Hochschule Fulda
University of Applied Sciences

Der Umschwung von Batch- auf Stream-Processing: Verschiedene Systeme und Technologien zur Verarbeitung von Daten in Echtzeit

Leon Gabriel Kratz

Übersichtsartikel
im Seminar *Data Science and Engineering* (AI5188)
im Studiengangs *Angewandte Informatik* (M. Sc.)
im Sommersemester 2024
an der Hochschule Fulda

Leon Gabriel Kratz
Matrikelnummer: 1337747
Betreuer: Prof. Dr. David James

Eingereicht am 3. Juni 2025

1 Einleitung

Durch die zunehmende Digitalisierung werden immer mehr Daten erzeugt. Wie eine Studie [1] aus dem Jahr 2022 zeigt, steigt die Menge der erzeugten Daten Jahr für Jahr. Für das Jahr 2027 wird eine Prognose von 284,3 Zettabyte geschätzt. Ein Zettabyte entspricht ungefähr einer Milliarde Terabytes. Durch die rapide steigende Datenmenge, die Jahr für Jahr erzeugt wird, stoßen herkömmliche Datenverarbeitungssysteme zunehmend an ihre Grenzen. Batch-Processing, das seit Jahrzehnten als Standardmethode zur Verarbeitung großer Datenmengen eingesetzt wird, erweist sich vor allem in Umgebungen, in denen Daten in Echtzeit verarbeitet werden müssen, als unzureichend. Diverse Anwendungsfälle, darunter die Verarbeitung von Social Media Daten für Empfehlungsdienste, das Aufbereiten von Sensordaten oder das Analysieren der Daten des Internet of Things, benötigen den Zugang zu kontinuierlich einströmenden Informationen, um schnelle Reaktionen und Datenanalysen durchzuführen. Das Verarbeiten von Daten in Echtzeit, auch Stream-Processing genannt, wird daher immer relevanter.
Der schnelle technologische Fortschritt hat die Art und Weise, wie Daten verarbeitet und analysiert werden, revolutioniert. Der Wandel von Batch- zu Stream-Processing stellte Unternehmen und Entwickler vor neue Herausforderungen, die dazu motivieren, neue Systeme zu entwickeln, die die aufkommenden Anforderungen erfüllen. In diesem Paper wird das Prinzip des Batch-Processings und des Stream-Processing vorgestellt und dargestellt, warum der Umschwung zur Verarbeitung von Daten in Echtzeit notwendig ist. Des Weiteren werden zugrundeliegende Technologien und Systeme vorgestellt, die das Stream-Processing geprägt und weiterentwickelt haben, darunter die Stream-Processing Engines und die Message-Broker-Systeme.

2 Batch-Processing

Batch-Processing entsprach lange Zeit dem Standard beim Verarbeiten von Big Data. Der Begriff Big Data wird in der Arbeit von Shahrivari [2] als Datenmengen beschrieben, die nicht von relationalen Datenbanksystemen verarbeitet werden können. Das sind in der Regel Datenmengen, die im Web, auf sozialen Netzwerken, durch das Internet of Things oder vielen weiteren Bereichen erzeugt werden.
Batch bedeutet übersetzt Stapel und Batch-Processing beschreibt die Methode, wie große Mengen an Daten in einem festen Intervall gesammelt und dann verarbeitet werden. In der Regel entspricht dieses Intervall einer gewissen Zeit oder Menge an Daten, die vor der Verarbeitung erreicht werden muss.
Diese Herangehensweise ist sehr effektiv und wird genutzt, um große Datenmengen zu bewältigen und analytische Berichte auf Basis der Daten zu erstellen. Typische Anwendungsfälle umfassen die Verarbeitung von Transaktionsdaten, die Erstellung von Monats- oder Jahresberichten oder die Durchführung komplexer Berechnungen auf großen Datensätzen. Neben den Vorteilen von Batch-Processing, führt Shahrivari in [2] die Nachteile dieser Datenverarbeitungsmethode auf. Die Schwächen liegen dabei auf

der Latenz, die bei der Verarbeitung entsteht, sowie der Notwendigkeit, die Daten vor der Verarbeitung zu sammeln. Dies kann zur Nutzung von veralteten Informationen und langsamen Reaktionszeiten führen.
Im Folgenden wird eines der bekanntesten Batch-Processing Systeme, Apache Hadoop, vorgestellt und die Stärken des Systems, sowie die Schwächen in Bezug auf die Echtzeit-Analyse von Daten, erläutert.

2.1 Apache Hadoop

Apache Hadoop [3] ist ein Framework für die verteilte Datenverarbeitung. Es nutzt den HDFS (Hadoop Distributed File Storage) als Speichereinheit und verarbeitet Daten basierend auf dem MapReduce-Programmiermodell [2]. Das Modell wurde von Google entwickelt und ermöglicht das verteilte Verarbeiten und Skalieren von Big Data.
In den Arbeiten von Shahrivari [2] und Garcia et al. [4] wird beschrieben, dass MapReduce aus den zwei zentralen Funktionen Map und Reduce besteht, die beim Verarbeiten von Daten ausgeführt werden. Ein MapReduce-Job erhält ein Key/Value-Pair als Input. Innerhalb der Map-Funktion wird der Input in ein Zwischenergebnis, bestehend aus kleineren Key/Value-Pairs, zerlegt. Im sogenannten Shuffling-Prozess werden die Zwischenergebnisse nach ihrem Schlüsselwert in einer Liste gruppiert. Die Reduce-Funktion verarbeitet die Zwischenergebnisse daraufhin in finale Key/Value-Pairs, die als Output gespeichert werden. Das Programmiermodell besitzt eine Master/Slave-Architektur, bei der ein Master-Knoten im Cluster die Aufgaben auf die Slave-Knoten verteilt und diese kontrolliert. Abbildung 1 stellt dar, wie ein Input auf die zugeteilten Map-Workern aufgeteilt wird, damit diese den Input verarbeiten können. Die Map-Worker erzeugen nach dem Shuffling-Prozess Zwischenergebnisse, die von den zugeteilten Reduce-Workern eingelesen und als Outputs in die Output-Files geschrieben werden.

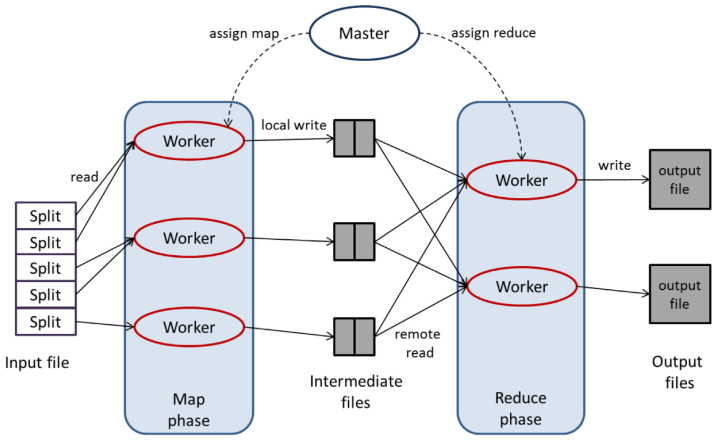

Abbildung 1: Ausführung eines MapReduce-Jobs mit der Master/Slave-Architektur (entnommen aus [2])

