Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Gerrit Thede

Big and Fast Data - Verarbeitung von Streaming Data

Grundlagen Vertiefung und Anwendungen 2

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Gerrit Thede

Big and Fast Data - Verarbeitung von Streaming Data

Grundlagen Vertiefung und Anwendungen 2

Eingereicht am: 29.08.2014

1 Big and Fast Data - Verarbeitung von Streaming Data

Neue Systeme für die Verarbeitung von „Big and Fast Data“ werden benötigt, da sich die Verarbeitung von Big Data mit dem MapReduce Ansatz bewährt hat, aber nicht für den Einsatz von schnell eintreffenden Streaming Daten geeignet ist. Hoch skalierbare Big Data Systeme haben die Eigenschaft, dass sie auf Knotenrechnern mit Standardhardware verteilt werden können. Als Basis hat sich das Hadoop Filesystem als de facto Standard entwickelt, das die zu verarbeitenden Daten und die Ergebnisse persistent verteilt speichern kann. Zur Verarbeitung der Daten wird häufig der MapReduce Ansatz verwendet, der in der Map Phase Daten auswertet und in der Reduce Phase die Ergebnisse zusammenfasst. Das System sorgt dafür, dass die Arbeitslast auf ein Cluster von Knoten verteilt wird, überwacht dabei die Ausführung und behandelt Ausfälle. In diesem Ansatz werden Offline-Daten verarbeitet, die vor der Verarbeitung im Hadoop Filesystem gespeichert werden müssen.

Dieser Ansatz eignet sich nicht für die Verarbeitung von schnell eintreffenden neuen Daten. Die Verarbeitung von einem neuen erweiterten Datenbestand muss den gesamten Datensatz neu auswerten, was zu erhöhten Antwortzeiten führt.

Ein „Big and Fast Data“-System soll eine kontinuierliche Verarbeitung von Datenströmen ermöglichen und dabei die Daten inkrementell auswerten und geringe Antwortlatenzen bieten.

In verschiedenen Ansätzen wird versucht, diese Anforderungen zu erfüllen. Ich möchte drei dieser Konzepte vorstellen. Die Ansätze haben die Gemeinsamkeit, dass MapReduce als grundlegender Ansatz verwendet wird und alte Anwendungen prinzipiell auf die neuen Systeme übertragbar sind.

2 Incoop: MapReduce for Incremental Computations

Das Incoop System [1] wurde auf dem ACM Symposium on Cloud Computing 2011 vorgestellt. Incoop beruht auf einer Erweiterung des Hadoop Frameworks und bietet ein MapReduce Framework, das inkrementell veränderte Daten effizient verarbeiten kann. Die Zielsetzung war, ein System zu erschaffen, das die MapReduce API beibehält und transparent für den Entwickler die höhere Komplexität der Algorithmen im System behandelt. Um diese Ziele zu erreichen wurden Änderungen am Hadoop Filesystem HDFS vorgenommen, eine Datenbankkomponente erschaf-

fen, die die Wiederverwendung von Zwischenergebnissen ermöglicht und ein inkrementelles MapReduce Verfahren implementiert.

Der Entwurf verwendet die Prinzipien von selbst-regulierenden Berechnungen, bei denen automatisch auf Veränderungen in den Eingabedaten reagiert wird. Dies wird erreicht, durch die Erstellung eines Abhängigkeitsgraphs zwischen Daten, Berechnungen und Folgeberechnungen. Ändern sich Teile der Eingabedaten zeigt der Graph an, welche Berechnungen erneut ausgeführt werden müssen.

Eine Anpassung des Dateisystems für eine stabile Aufteilung der Daten und eine feinere Aufteilung der MapReduce Tasks sorgen für eine hohe Wiederverwertbarkeit von Zwischenergebnissen.

Es wird gezeigt, dass bestehende MapReduce Programme für nicht inkrementelle Verarbeitung auf dem Incoop System ohne Änderung ausgeführt werden können und dabei eine Effizienssteigerung erlangen.

Anwendungsfälle sind die z.B. die Verarbeitung von inkrementell wachsenden Serverlogdateien.

