FernUniversität in Hagen

-

Seminar 01912

im Sommersemester 2011

„MapReduce und Datenbanken“

Thema 15

Strom- bzw. Onlineverarbeitung mit MapReduce

Referent: Jan Kristof Nidzwetzki

Inhaltsverzeichnis

1 Abstract

Das von Jeffrey Dean und Sanjay Ghemawat [DG04] vorgestellte MapReduce Framework ermöglicht es, große Datenmengen parallel auf mehreren Computern zu verarbeiten. Von seinem Aufbau her, gestattet es das MapReduce Framework zunächst nicht, für die *Strom- bzw. Onlineverarbeitung* eingesetzt zu werden. In dieser Semiararbeit werden drei Arbeiten *Map-Reduce Online* [CCA+09], *DEDUCE: At the Intersection of MapReduce and Stream Processing* [KAGW10] und *Beyond Online Aggregation: Parallel and Incremental Data Minding with Online Map-Reduce* [BAH10] vorgestellt, welche das Framework um diese Fähigkeiten erweitern. Der Schwerpunkt dieser Seminararbeit liegt auf der Arbeit *Map-Reduce Online*.

2 Einleitung

2.1 MapReduce

Durch das von [DG04] entwickelte MapReduce Framework ist es auch mit wenig Kenntnissen im Bereich der Parallelen- und Verteilten-Programmierung möglich, große Datenmengen auf einem Cluster von Computern effektiv zu verarbeiten. Um Tätigkeiten wie die notwendige Parallelisierung, Fehlertoleranz, Verteilung der Daten auf die Systeme und Lastverteilung muss sich der Programmierer nicht kümmern. Diese Aufgaben werden vom MapReduce-Framework übernommen.

2.2 Exkurs: Strom- bzw. Onlineverarbeitung

Unter dem Begriff der *Stromverarbeitung* versteht man das fortlaufende Verarbeiten von *Datenströmen*. Im Gegensatz zu traditionellen Auswertungen, auf statischen Datenbeständen, wird hierbei der Datenbestand fortlaufend neu ausgewertet.

2.2.1 Einsatzbereiche

Der Einsatz der *Stromverarbeitung* ist über all dort interessant, wo sich ändernde Datenbestände zeitnah ausgewertet werden müssen um Entscheidungen zu ermöglichen. Die Einsatzgebiete sind sehr weit gefächert und umfassen unter anderem die folgenden Bereiche:

- Finanzmärke (Auswertung von Aktienkursen)
- Medizintechnik (Überwachung von Körperfunktionen)
- Straßenverkehr (Auswertung von Verkehrsströmen)
- Radioastronomie (Auswertung von Telemetriedaten)
- Logistik (Auswertung von Positionsdaten)

Unter der dem Begriff der *Onlineverarbeitung* wird verstanden, dass die Verarbeitung der Daten im Dialog mit dem Anwender erfolgt. Die Onlineverarbeitung kann beispielsweise dadurch erreicht werden, dass dem Benutzer der Fortschritt seiner Berechnung angezeigt wird oder ihm ein vorläufiges Ergebnis präsentiert wird. Dieser Form der Datenverarbeitung steht die klassische Stapelverarbeitung (Batchverarbeitung) gegenüber.

2.2.2 IBM InfoSphere Streams

Die Firma *IBM* hat für die kontinuierliche Auswertung von Datenströmen das Produkt *IBM InfoSphere Streams* (ehemals *System S*) entwickelt. Die Auswertung der Datenströme wird mittels *Data-Flow Graphen* beschrieben. Ein *Data-Flow Graph* besteht aus einer Menge von *Processing Elements* (*PEs*) welche miteinander verbunden werden. Die *PEs* führen verschiedene Operationen auf den Datenströmen aus. *PEs* können auf verschiedenen Computern ausgeführt werden. Dies ermöglicht es, die Auswertung der Datenströme über mehrere Computer zu verteilen. Für weitere Einzelheiten zu dieser Software siehe [RM] und [BAH10].

Die in dieser Seminararbeit vorgestellte Arbeit *DEDUCE: At the Intersection of MapReduce and Stream Processing* erweitert das Produkt *IBM InfoSphere Streams* um die Fähigkeit, Daten mittels MapReduce auszuwerten.

3 Map-Reduce Online

In der Arbeit *Map-Reduce Online* [CCA+09] wird das Open Source MapReduce-Framework *Hadoop* um die Möglichkeit der *Onlineverarbeitung* (*Online Aggregation*) erweitert. Das klassische Hadoop unterstützt bislang nur die Form der *Batchverarbeitung*. Durch die *Onlineverarbeitung* ist es Anwendern möglich, vorzeitige Ergebnisse (*Early returns*) zu erhalten, während Ihr MapReduce-Job noch ausgeführt wird.

Die Autoren haben im Rahmen dieser Arbeit die Software *Hadoop Online Prototype* (*HOP*) entwickelt. Die Software *HOP* unterstützt neben der *Onlineverarbeitung* ebenfalls *Continuous Queries*, welche es ermöglichen, MapReduce-Programme für Dienste wie Ereignisüberwachung oder die Stromverarbeitung zu nutzen. Dabei behält *HOP* die von MapReduce bekannte Fehlertoleranz bei und erlaubt es, bestehende MapReduce-Programme auszuführen.

Zudem senkt *HOP* die Ausführungszeiten von klassischen MapReduce-Programmen, da sich nun die Map- und die Reduce-Phase überlappen und somit zur Verfügung stehende Ressourcen besser ausgelastet werden können.

3.1 Arbeitsweise des klassischen Hadoop

Das zu Hadoop gehörende Dateisystem *Hadoop Distributed File System* (*HDFS*) wird bei vielen Hadoop-Jobs dazu eingesetzt, die Eingabedaten für die Map-Funktion bereitzustellen. Die meisten Jobs speichern zudem das Ergebnis der Reduce-Funktion wieder im HDFS.

In einer Hadoop-Installation existiert eine *Master-Node*, der so genannte *JobTracker* und mehrere *Worker-Nodes*. Der JobTracker nimmt *Jobs* entgegen, zerteilt diese in einzelne *Tasks* und weist diese Tasks den *Worker-Nodes* zu. Auf den *Worker-Nodes* laufen wiederum ein oder mehrere Map- und Reduce-Tasks.

Eine detaillierte Beschreibung der Architektur von *Hadoop* ist z.B. in [Whi09] zu finden.