Hadoop kombiniert, laut [2], die Vorteile von MapReduce, darunter das simple Programmiermodell, die Skalierbarkeit und die Fehlertoleranz, mit zusätzlichen Features. Die Features umfassen verschiedene Scheduler, anspruchsvollere und komplexere Job-Definition mithilfe von YARN, ein Ressourcenmanagement-System, das die Verwaltung von Rechenressourcen in einem Hadoop-Cluster übernimmt [5], hochverfügbare Master-Maschinen, anpassbare Input- und Outputschnittstellen und vieles mehr. Shahravi erläutert in [2], dass Hadoop eine Basis für die Verarbeitung von Big Data bietet, auf der erweiternde Softwarelösungen aufgesetzt werden können. Beispielsweise kann HBase zur Speicherung von strukturierten Daten in großen Tabellen oder Pig und Hive für Data Warehousing installiert werden [2]. Ein weiteres Framework, welches auf Hadoop installiert werden kann, ist Apache Spark [6], das in dem folgenden Kapitel 3.1 thematisiert wird.
Neben der Flexibilität und den Vorteilen, die Apache Hadoop beim Verarbeiten von Big Data bietet, besitzt die Umsetzung auch Nachteile. Das Batch-Processing von Hadoop ermöglicht zwar einen hohen Durchsatz, ist allerdings dafür mit einer hohen Latenz verbunden. MapReduce-Jobs werden asynchron verarbeitet, sobald der gesamte Input für die Verarbeitung bereit ist, wodurch eine Verarbeitung je nach Datenmenge mehrere Stunden bis Tage andauern kann [2]. Dieses Verhalten ist für diverse Anforderungen und Systeme, wie in Kapitel 1 erläutert, zu langsam. Daher gibt es verschiedene Systeme und Architekturen, die eine Verarbeitung von Daten in Echtzeit ermöglichen, darunter Stream-Processing Engines und Message-Broker Systeme, die in den folgenden Kapiteln vorgestellt werden.

3 Stream-Processing Engines

Stream-Processing Engines sind Systeme, die für die Verarbeitung von Datenströmen in Echtzeit entwickelt oder erweitert wurden. Im Gegensatz zum Batch-Processing, ermöglichen Stream-Processing Engines die kontinuierliche und nahezu sofortige Analyse von eingehenden Daten. Diese Systeme spielen eine zentrale Rolle in modernen Anwendungsfällen, die schnelle Reaktionen auf Echtzeitdaten erfordern.
Zwei der bekanntesten Technologien in diesem Bereich sind Apache Spark [6] und Apache Flink [7]. Während Spark ursprünglich als Batch-Processing-Framework entwickelt wurde und später um Streaming-Funktionen erweitert wurde, ist Flink von Grund auf als Stream-Processing Engine konzipiert. Beide Systeme bieten leistungsstarke Werkzeuge für die Verarbeitung großer Datenmengen, unterscheiden sich jedoch in ihrer Architektur und ihren Ansätzen zur Echtzeitdatenverarbeitung. Im Folgenden werden die Eigenschaften und Unterschiede dieser beiden Engines erläutert.

3.1 Apache Spark

Apache Spark ist ein Framework, das für die Verarbeitung von Batch- und Stream-Daten im Bereich Big Data verwendet wird. Es wird erstmals in der Arbeit von Zaharia

et al. [8] vorgestellt und stellt eine neuere und schnellere Art dar, Big Data zu verarbeiten. In der Arbeit von Jonnalagadda et al. [9] erläutern die Autoren, dass Apache Spark oftmals als Erweiterung von Hadoop-Clustern, die in Kapitel 2.1 vorgestellt werden, verwendet wird. Dabei greift es auf den Speicher und die Verarbeitung der Daten zu. Durch die Verwendung von Spark kann ein Hadoop-Cluster für die Nutzung von iterativen Algorithmen, interaktiven Queries und Streaming mithilfe von Micro-Batches erweitert werden, während es weiterhin für Batch-Processing genutzt werden kann. In [9] wird dabei betont, dass Spark nicht abhängig von Hadoop ist und auch eigenständig ohne Hadoop und MapReduce oder mit anderen Systemen verknüpft genutzt werden kann.
Da das Speichersystem von Hadoop (HDFS), laut [2], dafür optimiert ist, einen hohen Durchsatz bei Schreib- und Lesevorgängen zu verarbeiten, ist die Performance beim Verarbeiten der Daten nicht hoch. Jonnalagadda et al. erläutern in ihrer Arbeit [9] ergänzend dazu, dass die Wiederverwendung der Daten zwischen den Berechnungen eine Speicherung im HDFS benötigt. Apache Spark löst dieses Problem durch In-Memory Computing, das in [2], [4], [8] und [9] vorgestellt wird. Beim In-Memory Computing nutzt das verteilte System einen gemeinsamen Arbeitsspeicher, wo Big Data gespeichert und verarbeitet werden kann. Das hat den Vorteil, dass eine höhere Bandbreite und eine niedrigere Zugriffslatenz beim Aufrufen der Daten bereitgestellt wird. Dabei werden in der Regel die Daten in dem Arbeitsspeicher gespeichert, auf die das Cluster häufig zugreift. Die Arbeiten von Shahrivari [2] und Garcia et al. [4] erläutern dazu ergänzend, dass Stream-Processing in Apache Spark dadurch erreicht wird, dass Micro-Batches, mit geringen Datenpaketen, schnell und kontinuierlich verarbeitet werden.
Damit Spark einen verteilten Arbeitsspeicher gemeinsam nutzen kann, nutzt es Resilient Distributed Datasets (RDD), welche in [4], [8] und [9] vorgestellt werden. RDDs sind die fundamentale Datenstruktur von Spark, die im Vergleich zu HDFS-Datenstrukturen auch im Arbeitsspeicher gespeichert werden kann. Sie entsprechen einer Sammlung von Objekten, die auf dem Cluster verteilt werden, beim Programmieren jedoch wahrgenommen werden, als ob sie auf einem einzigen Knoten liegen würden. RDDs sind unveränderlich, was bedeutet, dass sie nach dem Erstellen nicht mehr umgeschrieben werden können. Sie können entweder durch die Verarbeitung von bestehenden RDDs oder durch einen Verweis auf einen Datensatz in einem externen Speicher erstellt werden. Ein Spark-Cluster besteht aus einem Driver-Knoten und mehreren Worker-Knoten. Der Driver-Knoten, auf dem das Programm ausgeführt wird, verwaltet die Transformationen, die auf den Daten in den RDDs durchgeführt werden sollen, und sendet Partitionen der RDDs an die Worker-Knoten. Die Worker-Knoten führen die Transformationen dann auf ihrem Teil der Daten aus. Ein weiterer Vorteil, den Spark in Vergleich zu MapReduce bietet, ist, dass neben der Map- und Reduce-Funktion noch weitere Funktionen angeboten werden, die über das Framework ausführbar sind, wie beispielsweise das Filtern von Daten.
Grundsätzlich lässt sich Apache Spark in fünf verschiedene Komponenten unterteilen, die in den Arbeiten von Garcia et al. [4] und Jonnalagadda et al. [9] vorgestellt werden. Die grundlegende Komponente, die mit den anderen in Abbildung 2 dargestellt wird,

ist der Apache Spark Core. Der Core ist die allgemeine Execution Engine, auf der alle anderen Funktionalitäten aufgebaut sind. Er liefert das In-Memory Computing, sowie die Referenzen auf externe Speichersysteme. Über Spark SQL stellt das System interaktive Queries zur Verfügung. Diese unterstützen sowohl strukturierte als auch semistrukturierte Daten. Spark Streaming stellt die Verarbeitung von Datenströmen bereit. Dort wird die schnelle Scheduling-Fähigkeit des Cores genutzt, um Stream-Analysen durchzuführen, Micro-Batches zu verarbeiten und RDD-Transformation auf diesen auszuführen. Die Machine Learning Libary, MLib, und das verteilte Graphenverarbeitungs-Framework, GraphX, bilden die zwei anderen Komponenten des Systems.

