

Mit einem Geleitwort von  CONFLUENT



Anatoly
ZELENIN
Alexander
KROPP

Apache Kafka

Von den Grundlagen
bis zum Produktiveinsatz



HANSER

Zelenin/Kropp

Apache Kafka



Bleiben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:
www.hanser-fachbuch.de/newsletter



Anatoly Zelenin
Alexander Kropp

Apache Kafka

Von den Grundlagen
bis zum Produktiveinsatz

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autor und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2022 Carl Hanser Verlag München, www.hanser-fachbuch.de

Lektorat: Sylvia Hasselbach

Copy editing: Sandra Gottmann, Wasserburg

Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de

Umschlagrealisation: Max Kostopoulos

Titelmotiv: © Max Kostopoulos

Satz: Eberl & Kösel Studio GmbH, Altusried-Krugzell

Druck und Bindung: Eberl & Kösel GmbH, Altusried-Krugzell

Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702

Printed in Germany

Print-ISBN: 978-3-446-46187-1

E-Book-ISBN: 978-3-446-47046-0

E-Pub-ISBN: 978-3-446-47265-5

Inhalt

Geleitwort	XI
Über das Buch	XV
1 Einleitung	1
1.1 Einführung in Apache Kafka	2
1.2 Erste Schritte mit Kafka	5
2 Konzepte	11
2.1 Payload	12
2.1.1 Topics	12
2.1.1.1 Anzeigen von Topics	12
2.1.1.2 Erstellen, Anpassen und Löschen von Topics	14
2.1.2 Nachrichten	17
2.1.2.1 Nachrichtentypen	17
2.1.2.2 Datenformate	19
2.1.2.3 Keys und Values	19
2.1.3 Zusammenfassung	21
2.2 Kafka als verteiltes Log	22
2.2.1 Logs	22
2.2.1.1 Was genau ist eigentlich ein Log?	22
2.2.1.2 Grundlegende Eigenschaften eines Logs	23
2.2.1.3 Die Rolle des Logs in Kafka	24
2.2.2 Verteilte Systeme	26
2.2.2.1 Partitionierung	27
2.2.2.2 Consumer Groups	30
2.2.2.3 Replikation	33

2.2.3	Komponenten	36
2.2.3.1	Koordinationscluster	37
2.2.3.2	Broker	38
2.2.3.3	Clients	38
2.2.4	Kafka im Unternehmenseinsatz	38
2.2.5	Zusammenfassung	41
2.3	Zuverlässigkeit	41
2.3.1	Acknowledgements	42
2.3.1.1	Acknowledgement-Strategien in Kafka	43
2.3.1.2	Acknowledgements und ISR	44
2.3.1.3	Nachrichten-Zustellungsgarantien in Kafka	47
2.3.2	Replikation	49
2.3.2.1	Replikation vs. Backup	49
2.3.2.2	Leader-Follower-Prinzip	50
2.3.3	Zusammenfassung	55
2.4	Performance	56
2.4.1	Topics	58
2.4.1.1	Partitionierung und Keys	58
2.4.1.2	Skalierung und Lastverteilung	59
2.4.1.3	Wie viele Partitionen sollten wir haben?	60
2.4.1.4	Verändern der Anzahl an Partitionen	61
2.4.2	Producer Performance	62
2.4.2.1	Producer-Konfiguration	63
2.4.2.2	Producer-Performance-Test	64
2.4.3	Broker Performance	66
2.4.3.1	Die Aufgaben eines Brokers	66
2.4.3.2	Broker-Konfiguration und -Optimierung	67
2.4.4	Consumer Performance	68
2.4.4.1	Consumer-Konfiguration	68
2.4.4.2	Consumer-Performance-Test	69
2.4.5	Zusammenfassung	71
3	Kafka Deep Dive	73
3.1	Verbindung zu Kafka	74
3.2	Nachrichten produzieren und persistieren	75
3.2.1	Producer	76
3.2.1.1	Nachrichten produzieren	76

3.2.1.2 Serialisierung und Partitionierung	77
3.2.1.3 ACKs und deren Auswirkungen	78
3.2.2 Broker	79
3.2.2.1 Empfang und Persistierung von Nachrichten	80
3.2.2.2 Broker und ACKs	81
3.2.2.3 Optimierung	81
3.2.3 Daten- & Dateistrukturen	82
3.2.3.1 Metadaten, Checkpoints und Topics	82
3.2.3.2 Partitionen	83
3.2.3.3 Log-Dateien und -Indizes	86
3.2.3.4 Segmente	87
3.2.3.5 Gelöschte Topics	88
3.2.4 Replikation	89
3.2.4.1 In-Sync-Replicas	90
3.2.4.2 High Watermark	90
3.2.4.3 Auswirkungen von Verzögerungen bei der Replikation	92
3.2.5 Zusammenfassung	92
3.3 Nachrichten konsumieren	93
3.3.1 Consumer	93
3.3.1.1 Fetch-Request	94
3.3.1.2 Fetch vom nächsten Replica	94
3.3.2 Broker	95
3.3.3 Offsets	96
3.3.3.1 Verwaltung von Offsets	96
3.3.3.2 Praxisbeispiel Offsets	97
3.3.4 Consumer Groups	99
3.3.4.1 Funktionsweise von Consumer Groups	99
3.3.4.2 Kafka Rebalance Protocol	100
3.3.4.3 Verteilung der Partitionen auf die Consumer	101
3.3.4.4 Static Memberships	103
3.3.5 Zusammenfassung	104
3.4 Nachrichten aufräumen	104
3.4.1 Wieso müssen wir Nachrichten aufräumen?	105
3.4.2 Kafkas Aufräum-Methoden	105
3.4.3 Log Retention	106
3.4.3.1 Wann wird ein Log via Retention aufgeräumt?	107
3.4.3.2 Offset Retention	109

3.4.4	Log Compaction	109
3.4.4.1	Wann wird ein Log via Compaction aufgeräumt?	111
3.4.4.2	Wie funktioniert der Log Cleaner?	113
3.4.4.3	Tombstones	114
3.4.5	Zusammenfassung	115
3.5	Cluster-Management	115
3.5.1	Zookeeper-basiertes Cluster-Management	116
3.5.2	KRaft-basiertes Cluster-Management	118
3.5.3	Zusammenfassung	119
3.6	Verarbeitungsgarantien und Transaktionen	119
3.6.1	Idempotenz	120
3.6.2	Transaktionen	121
3.6.2.1	Transaktionen in Datenbanken	122
3.6.2.2	Transaktionen in Kafka	123
3.6.2.3	Transaktionen und Consumer	124
3.6.3	Zusammenfassung	125
4	Kafka im Unternehmenseinsatz	127
4.1	Kafka-Ökosystem	129
4.1.1	Kafka Connect	130
4.1.1.1	Wie funktioniert Kafka Connect?	130
4.1.1.2	Praxisbeispiel Distributed Mode	132
4.1.1.3	Skalierbarkeit und Ausfallsicherheit	134
4.1.2	Kafka Streams	135
4.1.2.1	Stream Processing	135
4.1.2.2	Wie funktioniert Kafka Streams?	136
4.1.2.3	KTables	138
4.1.3	Schema-Management	139
4.1.3.1	Schemas in Apache Kafka	139
4.1.3.2	Schemaanpassung und -kompatibilität	140
4.1.3.3	Schemaformate	141
4.1.3.4	Avro	141
4.1.3.5	Schema Registry	142
4.1.4	Sicherheit	143
4.1.4.1	Verschlüsselung	143
4.1.4.2	Authentifizierung und Autorisierung	144
4.1.4.3	Zookeeper	145

4.1.5	Desaster-Management	145
4.1.5.1	Was kann schon schiefgehen?	146
4.1.5.2	Backups in Kafka	147
4.1.5.3	Stretched Cluster	147
4.1.5.4	MirrorMaker	149
4.1.6	Zusammenfassung	152
4.2	Vergleich mit anderen Technologien	152
4.2.1	Klassische Messaging-Systeme	153
4.2.2	REST	155
4.2.3	Relationale Datenbanken	155
4.2.4	Kafka als Streaming-Plattform	157
4.2.5	Zusammenfassung	158
4.3	Kafka-Referenzarchitektur	159
4.3.1	Deployment-Modelle und Hardware-Anforderungen	159
4.3.1.1	Kafka auf eigener Hardware	160
4.3.1.2	Kafka in virtualisierten Umgebungen	160
4.3.1.3	Kafka in der Public Cloud selbst betreiben	160
4.3.1.4	Kafka in Kubernetes	160
4.3.1.5	Kafka as a Service	161
4.3.2	Broker	161
4.3.3	Koordinationscluster	162
4.3.4	Monitoring und Logging	162
4.3.5	Zusätzliche Werkzeuge	163
4.3.6	Zusammenfassung	164
5	Anhang: Kafka-Testumgebung aufsetzen	165
5.1	Betriebssysteme	166
5.2	Kafka herunterladen	166
5.3	Zookeeper starten	166
5.4	Kafka starten	168
5.5	Einzelne Broker stoppen	170
5.6	Umgebung aufräumen	170
Register	171	

Geleitwort

Technologie transformiert unseren Alltag und unsere Gesellschaft wie nie zuvor. Traditionelle, erfolgreiche Konzerne müssen sich neu erfinden und digitale Prozesse und Innovationen anbieten, um langfristig konkurrenzfähig zu bleiben. Ein wichtiger und stetig wachsender Teilbereich davon ist die Verarbeitung von Daten in Echtzeit: Ansonsten hat der Kunde bereits den Laden verlassen, bevor Empfehlungen oder Coupons via Push-Notification geschickt wurden. Ansonsten ist der Betrug passiert, bevor der Algorithmus warnen kann. Ansonsten ist die Maschine defekt, bevor die Anomalien der Sensoren erkannt wurden.

Diese Transformation geschieht über alle Industrien hinweg: Aus Läden erwachsen Webshops, Banken werden in Software-Services verwandelt, Taxibestellungen laufen über Software und auch Autos bestehen zu einem Großteil mittlerweile daraus. Wir können also zusammenfassend sagen: Unternehmen verwandeln sich in Daten-Infrastrukturen. Die Möglichkeit, diese Daten in Echtzeit verarbeiten zu können, ist essenziell für den zukünftigen Unternehmenserfolg. Eine moderne Infrastruktur für die Datenverarbeitung in Echtzeit ist Pflicht für den zukünftigen Unternehmenserfolg.

Paradigmenwechsel hin zu „Data in Motion“ mit Apache Kafka

Die Cloud ist die Zukunft für Datenzentren: flexibel, elastisch, skalierbar, immer auf dem neuesten Stand der Technik. Für die Datenverarbeitung findet mit Daten-Streaming gerade ein ähnlicher Paradigmenwechsel statt: Echtzeit, agil, skalierbar, Entkopplung verschiedener Anwendungen und Technologien.

Beim Daten-Streaming werden Daten kontinuierlich in Bewegung verarbeitet („Data in Motion“), anstatt erst alles in einer Datenbank, einem Data Warehouse oder einem Data Lake zu speichern, wo die Daten dann irgendwann (oftmals zu spät) in einem Batch-Prozess („Data at Rest“) verarbeitet werden.