Erweiterungen des Hadoop Frameworks

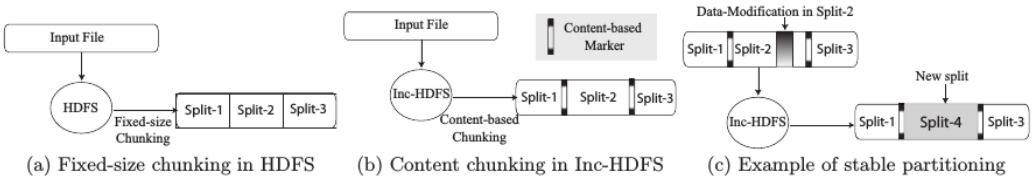
Incremental HDFS Das verwendete Dateisystem Inc-HDFS ist eine zu HDFS kompatible Modifikation, die beim Speichern von Input Dateien diese analysiert und gleiche Abschnitte des Inputs auch an gleicher Stelle gespeichert werden.

In HDFS werden große Input-Dateien beim Speichern in Blöcke fester Größe (z.B. 64MB) geteilt und wenn möglich auf verschiedenen Knoten gespeichert.

Dieses Verfahren eignet sich für Incoop nicht, denn ein kleiner Unterschied in der Eingabedatei kann dazu führen, dass sich die Blockgrenzen verschieben und der Inhalt der Blöcke unterschiedlich ist. Bei der Verarbeitung eines Blocks durch die Map Funktion wird dann ein unterschiedliches Ergebnis entstehen, dass nicht wiederverwendet werden kann.

Incoop bedient sich eines Verfahrens, das mit einer rollenden Hashfunktion (Rabin Fingerprint [5]) den Inhalt der Datei analysiert und die Blockgrenzen setzt, wenn ein bestimmtes Muster des Hashwerts vorliegt. Für jeden Block wird sein Hashwert gespeichert und wenn eine veränderte Eingabedatei einen unterschiedlichen Block enthält, muss nur dieser neue gespeichert werden.

Das Bild 1 zeigt die unterschiedlichen Teilungs-Strategien der beiden Dateisysteme.



(a) Fixed-size chunking in HDFS (b) Content chunking in Inc-HDFS (c) Example of stable partitioning

Abbildung 1: File Chunking Strategie [1]

Memoization Server Der Memoization Server ist die Komponente, die die Ergebnisse jedes ausgeführten Tasks referenziert. Er speichert einen Verweis vom Hash der Eingabedaten auf den persistenten Speicherort der Ergebnisdaten. Während eines Durchlaufs wird bei jeder Ausführung eines Tasks zuerst der Memoization Server nach einem bereits existierenden Ergebnis für die Eingabedaten gefragt um diese wiederzuverwenden. Da Map und Reduce deterministische Funktionen sind, liefern sie das selbe Ergebnis bei gleichem Input. Der Server verfügt über eine Garbage Collection und verwirft periodisch alte Einträge und entfernt auch die Daten aus dem persistenten Speicher.

Incremental MapReduce Der Unterschied in der Map Phase von Incoop zum klassischen MapReduce liegt darin, dass die Ergebnisse persistent gespeichert und als Referenz im Memoization Server abgelegt, anstatt verworfen zu werden, nachdem der Map Task beendet ist. Die Map Phase hat bei inkrementellen Ausführungen den Vorteil, dass Ergebnisse bereits im Memoization Server abgelegt sein können.

In der Incremental Reduce Phase werden die Ergebnisse der Map Tasks nach gleichen Keys der erzeugten Key-Value Paare gruppiert. Jeder Reduce Task holt sich alle Key-Value Paare, für die er zuständig ist und wendet die Reduce Funktion darauf an. Die Eingabedaten der Reduce Funktion bestehen also aus den Ergebnissen vieler Map Tasks. Es wird zunächst ein kollisionsresistenter Hash über die Map Ergebnisse gebildet und der Memoization Server gefragt, ob ein Ergebnis für diese Eingabedaten bereits gespeichert ist. Dabei zeigt sich ein Problem: Wenn auch nur ein Map Ergebnis nicht gefunden wird, muss die gesamte Reduce Funktion berechnet wer-

den. Um dies zu optimieren, wird eine sogenannte „Contraction Phase“eingeführt, in der die Granularität der Reduce Ergebnisse verkleinert wird.

