Knoten. Ein Spark Cluster wird dabei erstellt, die Daten (Text) werden zeilenweise eingelesen und als ein RDD gespeichert. Abbildung 3 zeigt die einfache Konfiguration des Knoten.

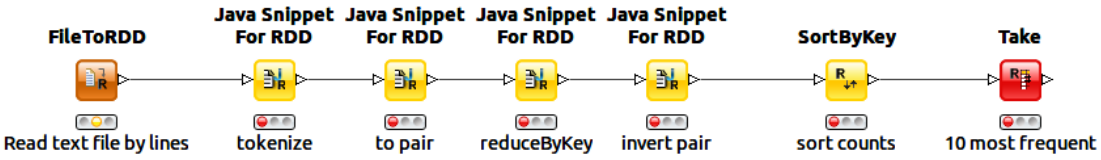


Abbildung 2: Word count mit Spark in KNIME. Ausgabe der 10 häufigsten Wörter eines Textes.

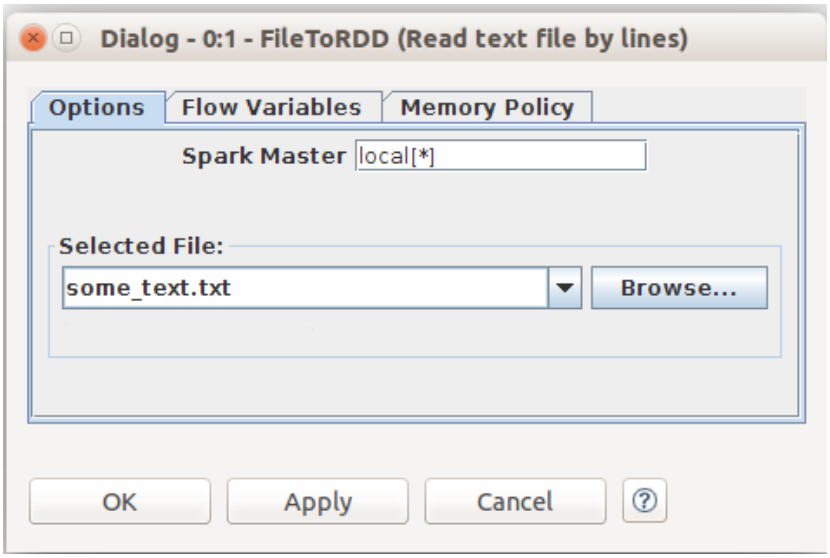


Abbildung 3: Dialogfenster des Knotens FileToRDD. Spark Master und Speicherort einer Textdatei müssen angegeben werden.

Als nächstes werden die eingelesenen Zeilen zu den einzelnen Wörtern zerlegt. Diese Aufgabe meistert die RDD Funktion *flatMap()*. Um die Funktionen zu benutzen, benötigt man den "JavaSnippetForRDD" Knoten. Folgender Java Code veranschaulicht die Aufspaltung der Textzeilen. Ein Filter für Sonderzeichen wäre an dieser Stelle eine hilfreiche Prozedur.

Listing 1: Anwendung der Funktion flatMap() an RDD mit Textzeilen

```
out_RDD = in_RDD.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public Iterable<String> call(String line) throws Exception {
        return Arrays.asList(line.split(" "));
    }
});
```

Die nächsten zwei Knoten sind das Map-Reduce Teil der Prozedur. Jedes Wort wird mit einem Zähler versehen. Daraus entsteht eine Menge von Schlüssel-Wert-Paaren, somit wird weiter mit PairRDD gearbeitet. Programmausdruck 2 zeigt eine mögliche Umwandlung eines RDD's in ein PairRDD. Die Aufzählung der gleichen Wörter erfolgt mittels *reduceByKey()* Methode. Als Parameter wird einfache Summenfunktion übergeben (Listing 3). Die RDD Funktion *countByKey()*, die eigentlich die gleiche Arbeit leistet, ist eine Aktion auf RDD und gibt somit kein RDD mehr zurück. Das Resultat ist eine Liste der (Wort-Zähler)-Paare. Das Sortieren der Liste kann der KNIME Knoten "Sorter" aus der Kategorie "Data Manipulation.Row.Transform" übernehmen.

Listing 2: Map Phase des Wordcounts

```
out_pairRDD = in_RDD.mapToPair(new PairFunction<String, String, Integer>() {
    @Override
    public Tuple2<String, Integer> call(String word) throws Exception {
        return new Tuple2(word, 1);
    }
});
```

Listing 3: Reduce Phase des Wordcounts

```
out_pairRDD = in_pairRDD.reduceByKey(new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer arg0, Integer arg1) throws Exception {
        return arg0 + arg1;
    }
});
```

In der Funktion *reduceByKey()* agieren die Wörter als Schlüssel. Für die Sortierung der Wörter nach ihrer Häufigkeit sollen umgekehrt die Zähler zu Schlüsseln werden. Das erledigt ein einfacher Konverter, der wieder in einem Java Snippet Knoten integriert ist (Listing 4).