Das Open Source Framework Apache Kafka hat sich dabei als De-facto-Standard für „Data in Motion“ etabliert. Vor über zehn Jahren wurde Kafka bei LinkedIn entwickelt, um sehr große Mengen von Logdaten („Big Data“) zu verarbeiten. Heute existieren zahlreiche verschiedene Anwendungsfälle für Kafka. Darunter fallen neben den analytischen Szenarien auch viele transaktionale Deployments wie Zahlungsplattformen, Trading-Apps, Retail-Anwendungen, Location-based Mobility Services, Supply Chain Management oder das operative Management von 5G Telco-Infrastrukturen.

Kafka ist so beliebt, weil es mehrere relevante Charakteristika bietet, die bisher auf verschiedene Frameworks oder Produkte verteilt waren: Messaging, Storage und Caching sowie Datenintegration (via Kafka Connect) und Datenverarbeitung (via Kafka Streams). Die Kombination aus hoher Skalierbarkeit, Zero Downtime und Zero Data Loss ermöglicht den Aufbau transaktionaler und analytischer Lösungen mit einer einzigen hochverfügbaren Infrastruktur. Dies differenziert Kafka von existierenden Messaging- (MQ), Integrations- (ETL/ESB) und Datenverarbeitungstools.

Kafka-Ökosystem: APIs, Tools, Automatisierung und SaaS

Die große weltweite Kafka Community erweitert das Kafka-Ökosystem um zusätzliche Skripte, Frameworks und kommerzielle Produkte. Die Schema Registry für Data Governance, ksqlDB als SQL-Abstraktionsebene für Stream Processing, kafkacat als Command Line Utility, cloud-native Kubernetes-Operatoren oder APIs für Programmiersprachen jenseits von der JVM wie beispielsweise C++, Go oder JavaScript ermöglichen den Aufbau von Kafka als zentrales Echtzeit-Nervensystem für das Unternehmen. In der Cloud besteht sogar die Möglichkeit, den Betrieb komplett auszulagern, da ein Serverless SaaS wie Confluent Cloud dem Nutzer unter kritischen SLAs ermöglicht, sich auf seine Kafka-Anwendungen zu fokussieren.

Moderne Microservice- und Data-Mesh-Architekturen sind mittlerweile der Standard, um agile Entwicklung sowie einen automatisierten Betrieb von Anwendungen zu ermöglichen. Kafka ermöglicht – im Gegensatz zu REST und Webservice-Architekturen – eine richtige Entkopplung verschiedener Anwendungen mittels Domain-driven Design (DDD). Auch in Zukunft wird nicht jede Anwendung Echtzeit sein. Nicht nur Legacy-Systeme, sondern auch Reporting für Business Intelligence und Modeltraining für Machine Learning bleiben batch-basiert. Kafka ermöglicht Backpressure Handling und unterstützt dadurch sowohl Echtzeit- als auch Batch- und Request-Response-Schnittstellen mit einer einzigen Datendrehscheibe.

Hybride Architekturen mit mehreren Kafka-Clustern sind dabei nicht die Ausnahme, sondern die Regel. Dabei gibt es diverse Anwendungsfälle wie beispielsweise Disaster Recovery, Aggregation, Migration oder globale Projekte. Der Betrieb erfolgt in unterschiedlichen Infrastrukturen, beispielsweise im Data Center, in (Multi-)Cloud oder auch für Edge Computing in einer Fabrik, Maschine oder Drohne. Bidirektionale Replikation in Echtzeit erlaubt analytische als auch transaktionale Kommunikation mit all den Vorteilen des Kafka-Protokolls.

Beispiele aus der Industrie

Abschließend noch ein paar Beispiele für Kafka-Deployments aus verschiedenen Branchen für verschiedene Anwendungsfälle, um die Vielfältigkeit von Kafka zu veranschaulichen:

Die Bank NORD/LB baut ihre konzernweite IT-Transformation auf Kafka auf, um Szenarien wie Betrugserkennung, Kundenbindung und Handelsplätze zu optimieren

Die Versicherung Generali verwendet Confluent, um Hunderte von Legacy-Datenbanken via Change Data Capture (CDC) in der neuen Kubernetes-basierten Anwendungslandschaft zu integrieren.

Die Deutsche Bahn nutzt das Kafka-Ökosystem, um in dessen Reiseinformationssystem Ereignisse aus diversen Schnittstellen zu integrieren und in Echtzeit zu korrelieren.

Der Zulieferer Bosch setzt auf Confluent Cloud für die asynchrone Kommunikation zwischen Anwendungen und Microservices in dessen IoT-Projekten für Echtzeit-Streaming und persistente Speicherung von Ereignissen.

Dieses Buch gibt einen sehr guten Überblick über die verschiedenen Komponenten von Kafka. Dabei werden die Grundkonzepte, APIs und Architekturen beschrieben, sodass jeder sein nächstes Kafka-Projekt erfolgreich gestalten kann. Viel Spaß und Erfolg dabei.

Kai Wöhner

Field CTO bei Confluent

Über das Buch

Wir haben lange nach einem Einstiegswerk für Apache Kafka¹ auf Deutsch gesucht. Als uns dann der Hanser Verlag gefragt hat, ob wir es schreiben wollen, war der Beschluss schnell gefasst. Nach einigen Startschwierigkeiten und mehr Zeit als geplant, halten Sie nun unser Buch in der Hand. Danke für Ihr Vertrauen.

Wir wissen aus eigener Erfahrung, dass es einfach ist, sich mit Kafka selbst zu überfordern. Gute Informationen zu Kafka auf Deutsch zu finden ist nicht immer leicht. In diesem Buch möchten wir Sie dabei unterstützen, einen guten Einstieg in Kafka zu finden, und Ihnen alles auf den Weg mitgeben, was Sie für Ihre Kafka-Mission benötigen.

Dieses Buch basiert auf den Schulungen, die wir für unsere Kunden geben. Uns ist es wichtig, nicht nur die fachlichen Grundlagen zu vermitteln, sondern auch unsere Erfahrungen zu teilen, wie Unternehmen Kafka erfolgreich bei sich einsetzen und was Sie dafür zusätzlich benötigen.

Während wir dieses Buch schreiben, befindet sich die Apache-Kafka-Welt im Umbruch. Neben zahlreichen Änderungen in Kafka 3.0, wird Zookeeper durch eine eigene Clusterkoordinationslösung ersetzt. Bis Redaktionsschluss wurde Kafka 3.0 noch nicht veröffentlicht, aber wir erwarten die Veröffentlichung in Kürze. Wir haben mit großer Sorgfalt darauf geachtet, dass alle relevanten zu erwartenden Änderungen im Buch vermerkt sind. Wir werden auf der Seite <https://streamcommit.de/buch/updates/> mögliche Fehler und unerwartete Änderungen vermerken.

Anatoly Zelenin und Alexander Kropp

August 2021

■ Wer sollte das Buch lesen?

Wir haben uns beim Schreiben dieses Buches an unseren Schulungsteilnehmern orientiert und möchten insbesondere diesen Zielgruppen das Bestmögliche auf ihrem Weg mitgeben. Wir glauben, dass dieses Buch allen ITlern, die sich mit Kafka beschäftigen möchten, einen guten Startpunkt gibt.

¹⁾ Apache Kafka <https://kafka.apache.org/> ist eine eingetragene Marke der Apache Software Foundation <https://apache.org/>.

IT-ArchitektInnen unterstützen wir dabei zu verstehen, wie sich Kafka in die Unternehmenslandschaft eingliedern kann und welche Konzepte sich bewährt haben. AdministratorInnen verstehen nicht nur die Kafka-Grundlagen, sondern lernen, wie sich Kafka am besten betreiben lässt und auch wie sie ihre Teams dabei unterstützen können, das Beste aus Kafka zu machen. Für EntwicklerInnen geben wir hier einen guten Überblick über die Konzepte und einen sehr kurzen Einstieg in die Programmierung mit Kafka. Dieses Buch ist aber kein Entwickler-Handbuch, da dies ein eigenes Buch wäre. Wir glauben auch, dass technische Teamleiter in diesem Buch viele interessante Konzepte lernen werden, um ihr Team bestmöglich zu unterstützen und Kafkas Wesen besser zu verstehen. Auch wenn Sie sich zu keiner dieser Gruppen zugehörig fühlen, wünschen wir Ihnen viel Freude bei der Lektüre dieses Werkes.

■ Konventionen im Buch

Kursiv stellen wir neu eingeführte Begriffe dar.

Sämtliche Code-Beispiele sind in der Unix-Shell-Syntax geschrieben.

Code-Beispiele im Fließtext stellen wir in Monospace dar.

Ausführliche Code-Beispiele werden folgendermaßen formatiert:

```
# Ein Kommentar  
$ ein befehl zum eintippen  
Ausgabe des Befehls  
> Manche Befehle benötigen zusätzliche Eingaben.
```

Zeilen, die mit einer Rauten (#) anfangen, sind unsere Kommentare und können ignoriert werden. Zeilen, die mit einem Dollar-Symbol (\$) beginnen, sind jene, die wir in unsere Kommandozeile eintippen können (ohne das Dollar-Symbol). Alles, was einzutippen ist, wird **fett** formatiert.

■ Nutzung von Code-Beispielen und Abbildungen

Dieses Buch enthält zahlreiche Code-Beispiele. Diese sollen das Verständnis erleichtern und die Konzepte praxisnah vermitteln. Code-Beispiele aus diesem Buch dürfen beliebig verwendet werden. Wir freuen uns über eine Quellenangabe, diese ist aber nicht Pflicht. Eine solche Quellenangabe kann wie folgt aussehen:

Apache Kafka: Von den Grundlagen bis zum Produktiveinsatz; von Anatoly Zelenin und Alexander Kropp; © 2022 Carl Hanser Verlag München

Um unsere Abbildungen in anderen Werken zu verwenden, bitten wir Sie, sich mit uns in Verbindung zu setzen:

buch@streamcommit.de

■ Danksagung

Unser Dank geht an unsere Familien, Freunde und Bekannte, die dieses Werk erst ermöglicht haben. Danke für eure Unterstützung in all den Jahren. Danke dafür, dass Ihr zu uns gehalten und uns wo immer möglich unterstützt habt!

Wir möchten der Apache Kafka Community, allen beteiligten Menschen und Unternehmen für die Erschaffung und die Pflege dieser Software danken.

Wir möchten Thomas Trepper dafür danken, dass er dieses Buchprojekt erst möglich gemacht hat und uns mit dem Hanser Verlag in Verbindung gebracht hat. Vom Hanser Verlag möchten wir insbesondere Sylvia Hasselbach für die Unterstützung und die fantastische Zusammenarbeit danken. Auch für die Geduld, die sie manchmal mit uns aufbringen musste.

Wir danken unseren Schulungsteilnehmern und Kunden für die Unterstützung und die Erfahrungen, die wir gemeinsam mit ihnen machen durften. Ohne ihre Unterstützung wäre das Buch weder finanziell noch inhaltlich zu stemmen gewesen.

Unser Dank geht auch an all die wunderbaren Menschen, die uns beim Erstellen des Buches direkt unterstützt haben. An Kai Wöhner für das Vorwort. An Thomas Natzschka für die textuelle Unterstützung. Natürlich auch an all diejenigen, die uns mit ihrem Feedback zum Buch unterstützt haben und es Korrektur gelesen haben: Inga Blundell, Walter Forkel, Daniela Griesinger, Tobias Heller, Vincent Latzko, Andrej Olunczek, Elin Rixmann, Thomas Trepper und David Weber.