Dazu wird die Combiner Funktion vom Original MapReduce Framework zweckentfremdet: Sie sorgt dafür, dass Übertragungsbandbreite gespart wird, indem bereits im Map Task auf dem ausführenden Knoten eine Zusammenfassung von Key-Value Paaren mit der Reduce Funktion erfolgt. So werden die eigentlichen Reduce Tasks kleiner und es erhöht sich die Wahrscheinlichkeit, Ergebnisse wiederzuverwenden (siehe Abb. 2).

Memoization aware Scheduler Der Hadoop Scheduler berücksichtigt für effiziente Ausführung die Verfügbarkeit, Cluster Topologie und Datenlokalität des Inputs. Damit auch die Lokalität der zwischengespeicherten Ergebnisse berücksichtigt wird, verwaltet der für Incoop angepasste Scheduler Taskqueues für jeden Knoten statt einer Queue für alle Knoten. So kann ein Task auf dem Knoten ausgeführt werden, auf dem die gespeicherten Ergebnisse liegen. Falls so die Lastverteilung ungünstig wird, reagiert der Scheduler und verlegt Tasks auf andere Knoten und ignoriert die Lokalität.

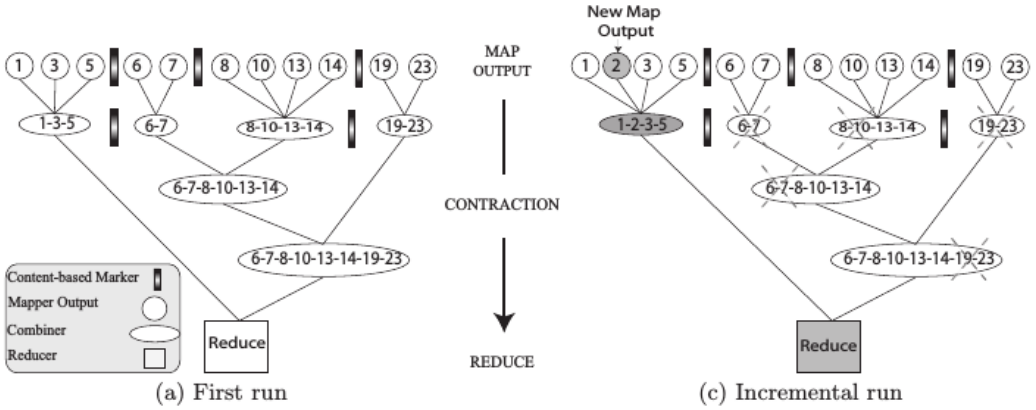


Abbildung 2: Incremental Reduce mit Contraction Phase [1]

Incoop Ergebnisse

Die Verfasser analysieren den Overhead des entwickelten Systems in Bezug auf MapReduce mit Hadoop. Es wird unterschieden zwischen dem Overhead des ers-

ten Durchlaufs und dem Overhead der folgenden Durchläufe nach einem dynamischen Update der Eingabedaten. Es zeigt sich, dass im initialen Lauf der Overhead durch die benötigte Zeit für die Hashfunktion und die Zeit für die Kommunikation mit dem Memoization Server linear erhöht wird. Speicheroverhead entsteht durch die zusätzliche Speicherung von Zwischenergebnissen. Bei einem dynamischen Update wird nur die Zeit benötigt, die das Hashen und Anfragen an den Memoization Server benötigt zusätzlich zu der Anzahl der neu zu berechnenden Tasks. Bei einem geringen Anteil an veränderten Daten in der Eingabedatei kann so eine Beschleunigung erwartet werden.

Die Implementierung wurde mit dem Einsatz von verschiedenen datenintensiven und berechnungsintensiven Programmen getestet. Dabei wurde eine Vergleichsmessung zum nicht inkrementellen Hadoop System mit unterschiedlich stark veränderten Anteilen am Eingangsdatensatz durchgeführt.