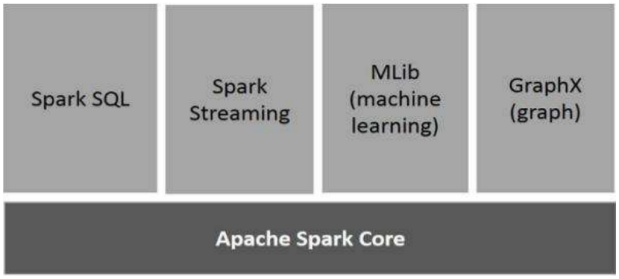

Abbildung 2: Architektur von Apache Spark (entnommen aus [9])

Die Abbildungen 3 und 4 zeigen beispielhaft den Vergleich zwischen einem iterativen Algorithmus in Hadoop und in Spark. Wie in der Abbildung zu erkennen, nutzt Spark den verteilten Arbeitsspeicher, der den Knoten im Cluster gemeinsam zur Verfügung steht, um effektiv Daten zwischenzuspeichern und einzulesen. Hadoop im Gegensatz verwendet den langsameren Festplattenspeicher, um die einzelnen Zwischenergebnisse zu speichern und wieder einzulesen.

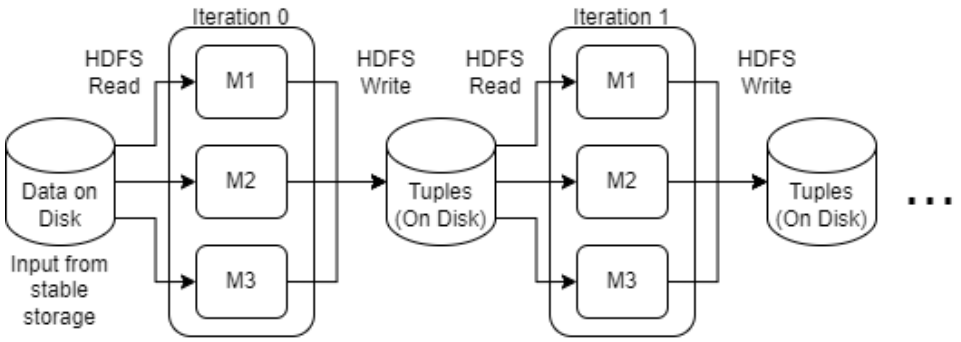

Abbildung 3: Durchführung von Iterationen in Apache Hadoop (entnommen aus [9])

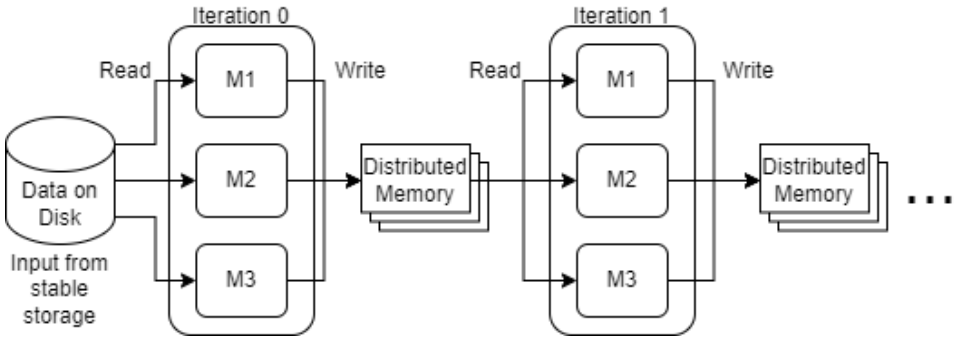

Abbildung 4: Durchführung von Iterationen in Apache Spark (entnommen aus [9])

3.2 Apache Flink

Während Apache Spark für Stream-Processing erweitert wurde, ist Apache Flink, das in den Arbeiten von Garcia et al. [4] und Carbone et al. [10] vorgestellt wird, eine Stream-Processing Engine, die explizit für Stream-Processing entwickelt wurde. Im Gegenteil zu Spark, wo das Stream-Processing mit Micro-Batches einen Spezialfall des Batch-Processings darstellt, ist das Batch-Processing in Flink ein Spezialfall des dort implementierten Stream-Processings[10].
Laut [4] fokussiert sich Flink darauf, viele Daten mit niedriger Latenz und hoher Fehlertoleranz in einem verteilten System zu verarbeiten. Das Hauptmerkmal von Flink liegt dabei auf der Verarbeitung der Daten in Echtzeit.

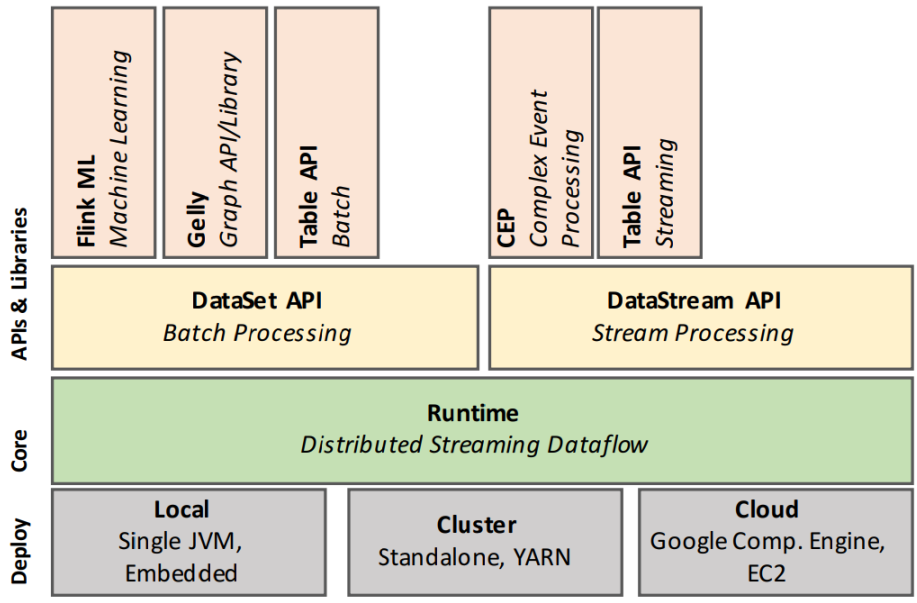

Abbildung 5: Architektur von Apache Flink (entnommen aus [10])

Die Architektur von Flink, die in [4] und [10] vorgestellt wird, besteht aus vier verschiedenen Schichten, die in Abbildung 5 dargestellt sind. Die unterste Schicht ist die Deployment-Schicht. Dort sind verschiedene Umgebungen gezeigt, auf denen Flink aufgesetzt werden kann. Das Herzstück der Stream-Processing Engine bildet die Core-Schicht, in welcher die verteilte Runtime-Engine liegt. Dort werden Programme, umgesetzt als Dataflow-Graphen, ausgeführt. Über der Core-Schicht liegen die zwei zentralen APIs von Flink, die DataSet-API und die DataStream-API. Die DataSet-API ist für das Batch-Processing zuständig. Durch sie werden endliche Datenströme beziehungsweise Datensets verarbeitet. Sie bildet durch die Verarbeitung von begrenzten Datenströmen das Batch-Processing als Spezialform des Stream-Processings an. Mithilfe der DataStream-API hingegen wird das Stream-Processing umgesetzt. Durch sie werden unbegrenzte und kontinuierliche Datenströme verarbeitet. Die Engine der Core-Schicht ist in der Lage, Programme der beiden APIs in Dataflows umzuwandeln und auszuführen. Die oberste Schicht bilden die Libraries und APIs basierend auf den zwei zentralen APIs. Dort stehen über der DataSet-API FlinkML für Machine Learning und Gelly für das Verarbeiten von Graphen zur Verfügung. Die DataStream-API bietet hingegen CEP für das Verarbeiten von komplexen Events. Die Table API zum Abfragen und Verarbeiten von Tabellen wird von beiden APIs bereitgestellt.