1

Einleitung

Sie haben einen soliden Abschluss in Raketenwissenschaft? Nein? Nun ja, falls Ihre Rakete jedoch den Namen Apache Kafka trägt, dann haben Sie nach der Lektüre dieses Buches einen Versprochen!

Zuallererst möchten wir, die Autoren dieses Buches, Ihnen jedoch danken. Wir danken Ihnen, dass Sie sich in Sachen Apache Kafka weiterbilden möchten und sind uns sicher, dass Sie mit den kommenden Inhalten Höhenflüge erleben und anschließend auch erzeugen werden. Um gleich mal bei unserer Metapher mit der Rakete zu bleiben.

Es ist dabei ganz gleich, ob Sie vor dem Aufschlagen dieses Buches noch nie etwas über Apache Kafka gehört oder ob Sie schon einige Erfahrungen mit Kafka gesammelt haben. Oder ob Sie vielleicht sogar an einer unserer Schulungen teilgenommen haben. Wir sind uns in jedem Fall sicher, dass Sie in den kommenden Kapiteln viel Nützliches für sich mitnehmen können.

Liegt Ihre Expertise in der IT oder leiten Sie ein Team und möchten Apache Kafka dort einsetzen? Oder Sie tun es bereits und möchten nun verstehen, was Kafka konkret ist und wie Sie es bestmöglich nutzen können? Gratulation, denn auch Sie sind hier absolut richtig, denn Sie werden nach der Lektüre dieses Buches nicht nur einiges an Expertise gesammelt haben, um Kafka erfolgreich zu betreiben und einzusetzen, sondern auch, um Ihr Team dabei zu unterstützen, wenn es mit Fragen oder Sorgen rund um Kafka zu Ihnen kommt. Natürlich nur so lange, wie es überhaupt Sorgen und Probleme mit Kafka hat. Denn auch diese lassen sich für und mit Ihrem Team lösen.

Und selbstverständlich hoffen wir auch, dass Ihnen die Beschäftigung mit den Inhalten Spaß bereitet. Scheuen Sie sich bitte nicht, uns bei Fragen oder Anregungen zu kontaktieren. Sie erreichen uns per E-Mail unter

buch@streamcommit.de

oder im Web unter

<http://streamcommit.de/buch>.

■ 1.1 Einführung in Apache Kafka

Aber was hat es denn nun überhaupt mit Apache Kafka auf sich? Diesem Programm, das quasi jeder namhafte (deutsche) Automobilhersteller einsetzt? Diese Software, dank derer wir mit unseren Schulungen und Workshops kreuz und quer durch Europa gereist sind und viele unterschiedliche Branchen – von Banken, Versicherungen, Logistik-Dienstleistern, Internet-Start-ups, Einzelhandelsketten bis hin zu Strafverfolgungsbehörden – kennengelernt haben? Warum setzen so viele unterschiedliche Unternehmen (und Behörden) Apache Kafka ein?

Wir teilen die Einsatzgebiete von Apache Kafka in zwei Kategorien ein: Die erste Gruppe von Anwendungsfällen ist die, für die Kafka einst gedacht war. Apache Kafka wurde nämlich anfangs bei LinkedIn entwickelt, um alle Events, die auf der LinkedIn-Website anfallen, in das zentrale Data Warehouse zu bewegen. LinkedIn war auf der Suche nach einem skalierbaren und auch bei sehr hoher Last performanten Messaging-System und hat letzten Endes Kafka dafür kreiert. So setzen heute sehr viele Unternehmen Kafka ein, um große Datenmengen von A nach B zu bewegen. Der Fokus liegt oft auf der benötigten Performance, der Skalierbarkeit Kafkas, aber natürlich auch auf der Zuverlässigkeit, die Kafka für die Zustellbarkeit und Persistierung der Nachrichten bietet.

Eine der Kernideen, die Kafka von klassischen Messaging-Systemen jedoch unterscheidet, ist, dass Kafka Daten auf Datenträgern persistiert. So können wir Daten in Kafka aufbewahren und einmal geschriebene Daten nicht nur mehrfach lesen, sondern auch Stunden, Tage oder gar Monate nachdem diese Daten geschrieben wurden.

Dies ermöglicht die zweite Kategorie der Apache-Kafka-Anwendungsfälle: Immer mehr Unternehmen nutzen Kafka als zentrales Werkzeug, um nicht lediglich Daten zwischen unterschiedlichsten Services und Anwendungen auszutauschen, sondern wenden Kafka als zentrales Nervensystem an, um mit Daten innerhalb von Unternehmen zu agieren.

Was aber meinen wir damit, dass Kafka das zentrale Nervensystem für Daten sein kann? Die Vision ist (wie in Bild 1.1), dass jedes Ereignis, das in einem Unternehmen stattfindet, in Kafka abgespeichert wird. Jeder andere Service (der natürlich die Berechtigung hat) kann nun auf dieses Ereignis asynchron reagieren und dieses Ereignis weiterverarbeiten.

Wir sehen zum Beispiel in vielen Unternehmen den Trend, dass es eine Trennung zwischen sogenannten *Altsystemen*, die für bestehende Geschäftsprozesse und Geschäftsmodelle essenziell sind, und der sogenannten *Neuen Welt* gibt, wo mit agilen Methoden neue Dienste entwickelt werden, die auch in Software abgebildet werden müssen. Oftmals setzen Unternehmen Kafka ein, um nicht nur als Schnittstelle zwischen alten und neuen Systemen zu agieren, sondern auch, um den neuen Services zu ermöglichen, untereinander Nachrichten auszutauschen.

Das liegt daran, dass Altsysteme oftmals nicht den neuen Anforderungen unserer KundInnen gewappnet sind. Batch-Systeme können den Drang nach Informationen, die immer „Sofort“ verfügbar sein sollen, nicht erfüllen. Wer möchte zum Beispiel heutzutage noch einen Tag oder gar mehrere Wochen darauf warten, dass sich der Kontostand nach einer Kreditkartentransaktion aktualisiert? Wir erwarten mittlerweile, dass wir unsere Pakete in Echtzeit verfolgen können. Moderne Pkw produzieren Unmengen von Daten, die insbesondere für die Vorbereitung auf das autonome Fahren an die Konzernzentrale geschickt und ausgewertet werden sollen. Kafka kann all diese Unternehmen dabei unterstützen, von

einer batch-orientierten Verarbeitung hin zu einer Datenverarbeitung in (nahezu) Echtzeit zu gelangen.

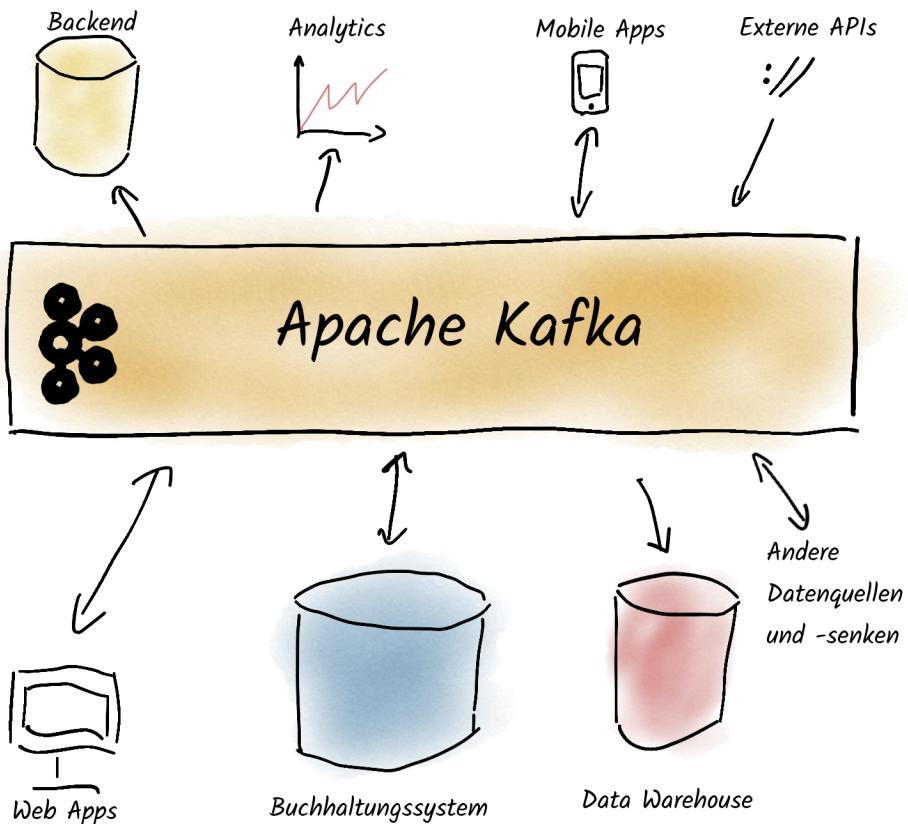


Bild 1.1 Kafka als zentrales Nervensystem für Daten im Unternehmen. Jedes Ereignis, welches im Unternehmen stattfindet, wird in Kafka gespeichert. Andere Services können auf diese Ereignisse asynchron reagieren und sie weiterverarbeiten.

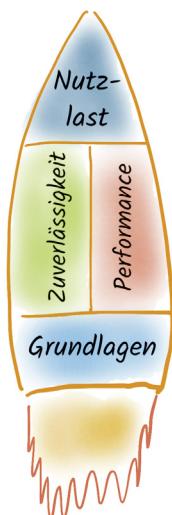
Aber auch die Art und Weise, wie wir Software schreiben, verändert sich. Statt immer mehr Funktionalität in monolithische Services zu stecken und dann diese wenigen Monolithen miteinander mittels Integration zu verbinden, brechen wir unsere Services in *Microservices* auf, um unter anderem die Abhängigkeit zwischen den Teams zu reduzieren. Dafür benötigen wir jedoch eine Art und Weise für den Datenaustausch, die möglichst asynchron ist. Dadurch können Services, auch wenn ein Service gerade in Wartung ist, unabhängig von diesem weiter funktionieren. Wir benötigen Methoden für die Kommunikation, die es erlauben, dass sich die Datenformate in einem Service unabhängig von anderen Services entwickeln können. Auch hierbei kann uns Apache Kafka unterstützen.

Ein anderer Trend, der vor allem durch Virtualisierung und immer breiteren Einsatz von Cloud-Architekturen ausgelöst wurde, ist das Zurückgehen von Spezialhardware. Es gibt im Gegensatz zu anderen Messaging-Systemen keine *Kafka-Appliances*. Kafka läuft auf handelsüblicher Hardware und benötigt keine ausfallsicheren Systeme. Kafka selbst ist so ent-

wickelt, dass es gut mit Ausfällen von Teilsystemen zurechtkommt. Dadurch ist die Zustellung von Nachrichten zuverlässig, auch wenn in unserem Rechenzentrum vielleicht gerade Chaos ausbricht.

Wie aber erreicht Kafka diese Zuverlässigkeit und Performance? Wie können wir Kafka für unsere Anwendungsfälle verwenden, und was ist beim Betrieb Kafkas zu beachten? Die Antworten auf diese Fragen und vieles mehr möchten wir Ihnen auf der Reise durch dieses Buch geben.