Die Beschleunigung durch Incoop war in jedem Szenario messbar und lag zwischen 3 bis 100 facher Beschleunigung bei 0% bis 25% veränderten Eingabedaten. Die berechnungsintensiven Programme profitierten am stärksten von der Beschleunigung.

3 Muppet: MapReduce-Style Processing of Fast Data

Das MapUpdate-Framework „Muppet“ [4], das bei den Walmart Labs entwickelt wurde, beschreibt eine Weiterentwicklung des MapReduce Prinzips um speziell schnelle, nicht endende Datenströme direkt als Eingang verwenden zu können.

Die Anforderungen an das System sollen ein einfaches Programmiermodell, hohe Skalierbarkeit, direkte Unterstützung von Datenströmen, eine geringe Verarbeitungslatenz und die ständige Abrufbarkeit von Live-Ergebnissen sein.

Für die Verarbeitung der Daten müssen Map- und Update-Funktionen definiert werden. Jede Funktion konsumiert und produziert Datenströme.

Die Update-Funktionen nutzen eine In-Memory Datenstruktur namens „Slate“ als Zustandsspeicher, die alle Key-Value Paare mit gleichem Key zusammenfasst.

Ein Stream besteht aus einer Sequenz von „Stream Events“ (Tupel mit [Stream-ID, Timestamp, Key, Value]). Die Map- und Update Funktionen registrieren sich

auf eine oder mehrere Stream-IDs und erhalten alle Events aus diesen Streams als chronologischen Input.

Eine Map-Funktion erzeugt Events für einen oder mehrere Stream-IDs mit neuen Timestamps. Die Update Funktionen legen für jeden Key ein neues Slate an, das eine Zusammenfassung aller bisher von der Update-Funktion gesehenen Keys enthält und dabei kontinuierlich aktualisiert wird.

Zur Verteilung der Berechnung legt das Muppet-System Arbeitsknoten an, auf denen die Map- oder Update-Funktionen ausgeführt werden. Mit einer Hash-Funktion werden die Events anhand des Keys auf die Eingangsqueues der Worker verteilt. Damit ein Worker das Ziel der erzeugten Events direkt ermitteln kann, haben alle Worker die gleiche Hashfunktion. Im Gegensatz zu MapReduce muss hier nicht der Umweg über den Master gemacht werden und die Latenz wird minimiert.

Die Slates werden in einem persistenten Key-Value Speicher abgelegt, der mit der verteilten Datenbank Cassandra [2] realisiert wird. Jeder Update-Worker legt bei einem neuen Key zunächst in der Cassandra Datenbank ein neues Slate an und verwaltet lokal einen Cache aller bekannten Slates. Die Frequenz der persistenten Speicherung kann konfiguriert werden. Slates kann eine Time-To-Live zugewiesen werden, die für eine Garbage-Collection bei selten auftretenden Keys sorgt.

Eine Anwendung kann Slates mit einem HTTP-Aufruf auslesen. Auf jedem Knoten läuft ein einfacher HTTP Server, der Anfragen direkt aus dem Slate-Cache beantwortet um aktuelle Ergebnisse zu liefern.

Fehlerbehandlung Die Erkennung von ausgefallenen Arbeitsknoten erfolgt nicht durch den Master, sondern geschieht, wenn ein Knoten die Zielqueue eines Ergebnistupels nicht erreichen kann. Dann werden mit dem Umweg über den Master alle Knoten über den Ausfall informiert. Events und Slates, die auf dem ausgefallenen Knoten noch in der Queue oder nicht persistent gespeichert wurden, gehen verloren. Dies wird in Kauf genommen, um die Latenz gering zu halten und muss bei den Anwendungen berücksichtigt werden. Bei einem Queue Overflow werden entweder Events verworfen und geloggt um sie später zu verarbeiten oder sie können an einen speziellen Overflow Stream geleitet werden, um von anderen Knoten verarbeitet zu werden.

Ergebnisse Die Entwickler beschreiben eine Optimierung des Systems für eine bessere Arbeitsauslastung bei Mehrkern-Rechnern durch Optimierung von Queues und die Zusammenlegung von Map- und Update-Knoten. Wenn ein Knoten beide Funktionen ausführen kann, spart dies Ressourcen beim Datenaustausch der Ergebnisse.