3.1.1 Probleme der Onlineverarbeitung

Die Architektur von *Hadoop* sieht es vor, dass die Ausgabe der Map-Funktion und die Ausgabe der Reduce-Tasks erst vollständig vorliegen müssen, bevor diese Daten weiterverarbeitet werden können. Dies vorgehen erlaubt es *Hadoop* mit recht einfachen Mitteln für Fehlertoleranz zu sorgen. Falls ein Map- oder Reduce-Task aufgrund eines Fehlers nicht vollständig ausgeführt werden kann, muss der *JobTracker* von *Hadoop* lediglich den Task erneut starten. Die Ausgabe des fehlgeschlagenen Tasks wird von *Hadoop* verworfen und nur das Ergebnis des neuen Tasks wird zur Weiterverarbeitung genutzt.

Erst wenn es möglich ist, Ergebnisse von Map- oder Reduce-Tasks umgehend zu nutzen, kann eine Online-Verarbeitung der Daten erfolgen. Wie dieses genau durchgeführt werden kann, ohne die Fehlertoleranz zu beeinträchtigen, wird in den kommenden Abschnitten erläutert.

Die Ausgaben eines Map-Tasks bestehen aus einzelnen *Records*. Es existiert in *Hadoop* eine *Partition-Funktion*, welche für jeden *Record* die *Partition* ermittelt, zu der der *Record* gehört. Jedem Reduce-Task wird beim Start eine *Partition* zugewiesen, die er verarbeiten soll. Die *Partition-Funktion* bestimmt somit, wie die einzelnen Records auf die *Reduce-Tasks* aufgeteilt werden.

Die erste Aufgabe eines Reduce-Tasks ist es, alle für seine *Partition* relevante Daten der Map-Tasks einzusammeln. Da diese Daten nicht im *HDFS* vorhanden sind, müssen diese von den entsprechenden Rechnern, auf den der Map-Tasks lief, abgeholt werden[1]. Der *JobTracker* sorgt dafür, dass alle *TaskTracker* bekannt sind, auf welchen Ausgaben eines Map-Tasks liegen.

Der Reduce-Task kann erst mit der Verarbeitung der Daten beginnen, wenn alle Daten für die von ihm zu bearbeitende Partition vorliegen.

3.2 Pipelining

Um die *Strom- bzw. Onlineverarbeitung* zu ermöglichen, wird die Entkopplung der Map- und Reduce-Tasks aufgehoben. Statt es dem Reduce-Tasks zu überlassen, die Daten der Map-Tasks einzusammeln, werden bei *HOP* die Ausgaben der Map-Tasks direkt an die Reduce-Tasks weitergeleitet (*Pipelining*).

Sobald ein Client einen neuen Job übermittelt, sorgt *HOP* dafür, dass jeder Reduce-Tasks umgehend alle Map-Tasks kontaktiert und eine TCP-Verbindung zu ihnen aufbaut. Über diese Verbindung leitet der Map-Task umgehend die von ihm produzierten Daten an den zuständigen Reduce-Task weiter. Der Reduce-Task nimmt diese Daten an und speichert diese in einem Pufferspeicher zwischen. Sobald der Reduce-Tasks darüber benachrichtigt wird, dass alle Map-Tasks beendet sind, führt er, wie im klassischen Hadoop, die Reduce-Funktion aus.

Dieses Design vertraut darauf, dass den gestarteten Jobs sofort von *Hadoop* genügend freie Map- und Reduce-Tasks zur Verfügung gestellt werden können. Zudem wird davon ausgegangen, dass eine beliebige Anzahl von TCP-Verbindungen aufgebaut werden kann.

[1] Dies geschieht bei *Hadoop* durch das Protokoll HTTP.

Beide Annahmen sind in der Praxis nicht immer zu erfüllen. Um das *Pipelining* trotzdem zu ermöglichen, bedient sich *HOP* eines einfachen Tricks: wenn nicht genügend freie Ressourcen zur Verfügung stehen, um alle Reduce-Tasks sofort zu starten, schreiben die Map-Tasks die Daten, wie im klassischen *Hadoop*, auf die lokale Festplatte. Sobald der Reduce-Task ausgeführt werden kann, sammelt dieser die Daten von den Map-Tasks ein.

Um die Anzahl der verwendeten TCP-Verbindungen zu reduzieren, lässt sich die Anzahl der maximalen TCP-Verbindungen der Reduce-Tasks festlegen. Müssen mehr Map-Tasks kontaktiert werden, als dieses Limit zulässt, so baut der Reduce-Task so viele TCP-Verbindungen auf, wie maximal möglich. Die Daten von den restlichen Map-Tasks werden wieder wie im klassischen Hadoop eingesammelt, sobald der Map-Task beendet ist.

3.2.1 Weitere Probleme beim Pipelining

Da die vom Map-Task produzierten Daten sofort an den Reduce-Task übermittelt werden, kann die aus dem klassischen Hadoop bekannte *Combiner-Funktion* nicht angewendet werden. Sie sorgt dafür, dass Ergebnisse des Map-Tasks zusammengefasst werden und weniger Daten zwischen dem Map- und Reduce-Tasks zu übertragen sind.

Zudem werden im klassischen *Hadoop* die Daten für die Reduce-Tasks von den Map-Tasks vorab sortiert. Da in *HOP* die Ausgaben der Map-Tasks direkt an die Reduce-Tasks übermittelt werden, müsste diese Arbeit ebenfalls von den Reduce-Tasks erledigt werden.

Um nicht zu viel Arbeit von den Map-Tasks auf die Reduce-Tasks zu übertragen und um das Netzwerk nicht übermäßig zu belasten, werden die Daten von den Map-Tasks nicht unmittelbar an die Reduce-Tasks übertragen. Stattdessen hält der Map-Tasks eine gewisse Menge an Ausgaben im Speicher. Wenn diese Datenmenge einen Schwellwert übersteigt, wird die *Combiner-Funktion* angewendet und der Speicherinhalt nach *Partition* und *Map-Key* sortiert in eine Datei geschrieben.

In *HOP* wurde der *Task-Tracker* um Funktionen für die Verarbeitung derartiger Dateien erweitert. Zudem wurde das Übertragen der Daten von dem Map-Task in den *Task-Tracker* verlegt. Der Map-Task benachrichtigt lediglich den *Task-Traker*, dass eine neue Datei mit Ausgaben vorhanden ist. Der *Track-Tracker* übermittelt nun die Datei umgehend an den zuständigen Reduce-Task.