Wir finden, dass sich Kafka auf gewisse Weise gut mit einer Rakete vergleichen lässt: Sie ist in aller Munde, ist zuverlässig, performant, aber auch bekannt für ihre komplexe Bauart. Wir werden die Raketenanalogie ein wenig ausreizen und Sie mit unserer Kafka-Rakete auf eine Mission mitnehmen.



Deswegen möchten wir in Kapitel 2 die Konzepte unserer Apache Kafka-Rakete vorstellen. Ziel einer jeden Rakete ist nämlich nie die Rakete selbst. Es geht vielmehr darum, unsere Nutzlast in den Orbit, zum Mond oder gar zum Mars zu befördern. Bei Kafka ist nie das Ziel, Kafka selbst einzusetzen, sondern unser Business zu unterstützen, indem wir Nachrichten zuverlässig und performant zwischen Systemen austauschen. Wie diese Nachrichten aussehen, wie wir diese Nachrichten einteilen und verschicken können, entdecken wir im Kapitel 2.1 „Payload“. Wie auch bei echten Raketen können wir Kafka nicht erfolgreich auf lange Zeit betreiben, wenn wir die Grundlagen und die Hintergründe des Systems nicht verstehen. Kafka ist nämlich ein verteiltes Log, und was dies bedeutet, das erfahren wir im Kapitel 2.2 „Kafka als verteiltes Log“. Dieses verteilte Log muss zwei Kernanforderungen erfüllen: Nachrichten zuverlässig Zustellen – und dies darüber hinaus sehr schnell. Wir schauen uns das genauer in den Kapiteln 2.3 „Zuverlässigkeit“ und 2.4 „Performance“ an.

In Kapitel 3, nämlich dem Kafka Deep Dive, gehen wir weiter in die Tiefe. Wir sehen uns im Kapitel 3.1 „Verbindung zu Kafka“ an, wie unsere Kafka-Clients sich mit Kafka verbinden, im Kapitel 3.2 „Nachrichten produzieren und persistieren“, wie Nachrichten produziert und gespeichert werden. Im Kapitel 3.3 „Nachrichten konsumieren“ verdeutlichen wir, wie wir diese Nachrichten auch wieder lesen können. Wir setzen uns im Kapitel 3.4 „Nachrichten aufräumen“ auch damit auseinander, wie wir Nachrichten wieder löschen, und im Kapitel 3.5 „Cluster-Management“, wie Kafka als verteiltes System sich selbst koordiniert. Zu guter Letzt widmen wir uns in diesem Kapitel auch den Verarbeitungsgarantien und Transaktionen.

Im letzten Kapitel dieses Buches, „Kafka im Unternehmenseinsatz“, möchten wir darauf eingehen, wie Kafka erfolgreich in Unternehmen angewendet wird und was es abseits von Kafka für einen erfolgreichen Einsatz benötigt. Wir teilen mit Ihnen in diesem Kapitel auch einige Erfahrungen, die wir mit Kafka gemacht haben. Wir sehen uns im Kapitel 4.1 „Kafka-Ökosystem“ genauer das Ökosystem um Kafka an. Das heißt, wie wir Kafka mit anderen Systemen verbinden und Daten in Kafka verarbeiten. Für Architekten ist es immer wichtig zu wissen, wie sich Kafka mit anderen Systemen vergleicht. Das möchten wir im Kapitel 4.2 „Vergleich mit anderen Technologien“ schildern.

Zu guter Letzt haben wir im Kapitel 4.3 „Kafka-Referenzarchitektur“ noch weitere Empfehlungen für den Einsatz von Kafka gesammelt. Welche Hardware wird benötigt? Wo können

wir Kafka betreiben? Wie automatisieren wir den Betrieb? Die Antworten auf all diese Fragen werden besprochen.

Aber bevor wir uns in all die angekündigten Details stürzen, lassen Sie uns gemeinsam unsere Apache Kafka-Rakete starten und einen Blick darauf werfen, wie sie sich in einem ersten Testflug bewährt. Jede Raketenwissenschaft muss schließlich irgendwann einmal beginnen. Auch die Ihre.

In diesem Sinne: 3, 2, 1 ... Zündung!

■ 1.2 Erste Schritte mit Kafka

Bevor wir uns in die Details stürzen, starten wir gemeinsam unsere *Kafka-Rakete* und wagen einen ersten Testflug. Wir gehen davon aus, dass Kafka schon installiert ist. Wir haben die genaue Installationsanleitung im Anhang beschrieben.

Bevor wir ergründen, warum die Kafka-Rakete überhaupt fliegen kann und wie sie es tut, zünden wir gemeinsam die Triebwerke und machen einen kleinen Testflug, wie in Bild 1.2 dargestellt.

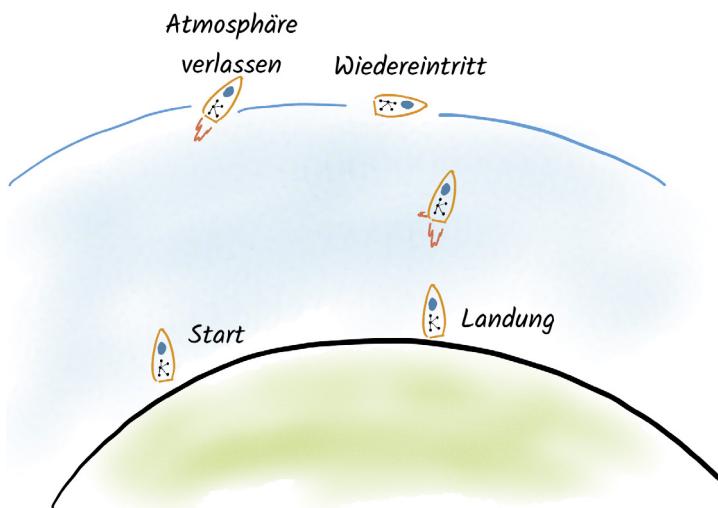


Bild 1.2 Unser Beispielflug mit unserer Apache Kafka-Rakete

Für unseren Flug ins All starten wir unsere (natürlich wiederverwendbare) Rakete, verlassen kurz die Atmosphäre, um den Wiedereintritt zu üben, und landen die Rakete sanft und sicher auf einem Landeplatz.

Wir möchten hierzu in Kafka alle Flugphasen erfassen und erstellen dazu als Erstes ein *Topic*. Topics sind insofern ähnlich zu Tabellen in Datenbanken, als dass wir in Topics eine Ansammlung von Daten zu einem bestimmten Thema abspeichern. In unserem Fall sind es Flugdaten, daher nennen wir das Topic entsprechend *flugdaten*:

```
$ kafka-topics.sh \
  --create \
  --topic flugdaten \
  --partitions 1 \
  --replication-factor 1 \
  --bootstrap-server
localhost:9092
Created topic flugdaten.
```

Mit dem Befehl `kafka-topics.sh` verwalten wir unsere Topics in Kafka. Hier weisen wir Kafka mit dem Argument `--create` an, das Topic `flugdaten` (`--topic flugdaten`) zu erstellen. Zunächst starten wir mit einer *Partition* (`--partitions 1`) und ohne die Daten zu replizieren (`--replication-factor 1`). Als Letztes geben wir an, zu welchem *Kafka-Cluster* sich `kafka-topics.sh` verbinden soll. In unserem Fall nutzen wir unser lokales Cluster, welches standardmäßig auf dem Port 9092 hört (`--bootstrap-server localhost:9092`). Der Befehl bestätigt uns die erfolgreiche Erstellung des Topics. Sollten wir hier Fehler bekommen, liegt dies oft daran, dass Kafka noch nicht gestartet und somit nicht erreichbar ist, oder daran, dass das Topic schon existiert.

Wir haben jetzt also einen Ort, an dem wir unsere Daten abspeichern können. Der Bordcomputer der Rakete schickt uns kontinuierlich Aktualisierungen zum Flugzustand der Rakete. Für unsere Simulation nutzen wir dafür das Kommandozeilenwerkzeug `kafka-console-producer.sh`. Dieser *Producer* und auch andere nützliche Werkzeuge werden mit Kafka direkt ausgeliefert. Der Producer verbindet sich zu Kafka, nimmt Daten von der Kommandozeile entgegen und schickt sie als Nachrichten in ein Topic (konfigurierbar über den Parameter `--topic`). Schreiben wir die Nachricht `Countdown gestartet` in unser eben erstelltes Topic `flugdaten`:

```
$ echo "Countdown gestartet" | kafka-console-producer.sh \
  --topic flugdaten \
  --bootstrap-server=localhost:9092
```

Unsere Bodenstation möchte diese Daten jetzt lesen und zum Beispiel auf einem großen Bildschirm ausgeben, damit wir sehen, ob die Rakete wirklich so funktioniert, wie wir das von ihr erwarten. Schauen wir uns also an, was bisher passiert ist. Um unsere eben gesendete Nachricht wieder zu lesen, starten wir den `kafka-console-consumer.sh`, welcher ebenfalls Teil der Kafka-Familie ist:

```
$ timeout 10 kafka-console-consumer.sh \
  --topic flugdaten \
  --bootstrap-server=localhost:9092
Processed a total of 0 messages
```

Wenn wir den `kafka-console-consumer.sh` starten, läuft der standardmäßig weiter, bis wir ihn aktiv (zum Beispiel mit **STRG+C**) abbrechen. Das wäre auch das gewünschte Verhalten, wenn wir den aktuellen Zustand der Rakete wirklich irgendwo anzeigen wollten. In unserem Beispiel nutzen wir den Befehl `timeout`, damit der Consumer automatisch nach spätestens zehn Sekunden abbricht. Bei dem Consumer müssen wir wieder angeben, welches Topic dieser benutzen soll (`--topic flugdaten`).

Etwas überraschend wird keine Nachricht angezeigt. Dies liegt daran, dass der `kafka-console-consumer.sh` standardmäßig am Ende des Topics zu lesen beginnt und nur neue Nachrichten ausgibt. Um auch bereits geschriebene Daten anzuzeigen, müssen wir das Flag `--from-beginning` benutzen:

```
$ timeout 10 kafka-console-consumer.sh \
--topic flugdaten \
--from-beginning \
--bootstrap-server=localhost:9092
Countdown gestartet
Processed a total of 1 messages
```

Diesmal sehen wir die Nachricht *Countdown gestartet!* Was ist also passiert? Mit dem Befehl `kafka-topics.sh` haben wir das Topic `flugdaten` in Kafka erstellt und mit dem `kafka-console-producer.sh` die Nachricht *Countdown gestartet* produziert. Diese Nachricht haben wir dann mit dem `kafka-console-consumer.sh` wieder gelesen. Dieser Datenfluss ist in Bild 1.3 dargestellt. Ohne weitere Angaben fängt der `kafka-console-consumer.sh` immer am Ende an zu lesen, das heißt, wenn wir alle Nachrichten lesen möchten, müssen wir das Flag `--from-beginning` benutzen.



Bild 1.3 In unserem Beispiel produzieren wir Daten mit dem `kafka-console-producer.sh` in das Topic `flugdaten`, und mit dem `kafka-console-consumer.sh` können wir diese Daten wieder lesen.