Es wird von einem erfolgreichen und leistungsfähigen Einsatz von Anwendungen des Muppet-Systems mit Antwortzeiten von unter 2 Sekunden bei einer Verarbeitung 100 Millionen Tweets pro Tag berichtet.

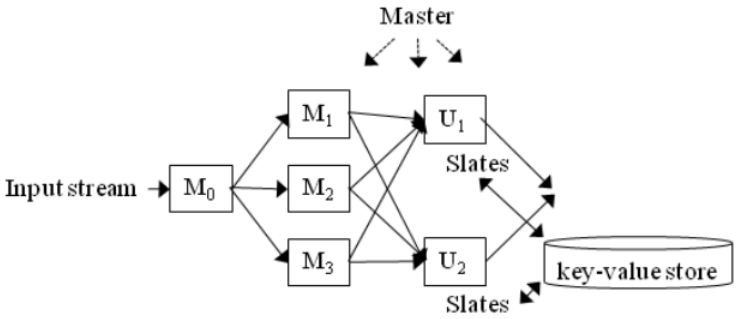


Abbildung 3: Muppet System [4]

4 Discretized Streams: Fault-Tolerant Streaming Computation at Scale

Das Open Source Projekt „Spark Streaming System“ [3] verwendet für das Verarbeiten von Datenströmen einen Ansatz mit zeitdiskretisierten Datenströmen [6]. Dieser Ansatz hat das Ziel, ein Modell zu entwickeln, das in einem hochskalierbaren verteilten System eine Fehlerbehebung durch Ausfälle ohne lange Erholungs- und Replikationszeiten ermöglicht und langsame Nachzügler Knoten toleriert. Es werden Latenzen im Sekundenbereich für Berechnungen und Erholung angestrebt.

In Clustern mit einer großen Anzahl von Knoten treten unausweichlich Ausfälle ein, daher ist bei einem Streaming System eine kurze Erholungszeit noch wichtiger als bei einem Stapelverarbeitungssystem, denn dort fällt eine etwas längere Bearbeitungszeit nicht so sehr ins Gewicht.

Der Unterschied zu verbreiteten Ansätzen anderer Streaming Systeme ist, dass Spark ohne langlebige zustandsbehaftete Operatoren auskommt. Um Probleme mit ausgefallenen Knoten zu beheben, kann mit Replikation von Knoten oder mit Backup darauf reagiert werden. Bei Replikation ist doppelte Hardware nötig und ein Backup erzeugt eine lange Wartezeit bis ein Ersatz-Knoten im System wieder verfügbar ist.

Das D-Stream Modell versucht diese Herausforderungen zu lösen. Ein D-Stream besteht aus einer Sequenz zustandsloser, deterministischer Berechnungen auf Daten aus kleinen Zeitintervallen.

Um geringe Antwortzeiten zu realisieren werden im Gegensatz zum Hadoop System die Zwischen-Ergebnisse nicht persistent auf einem verteilten und replizierten Festplatten Speichersystem sondern im Arbeitsspeicher der Knoten gehalten. Dazu wird die Datenstruktur RDD (Resilient Distributed Datasets, [7]) verwendet, die es ermöglicht, Daten bei einem Verlust mit einem Abstammungsgraph über die für die Erstellung verwendeten Operationen wiederherzustellen. Fällt ein Knoten aus, erzeugen alle Knoten des Clusters parallel Teile des verlorenen RDD und sorgen für eine schnelle Erholungszeit.

Zusätzlich bietet das System die Möglichkeit, auf einen langsam arbeitenden Knoten zu reagieren und die Berechnungen auf anderen Knoten im Cluster anstoßen zu lassen ohne auf das Ergebnis lange zu warten.

4.1 Diskretisierte Streams

Der Eingangsstrom wird in kleine Zeitintervalle aufgeteilt, in denen die eingetroffenen Daten eines Intervalls den Eingabedatensatz des Clusters bilden. Der Datensatz wird als unveränderliches, partitioniertes RDD im Cluster gespeichert. RDDs werden anhand eines Hashs auf den Knoten im Cluster partitioniert gespeichert. Zusätzlich enthält ein RDD auf Partitionsebene die Information, aus welchen Daten und deterministischen Operationen es entstanden ist.