Bevor der Map-Task eine neue Datei dem *Task-Tracker* meldet, wird der *Task-Tracker* nach der Anzahl der noch nicht übertragenen Dateien gefragt. Übersteigt das Ergebnis einen Schwellwert, so registriert der Map-Task die Datei vorerst nicht beim *Task-Tracker*. Erst wenn die Anzahl der ungesendeten Dateien unter diesen Schwellwert sinkt, fügt der Map-Task alle angesammelten Dateien zusammen. Die zusammengefassten Daten werden sortiert, der *Combiner-Funktion* unterworfen und das Ergebnis in eine neue Ausgabedatei geschrieben. Lediglich diese eine Datei wird dann dem *Task-Tracker* gemeldet.

Durch dieses Vorgehen wird ein Teil der Arbeitslast zwischen den Map- und Reduce-Tasks dynamisch verteilt. Je nachdem, ob das System für den Map-Task oder Reduce-Task stärker ausgelastet ist, wird dem stärker belasteten System ein Teil der Arbeitslast abgenommen.

3.2.2 Pipelining zwischen verschiedenen MapReduce-Jobs

Viele gängige Berechnungen bestehen aus mehreren hintereinander ausgeführten Map-Reduce Jobs. In der traditionellen Hadoop-Architektur wird das Ergebnis eines Jobs im *HDFS* gespeichert. Sobald dieses Ergebnis vollständig vorliegt, wird es vom nächsten Map-Task als Eingabe verwendet.

Durch *HOP* ist es möglich, dass die Reduce-Tasks ihre Ergebnisse direkt in die Map-Tasks des nächsten Jobs weiterleiten. Das aufwändige Schreiben des Ergebnisses des ersten Jobs ins *HDFS* entfällt somit.

Zu beachten ist jedoch, dass sich der Reduce-Task des ersten Jobs nicht mit dem Map-Task des zweiten Jobs überlappen kann, da das vollständige Ergebnis des ersten Jobs erst nach Beendigung des Reduce-Tasks vorhanden ist. Erst in diesem Moment können die Daten vom nachfolgenden Job gelesen werden. Diese Einschränkung verhindert ein effektives Pipelining zwischen zwei verschiedenen Jobs.

Im Abschnitt 3.3.1 wird beschrieben, wie mittels *Snapshot-Outputs* trotzdem schon vorab die Daten des ersten Jobs vom zweiten Job verarbeitet werden können.

3.2.3 Fehlertoleranz

Auch *HOP* ist in der Lage, mit fehlgeschlagenen Map- und Reduce-Tasks umzugehen. Um einen fehlgeschlagenen Map-Task ausgleichen zu können, führen die Reduce-Tasks Buch darüber, von welchem Map-Task sie welche Daten erhalten haben. Den Reduce-Tasks ist es nur erlaubt, Daten von dem gleichen Map-Task zu kombinieren. Erst wenn der Reduce-Task die Information erhält, dass der Map-Task vollständig durchgeführt worden ist, dürfen die Daten aus diesem Map-Task mit Daten aus andern abgeschlossenen Map-Tasks kombiniert werden. Falls ein Map-Task fehlschlägt, kann der Reduce-Task alle Daten die er bis zum Ausfall von dem Map-Task erhalten hat, vollständig ignorieren. Der *Task-Tracker* muss nun lediglich einen neuen Map-Task starten, der die gleiche Berechnung wiederholt, um den Ausfall zu kompensieren.

Wenn ein Reduce-Task fehlschlägt, wird ein neuer Reduce-Task für die gleiche *Partition* gestartet. Die Map-Tasks müssen nun den neuen Reduce-Task erneut mit Ihren Ausgaben versorgen. Damit der Map-Task die Berechnung nicht vollständig wiederholen muss, speichert er seine Ergebnisse solange zwischen, bis der zugehörige Reduce-Task erfolgreich beendet worden ist.

Diese Art der Fehlertoleranz ist sehr einfach zu implementieren. Jedoch besitzt Sie eine Limitierung. Die Daten der Map-Tasks können erst von den Reduce-Tasks verarbeitet werden, wenn der Map-Task erfolgreich beendet worden ist. Um diese Limitierung zu umgehen wurde das Konzept der *Checkpoints* eingeführt. Ein Map-Task informiert periodisch den *JobTracker* dass dieser an dem *Offset* x in den Eingabedaten angekommen ist. Der *JobTracker* Informiert darauf hin alle Reduce-Tasks, die Daten von diesem Map-Task konsumieren, dass diese alle Daten verarbeiten können, die vor diesem *Offset* liegen. Wenn ein Map-Task fehlschlägt, muss dieser lediglich an dem zuletzt bekannten *Offset* seine Arbeit aufnehmen.

3.3 Online Aggregation

MapReduce wurde entwickelt, um Daten mittels *Batchverarbeitung* zu verarbeiten. Häufig würden Benutzer jedoch gerne MapReduce verwenden, um interaktiv Daten auszuwerten. Derzeit sieht der typische Arbeitsablauf wie folgt aus: ein MapReduce-Job wird gestartet. Sobald das Ergebnis vorliegt, schaut der Benutzer sich diese Daten an, um zu überprüfen, ob diese seinen Vorstellungen entsprechen. Danach wird entweder ein veränderter Job gestartet, falls das Ergebnis nicht den Vorstellungen des Benutzers entspricht oder die Daten werden weiterverarbeitet.

Das klassische *Hadoop* bietet keine Möglichkeiten, Daten interaktiv auszuwerten. Die Ausgabe eines Jobs kann erst angesehen werden, wenn der Job abgeschlossen worden ist. Häufig wünschen sich jedoch Anwender, ein vorläufiges Ergebnis (*Early return*) ansehen zu können. Dieses Ergebnis ist zwar noch nicht sehr genau, häufig genügt es jedoch für die Einschätzung, ob der laufende MapReduce-Job die gewünschten Ergebnisse liefert.

3.3.1 Online Aggregation innerhalb eines MapReduce-Jobs

In *HOP* werden die vom Map-Task produzierten Daten umgehend an die Reduce-Tasks weitergeleitet. Jedoch kann der Reduce-Task erst mit seiner Arbeit anfangen, wenn alle Daten von den Map-Tasks vorliegen. Um die *Online Aggregation* in *HOP* zu ermöglichen, wurden *Snapshots* eingeführt. In einem *Snapshot* sind alle bislang von den Map-Tasks gelieferten Daten enthalten. Der Reduce-Task beginnt seine Arbeit auf diesen Daten und liefert ein vorläufiges Ergebnis, welches im *HDFS* abgelegt wird.