Interessanterweise, beziehungsweise anders als in vielen Messaging-Systemen, können wir Nachrichten nicht nur einmal lesen, sondern beliebig oft. Dies können wir nutzen, um zum Beispiel mehrere unabhängige Bodenstationen mit dem Topic zu verbinden, sodass alle die gleichen Daten lesen können. Oder es gibt unterschiedliche Systeme, die alle die gleichen Daten benötigen. Wir können uns vorstellen, dass wir nicht nur einen Anzeigebildschirm haben, sondern zusätzlich andere Services, wie zum Beispiel einen Service, der die Flugdaten mit aktuellen Wetterdaten abgleicht und entscheidet, ob etwas unternommen werden muss. Vielleicht möchten wir aber auch nach dem Flug die Daten analysieren und benötigen dafür die historischen Flugdaten. Dazu können wir den Consumer einfach mehrmals ausführen und bekommen jedes Mal das gleiche Ergebnis.

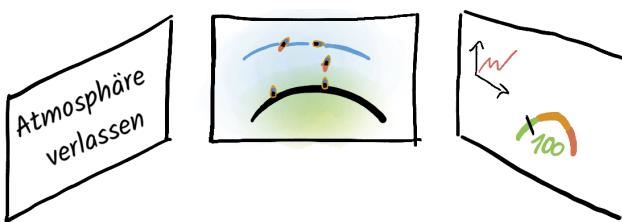


Bild 1.4 So könnte unser Kontrollzentrum für unser Beispiel aussehen.

Wir möchten jetzt aber gerne den aktuellen Zustand der Rakete in unserem Kontrollzentrum, welches in Bild 1.4 schemenhaft dargestellt ist, anzeigen, und zwar so, dass sich die Anzeige sofort aktualisiert, wenn es neue Daten gibt. Dazu starten wir den `kafka-console-consumer.sh` (ohne Timeout) in einem Terminal-Fenster. Sobald neue Daten verfügbar sind, holt sich der Consumer diese von Kafka und zeigt sie auf der Kommandozeile an:

```
# Nicht vergessen: STRG+C nutzen, um den Consumer zu stoppen
$ kafka-console-consumer.sh \
  --topic flugdaten \
  --bootstrap-server=localhost:9092
```

Um den Producer auf Raketen-Seite zu simulieren, starten wir jetzt den `kafka-console-producer.sh`. Der Befehl stoppt so lange nicht, bis wir **STRG+D** drücken und damit das *EOF-Signal* an den Producer senden:

```
# Nicht vergessen: STRG+D nutzen, um den Producer zu stoppen
$ kafka-console-producer.sh \
  --topic flugdaten \
  --bootstrap-server=localhost:9092
```

Der `kafka-console-producer.sh` schickt pro Zeile, die wir schreiben, eine Nachricht an Kafka. Das heißt, wir können nun Nachrichten in das Terminal mit dem Producer eintippen:

```
# Producer Fenster:
> Countdown beendet
> Liftoff
> Atmosphäre verlassen
> Vorbereitung Wiedereintritt
> Wiedereintritt erfolgreich
> Landung erfolgreich
```

Wir sollten diese auch zeitnah im Fenster mit dem Consumer sehen:

```
# Consumer Fenster:
Countdown beendet
Liftoff
Atmosphäre verlassen
Vorbereitung Wiedereintritt
Wiedereintritt erfolgreich
Landung erfolgreich
```

Stellen wir uns vor, dass ein Teil unserer Bodencrew im Homeoffice ist und den Flug von zu Hause aus verfolgen möchte. Dazu starten sie unabhängig voneinander ihren Consumer. Wir können das simulieren, indem wir in einem weiteren Terminalfenster einen `kafka-console-consumer.sh` starten, der alle Daten von Beginn an anzeigt:

```
# Fenster Consumer 2
$ kafka-console-consumer.sh \
  --topic flugdaten \
  --from-beginning \
  --bootstrap-server=localhost:9092
Countdown gestartet
Countdown beendet
Liftoff
Atmosphäre verlassen
Vorbereitung Wiedereintritt
Wiedereintritt erfolgreich
Landung erfolgreich
```

Wir sehen hier, dass Daten, die einmal geschrieben werden, parallel von mehreren Consumern gelesen werden können, ohne dass die Consumer miteinander reden müssen oder die Consumer sich vorher bei Kafka registrieren müssen. Kafka löscht erst einmal keine Daten, das heißt, dass wir auch später noch einen Consumer starten können, der historische Daten lesen kann.

Sagen wir nun, dass wir nicht nur eine Rakete gleichzeitig fliegen lassen möchten, sondern mehrere. Kafka hat damit kein Problem und kann ohne Weiteres Daten von vielen Producern gleichzeitig verarbeiten. Starten wir also einen weiteren Producer in einem weiteren Terminalfenster:

```
# Producer für Rakete 2
$ kafka-console-producer.sh \
    --topic flugdaten \
    --bootstrap-server=localhost:9092
> Countdown gestartet
```

Wir sehen alle Nachrichten von allen Producern in allen unseren Consumern in der Reihenfolge, in der die Nachrichten produziert wurden, auftauchen:

```
# Fenster Consumer 1
[...]
Landung erfolgreich
Countdown gestartet
# Fenster Consumer 2
[...]
Landung erfolgreich
Countdown gestartet
```

Nun ergibt sich aber das Problem, dass wir die Nachrichten von Rakete 1 und 2 nicht voneinander unterscheiden können. Wir könnten in jede Nachricht schreiben, von welcher Rakete die Nachricht geschickt wurde. Dazu aber später mehr.

Bevor wir weitermachen, sollten wir noch den Flug unserer zweiten Rakete abbrechen, da wir sie erst später starten lassen wollen:

```
# Producer für Rakete 2
[...]
> Countdown abgebrochen
```

Wir haben jetzt erfolgreich eine Rakete testweise gestartet und wieder gelandet. Dabei haben wir einige Daten in Kafka geschrieben. Dazu haben wir zuerst ein Topic *flugdaten* mit dem Kommandozeilenwerkzeug *kafka-topics.sh* erstellt, in welches wir alle Flugdaten für unsere Rakete schreiben. In dieses Topic haben wir mithilfe von *kafka-console-producer.sh* einige Daten produziert. In unserem Fall waren dies Informationen über den aktuellen Status der Rakete. Diese Daten konnten wir mithilfe des *kafka-console-consumer.sh* lesen und anzeigen. Wir sind sogar weiter gegangen und haben parallel mit mehreren Producern Daten produziert und mit mehreren Consumern Daten gleichzeitig gelesen. Mit dem Flag *--from-beginning* im *kafka-console-consumer.sh* waren wir in der Lage, auf historische Daten zuzugreifen. Wir haben so schon drei Kommandozeilenwerkzeuge kennengelernt, die mit Kafka ausgeliefert werden.

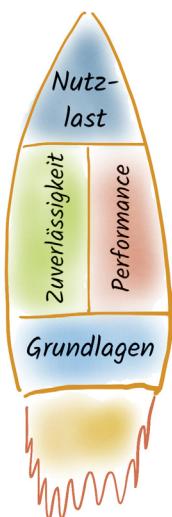
Nachdem wir diese Erfahrungen gesammelt haben, können wir nun alle geöffneten Terminals schließen. Producer beenden wir mit **STRG+D** und Consumer mit **STRG+C**. Dieses Beispiel soll nicht darüber hinwegtäuschen, dass Kafka überall da eingesetzt wird, wo größere Datenmengen verarbeitet werden. Aus unserer Schulungserfahrung wissen wir, dass Kafka bei vielen Automobilherstellern, Supermarktketten, Logistikdienstleistern und selbst in vielen Banken und Versicherungen intensiv eingesetzt wird.

Nachdem wir in diesem Kapitel einen ersten Überblick über Kafka erhalten und ein einfaches Testflug-Szenario durchexerziert haben, werden wir im nächsten Kapitel tiefer in die

Materie einsteigen und die Kafka-Architektur näher untersuchen. Wir werden uns anschauen, wie Kafka-Nachrichten strukturiert sind und wie genau sie in Topics organisiert werden. In diesem Zusammenhang werden wir uns auch mit der Skalierbarkeit und der Zuverlässigkeit von Kafka beschäftigen. Außerdem werden wir mehr über Producer, Consumer und das Kafka-Cluster selbst erfahren.

2

Konzepte



In der Einleitung haben wir uns einen Überblick über Kafka verschafft und Erfahrungen damit gesammelt, wie wir Nachrichten produzieren und konsumieren. Wir haben Kafka mit einer Rakete verglichen. Bei Raketen geht es darum, Nutzlasten in einen Orbit oder auf Himmelskörper zu transportieren. Die Rakete selbst ist nur dazu da, dieses Ziel so schnell und günstig wie möglich zu erreichen. Bei Kafka ist dies ähnlich. Es geht darum, Business-Probleme zu lösen. Im Fall von Kafka sind dies der Transport und die Speicherung von großen Mengen an Nachrichten. Auch hier drängt sich der Vergleich zu Raketen auf. Wir benötigen eine Menge an Konzepten und Infrastruktur, um unser Ziel möglichst schnell, sicher und kostengünstig zu erreichen.

In den folgenden Kapiteln werden wir unser Raketenmodell mit seinen unterschiedlichen Stufen genauer anschauen. Wir starten mit der eigentlichen Nutzlast, unseren Nachrichten, und den Nutzlastbehältern, den Topics. Sobald wir uns einen Überblick erarbeitet haben, was wir letztendlich transportieren möchten, bauen wir die Rakete Stück für Stück auf. Die Grundlage, um Raketen ins All zu schicken, ist ein sehr solides

Verständnis aller zugrundeliegenden Konzepte. Zum Glück ist Kafka keine Raketenwissenschaft, aber dennoch ist es sehr nützlich, einiges an Grundwissen zu besitzen, um Kafka erfolgreich zu betreiben.

Kafka ist darauf ausgelegt, auf günstiger, nichtspezialisierter Hardware zu laufen. Im Einzelnen ist diese Hardware oft zuverlässig genug, aber spätestens sobald wir viele Rechner zusammenschließen, sind Fehler und Ausfälle an der Tagesordnung. Kafka selbst legt dementsprechend viel Wert auf Zuverlässigkeit, um auch den Ausfall von größeren Teilsystemen ohne Weiteres zu verkraften. Einer unserer zwei Treibstofftanks der Modellrakete ist deshalb mit *Zuverlässigkeit* beschriftet. Der andere Tank heißt *Performance*. Kafka wurde ursprünglich dafür entwickelt, sehr große Datenmengen zwischen Systemen auszutauschen. Die Kafka-Entwickler mussten einige interessante Werkzeuge benutzen, um dies in Nahe-Echtzeit zu ermöglichen.

■ 2.1 Payload

Ganz egal wie groß eine Rakete ist oder wie schnell sie fliegen kann, am Ende zählt nur, dass sie die Nutzlast zuverlässig ans Ziel bringt. Die Nutzlast in Kafka besteht aus Daten, und genau hier setzt dieses Kapitel an. Wir werden uns ansehen, wie Kafka Daten verpackt und auch welche Art von Daten wir eigentlich mit Kafka transportieren.