Sobald das Intervall abgelaufen ist, wird der Datensatz mit parallelen deterministischen Map, Reduce und GroupBy Berechnungen verarbeitet (siehe Abb. 4). Endergebnisse werden persistent gespeichert. Zwischenergebnisse werden in RDDs gespeichert. Die Zwischenergebnisse können gemeinsam mit dem Eingabedatensatz des nächsten Zeitintervalls verarbeitet werden.

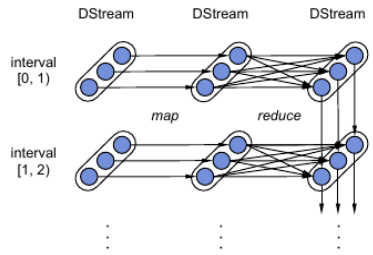


Abbildung 4: Abstammungsgraph für RDDs. Jedes Oval ist ein RDD, die Partitionierung wird als Kreis dargestellt. Jede Sequenz der RDDs ist ein D-Stream. [6]

Im Programmiermodel von Spark wird das D-Stream Interface verwendet. Jede Transformations-Berechnung erzeugt wieder einen D-Stream, der von den Map, Reduce und GroupBy Funktionen als Eingabe verwendet werden kann.

Fehlerbehandlung Wenn ein Knoten ausfällt, werden die RDD Partitionen, die sich auf dem Knoten befanden neu berechnet, indem die Tasks erneut ausgeführt werden, aus denen sie erzeugt wurden. Damit die Neuberechnung nicht unendlich viele Schritte beinhaltet, werden periodisch Checkpoints eingeführt, bei denen die RDDs in einem persistenten Speicher im Cluster gesichert werden. Da die Neuberechnung parallel durch mehrere Knoten ausgeführt werden kann und sehr schnell ist, kann der Sicherungsabstand großzügig (z.B. alle 30s) gewählt werden.

Dieser Mechanismus kann auch eingesetzt werden, wenn ein Knoten zu langsam arbeitet. Es kann spekulativ eine Neuberechnung angestoßen werden.

System Architektur Das Spark Streaming System verwendet eine modifizierte Architektur des ursprünglichen Spark Systems zur Stapelverarbeitung. Der Master verwaltet den D-Stream und RDD Abstammungsgraph und verteilt Tasks auf die Worker. Unbenötigte Teile des Abstammungsgraph werden nach einem Checkpoint verworfen.

Worker empfangen Eingangsdaten als Stream oder periodisch aus einem Dateisystem. Der Client sendet die Daten an zwei Worker, damit die Daten repliziert gespeichert sind. Sie speichern neue RDD Partitionen und führen die Tasks aus.

Alle Daten sind im Blockspeicher der Worker gespeichert, der Master führt nur Referenzen auf die Datensätze. Jeder Block hat eine eindeutige ID und kann auf mehreren Workern abgelegt sein, z.B. wenn er durch eine Berechnung auf verschiedenen Workern erzeugt wurde. Der In-Memory Blockstore verwendet eine Least-Recently-Used-Strategie, um Blocks auf die Festplatte auszulagern.

Die Uhren der Worker sind synchronisiert. Endet ein definiertes Zeitintervall senden die Worker alle empfangenen Block-IDs des Intervalls an den Master. Der Task-Scheduler des Masters startet dann die Tasks und optimiert nach Datenlokalität und fasst mit Pipelining Operationen zusammen.