Anwender möchten bei solchen *Snapshots* häufig wissen, wie genau dieser ist. Das Problem, zu ermitteln, wie genau ein solches Zwischenergebnis ist, ist selbst für normale SQL Abfragen nur schwer zu bestimmen. *HOP* gibt daher nur den Fortschritt der laufenden Berechnung an. Eine genauere Betrachtung dieser Fragestellung wird in der Arbeit *Beyond Online Aggregation: Parallel and Incremental Data Minding with Online Map-Reduce* [BAH10] nachgegangen, welche im Abschnitt 4.2 kurz vorgestellt wird.

Der Anwender kann in *HOP* definieren, wie oft ein solcher *Snapshot* berechnet werden soll. So kann er beispielsweise einen *Snapshot* bei 20% , 40%, 60% und 80% Fortschritt eines Jobs anfordern. Darüber hinaus kann der Benutzer angeben, ob nur Daten von vollständig abgeschlossenen Map-Tasks in das Ergebnis einfließen sollen oder ob auch Daten aus noch laufenden Map-Tasks in das Ergebnis aufgenommen werden sollen.

3.3.2 Online Aggregation zwischen mehreren MapReduce-Jobs

Wie bereits beschrieben, erweitert *HOP* das klassische *Hadoop* um die Möglichkeit, Daten aus den Map-Task direkt an den Reduce-Task weiterzuleiten. Diese Erweiterung kann ebenfalls dazu genutzt werden, um die *Online Aggregation* zwischen zwei oder mehr Jobs zu ermöglichen.

Im folgenden Beispiel wird angenommen, dass die zwei Jobs $j1$ und $j2$ ausgeführt werden sollen und $j2$ die Ausgaben von $j1$ als Eingabe ließt. Um die *Online Aggregation* über mehrere Jobs hinweg zu ermöglichen, produziert der Job $j1$ wie im Abschnitt 3.3.1 erläutert regelmäßige *Snapshots*. Der *Snapshot* wird wie bereits beschreiben im *HDFS* abgelegt. Zu-

dem werden die Daten direkt an die Map-Tasks des Jobs $j2$ weitergeleitet. Der Task $j2$ berechnet nun, wie ein normaler Map-Reduce Job, sein Ergebnis. Mit dieser Methode lassen sich auch mehr als zwei Jobs an einander Reihen.

Dieses Verfahren baut jedoch darauf auf, dass alle Berechnungen von $j2$ für jeden *Snapshot* von $j1$ vollständig wiederholt werden müssen. Bereits berechnete Zwischenergebnisse lassen sich bislang nicht weiterverwenden. Eine genaue Untersuchung wie Zwischenergebnisse bei bestimmten Funktionen weiterverwendet werden können, steht bislang noch aus.

3.3.3 Einsatz in der Praxis

Im Rahmen der Entwicklung von *HOP* wurde die Online Aggregation an einem praktischen Beispiel getestet. In 5,5 GB der englischsprachigen Wikipedia sollte ermittelt werden, welches die K häufigsten Wörter sind. Diese Berechnung wurde in der *Amazon Elastic Computer Cloud* (EC2) durchgeführt. Dort ist es möglich, sich für eine gewisse Zeit, eine bestimmte Menge an Ressourcen zu mieten.

Zum Einsatz für diese Berechnung kamen 60 Systeme vom Typ *high-CPU Medium* mit jeweils 1,7 GB Arbeitsspeicher und 2 *virtual Cores*. Ein *virtueller Core* entspricht der Leistung eines 2,5 GHz Intel Xeon Prozessor aus dem Jahre 2007.

Die 5,5 GB an Eingabedaten wurden ins *HDFS* kopiert. Auf diesen Daten wurden zwei MapReduce-Jobs durchgeführt. Der erste Job zählt die Häufigkeit der Wörter und der zweite Job ermittelt die K häufigsten Wörter. In dem Job werden oft *Snapshots* erzeugt und ausgewertet. Ziel ist es zu ermitteln, nach wie vielen Sekunden des Jobs bereits die finalen k häufigsten Wörter feststehen.

Die Ergebnisse dieser Berechnungen sind in Abbildung 1 zu sehen. So ist zu sehen, dass nach ca. 30 Sekunden schon die Top 5 Wörter, nach ca. 50 Sekunden die Top 10 Wörter und nach ca. 80 Sekunden die Top 20 Wörter ermittelt waren.

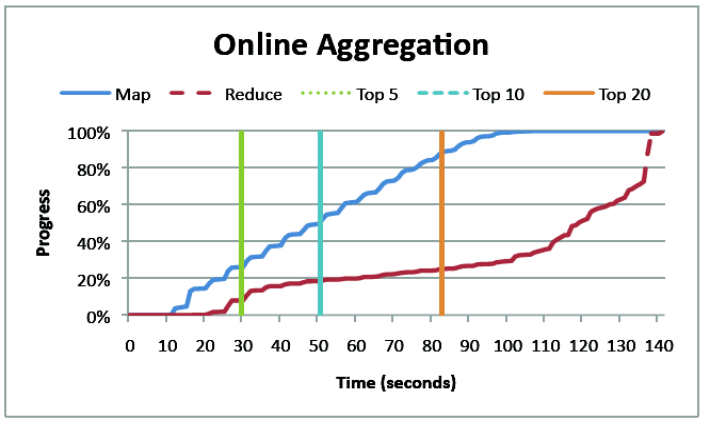

Abbildung 1: Häufigste K Wörter in 5.5 GB Text der englischsprachigen Wikipedia. Quelle: [CCA+09] S. 07

3.4 Continuous Queries

MapReduce wird häufig dazu eingesetzt, Datenströme wie Logdateien auszuwerten. Durch die Architektur von *Hadoop* ist es nur möglich, die Logdateien als Momentaufnahme zu verarbeiten. Veränderungen an den Logdateien können erst mit der nächsten Auswertung mit berücksichtigt werden. Wünschenswert wäre es, derartige Ströme von Eingaben kontinuierlich in Echtzeit auszuwerten.

Bei jeder Auswertung der Logdatei muss zudem die komplette Berechnung wiederholt werden, sofern der Anwender nicht selber dafür sorgt, die Ergebnisse der letzten Berechnung zwischenzuspeichern und mit der nächsten Berechnung wieder zu laden. *HOP* bietet für die *Stromverarbeitung* eine Alternative an. In *HOP* können Jobs fortwährend laufen und neue Daten auswerten, sobald diese verfügbar sind.