2.1.1 Topics

Eine wichtige Komponente sind die Nutzlastbehälter. Diese stellen sicher, dass Nutzlasten unabhängig von ihrer Form und Struktur sicher transportiert werden können. Je nachdem, wie groß oder schwer eine Nutzlast ist, muss sie auf mehrere Behälter oder sogar auf mehrere Raketen aufgeteilt werden. Für den Aufbau der Internationalen Raumstation waren zum Beispiel ca. 40 Flüge nötig. Außerdem sind besonders wichtige Komponenten zur Ausfallsicherung mehrfach vorhanden. Das alles erfordert Planung. Diese Aufgabe wird in Kafka von *Topics* übernommen.

2.1.1.1 Anzeigen von Topics

Schauen wir uns das Topic `flugdaten` aus dem letzten Kapitel etwas genauer an:

```
$ kafka-topics.sh \
  --describe \
  --topic "flugdaten" \
  --bootstrap-server localhost:9092
Topic: flugdaten      PartitionCount: 1      ReplicationFactor: 1      Configs:
      Topic: flugdaten      Partition: 0      Leader: 3      Replicas: 3      Isr: 3
```

Wir nutzen den aus dem letzten Kapitel bekannten Befehl `kafka-topics.sh`, mit dem wir unsere Topics in Kafka verwalten. Mit dem Argument `--describe` können wir uns Topics genauer ansehen. Die Argumente `--topic` und `--bootstrap-server` sind uns ebenfalls bereits aus dem vorherigen Kapitel bekannt und dienen dazu, das Topic `flugdaten` (`--topic flugdaten`) auf unserem lokalen Kafka-Cluster (`--bootstrap-server=localhost:9092`) auszuwählen. Werfen wir nun einen Blick auf die Ausgabe. In der ersten Zeile finden wir allgemeine Informationen zu unserem `flugdaten`-Topic. Das Feld `PartitionCount` gibt die Anzahl an Partitionen in unserem Topic an und entspricht dem bei der Erstellung des Topics gesetzten Wert (`--partitions 1`). Der `ReplicationFactor` definiert, wie oft Nachrichten redundant gespeichert werden. Ein `ReplicationFactor` von 1 bedeutet, dass Nachrichten nicht repliziert werden. Es gibt also keinerlei Ausfallsicherheit.

Weitere Konfigurationseinstellungen für das Topic finden sich unter `Configs`. In unserem Beispiel ist das Feld leer, weil wir an der Standardkonfiguration nichts geändert haben. `Configs` werden wir im Verlauf des Buches auch noch ausführlicher behandeln. Alle weiteren Zeilen enthalten Informationen zu den jeweiligen Partitionen und den dazugehörigen `Replicas`. Da wir nur eine Partition haben, gibt es lediglich einen Eintrag zu Partition 0 (`Partition: 0`).

Um zu verstehen, was es mit den weiteren Zahlen, die bei den anderen Eigenschaften als Wert hinterlegt sind, auf sich hat, müssen wir einen kurzen Umweg über die Architektur

von Kafka gehen. Bisher haben wir immer von unserem Kafka-Cluster gesprochen. Was meinen wir damit eigentlich? Kafka besteht aus mehreren Komponenten, welche in Bild 2.1 dargestellt sind.

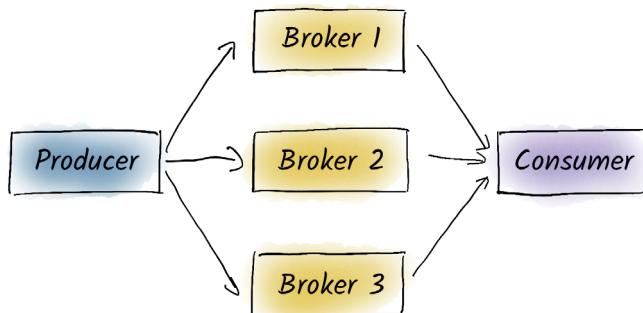


Bild 2.1 Eine typische Kafka-Umgebung besteht aus dem Kafka-Cluster selbst und den Producern und Consumern, die Daten nach Kafka schreiben und lesen. Vor Kafka 3.0 benötigten wir noch zusätzlich ein Zookeeper-Ensemble als Koordinationscluster.

Producer und *Consumer* haben wir bereits aktiv im letzten Kapitel kennengelernt. *Broker* sind verantwortlich für die Verarbeitung und Speicherung der Nachrichten im Kafka-Cluster.

Die weiteren Zahlen in unserer Topic-Beschreibung beziehen sich jeweils auf die ID unserer Broker. *Replicas* gibt an, auf welchen Brokern die Partition repliziert ist. Die Abkürzung *ISR* steht für *in-sync Replicas* und zeigt an, welche Broker auf dem aktuellen Stand sind. Das Feld *Leader* gibt an, welcher Broker die Hauptverantwortung für die Partition trägt. Da wir nur einen *ReplicationFactor* von 1 haben und daher die Partition nicht repliziert wird, findet sich in den jeweiligen Einträgen auch nur jeweils die ID eines unserer drei Broker. Kafka versucht bei der Erstellung eines neuen Topics, die Partitionen und *Replicas* gleichmäßig auf alle Broker aufzuteilen, um so eine gleichmäßige Lastenverteilung zu gewährleisten. Da wir bisher nur ein Topic erstellt haben, ist die Auswahl des Brokers zufällig. In unserem Beispiel wurde der Broker mit der ID 3 ausgewählt.

Eine wichtige Frage haben wir bisher außer Acht gelassen. Woher wissen wir, welche Topics es in unserem Kafka-Cluster gibt? Für unser Beispiel lässt sich die Frage zwar relativ einfach beantworten, denn wir haben bisher nur das Topic *flugdaten* erstellt. In der Praxis kann ein Kafka-Cluster aus sehr vielen Topics, die von verschiedenen Personen angelegt worden sind, bestehen. Um auch hier noch den Überblick zu behalten, können wir uns mit dem Befehl *kafka-topics.sh* alle Topics in unserem Kafka-Cluster anzeigen lassen:

```
$ kafka-topics.sh \
--list \
--bootstrap-server=localhost:9092
__consumer_offsets
flugdaten
```

Hierfür verwenden wir das Argument *--list* und geben wieder unser Kafka-Cluster an (*--bootstrap-server=localhost:9092*). Es spielt keine Rolle, von welchem Broker das Topic tatsächlich verwaltet wird. Alternativ könnten wir also auch mit unseren anderen

beiden Brokern sprechen (`--bootstrap-server=localhost:9093` bzw. `--bootstrap-server=localhost:9094`). Im Ergebnis fällt das Topic `_consumer_offsets` auf. Dieses Topic wurde automatisch von Kafka angelegt, was an den doppelten Unterstrichen am Anfang des Namens zu erkennen ist¹⁾. Dieses Topics speichert für jeden Consumer die aktuelle Leseposition. Wir können uns *Offsets* als eine Art Lesezeichen für unsere Consumer vorstellen. In unserem ersten Beispiel haben wir sogar bereits Offsets benutzt, erinnern wir uns kurz zurück:

```
# Fenster Consumer 2
$ kafka-console-consumer.sh \
  --topic flugdaten \
  --from-beginning \
  --bootstrap-server=localhost:9092
Countdown gestartet
Countdown beendet
Liftoff
Atmosphäre verlassen
Vorbereitung Wiedereintritt
Wiedereintritt erfolgreich
Landung erfolgreich
```

Standardmäßig fängt der `kafka-console-consumer.sh` immer am Ende eines Topics zu lesen an, das heißt, er liest nur neue Nachrichten. Durch das Flag `--from-beginning` verändern wir den Offset unseres Consumers und setzen ihn auf 0, wodurch auch alle bereits produzierten Nachrichten gelesen werden.

2.1.1.2 Erstellen, Anpassen und Löschen von Topics

Nachdem wir im letzten Abschnitt gelernt haben, wie wir uns Topics und deren Eigenschaften anzeigen lassen können, schauen wir uns nun genauer an, wie wir Topics erstellen und anpassen können. Damit wir frisch starten können, löschen wir als Erstes unser Topic `flugdaten`:

```
$ kafka-topics.sh \
  --delete \
  --topic "flugdaten" \
  --bootstrap-server=localhost:9092
```

Wir verwenden hierfür den Befehl `kafka-topics.sh` mit dem Argument `--delete`.

Wenn wir uns jetzt die Topics in unserem Kafka-Cluster anzeigen, stellen wir fest, dass das Topic `flugdaten` nicht mehr vorhanden ist.

```
$ kafka-topics.sh \
  --list \
  --bootstrap-server=localhost:9092
__consumer_offsets
```

Das `__consumer_offsets`-Topic wurde allerdings nicht gelöscht, denn dies ist ein internes Kafka-Topic und unabhängig von einzelnen Topics. Jedoch wurden die Offsets für das gelöschte Topic aufgeräumt. Falls wir jetzt mit dem Skript `kafka-topics.sh` versuchen wür-

¹⁾ Wobei wir theoretisch auch selbst Topics mit doppelten Unterstrichen anlegen könnten

den, unser gelöscht *flugdaten*-Topic aufzurufen, bekämen wir eine entsprechende Fehlermeldung:

```
$ kafka-topics.sh \
--describe \
--topic "flugdaten" \
--bootstrap-server localhost:9092
Error while executing topic command : Topic 'flugdaten' does not exist as expected
```

Nachdem unser Kafka-Cluster wieder, abgesehen vom *__consumer_offsets*-Topic, leer ist, können wir unser *flugdaten*-Topic neu erstellen:

```
$ kafka-topics.sh \
--create \
--topic "flugdaten" \
--replication-factor 2 \
--partitions 2 \
--bootstrap-server=localhost:9092
```

Im Prinzip hat sich im Vergleich zu dem Befehl aus der Einleitung nicht viel geändert. Wir verwenden wieder den uns mittlerweile gut bekannten Befehl `kafka-topics.sh` in Kombination mit dem Argument `--create`. Spannend wird es allerdings beim *ReplicationFactor* und der Anzahl an Partitionen. Im Vergleich zu unserem einführenden Beispiel wählen wir hier jeweils einen Wert von 2 anstelle von 1. Werfen wir nun mittels `--describe` einen genaueren Blick auf unser neues Topic *flugdaten*:

```
$ kafka-topics.sh \
--describe \
--topic "flugdaten" \
--bootstrap-server localhost:9092
Topic: flugdaten      PartitionCount: 2      ReplicationFactor: 2      Configs:
          Topic: flugdaten      Partition: 0      Leader: 3      Replicas: 3,2      Isr: 3,2
          Topic: flugdaten      Partition: 1      Leader: 1      Replicas: 1,3      Isr: 1,3
```

Wenig überraschend sehen wir hier nun einen `PartitionCount` und einen `ReplicationFactor` von 2. Interessanter sind die darauffolgenden beiden Zeilen. Jede Zeile steht hierbei für eine Partition, und die Partitionen sind mit null beginnend durchnummieriert. Partition 0 befindet sich auf den Brokern mit den IDs 3 und 2 (`Replicas: 3,2`), wobei Broker 3 hierbei die Rolle des Leaders einnimmt (`Leader: 3`) und somit die Hauptverantwortung für Partition 0 trägt. Partition 1 befindet sich ebenfalls auf Broker 3 und zusätzlich auf Broker 1 (`Replicas: 1,3`), wobei Broker 1 diesmal die Rolle des Leaders innehat (`Leader: 1`). Die Verteilung der einzelnen Partitionen und Replicas ist kein reiner Zufall, denn Kafka versucht, die Last gleichmäßig auf alle Broker zu verteilen. Ganz egal, wie oft wir das Topic löschen und wieder neu erstellen würden, in unserem kleinen Beispiel wäre nie dieselbe Broker Leader von beiden Partitionen gleichzeitig. Welcher Broker welche Partition anführt, kann sich allerdings sehr wohl ändern. Ähnlich verhält es sich mit der Verteilung der Replicas auf die Broker. Es würde nie dazu kommen, dass die insgesamt vier Replicas auf nur zwei Broker verteilt werden. Lediglich die exakte Aufteilung auf die Broker kann sich ändern.