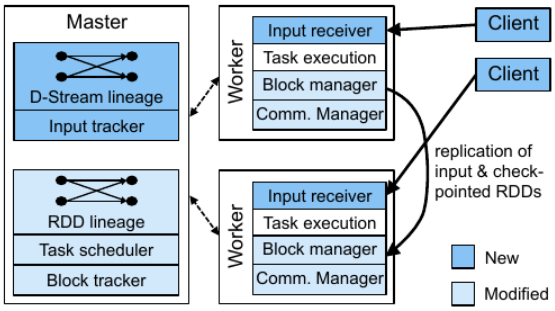


Abbildung 5: Spark Streaming Komponenten [6]

Ergebnisse

Es wird eine ausführlicher Betrachtung der Performance von Spark gezeigt, bei der die Skalierbarkeit des Systems demonstriert wird. Bei einem simulierten Ausfall eines Knotens zeigt sich, dass sich die Latenzen kurz nach dem Ausfall erhöhen, aber durch die parallele Berechnung schnell wieder normalisieren (siehe Abb. 6), obwohl gleichzeitig weiterhin Streams ohne Rückstau verarbeitet werden.

Obwohl das diskretisieren der Streams eine minimale Latenz automatisch vorgibt, gelingt es Spark, Latenzen im Bereich von 1 - 2 Sekunden zu halten und genügt damit den Anforderungen vieler Anwendungen.

Die Architektur von Spark macht es möglich, als Eingabe auch historische Daten zu verwenden um sie in die Analyse von Streams mit einfließen zu lassen.

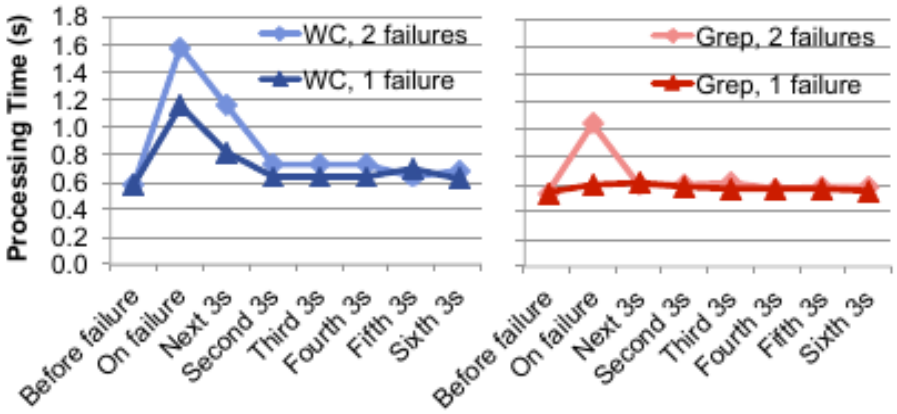


Abbildung 6: Latenz bei simuliertem Ausfall von Workern [6]

5 Zusammenfassung

Die vorgestellten Konzepte zur Verarbeitung von „Fast Data“ haben unterschiedliche Strategien, um das gewünschte Ziel der schnellen Verarbeitung von Datenströmen zu erreichen. Incoop hat einen interessanten Ansatz, der durch die Transparenz des Systems eine Umsetzung von bereits existierenden Anwendungen am einfachsten ermöglicht. Das Muppet-System stellt einen performanten Ansatz dar, der aber auf Fehlertoleranz geringeren Wert legt und geringe Latenzen höher bewertet. Das Spark Streaming System stellt das Konzept mit dem höchsten Anspruch an Skalierbarkeit und Fehlertoleranz dar.

Literatur

[1] Pramod Bhatotia u. a. „Incoop: MapReduce for Incremental Computations“. In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: ACM, 2011, 7:1–7:14. ISBN: 978-1-4503-0976-9. DOI: 10.1145/2038916.2038923. URL: http://doi.acm.org/10.1145/2038916.2038923.

[2] The Apache Software Foundation. *Apache Cassandra Database*. http://cassandra.apache.org. 2014.

[3] The Apache Software Foundation. *Apache Spark*. http://spark.apache.org. 2014.

[4] Wang Lam u. a. „Muppet: MapReduce-style Processing of Fast Data“. In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), S. 1814–1825. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=2367502.2367520.

[5] M. O. Rabin. *Fingerprinting by Random Functions*. Cambridge, MA, 1981.

[6] Matei Zaharia u. a. „Discretized Streams: Fault-tolerant Streaming Computation at Scale“. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, S. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://doi.acm.org/10.1145/2517349.2522737.

[7] Matei Zaharia u. a. „Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing“. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, S. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.