Die Umsetzung der *Continuous Queries* für die Stromverarbeitung gestaltet sich durch die bisher implementierten Funktionen einfach. Genau wie bei der *Online Aggregation* werden die Ausgaben der Map-Tasks direkt an die Reduce-Tasks weitergeleitet. Lediglich um eine API-Funktion musste *HOP* erweitert werden. Diese API-Funktion sorgt dafür, dass die Map-Tasks ihr aktuelles Ergebnis unmittelbar an die Reduce-Tasks weiterleiten.

Für die *Stromverarbeitung* wird die Reduce-Funktion periodisch aufgerufen und mit neuen Daten versorgt. Wie oft die Reduce-Funktion aufgerufen wird, hängt von der jeweiligen Anwendung ab. Der Aufruf der Reduce-Funktion kann entweder in einem festen Zeitintervall erfolgen, bei Auftreten eines bestimmten Wertes in der Eingabe der Map-Funktion oder beim erreichen einer bestimmten Anzahl von unverarbeiteten Eingabedaten.

Das Ergebnis der Reduce-Tasks kann genau wie bei der *Online Aggregation* in *HDFS* geschrieben werden. Auch alternative Ausgabemöglichkeiten sind in der Praxis denkbar. Ein Beispiel hierfür ist die im Abschnitt 3.4.2 beschriebene Anwendung.

3.4.1 Fehlertoleranz

In dem vom *Hadoop* implementierten Fehlertoleranz-Modell ist es einfach, mit fehlgeschlagenen Map- oder Reduce-Tasks umzugehen. Im Bereich der *Stromverarbeitung* ist dies jedoch nicht so einfach möglich. Es ist in der Praxis nicht realisierbar, alle von der Map-Funktion erzeugten Daten aufzubewahren und bei einem Fehler erneut auszuwerten.

Viele Reduce-Tasks benötigen keine vollständige Historie, um Ihre Arbeit fortsetzen zu können. So benötigt ein Reduce-Task, der einen gleitenden 30 Sekunden Durchschnitt berechnet, nur die Eingaben der letzten 30 Sekunden um nach einem Ausfall weiterarbeiten zu können.

Um Reduce-Tasks in Ihrer Fehlertoleranz zu unterstützen, wurde in *HOP* der *JobTracker* erweitert. Dieser speichert nun, welche Eingabedaten von den Reduce-Tasks verarbeitet worden sind und wie lange diese Daten vorgehalten werden müssen. Sobald bestimmte Daten nicht mehr vorgehalten werden müssen, informiert der *JobTracker* die Map-Tasks, dass diese Daten entfernt werden können. In unserem Beispiel mit der Berechnung des gleitenden Durchschnitts müssen nur Daten der letzten 30 Sekunden vorgehalten werden. Ältere Daten sind für die Berechnung nicht relevant.

Es gibt jedoch auch Berechnungen, die eine vollständige Historie der Eingabedaten benötigen, um nach einem Fehler korrekt weiterarbeiten zu können. Diese Map-Tasks müssen derzeit Ihren eigenen Zustand im *HDFS* speichern und bei einem Neustart diesen Zustand korrekt wieder einlesen. *HOP* könnte prinzipiell derartige Aufgaben unterstützen. Derartige Funktionen sind jedoch bislang noch nicht in *HOP* implementiert.

3.4.2 Beispielanwendung: Aufbau eines Monitoring Systems

Im Rahmen dieser Arbeit wurde die Nutzung der *Stromverarbeitung* im praktischen Einsatz erprobt. Für einen HOP-Cluster wurde ein Monitoring System entworfen, welches sich die *Stromverarbeitung* von *HOP* zu nutze macht. Auf jedem System in diesem Cluster kommt ein *Agent* zum Einsatz, welcher einen Map-Task darstellt, der die Eingabedaten für das Monitoring System liefert. Als Eingabedaten werden Informationen über die Speicher- und CPU-Auslastung, IO-Operationen, etc. eingelesen.

Alle *Agents* liefern Ihre Ergebnisse an einen *Aggregator*, welcher als Reduce-Tasks implementiert worden ist. Dieser vergleicht die Systemlast der letzten 20 Sekunden mit der Systemlast des gesamten Clusters in den letzten 120 Sekunden. Weichen die Werte eines *Agents* um mehr als zwei Standardabweichungen vom Durchschnitt des gesamten Systems ab, so wird ein Alarm erzeugt.

Das ganze System wurde wieder auf einem Amazon *EC2* Cluster mit 7 Systemen eingesetzt. Auf diesen Systemen wurde ein MapReduce-Job ausgeführt, welcher auf 5,5 GB an Daten der Wikipedia die Anzahl der Wörter zählt. 10 Sekunden nachdem dieser Job gestartet worden ist, wurde auf einem der Systeme ein Programm gestartet, was die Systemlast deutlich ansteigen ließ. Das Monitoring System meldete diesen Ausreißer in der Auslastung nach weniger als 5 Sekunden.

3.5 Performance von Map-Reduce Online

In einem MapReduce-Job stellt die Map-Phase den größten Teil der Arbeit dar. Die Map-Funktion wird auf alle Eingabedaten angewendet. Danach werden die Ausgaben der Map-Funktion sortiert und dem *TaskTracker* gemeldet.

Die Reduce-Funktion besteht aus drei Phasen. In der ersten Phase *Shuffle* werden die Daten des Map-Tasks eingesammelt und sortiert. In der *Reduce* Phase werden die Daten durch die Reduce-Funktion verarbeitet und in der anschließenden *Commit* Phase ausgegeben. 75% der gesamten Arbeit des Map-Tasks entfallen auf die *Shuffle* Phase. Die restlichen 25% entfallen auf die *Reduce* und *Commit* Phase.

Durch das in HOP eingeführte *Pipelining* zwischen den Map- und den Reduce-Tasks ist es dem Reduce Task möglich, die eintreffenden Daten kurz nach Ihrem eintreffen zu sortieren. Wenn der letzte Map-Task seine Arbeit abgeschlossen hat und die Daten an den Reduce-Task übermittelt worden sind, muss dieser die Daten noch ein letztes mal sortieren und kann danach in die *Reduce* Phase eintreten. Im klassischen Hadoop werden die Daten erst dann sortiert, wenn alle Daten der Map-Tasks vorliegen.

3.5.1 Performance-Tests

Als Performance-Test wurde wieder ein MapReduce-Job auf einem *Amazon EC2 Cluster* ausgeführt. Zum Einsatz kamen, bei diesem Test, 10 Systeme mit jeweils 16 GB Arbeitsspeicher und vier virtuellen Cores. Als MapReduce-Job wurde ein Wordcount über 10 GB an Eingabedaten ausgeführt. Einmal kam *Hadoop* zum Einsatz, das andere mal wurde *HOP* mit Pipelining eingesetzt. Der Job wurde jeweils zwei mal ausgeführt. Einmal mit 20 Map-Tasks und 5 Reduce-Tasks und einmal mit 20 Map-Tasks und ebenfalls 20 Reduce-Tasks. Die Ergebnisse sind in den Abbildungen 2 und 3 dargestellt.