Was aber passiert, wenn wir im Nachhinein feststellen, dass wir die Anzahl an Partitionen und Replicas ändern müssen, weil zum Beispiel die Performance nicht mehr ausreicht oder die Zuverlässigkeit ungenügend ist? Das Topic jedes Mal zu löschen und neu zu erstellen ist, abgesehen davon, dass es unpraktikabel ist, auch nicht immer möglich, denn wir wür-

den hierbei alle Daten verlieren. Zum Glück hat Kafka auch hierfür eine Lösung. Erhöhen wir zunächst die Anzahl der Partitionen auf 3:

```
$ kafka-topics.sh \
--alter \
--topic "flugdaten" \
--partitions 3 \
--bootstrap-server=localhost:9092
```

Um das Topic anzupassen, verwenden wir den `kafka-topics.sh`-Befehl mit dem Argument `--alter`. Wir wählen das Topic `flugdaten` (`--topic "flugdaten"`) und setzen die Anzahl an Partitionen auf 3 (`--partition 3`). Nachdem wir den Befehl ausgeführt haben, können wir die Änderungen genauer betrachten:

```
$ kafka-topics.sh \
--describe \
--topic "flugdaten" \
--bootstrap-server localhost:9092
Topic: flugdaten      PartitionCount: 3      ReplicationFactor: 2      Configs:
          Topic: flugdaten    Partition: 0    Leader: 3      Replicas: 3,2      Isr: 3,2
          Topic: flugdaten    Partition: 1    Leader: 1      Replicas: 1,3      Isr: 1,3
          Topic: flugdaten    Partition: 2    Leader: 2      Replicas: 2,3      Isr: 2,3
```

Wir können sehen, dass der `PartitionCount` auf 3 erhöht wurde. Außerdem sehen wir eine neue Zeile am Ende der Ausgabe, welche Informationen zur neu erstellten Partition 2 enthält. Die Auswahl der Leader und Replicas der bereits vorhandenen Partitionen wird durch `--alter` nicht verändert. In unserem Beispiel wurde für die neue Partition Broker 2 als Leader und zusätzlich Broker 3 als weiteres Replica ausgewählt. Kafka versucht bei der Erstellung eines neuen Topics, die Partitionen und Replicas gleichmäßig auf alle Broker aufzuteilen, um so eine gleichmäßige Lastenverteilung zu gewährleisten. Da wir bisher nur ein Topic erstellt haben und die Partitionen gleichmäßig verteilt sind, ist die Auswahl der Broker und auch des Leaders zufällig. Anstelle von Broker 2 hätte übrigens auch Broker 3 als Leader für Partition 2 ausgewählt werden können, womit auch die Leader ungleichmäßig verteilt gewesen wären. Hätten wir allerdings die Anzahl an Partitionen z.B. auf 5 erhöht, so wären die drei neuen Partitionen; ähnlich wie beim Erstellen eines Topics, gleichmäßig auf die Broker aufgeteilt worden.

Wenn wir den *Replication Factor* verändern wollen, müssen wir beim `kafka-topics.sh`-Befehl entweder das Argument `--replica-assignment` verwenden oder auf den Befehl `kafka-reassign-partitions.sh` zurückgreifen.



Tipp

Es ist nicht möglich, die Anzahl der Partitionen zu verringern, da wir ansonsten die Daten in den Partitionen verlieren würden. Wir können die Daten auch nicht vorher auf eine andere Partition verschieben, da das einem der Grundprinzipien von Kafka widersprechen würde. In einem späteren Kapitel werden wir auch genauer erfahren, warum.

2.1.2 Nachrichten

Wir haben im letzten Abschnitt die Topics, also unsere Nutzlastbehälter, kennengelernt. Es stellt sich weiter die Frage, was wir in den Nutzlastbehältern transportieren können. Topics verhalten sich ähnlich wie echte Nutzlastbehälter. Solange die Nutzlast kompatibel zum Behälter ist und weder die Maximalgröße noch das Maximalgewicht überschreitet, können wir prinzipiell alles transportieren. Im ersten Kapitel haben wir Zeichenketten, also Strings, an Kafka geschickt und von Kafka empfangen. Kafka ist im Grunde agnostisch bezüglich des Datentyps. Das heißt konkret, dass Kafka intern nur mit Byte-Arrays arbeitet und somit für alle möglichen Arten von *Nachrichten* geeignet ist.

Wichtig ist nur zu bedenken, dass Kafka darauf optimiert ist, viele kleine Nachrichten zu verarbeiten. Die Maximalgröße von Nachrichten ist auf 1 MB festgesetzt. Wir können dies zwar anpassen, sollten es aber vermeiden, weil die Performance und die Ressourcenauslastung durch größere Nachrichten stark leiden. Was genau bedeutet aber *viele kleine Nachrichten*? Im Extremen bedeutet das, dass Firmen wie LinkedIn im Jahr 2019 etwa 7 Billionen (nicht Milliarden) Nachrichten am Tag in Kafka verarbeiten. Dabei ist zu beachten, dass LinkedIn nicht nur einen Kafka-Cluster, sondern nach eigenen Aussagen etwa 100 Kafka-Cluster zu diesem Zeitpunkt hatte². Aber auch kleinere Installationen ermöglichen uns, sehr viele Nachrichten sehr schnell zu verarbeiten. Wir sollten nur darauf achten, dass unsere Nachrichten unter der 1-MB-Grenze bleiben. Es ist zum Beispiel nicht sinnvoll, mit Kafka größere Dateien zwischen Systemen zu übertragen. Für solche Anwendungsfälle sollten wir andere Lösungen bevorzugen. Die Rakete kann uns zum Beispiel die aktuellen Messwerte der Sensoren und die aktuellen Zustände der Systeme bereitstellen.

2.1.2.1 Nachrichtentypen

Welche Daten schreiben wir üblicherweise in Kafka? Das ist zwar sehr vom Anwendungsfall abhängig, aber erfahrungsgemäß können wir drei Arten von Nachrichten unterscheiden:

Zustände: Wir beschreiben in der Nachricht den aktuellen Zustand eines Subjekts oder Objekts. Die Fragestellung, die sich dahinter verbirgt, beschreibt das Interesse, wie die zu beobachtende Welt gerade aussieht. Was sind die aktuellen Messwerte meiner Sensoren? Was ist der aktuelle Warenbestand eines Artikels? Zum Beispiel nutzen wir das Topic *sensordaten*, um aktuelle Sensordaten zu schreiben:

```
# Eine Nachricht pro Zeile
{"sensor": "Tanktemperatur", "messwert": "-20°C"}
 {"sensor": "Außentemperatur", "messwert": "-40°C"}
 {"sensor": "Höhe", "messwert": "2500 m"}
```

²⁾ <https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages>

Oder statt Messwerte zu sammeln, nutzen wir in unserer Stammdatenverwaltung für Astronauten ein Topic namens *astronauten*, in das wir Informationen zu diesen Astronauten ablegen:

```
# Eine Nachricht pro Zeile
[{"id": 1, "name": "Alice", "Nächster Flug": "2030-01-01", "Geburtsdatum": "1990-05-07"},
 {"id": 2, "name": "Bob", "Nächster Flug": "2022-05-03", "Geburtsdatum": "2002-01-08"},
 {"id": 3, "name": "Chris", "Nächster Flug": "2022-05-03", "Geburtsdatum": "1960-10-22"}]
```

Deltas: Oft ist es bei jeder kleinen Veränderung nicht erforderlich, den gesamten Zustand neu zu schreiben. In diesen Fällen kann es sinnvoll sein, nur die Veränderung des Zustands in die Nachricht zu setzen. Vielleicht haben wir auch einen Sensor, der keine absoluten Werte liefert, sondern die Veränderung seit dem letzten Messwert. In unserem Sensor-daten-Szenario könnten wir uns zum Beispiel folgende Werte vorstellen:

```
{"sensor": "gyroskop", "delta_pitch": "+0.5°", "delta_yaw": "-0.5°", "delta_roll": "0°"}
 {"sensor": "gyroskop", "delta_pitch": "+0.01°", "delta_yaw": "-0.7°", "delta_roll": "0°"}
```

Bei unseren Astronauten gab es eine erfreuliche Änderung. Die Flüge wurden alle vorverlegt. Das heißt, wir können folgende Nachrichten in unser *astronauten*-Topic schreiben:

```
# Eine Nachricht pro Zeile
[{"id": 1, "Nächster Flug": "2025-01-01"},
 {"id": 2, "Nächster Flug": "2022-04-03"},
 {"id": 3, "Nächster Flug": "2022-04-03"}]
```

Wir sparen hier unter Umständen sehr viel Datenvolumen, weil wir auf die häufige Wiederholung der sich nicht verändernden Werte verzichten. Außerdem ist es mithilfe von Deltas einfacher herauszufinden, was sich konkret seit der letzten Nachricht verändert hat.

Ereignisse: Zu guter Letzt bilden Ereignisse (Englisch: *Events*) die Antwort auf die Frage „*Was ist passiert?*“ ab. Wir können zum Beispiel beschreiben, dass ein Sensorwert einen kritischen Wert überschritten oder dass ein Astronaut einen benötigten Test bestanden hat. Basierend auf diesen Ereignissen kann unser System entsprechend reagieren, das heißt zum Beispiel, den Flug abbrechen oder automatisch goldene Anstecknadeln verschicken.

Oft verwenden wir Zustände und Deltas als Rohdaten und können basierend darauf in ein weiteres Topic unsere Ereignisse produzieren. Wenn wir uns an unsere ersten Schritte mit Kafka zurückinnern, können wir unser *flugdaten*-Topic als ein Ereignis-Topic betrachten. Wir haben damals folgende Nachrichten geschrieben:

```
# Eine Nachricht pro Zeile
Countdown beendet
Liftoff
Atmosphäre verlassen
Vorbereitung Wiedereintritt
Wiedereintritt erfolgreich
Landung erfolgreich
```

Wenn wir diese Nachrichten mit dem jetzigen Wissen betrachten, können wir darin Ereignisse erkennen. Wir beschreiben hier die Antwort auf die Frage „*Was ist gerade passiert?*“.

2.1.2.2 Datenformate

Nachdem wir nun eine Idee haben, welche Arten von Nachrichten wir in Kafka speichern möchten, sollten wir uns überlegen, welches Datenformat wir dafür benutzen wollen. Kafka arbeitet intern ausschließlich mit Byte-Arrays und kann daher die Daten, die wir versenden, nicht interpretieren. Somit können keine Funktionen zur Manipulation der Daten verwendet werden – dies ist einer der vielen Gründe, warum Kafka so performant ist.