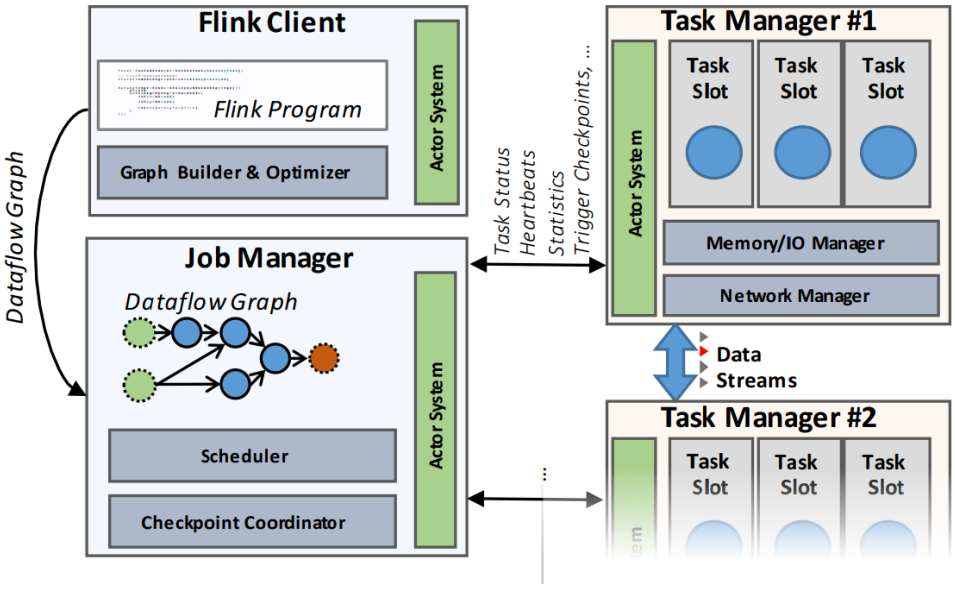

Abbildung 6: Aufbau eines Flink-Clusters (entnommen aus [10])

Apache Flink verarbeitet seine Programme, wie bereits beschrieben, in einem verteilten Cluster. Ein Flink-Cluster, das in [4] präsentiert und in Abbildung 6 dargestellt wird, besteht aus drei Komponenten, darunter dem Flink-Client, dem Job-Manager und dem Task-Manager. Der Flink-Client beginnt damit, den gegebenen Programmcode im Graph-Builder in einen Dataflow-Graphen zu transformieren. Handelt es sich um

ein Programm der DataSet-API, durchläuft das Programm im Optimizer zusätzlich eine Query-Optimierung. Nachdem der Flink-Client den Dataflow-Graphen erstellt hat, wird dieser an den Job-Manager weitergeben. Der Job-Manager ist die koordinierende Einheit im Cluster. Er verteilt die Ausführung des Graphen auf die einzelnen Task-Manager, überwacht den Zustand der Ausführungen, führt Checkpoints durch und unterstützt die Wiederherstellung der Dataflow-Ausführung bei Ausfällen. Flink bietet die Möglichkeit, die Verfügbarkeit der Job-Manager und der Ausführung zu erhöhen, indem Checkpoints minimale Metadaten auf einem ausfallsicheren Speicher gespeichert werden. Im Fehlerfall kann ein Ersatz-Job-Manager diesen Checkpoint nutzen, um eine Ausführung wiederherzustellen. Die letzte Komponente in einem Flink-Cluster sind die Task-Manager. Diese verarbeiten die Daten, führen einen oder mehrere Operatoren durch und produzieren Datenströme. Sie besitzen einen eigenen Speicher, um Daten des Datenstroms zu speichern, und eine Netzwerkverbindung, um Datenströme zwischen den Operatoren auszutauschen.

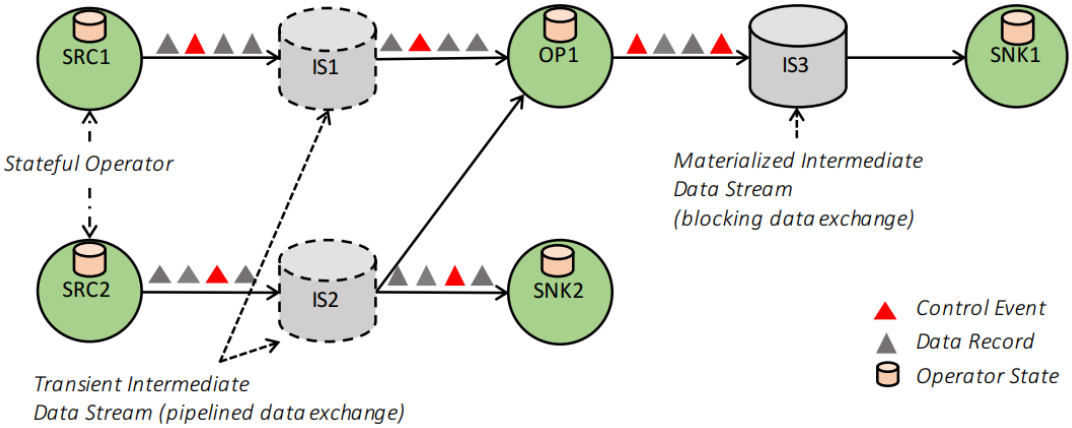

Abbildung 7: Simpler Dataflow in Flink (entnommen aus [10])

Als zentrale Repräsentation von Programmen nutzt Flink die bereits erwähnten Dataflow-Graphen. Sie werden in der Arbeit von Carbone et al. [10] vorgestellt und bilden das Programm, das letztendlich von der Runtime-Engine ausgeführt wird. Dataflow-Graphs sind gerichtete azyklische Graphen, auch DAG genannt, und beinhalten zwei verschiedene Komponenten. Die erste Komponente bilden die zustandsbehafteten Operatoren, die die gesamte Verarbeitungslogik ausführen. Sie beinhalten unter anderem verschiedene Funktionen, wie das Filtern, das Hash-Joinen oder das Windowing, und entsprechen oftmals Standardversionen von bekannten Algorithmen. Die zweite Komponente sind die Datenströme, die von den verschiedenen Operatoren erzeugt und konsumiert werden. Sie repräsentieren die Daten, die im Verlauf des Programms verarbeitet werden. Die Ausführung eines Dataflow-Graphen geschieht datenparallel. Das bedeutet, dass die Operatoren in eine oder mehrere parallele Instanzen, auch Subtasks genannt, aufgeteilt werden. Der Datenstrom wird ebenfalls in mehrere Stream-Partitionen unterteilt, die dann von den Subtasks verarbeitet werden. Abbildung 7 zeigt den Aufbau ei-