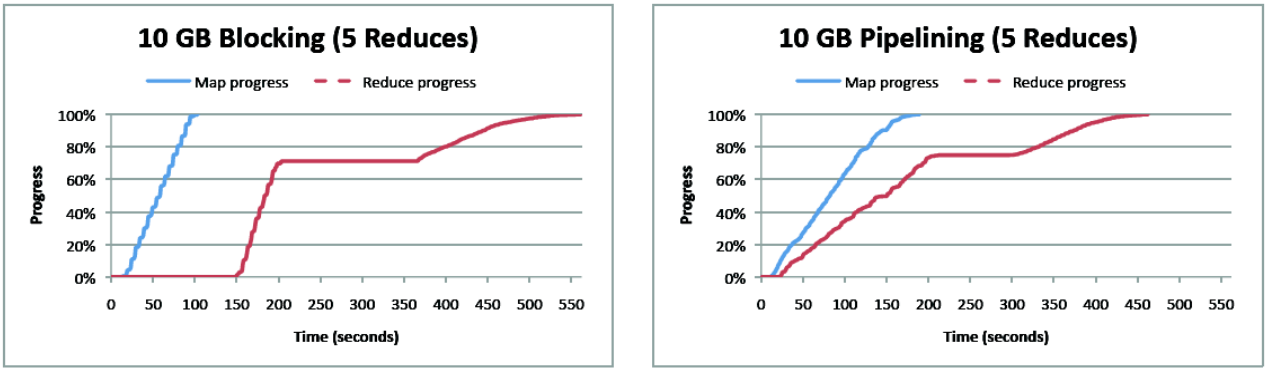

Abbildung 2: Wordcount über 10 GB mit 20 Map-Tasks und 5 Reduce-Tasks. Links: Hadoop ohne Pipelining (551 Sekunden), Rechts: HOP mit Pipelining (462 Sekunden). Quelle: [CCA+09] S. 11

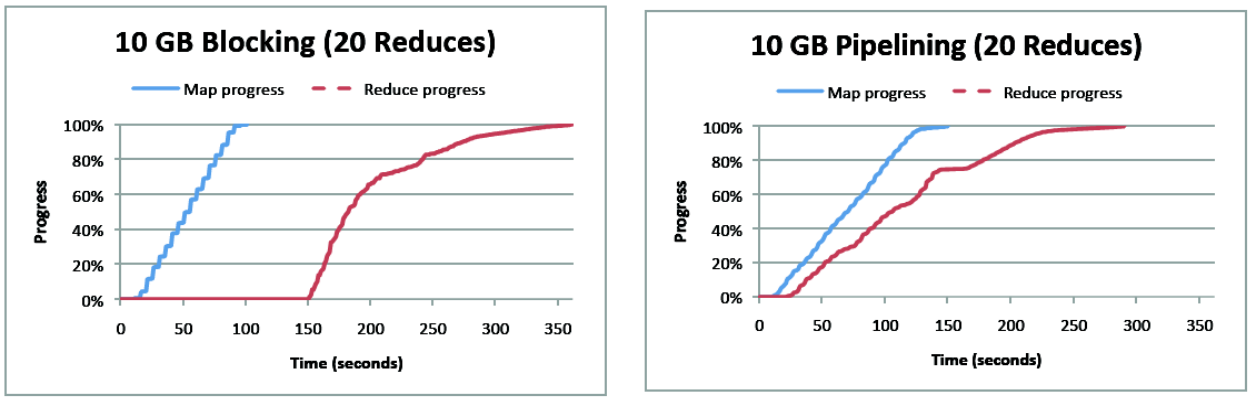

Abbildung 3: Wordcount über 10 GB mit 20 Map-Tasks und 20 Reduce-Tasks. Links: Hadoop ohne Pipelining (361 Sekunden), Rechts: HOP mit Pipelining (290 Sekunden). Quelle: [CCA+09] S. 11

Sowohl beim klassischen *Hadoop* als auch bei *HOP* ist im ersten Test, bei der Ausführung des Reduce-Taks, nach 75% eine gewisse Zeit zu erkennen, in der der Job scheinbar nicht fortschreitet. Der Map-Tasks hat zu diesem Zeitpunkt alle Daten eingesammelt und sortiert diese, bevor die *Reduce* Phase gestartet wird. Es ist jedoch deutlich zu erkennen, dass durch das *Pipelining* das sortieren schneller abgeschlossen werden kann, da die Daten schon nach

dem Eintreffen vorsortiert worden sind. Der Job welcher *Pipelining* nutzen konnte, kann somit schneller abgeschlossen werden.

Der gleiche MapReduce-Job wurde nochmals laufen gelassen. Dieses mal jedoch mit 20 Map- und 20 Reduce-Tasks. Die Ergebnisse sind in der Abbildung 3 zu sehen. Durch die größere Anzahl von Reduce-Tasks ist die Datenmenge für jeden Reduce-Tasks geringer, die verarbeitet werden muss. Auch in diesem Setup kann der Job, welcher *Pipelining* nutzen konnte schneller abgeschlossen werden.

Die Autoren der Arbeit haben noch weitere vergleiche zwischen dem klassischen *Hadoop* und *HOP* mit *Pipelining* durchgeführt. Diese Tests haben ergeben, dass durch *Pipelining* die Auslastung auf den Systemen gesteigert wird und somit die MapReduce Jobs schneller abgeschlossen werden können. Somit ist *HOP* auch für den Einsatz mit klassischen MapReduce-Jobs interessant.

4 Weitere Arbeiten

Auch andere Gruppen haben sich mit der Nutzung von MapReduce zur *Strom- bzw. Onlineverarbeitung* auseinandergesetzt. Zwei weitere Arbeiten werden in diesem Abschnitt kurz vorgestellt. In dem Abschnitt 5 werden alle drei Arbeiten miteinander verglichen.

4.1 DEDUCE: At the Intersection of MapReduce and Stream Processing

Die Arbeit *DEDUCE: At the Intersection of MapReduce and Stream Processing* [KAGW10] erweitert das von IBM angebotene und Abschnitt 2.2.2 vorgestellte System *IBM InfoSphere Streams* (*ehemals System S*) um die Möglichkeit, Daten mittels MapReduce zu verarbeiten.