Listing 4: Vertauschen der Schlüssel-Wert-Paaren

```
out_pairRDD = in_pairRDD.mapToPair(new PairFunction<Tuple2<String, Integer>,
        Integer, String>() {
    @Override
    public Tuple2<Integer, String> call(Tuple2 arg0) throws Exception {
        return new Tuple2(arg0._2, arg0._1);
    }
});
```

Nun werden die Wörter mit dem Knoten "SortByKey" sortiert. Die Vorschau des Knotens zeigt sofort die ersten 10 Einträge des Resultats. Mit dem Knoten "Take" können die häufigsten (oder die seltensten) Wörter für die weitere Bearbeitung exportiert werden. Auf den Abbildungen 4 und 5 sind die Konfigurationsfenster der beiden Knoten zu sehen.

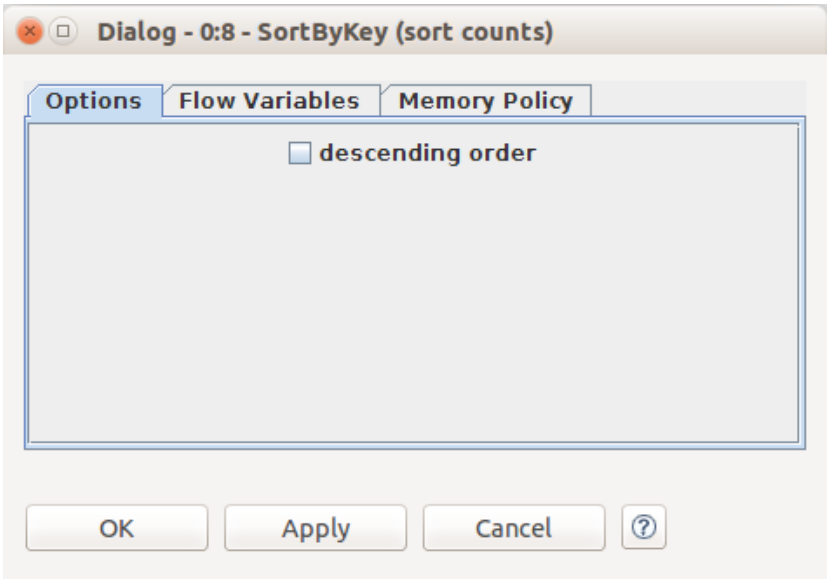


Abbildung 4: Konfiguration des Knotens "SortByKey". Absteigend oder aufsteigende Sortierung sind möglich.

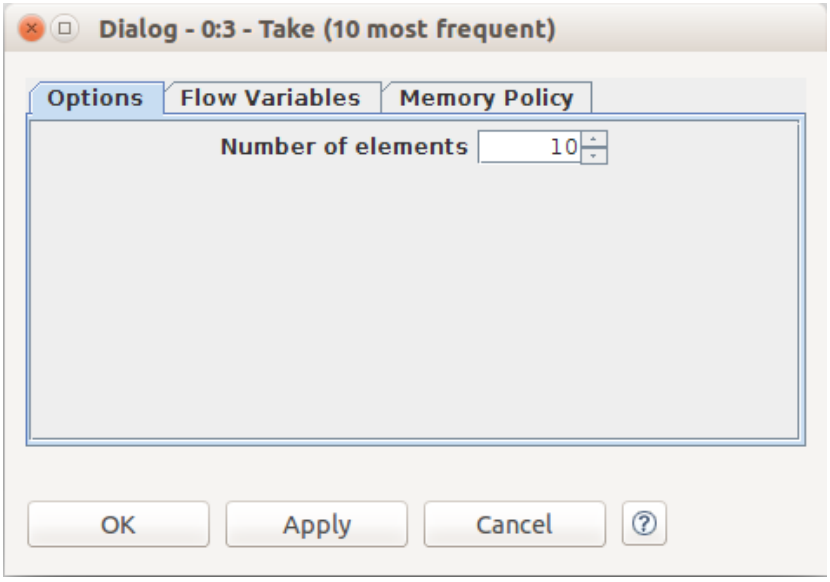


Abbildung 5: Konfiguration des Knotens "Take". Anzahl der zu exportierenden Elemente müssen angegeben werden.

7 Literatur

[MZ12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das Ankur Dave Justin Ma Murphy McCauley Michael J. Franklin Scott Shenker Ion Stoica: *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2–2, Berkeley, CA, USA, 2012. USENIX Association.