nes simplen Dataflow-Graphen. Dort ist dargestellt, wie die verschiedenen Operatoren einen Datenstrom erzeugen, der daraufhin zwischen den anderen Operatoren ausgetauscht und konsumiert wird.
Apache Flink besitzt damit diverse Ähnlichkeiten zu Apache Spark. Beide Systeme arbeiten auf verteilten Clustern und können Batch- und Stream-Processing Aufgaben erledigen. Zudem bieten beide Engines die Unterstützung für verschiedene Technologien über ihre APIs an. Während Apache Flink das Batch- und Stream-Processing in zwei APIs aufteilt, ermöglicht Apache Spark die Umsetzung von Stream-Processing als Micro-Batches, weshalb sowohl Batch- als auch Stream-Processing unter einer API erreichbar sind. Weiterhin unterscheiden sich die Systeme in ihrer Geschwindigkeit. Da Spark ursprünglich nicht für Stream-Processing entwickelt wurde, kommt es beim Verarbeiten des Datenstroms zu höheren Latenzzeiten als bei Apache Flink. In [4] wird dargestellt, dass Spark bei der Verarbeitung nur nahezu Echtzeit erreicht, während Flink die Daten in Echtzeit verarbeiten kann. Damit zeigt sich, dass Apache Spark das bessere System für die Verarbeitung von Batches ist, während Flink für das Verarbeiten von Datenströmen eingesetzt werden sollte.
Damit Flink einige Vorteile des Systems nutzen kann, benötigt es Message-Queues, die eine beliebige Wiedergabe des Datenstroms ermöglichen. Solche Message-Queues lassen sich als Message-Broker-Systeme, die im folgenden Kapitel vorgestellt werden, umsetzen.

4 Message-Broker-Systeme

Message-Broker-Systeme haben sich in Bezug auf Stream-Processing als eine zentrale Komponente der Infrastruktur etabliert. Sie ermöglichen die Verarbeitung von Datenströmen in Echtzeit und stellen sicher, dass große Mengen von Nachrichten effizient und zuverlässig zwischen verteilten System übertragen werden können. Im Gegensatz zum klassischen Batch-Processing bieten die Systeme die Möglichkeit, Daten kontinuierlich zu empfangen und sofort weiterzuleiten. Dadurch können moderne Anwendungen, die auf schnelle Reaktionszeiten und fortlaufende Datenverarbeitung angewiesen sind, effektiv bereitgestellt werden. In diesem Kapitel werden zwei der bekanntesten Message-Broker-Systeme vorgestellt, darunter Apache Kafka [11], ein etabliertes System, das maßgeblich zur Popularität von Stream-Processing beigetragen hat, und Redpanda [12], eine moderne Alternative, die durch Optimierungen eine noch effizientere Datenverarbeitung als Kafka ermöglicht.

4.1 Apache Kafka

Apache Kafka [11] ist eine weitverbreitete Infrastruktur, die in Bereichen des Stream-Processings implementiert wird [13]. Das System wurde im Jahr 2010 von LinkedIn entwickelt wurde und in der Arbeit von Goodhope et al. [14] vorgestellt. Wie in der Arbeit erläutert, hatte LinkedIn das Problem, dass die herkömmlichen Data Warehouses

und Hadoop-Umgebungen, die die Daten als Batches verarbeiten, nicht effizient genug waren, um die Funktionalitäten zu bedienen, die auf Aktivitätsdaten angewiesen waren. Dazu zählten zum Beispiel Werbungs- oder Empfehlungssysteme. Das Unternehmen wollte daher ein System entwickeln, dass das hohe Volumen der Aktivitätsdaten in Echtzeit verarbeitet und für entsprechende Anwendungen zugänglich macht. Zudem sollte das Problem der alten Architektur, dass bestimmte Daten, wie Business-Metriken, nur für bestimmte Anwendungen, wie Monitoringsysteme, zur Verfügung standen, mit dem neuen System behoben werden. Das neue Kafka-System sollte die Daten daher nicht nur in Echtzeit erreichbar, sondern auch für verschiedene Anwendungen einfach zugänglich machen [14].
Die Kafka-Systeme, die in den Arbeiten [13], [14] und [15] vorgestellt werden, bestehen aus drei großen Komponenten, dem Producer/Publisher, dem Kafka-Cluster/Broker und dem Consumer/Subscriber. Der Producer ist die Einheit des Clusters, die Datenpakete, auch Messages genannt, zu einem bestimmten Topic veröffentlicht und diese an die Broker weiterleitet. Der Broker hingegen verwaltet die verschiedenen Topics, während der Consumer diese ausliest und verarbeitet. Jedes Kafka Topic besteht aus verschiedenen Partitionen. Ein Broker hostet beliebig viele Partitionen für jedes Topic. Goodhope et al. [14] erläutert ergänzend, dass jede Partition aus einem sortierten Write-Ahead-Log besteht und zwei verschiedene Operationen anbietet. Entweder können Messages an das Ende vom Log angefügt werden oder ab einer bestimmten Message-ID sequentiell ausgelesen werden. Beide diese Operationen stehen in verschiedenen Programmiersprachen zur Verfügung.
Wenn ein Producer eine Message an einen Broker sendet, wird diese an eine der Partitionen des Topics angehängt. In der Arbeit von Thein [15] wird erläutert, dass Kafka dabei entweder eine zufällige Verteilung auf die Partitionen per Round-Robin-System oder eine feste Verteilung anhand eines Key-Value-Pairs anbietet. Bei einer Verteilung mithilfe des Key-Value-Pairs werden die Messages anhand des gehashten Schlüsselwertes einer Partition zugeteilt. In [14] wird dazu ergänzend erklärt, dass das zum einen den Vorteil bietet, dass die Partitionen bereits einem Kontext unterliegen, beispielsweise einer User-ID. Zum anderen bleibt die zeitlich korrekte Reihenfolge innerhalb einer Partition erhalten, da die Messages nur anhängend gespeichert werden können. Falls für den Schlüsselwert noch keine Partition existiert, wird eine neue erstellt.
Kleppmann et al. erklärt in [13], dass die Broker zu jeder Partition ein Offset verwalten. Das Offset entspricht einer Markierung, die sequentiell abhängig von den Messages in einer Partition angehoben wird. Es wird weiterhin ausgeführt, dass Partitionen auf mehrere Broker-Knoten repliziert werden, damit die Messages verlustfrei und fehlertolerant auf den Brokern gespeichert sind. Für jede Partition wird ein Leader-Knoten bestimmt, der die Lese- und Schreibzugriffe verwaltet. Dieser serialisiert den Schreibvorgang und repliziert ihn dann synchron auf die Replika. Kafka bietet, laut [13], zudem die Möglichkeit, flexibel Broker-Knoten zu einem Cluster hinzuzufügen und Partitionen neu zu verteilen, ohne, dass die Anzahl der Partitionen eines Topics angepasst werden müssen.