4.1.1 SPADE

SPADE die *Stream Processing Application Declarative Engine* stellt im IBM *System S* die verwendete Sprache zum beschreiben einer Anwendung dar. *SPADE* selber besitzt die Möglichkeit, durch *UBOPs - user-defined build-in operators* erweitert zu werden.

Diese Fähigkeit macht sich *DEDUCE* zu nutze. *DEDUCE* führt einen MapReduce-Operator ein, welcher als Eingabe eine Liste von Dateien und Verzeichnissen erwartet und als Ausgabe die im *System S* enthaltenen *stream processing operators* mit Daten versorgt. Dieser Operator erlaubt es zudem, kaskadiert eingesetzt zu werden um auf diese Weise die Ausgabe eines MapReduce-Jobs als Eingabe eines neuen MapReduce-Jobs zu verwenden. Hierbei ist zu beachten, dass der DEDUCE selber nicht für die *Strom- bzw. Onlineverarbeitung* eingesetzt wird, sondern lediglich als Datenlieferant eingesetzt wird.

Die Daten für diesen MapReduce-Opreator werden aus dem schon bekannten *Hadoop Distributed Filesystem - HDFS* gelesen. An einer Unterstützung für die OpenSource Implementation des Google Filesystems *Kosmos File System - KFS* wird derzeit gearbeitet.

4.1.2 Eine Beispielanwendung

Die Autoren von *DEDUCE* beschreiben in Ihrer Arbeit eine Beispielanwendung für den MapReduce-Operator. Diese Anwendung wird dazu eingesetzt, Aktienmärkte auszuwerten und Aktien mit bestimmten Kriterien zu ermitteln. Die *Stromverarbeitung* des *System S* wird dazu eingesetzt, die sich ständig verändernden Daten auf einem Aktienmarkt wie Preis und Umsatz auszuwerten. Diese Daten werden mit aktuellen Nachrichten kombiniert.

Für die Auswertung dieser Nachrichten, wie Analysen und Ad-Hoc-Meldungen[2], wird ein MapReduce-Job eingesetzt. Dieser wertet periodisch die leicht mehrere Gigabyte umfassenden Daten aus und stellt das Ergebnis für die weitere Analyse zur Verfügung. Ein Schema dieser Anwendung ist in der Abbildung 4 dargestellt.

In diesem Schema sind drei wichtige Komponenten zu erkennen. Unten links ist der MapReduce-Job dargestellt. Dieser wertet periodisch Daten aus externen Quellen aus. Das Ergebnis wird durch einen *ModelReader* gelesen, welcher diese Daten aus dem Dateisystem ließt zur weiteren Verarbeitung im *System S* bereitstellt. In dem Schema ist zudem ein *Normalizer* zu sehen. Dieser ist dafür zuständig, die aus der *Stromverarbeitung* stammenden Ergebnisse mit den Daten des MapReduce-Jobs zu kombinieren.

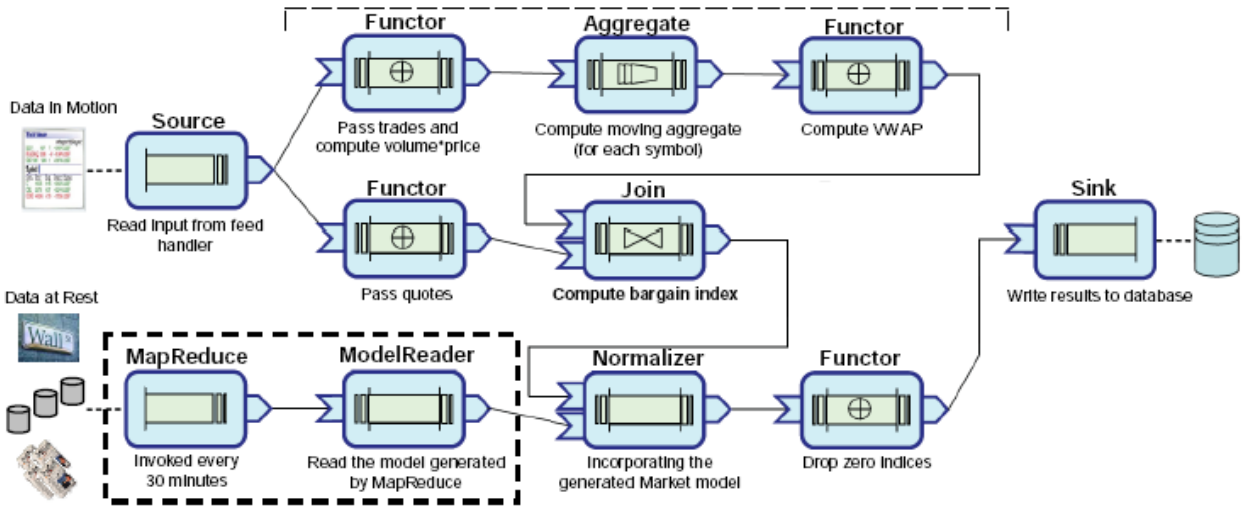

Abbildung 4: Schematische Darstellung einer Anwendung in System S mit MapReduce-Operator. Quelle: [KAGW10] S. 5

4.2 Beyond Online Aggregation: Parallel and Incremental Data Minding with Online Map-Reduce

In der Arbeit *Beyond Online Aggregation: Parallel and Incremental Data Minding with Online Map-Reduce* [BAH10] wird ein MapReduce-Framework zur *Strom bzw. Onlineverarbeitung* vorgestellt, um Probleme wie die Angabe der Genauigkeit einer vorläufigen Berechnung oder die Ermittlung der Laufzeit des gesamten Jobs untersuchen zu können.

[2] § 15 WpHG verpflichtet börsennotierte Unternehmen zur sofortigen Veröffentlichung von Nachrichten, die den Aktienkurs erheblich beeinflussen können. Diese Nachrichten werden als Ad-Hoc-Meldung bezeichnet.

4.2.1 Architektur des MapReduce Framework

Das von den Autoren entwickelte MapReduce-Framework basiert auf einer *Shared-Memory Architektur* und ist als Testumgebung zur Untersuchung der oben genannten Fragestellungen gedacht. Die Daten werden in diesem Framework lediglich im Arbeitsspeicher gehalten. Dies ist der Grund dafür, dass diese Implementation nicht in einem Cluster von Computern eingesetzt werden kann und nur ebenfalls nur eine sehr geringe *Fehlertoleranz* aufweist. Die Autoren sehen diese Implementation als reine Testumgebung an und empfehlen für den produktiven Einsatz das in Abschnitt 3 vorgestellte *Hadoop Online Prototype - HOP*.