Des Weiteren erläutert Kleppmann et al. in [13], dass ein Kafka Topic standardmäßig eine Message für eine Woche speichert, bis diese verworfen wird. Alternativ kann diese Zeitspanne entweder angepasst werden oder durch eine Log-Rotation ersetzt werden, bei welcher der Speicher erst vollständig gefüllt wird, bis dann die ältesten Messages für die Neusten gefiltert und gelöscht werden.
Consumer sind die Einheiten im System, die die Messages aus den Topics konsumieren. In den Arbeiten von Kleppmann et al. [13] und Goodhope et al. [14] wird dargestellt, wie ein Consumer ein Topic verarbeitet. Beim Erzeugen eines Consumers wird dieser einer Consumer-Group zugeordnet. Eine Consumer-Group entspricht einem verteilten System, bestehend aus verschiedenen Consumer-Clients. Jeder Consumer-Client liest mindestens eine Partition, kann aber auch mehrere Partitionen eines Topics einlesen, wobei zwei Clients der gleichen Gruppe nicht die gleiche Partition einlesen können. Dadurch wird verhindert, dass die Messages einer Partition mehrfach von der gleichen Consumer-Group eingelesen werden. Während des Lesevorgangs wird die komplette Partition sequentiell eingelesen. Die Consumer-Clients speichern für jede Partition das Offset in einem festen Speicher, mit welchem sie markieren, bis zu welcher Message sie die Partition bereits gelesen haben. Das dient dazu, dass im Fall eines Absturzes oder Neustarts, der Client nicht die gesamte Partition erneut lesen muss, sondern ab dem letzten gespeicherten Offset weiterlesen kann. Nachdem ein Consumer-Client eine Message eingelesen hat, wird diese auch den anderen Clients der Consumer-Group zur Verfügung gestellt. Dadurch kann eine Consumer-Group die Partitionen der Topics effizient und parallel einlesen. Wird ein neuer Client in einer Consumer-Group hinzugefügt, werden die Partitionen von Kafka neu auf die verschiedenen Client-Prozesse verteilt. Es kann maximal so viele Clients geben, wie es auch Partitionen in einem Topic gibt. Damit ist die maximale Parallelität erreicht, wenn es genauso viele Consumer-Clients gibt, wie Partitionen in einem Topic. Die horizontale Skalierbarkeit, die durch die Anzahl an Partitionen gegeben wird, ist ein zentraler Vorteil von Kafka, da die Ressourcen damit an den Bedarf und den Workload angepasst werden können. Ein weiterer Vorteil, der von [13] betont wird, ist, dass sich die einzelnen Teile des Systems beim Lesen und Schreiben nicht beeinflussen. Ein Consumer-Client, der nur sehr langsam seine Partition einliest, beeinflusst weder Publisher, die auf dem Topic oder der Partition Messages veröffentlichen, noch andere Consumer-Clients, die die Partition für ihre Consumer-Group einlesen. Damit ist das System entkoppelt, wodurch Abstürze einzelner Komponenten keinen direkten Einfluss auf das gesamte System haben.
Abbildung 8 zeigt den Aufbau eines Kafka Topics vereinfacht. Sie stellt dar, wie die Topis in verschiedene Partitionen aufgeteilt werden. Zwei Producer-Clients publizieren verschiedene Messages auf die Partitionen der Topics A und B, während eine Consumer-Group, bestehend aus zwei Consumer-Clients, die Messages der Partitionen des Topics B sequentiell einliest. Der erste der beiden Consumer-Clients liest dabei die Partitionen 0 und 1 ein und speichert die entsprechenden Offsets zu diesen. Der zweite Consumer-Client hingegen liest die Partition 2 ein und speichert das entsprechende Offset. Würde einer der beiden Clients abstürzen, könnte er nach dem Neustart, das Einlesen wieder am gespeicherten Offset beginnen.

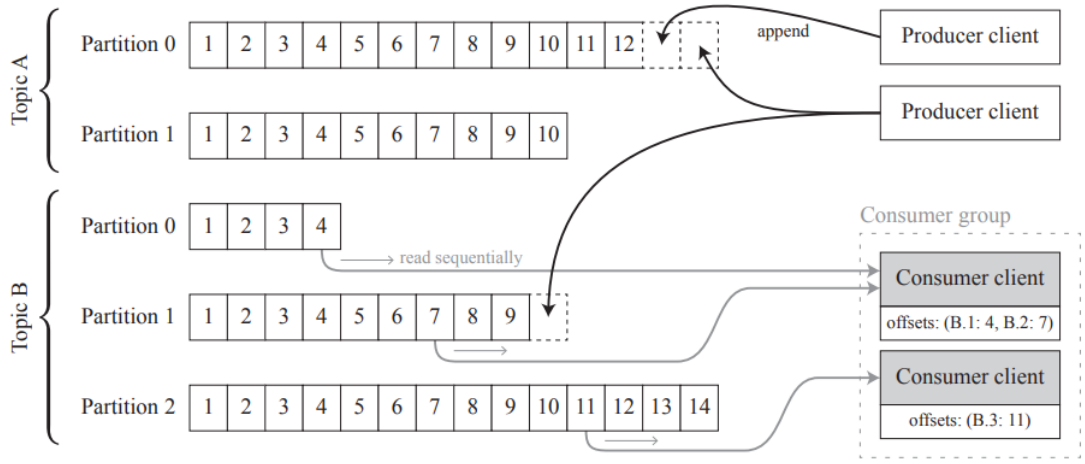

Abbildung 8: Vereinfachter Aufbau von Kafka Topics in mehrere Partitionen (entnommen aus [13])

Apache Kafka wurde dafür entwickelt, einen hohen Durchsatz von Messages speichern und verarbeiten zu können. Die Arbeit [14] stellt genauer dar, wie LinkedIn das System in ihre Infrastruktur einbettet. Laut der Arbeit verarbeitet das System bis zu 10 Milliarden Messages pro Tag, wobei der höchste Durchsatz bei 172.000 Messages pro Sekunde liege. Dabei seien 40 Echtzeit-Consumer im Einsatz gewesen, wovon acht für Monitoring und 32 für andere Produktfeatures oder Tools eingesetzt worden sind. Zudem soll jede im Live-Datenzentrum veröffentlichte Message im Durchschnitt 5,5x konsumiert worden sein. Abbildung 9 zeigt den Aufbau eines Datenclusters bei LinkedIn. Dort wird dargestellt, wie die Daten in verschiedenen Live-Datenzentren erzeugt und für Services zum Verarbeiten bereitgestellt werden. Ein zentrales Cluster, das Offline-Datenzentrum, liest die Daten der Live-Datenzentren wie ein Consumer ein und speichert diese. Von dort aus werden die Daten für Aufbereitungen und Analysen an das Hadoop-Cluster weitergeleitet, welches diese im Data Warehouse speichert.
Damit Apache Kafka einen hohen Durchsatz von Messages erreichen kann, hat LinkedIn verschiedene Optimierungen in das System eingebaut. Eine dieser Optimierungen, die in [14] dargestellt wird, ist die Nutzung von Page-Caches, einem virtuellen Speicher des Betriebssystems, der Datenpakete buffert, um ihren Schreibvorgang zu verzögern. Damit entzieht sich Kafka von der Aufgabe, die Daten auf die Festplatten der Systeme zu schreiben und überlässt diese dem I/O-Scheduler des jeweiligen Betriebssystems. Das hat den Vorteil, dass kleine Schreiboperationen in einen großen linearen Schreibvorgang zusammengefasst werden können. Da die Bandbreite der Festplatten zur Entwicklungszeit von Kafka ein Bottleneck bei der Speicherung der Daten war, dient der Page-Cache dazu, kostenintensive Schreibvorgänge zu minimieren und zu optimieren. Des Weiteren bietet der Page-Cache eine Fehlertoleranz, da Daten nicht mehr beim Anwendungsabsturz, sondern nur beim Absturz des gesamten Betriebssystems und damit beim Löschen des Page-Caches verloren gehen. Die Daten werden beim Speichern im

Page-Cache gesammelt, bis ein bestimmter konfigurierbarer Flush-Threshold erreicht wird.