4.2.2 Ermittlung des Job-Fortschritts und der Genauigkeit

Es wird vorgeschlagen, dass der Entwickler eines MapReduce-Jobs bestimmte Methoden implementiert, welche es dem MapReduce-Framework ermöglichen, Aussagen über den die Genauigkeit einer vorläufigen Berechnung oder die noch zu erwartende Laufzeit zu geben. Diese Methoden müssen vom jeweiligen Entwickler implementiert werden, da jeder Job unterschiedliche Charakteristiken aufweist und ohne diese Angabe keine genaue Ermittlung dieser Werte möglich wäre.

Das vorgestellte MapReduce-Framework ist in der Lage *convergence curves* zu zeichnen, welche es dem Benutzer erlauben, die Genauigkeit eines vorläufigen Ergebnisses abschätzen zu können. Eine *convergence curve* stellt einen Graphen dar. Auf der X-Achse wird der Fortschritt des Jobs in dem Intervall von [0,1] angegeben auf der Y-Achse hingegen der Wert der $dif()$-Funktion.

Sobald das MapReduce-Framework ein Zwischenergebnis eines Jobs berechnet hat, wird zusätzlich die Signatur $sig()$ berechnet. Dabei handelt es sich um eine durch den Entwickler des MapReduce-Jobs bereitgestellte Funktion, die (sofern möglich) alle für das Ergebnis relevanten Informationen zusammenfasst. Wenn beispielsweise die häufigsten K-Wörter in einem Text ermittelt werden sollen, würde die Funktion genau diese Wörter zurück liefern. Wenn hingegen die Anzahl der Wörter ermittelt werden soll, liefert die Funktion nur die bislang gezählten Wörter zurück. Diese Funktion wurde eingeführt, um eine speichersparende Repräsentation des Zwischenergebnisses zu erhalten.

Die Funktion $dif()$ nimmt als Parameter zwei derartige Signaturen entgegen und berechnet den Abstand dieser beiden Signaturen. Um eine *convergence curve* zu Zeichnen, wird der Abstand aller ermittelten Zwischenergebnisse mit dem letzten berechneten Ergebnis ermittelt und die Differenz als Graph dargestellt.

In der Abbildung 5 ist eine *convergence curve* für einen MapReduce-Job dargestellt, welcher die Wörter eines Textes zählt. Man sieht, dass die Abweichung eines vorläufigen Ergebnisses sehr stark abnimmt.

5 Vergleich der vorgestellten Arbeiten

In der Arbeit *DEDUCE: At the Intersection of MapReduce and Stream Processing* wird das vom IBM entwickelte *System S* um die Möglichkeit ergänzt, Daten mittels MapReduce zu verarbeiten. Die im *System S* enthaltene Beschreibungssprache *SPADE* wird um

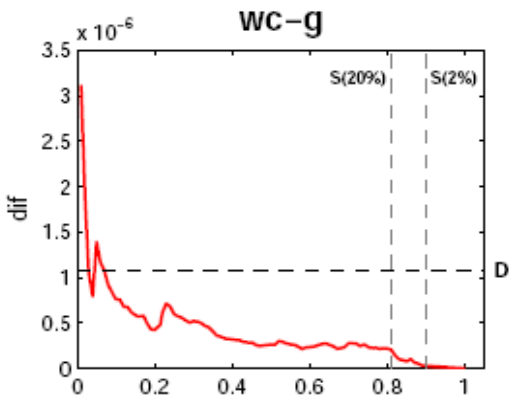

Abbildung 5: Wordcount über den Text der englischen Bücher auf Gutenberg Books (http://www.gutenberg.org). Quelle: [BAH10] S. 5

einen MapReduce-Operator erweitert. Dieser eignet sich dafür, bestehende Daten periodisch auszuwerten und die Ergebnisse an die Stromverarbeitungs-Operatoren weiterzuleiten. Eine *Strom- bzw. Onlineverarbeitung* mittels MapReduce findet jedoch nicht statt.

Die Arbeit *Beyond Online Aggregation: Parallel and Incremental Data Minding with Online Map-Reduce* hingegen stellt ein MapReduce-Framework vor, welches um Aspekte wie Fehlertoleranz oder Skalierbarkeit reduziert worden ist, sich jedoch gut zur Evaluation von Erweiterungen eignet. Die Autoren dieses Systems nutzen es, um die Genauigkeit von vorläufigen Ergebnissen (*Online Aggregation / Snapshots*) zu Berechnen sowie Aussagen über den Berechnungsfortschritt zu geben.

Die Arbeit *Map-Reduce Online* beschreibt eine Erweiterung der Software *Hadoop*. Diese Erweiterung mit dem Namen *Hadoop Online Prototype* (*HOP*) erweitert *Hadoop* um die Möglichkeiten der *Strom bzw. Onlineverarbeitung*.

HOP ermöglicht es, Daten aus Map-Tasks schnell an Reduce-Tasks weiterzuleiten. Hierdurch werden Jobs schneller ausgeführt, da sich nun die Bearbeitung der Map- und Reduce-Tasks überlappen kann. Zudem führt *HOP* das Konzept der *Snapshots* ein. Hierbei handelt es sich um vorzeitige Ergebnisse eines Jobs. Diese erlauben es, MapReduce-Jobs auch in der Online-Verarbeitung einzusetzen. Darüber hinaus werden *Continuous Queries* eingeführt, welche *HOP* die Fähigkeit geben, Datenströme auszuwerten. All diese Erweiterungen bewahren die von *Hadoop* bekannte Fehlertoleranz.

Literatur

[BAH10] Joos-Hendrik Böse, Artur Andrzejak, and Mikael Högqvist. Beyond online aggregation: parallel and incremental data mining with online map-reduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, MDAC '10, pages 3:1–3:6, New York, NY, USA, 2010. ACM.

[CCA+09] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[KAGW10] Vibhore Kumar, Henrique Andrade, Bugra Gedik, and Kun-Lung Wu. Deduce: at the intersection of mapreduce and stream processing. In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Léger, Felix Naumann, Anastasia Ailamaki, and Fatma Özcan, editors, *EDBT*, volume 426 of *ACM International Conference Proceeding Series*, pages 657–662. ACM, 2010.

[RM] Roger Rea and Krishna Mamidipaka, editors. *IBM InfoSphere Streams - Redefining Real Time Analytics*. IBM Software Group.

[Whi09] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, first edition edition, june 2009.