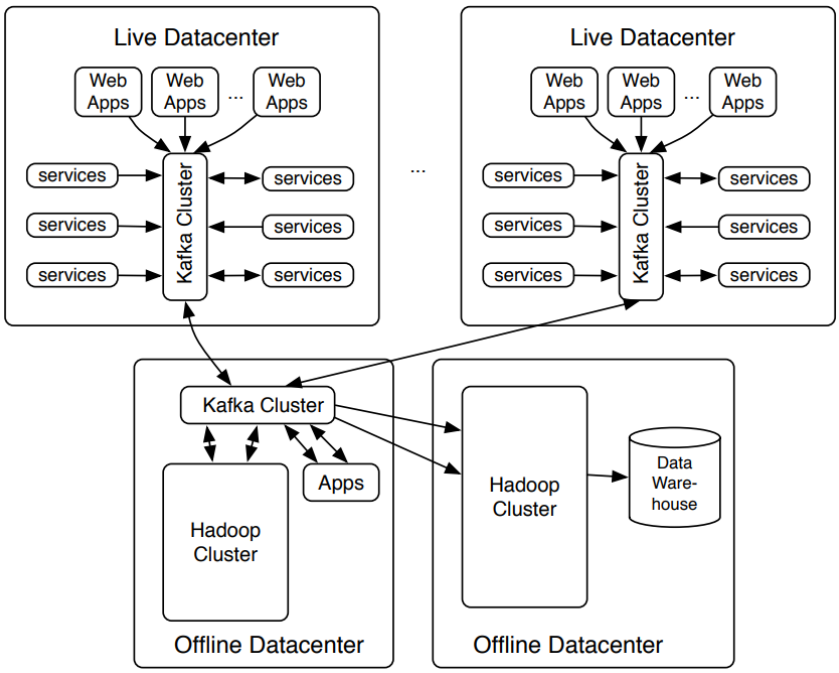

Abbildung 9: Darstellung eines Apache Kafka Clusters bei LinkedIn mit Live- und Offline-Datenzentren (entnommen aus [14])

Neben dem Page-Caching stellt Goodhope et al. in [14] das Batching vor, welches Kafka nutzt, um den Durchsatz bei vielen kleinen Messages zu verbessern. Dabei werden die Messages vom Producer erst gesammelt, bevor sie verschickt werden. Zwar steigt dadurch die Latenz geringfügig, der Durchsatz verbessert sich allerdings maßgebend. Abbildung 10 zeigt, wie der Durchsatz sich durch das Batching und das Page-Caching innerhalb der großen drei Systemkomponenten verbessert. Kafka bietet ebenfalls die Möglichkeit, die Verarbeitung auf Anwendungsebene so anzupassen, dass die Messages synchron und nicht asynchron als Batches verarbeitet werden. Auf der Consumer-Seite nutzt das System ebenfalls einen Batch-Vorgang, um die Anzahl der Lesevorgänge zu optimieren. Der Broker erhält dabei vom Consumer eine Buffersize und eine Message-ID, die die Startposition markiert. Er füllt die Antwort dann mit Messagedaten bis die entsprechende Buffersize erreicht ist und sendet sie an den einlesenden Consumer-Client. Neben den genannten Optimierungen nutzt Kafka ebenfalls Datenkomprimierung, um die Leistungsfähigkeit des Systems zu verbessern [14].
Zu Beginn diente Kafka der effizienten, verteilten und kontinuierlichen Speicherung der Daten. Das Kafka-System selbst diente nicht dazu, die Daten eigenständig zu verarbeiten. Systeme, wie Apache Flink und Apache Spark, die in Kapitel 3.1 und 3.2 vorgestellt werden, oder Apache Samza, das in der Arbeit von Kleppmann [13] dargestellt wird, können genutzt werden, um die Datenströme, die durch das Kafka-System fließen, zu

verarbeiten. Im Jahr 2016 wurde das System um Kafka Streams erweitert. Ein Teil der Client-Libary von Apache Kafka, der es ermöglicht Applikationen und Mikroservices zu bauen, die den Input der Kafka-Cluster verarbeiten und diesen als Output im Cluster wieder speichern können [16].

Batching Type	Improvement	Default Threshold
Producer	3.2x	200 messages or 30 seconds
Broker	51.7x	50000 messages or 30 seconds
Consumer	20.1x	1 megabyte

Abbildung 10: Verbesserung des Durchsatzes durch die Nutzung von Batching und Page Caching (entnommen aus [14])

4.2 Redpanda

In dem Beitrag von Gallego [17] wird erläutert, dass durch die Weiterentwicklung von SSDs und der steigenden Bandbreite für Lese- und Schreiboperationen der Festplatten, ein neues Bottleneck beim Speichern und Verarbeiten von Datenströmen entstanden ist, die CPU-Verarbeitungszeit. Das Daten-Streaming System Redpanda [12], dessen Architektur in den Webquellen [17] und [18] vorgestellt wird, bietet eine direkte Alternative zu Apache Kafka. Das System ist optimiert für den CPU-Workflow, um eine bessere Performance und eine schnelle Tail-Latency erreichen. Tail-Latency beschreibt die Latenz, die bei den langsamsten Operationen im System auftritt. Neben der Optimierung ist Redpanda vollständig kompatibel mit der API von Apache Kafka, weshalb bestehende Kafka-Anwendungen in der Regel ohne Änderungen auf Redpanda wechseln können [17].
Redpanda ist, laut [18], implementiert in C++ und nutzt das Seastar Framework [19], um einen direkteren und kontrollierteren Zugriff auf das Speicher-Management und die Lower-Level-Features des Arbeitsspeichers und der Festplatte zu erhalten. Während Apache Kafka, wie im vorigen Kapitel beschrieben, den Page-Cache des Betriebssystems nutzt, um effizient Daten des Datenstroms auf der Festplatte zu speichern, umgeht Redpanda den Page-Cache vollständig [17]. Abbildung 11 zeigt abstrahiert den Aufbau der Architektur von Apache Kafka und Redpanda. Wie in der Abbildung zu sehen, wird weder ein Page-Cache noch ein virtueller Speicher genutzt, um die Daten vor dem Schreibvorgang auf der Festplatte zwischenzuspeichern. Anstatt die Speicherung der Daten dem Betriebssystem zu überlassen, implementiert Redpanda den Speichervorgang eigenständig. Dieses Verhalten bietet Redpanda im Vergleich zu Apache Kafka einige Vorteile. Gallego erläutert dazu in [17], dass zum einen dadurch sichergestellt wird, dass kein Datenverlust stattfindet. Sobald die Daten in Kafka in den Page-Cache geschrieben wurden, behandelt Kafka diese als erfolgreich gespeichert. Dieses Verhalten führt beim Absturz vom Betriebssystem zu einem Datenverlust, da der Page-Cache

nach einem Absturz nicht wiederhergestellt werden kann. Durch das Umgehen des Page-Caches und das direkte Speichern der Daten auf der Festplatte, versichert Redpanda, dass es zu keinem Datenverlust kommt. Zum anderen bietet das Überspringen des Page-Caches den Vorteil, dass insgesamt weniger Schreibe- und Leseoperationen notwendig sind, um die Daten auf der Festplatte zu speichern. Dadurch kann sichergestellt werden, dass zusätzliche Latenzzeiten durch nicht notwendige Operationen vermieden werden und der Page-Cache nicht zu einem Engpass in der Verarbeitung der Daten wird.

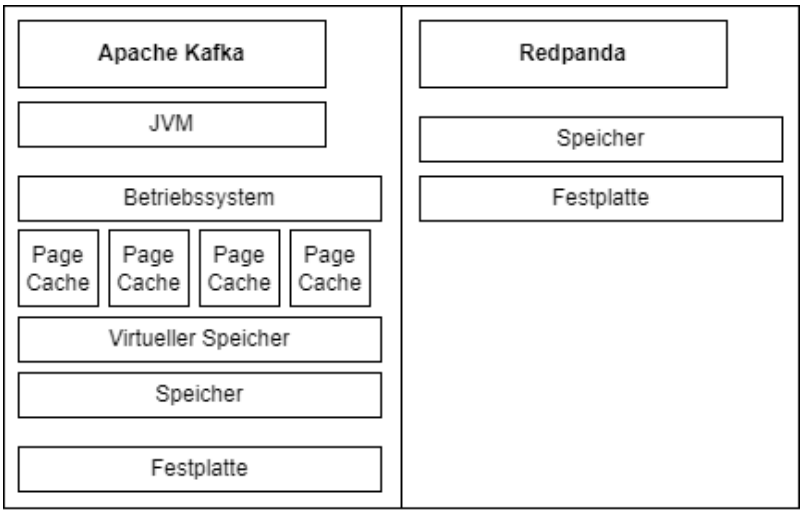

Abbildung 11: Abstrahierter Aufbau der Architektur von Apache Kafka und Redpanda

Damit Redpanda den Page-Cache effizient umgehen kann, arbeitet es mit einer Thread-Per-Core-Architektur. Bei dieser Architektur, die in dem Beitrag von Gallego [18] vorgestellt wird, erhalten die Kerne der CPU eine bestimmte Menge an Partitionen eines Kafka-Topics. Einer der Kerne, der Source-Core, verbindet sich mit einem Kafka-Client, um die Daten des Datenstroms per Request zu erhalten. Im Metadaten Cache wird die Anfrage validiert und dann durch den Partitionen-Router dem entsprechenden Kern zugeordnet. Über Structured Message Passing (SMP), sendet der Source Core die Requests in die Message Queue des entsprechenden Kerns, wo diese dann asynchron verarbeitet und auf der Festplatte gespeichert werden. Das SMP wird durch das Seastar Framework bereitgestellt und bietet die Möglichkeit, strukturierte Nachrichten zwischen Kernen auszutauschen. Durch die minimale Kommunikation zwischen den Kernen und das Einteilen der Kerne auf bestimmte Partitionen, werden die zeitintensiven Kontextwechsel, die beim Wechsel von Threads aus verschiedenen Prozessen entstehen, verhindert. Dadurch wird sichergestellt, dass die CPU-Rechenleistung für das tatsächliche Rechnen genutzt wird und nicht durch das Laden von Zustandsinformationen aufgehalten wird.
Zudem verwendet Redpanda eine optimierte Speicherverwaltung, die ergänzend in [18] vorgestellt wird. Sie ermöglicht es, dass der Arbeitsspeicher effizient auf die ver-

schiedenen CPU-Kerne verteilt wird. Jeder Kern erhält dabei seinen eigenen, fest zugewiesenen Speicherbereich, auf den er direkt zugreift, wodurch Daten nicht ständig zwischen den Kernen hin- und hergeschoben werden müssen. Dadurch hat jeder Kern seinen eigenen Speicherbereich, mit dem er unabhängig und lokal Daten speichern kann, ohne dass Zeitaufwand für Synchronisierungen zwischen Kernen, wie in klassischen Multithreading-Systemen, anfällt.
Aufgrund der diversen Optimierungen und dem Zugriff auf das Speicher-Management durch C++ arbeitet Redpanda schneller als Apache Kafka, weshalb es sich, laut [20], besonders für kosteneffiziente und reaktionsschnelle Anwendungen eignet. Apache Kafka hingegen bietet durch seine Langlebigkeit den Vorteil, dass es eine reichhaltigere Integration und eine größere Community besitzt.

5 Zusammenfassung

In diesem Übersichtsartikel wurde der Übergang von Batch- zu Stream-Processing beschrieben, insbesondere im Kontext der modernen Anforderungen an die Datenverarbeitung. Die traditionelle Batch-Verarbeitung, die anhand des Systems Apache Hadoop dargestellt wurde, erweist sich aufgrund von hohen Latenzzeiten als ungeeignet für die Echtzeitverarbeitung von Daten.
Auch Lösungen, wie Apache Spark, die ein Batch-Processing System für Stream-Processing erweiterten, erreichten nicht die Leistungen und Reaktionszeiten, die für die heutige Verarbeitung von Daten notwendig ist. Während Apache Spark durch die Erweiterung mit Micro-Batches eine Verbesserung gegenüber klassischen Batch-Systemen bietet, stößt es dennoch an Grenzen, da es nur eine nahezu Echtzeitverarbeitung ermöglicht. Im Gegensatz dazu wurde Apache Flink als Stream-Processing Engine von Grund auf für die Verarbeitung kontinuierlicher Datenströme entwickelt. Flink bietet damit geringere Latenzzeiten, weshalb es sich für das Verarbeiten von Datenströmen besser eignet, als Apache Spark.
Neben den Stream-Processing Engines spielen auch Message-Broker-Systeme eine zentrale Rolle. Apache Kafka, ein etabliertes System für verteilte Datenströme, hat maßgeblich zur Popularität des Stream-Processings beigetragen. Es ermöglicht die effiziente Übertragung und Speicherung von großen Mengen an Messages. Redpanda hingegen, als moderne Alternative zu Kafka, optimiert den Prozess durch gezielte Verbesserungen wie dem Umgehen des Page-Caches und die gezielte Nutzung moderner Hardware, was zu niedrigeren Latenzzeiten und höherer Performance führt.
Insgesamt zeigt dieser Artikel, dass der Übergang von Batch- zu Stream-Processing unvermeidlich ist, um den heutigen Anforderungen an Geschwindigkeit, Skalierbarkeit und Echtzeitverarbeitung gerecht zu werden. Technologien wie Apache Flink, Apache Spark, Apache Kafka und Redpanda spielen dabei eine entscheidende Rolle und treiben die Entwicklung in der Datenverarbeitung entscheidend voran.

Literatur

[1] IDC. Volumen der jährlich generierten/replizierten digitalen datenmenge weltweit von 2010 bis 2022 und prognose bis 2027 (in zettabyte). Statista, Mai 2023. Zugriff am 16. September 2024.

[2] Saeed Shahrivari. Beyond batch processing: towards real-time and streaming big data. *Computers*, 3(4):117–129, 2014.

[3] Apache hadoop. `https://hadoop.apache.org/`.

[4] Diego García-Gil, Sergio Ramírez-Gallego, Salvador García, and Francisco Herrera. A comparison on scalability for batch big data processing on apache spark and apache flink. *Big Data Analytics*, 2:1–11, 2017.

[5] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[6] Apache spark. `https://spark.apache.org/`.

[7] Apache flink. `https://flink.apache.org/`.

[8] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, pages 15–28, 2012.

[9] V Srinivas Jonnalagadda, P Srikanth, Krishnamachari Thumati, Sri Hari Nallamala, and K Dist. A review study of apache spark in big data processing. *International Journal of Computer Science Trends and Technology (IJCST)*, 4(3):93–98, 2016.

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4), 2015.

[11] Apache kafka. `https://kafka.apache.org/`.

[12] Redpanda. `https://www.redpanda.com/`.

[13] Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. 2015.

[14] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building linkedin's real-time activity data pipeline. *IEEE Data Eng. Bull.*, 35(2):33–45, 2012.

[15] Khin Me Me Thein. Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.

[16] Kafka streams. https://kafka.apache.org/documentation/streams/.

[17] Roko Kruze Alexander Gallego. The kafka api is great - now let's make it fast! *Redpanda Blog*, 2021. https://www.redpanda.com/blog/when-to-choose-redpanda-vs-kafka.

[18] Alexander Gallego. Thread-per-core buffer management for a modern kafka-api storage system. *Redpanda Blog*, 2020. https://www.redpanda.com/blog/engineering-redpanda-multi-core-hardware.

[19] Seastar framework. https://docs.seastar.io/master/index.html.

[20] Roko Kruze Alexander Gallego. When to choose redpanda instead of apache kafka. *Redpanda Blog*, 2023. https://www.redpanda.com/blog/redpanda-faster-safer-than-kafka.