Da Kafka immer mehr in Microservice-Architekturen eingesetzt wird, finden wir in Kafka-Tops oft *JSON*-Daten wieder. *JSON* hat den Vorteil, dass sowohl Mensch als auch Maschine dieses Datenformat einfach lesen können. Aber ähnlich wie bei *XML* kann es vorkommen, dass sich die Datenmenge stark aufbläht und wir auch mit der Komprimierung von Nachrichten nicht weit kommen. Wenn die Minimierung der Datengröße sehr wichtig ist, können wir Binärformate wie zum Beispiel *Google Protocol Buffers (ProtocolBuf³)* oder das in der Kafka Community verbreitete *Apache Avro⁴* verwenden. Unserer Erfahrung nach muss fast jedes Topic mindestens einmal in seinem Leben von einem Menschen ausgelesen werden, was bei binären Datenformaten umständlich ist. Wichtiger als die Diskussion über das Datenformat ist jedoch die möglichst einheitliche Wahl des Formats. Nicht für jedes Topic sollte ein unterschiedliches Datenformat verwendet werden. Je mehr Topics wir in einem Kafka-Cluster haben und je älter der Cluster wird, desto wichtiger wird es, die Datenformate zu dokumentieren. Binäre Datenformate benötigen immer ein Schema, damit sie gelesen werden können, aber es ist ebenso sinnvoll für *JSON* (zum Beispiel *JSON Schema⁵*) und *XML* (z. B. *XSD⁶* oder *DTD⁷*), konsistente Schemas zu benutzen.

2.1.2.3 Keys und Values

Wenn wir uns die Nachrichten genauer anschauen, können wir feststellen, dass Nachrichten in Kafka nicht nur aus einem Wert bestehen. Denn abgesehen von Metadaten besteht jede Kafka-Nachricht aus zwei Werten, einem Key und einem Value. Keys sind jedoch optional. Das heißt, wenn wir im *kafka-console-producer.sh* nicht explizit Keys angeben, werden Nachrichten ohne Keys produziert. Starten wir nun den *kafka-console-producer.sh*, sodass wir Nachrichten mit Keys produzieren können:

```
# Zuvor haben wir das Topic flugdaten_mit_keys erstellt
$ kafka-console-producer.sh \
  --topic flugdaten_mit_keys \
  --property parse.key=true \
  --property key.separator=: \
  --bootstrap-server=localhost:9092
```

Zusätzlich zu den üblichen Angaben wie Topic und Kafka-Cluster übergeben wir zwei weitere Parameter. Mit `--property parse.key=true` aktivieren wir im *kafka-console-producer.sh* das Parsen von Keys, und mit der Einstellung `--property key.separator=:` setzen wir den Doppelpunkt (:) als Separator zwischen Keys und Values. Produzieren wir nun ein paar Nachrichten:

³⁾ <https://developers.google.com/protocol-buffers>

⁴⁾ <http://avro.apache.org/>

⁵⁾ <http://json-schema.org/>

⁶⁾ https://de.wikipedia.org/wiki/XML_Schema

⁷⁾ <https://de.wikipedia.org/wiki/Dokumenttypdefinition>

```
# Producer Fenster:
> rakete1:Countdown beendet
> rakete2:Countdown gestartet
> rakete1:Liftoff
> rakete2:Countdown beendet
```

Wenn wir den `kafka-console-consumer.sh` nun wie bisher starten, sehen wir keinen Unterschied, da der Consumer standardmäßig keine Keys anzeigt:

```
$ kafka-console-consumer.sh \
--from-beginning \
--topic flugdaten_mit_keys \
--bootstrap-server=localhost:9092
Countdown beendet
Countdown gestartet
Liftoff
Countdown beendet
Processed a total of 4 messages
```

Wir brechen den Befehl wieder mit **STRG+C** ab. Um die Keys zu sehen, müssen wir dies explizit anschalten:

```
$ kafka-console-consumer.sh \
--from-beginning \
--topic flugdaten_mit_keys \
--property print.key=true \
--bootstrap-server=localhost:9092
rakete1    Countdown beendet
rakete2    Countdown gestartet
rakete1    Liftoff
rakete2    Countdown beendet
Processed a total of 4 messages
```

Nun zeigt uns der Consumer die Nachrichten inklusive Key an.

Wofür benötigen wir aber Keys? In Key-Value-Datenbanken ist dies offensichtlich. Wir speichern dort Daten immer zu einem konkreten Key, unter dem wir die geschriebenen Daten auch wieder abrufen oder manipulieren zu können. In Kafka ist dies nicht ganz so offensichtlich. Erstens ist der Key optional, und wie wir aus der Ausgabe des `kafka-console-consumer.sh` sehen können, werden alte Nachrichten mit demselben Key nicht gelöscht.

Wir werden im nächsten Kapitel Kafka als verteilten Log kennenlernen und feststellen, dass üblicherweise Topics aus mehr als nur einer Partition bestehen. Wir können Keys benutzen, um zu garantieren, dass Nachrichten in der gleichen Partition landen. Denn Kafka garantiert die Reihenfolge von Nachrichten nur innerhalb einer Partition. Wenn wir unser Topic `flugdaten` mit mehr als einer Partition erstellt hätten, könnten wir nicht garantieren, dass die Nachricht `Countdown beendet` vor der Nachricht `Liftoff` gelesen werden würde. Im besten Fall würde das zu Verwirrung und Gelächter führen, im schlechtesten Fall wahrscheinlich zum Absturz unserer Rakete. Wenn wir das Topic `flugdaten_mit_keys` nun mit mehr als einer Partition erstellt hätten, würden wir garantieren, dass die Nachrichten zu `rakete1` immer in der richtigen Reihenfolge eintreffen. Aber wir hätten keine Garantie, welche der beiden Nachrichten zuerst gelesen werden würde:

```
rakete2:Countdown gestartet
rakete1:Countdown beendet
```

In den allermeisten Fällen können wir akzeptieren, dass die Nachrichten zu unterschiedlichen Keys nicht immer in der richtigen Reihenfolge sind. Es reicht meistens aus, garantieren zu können, dass die Nachrichten zu einem Key korrekt sortiert sind.

Mithilfe von Keys garantieren wir nicht nur, dass die Reihenfolge unserer Nachrichten korrekt ist, sondern können mithilfe von *Log Compaction* auch alte Nachrichten mit demselben Key löschen. Stellen wir uns zum Beispiel vor, dass wir in unserem Topic *astronauten* die ID in den Key und nicht in den Value schreiben und immer die aktuellen Daten zu den Astronauten abspeichern:

```
[...]
1: {"name": "Alice", "Nächster Flug": "2030-01-01", "Geburtsdatum": "1990-05-07"}
[...]
1: {"name": "Alice", "Nächster Flug": "2025-01-01", "Geburtsdatum": "1990-05-07"}
[...]
1: {"name": "Alice", "Nächster Flug": "2024-05-07", "Geburtsdatum": "1990-05-07"}
```

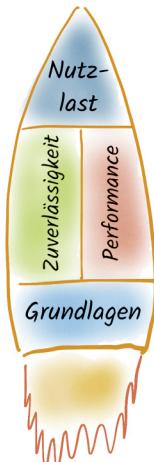
Die alten Nachrichten sind für uns nicht immer von Wert, und wir können mithilfe von Log Compaction alte Nachrichten mit demselben Key löschen, um Festplattenplatz freizuräumen. Dann würden wir nur noch den aktuellen Wert behalten:

```
1: {"name": "Alice", "Nächster Flug": "2024-05-07", "Geburtsdatum": "1990-05-07"}
```

2.1.3 Zusammenfassung

In diesem Kapitel haben wir die ersten Konzepte von Kafka kennengelernt und sind hierbei bereits mit einigen Fachbegriffen aus dem Kafka-Universum in Berührung gekommen, sodass für uns Consumer, Producer oder Broker keine Unbekannten mehr sind. Wir haben erfahren, dass Daten in Kafka in sogenannten Topics strukturiert werden und dass die Daten selbst in Nachrichten aufgeteilt werden. Wir wissen nun auch, dass wir über diese Topics grundlegende Eigenschaften von Kafka definieren können, wie zum Beispiel den *Replication Factor* oder die Anzahl an Partitionen. Außerdem haben wir erste Erfahrungen im Anzeigen, Erstellen, Anpassen und Löschen von Topics gesammelt. Im zweiten Abschnitt haben wir uns intensiv mit den Nachrichten an sich in Kafka beschäftigt. Wir haben gelernt, was genau Kafka unter einer Nachricht versteht und in welche Typen wir unsere Nachrichten klassifizieren können. Weiterhin wissen wir jetzt, dass Kafka intern nur mit Byte-Arrays arbeitet und wir verschiedene Möglichkeiten haben, diese Byte-Arrays zu serialisieren beziehungsweise zu deserialisieren. Zum Abschluss haben wir uns noch angesehen, wie wir unsere Nachrichten mit Keys versehen können und welche Auswirkungen dies auf die Speicherung unserer Daten beziehungsweise Nachrichten hat.

■ 2.2 Kafka als verteiltes Log



Nachdem wir im ersten Kapitel unsere Rakete haben fliegen lassen und den Flug in Kafka nachvollzogen haben, fokussieren wir uns im folgenden Kapitel auf das Verständnis der Grundlagen. In unserem Raketenmodell ist dies die erste Stufe. Was ist Kafka für ein System? Warum funktioniert es so, wie es ist? Welche Auswirkungen hat ein System wie Kafka auf die IT-Architektur in meinem Unternehmen? Diese und weitere Fragen möchten wir in diesem Kapitel behandeln.

Wir kommen mit Kafka auch ohne die Antworten auf diese Fragen recht weit, doch spätestens abseits von Sandkastenprojekten ist es essenziell, die genaue Arbeitsweise zu verstehen. Für die meisten ITler sind die Kafka-Konzepte relativ unbekannt. Aber das Verstehen dieser Konzepte führt oft zu Aha-Effekten und Erkenntnissen, wie wir unsere IT-Architektur, die Entwicklung und den Betrieb noch weiter optimieren können.

2.2.1 Logs

Wir beschreiben Kafka gerne als ein Log. Obwohl der Begriff des Logs für viele im Zusammenhang mit Kafka neu ist, begegnen uns Logs ständig im (IT-)Alltag.

2.2.1.1 Was genau ist eigentlich ein Log?

Unsere Betriebssysteme produzieren System-Logs. Wenn wir für diese Systeme zuständig sind, nutzen wir die Logs regelmäßig, um den Status der Systeme zu überprüfen und in Fehlerfällen zu verstehen, wie es zu diesen Fehlern kommen konnte. Vielleicht setzen wir sogar Log-Überwachungssysteme ein, die bei bestimmten Ereignissen Alarm schlagen und uns notfalls (aber hoffentlich nicht) mitten in der Nacht wecken, damit wir auf die Fehler reagieren können.

Dasselbe gilt auch für unsere Applikationen und Services. Sie produzieren Log-Ereignisse, in denen wir sehen können, was gerade passiert, ob es zu Fehlern kommt oder ob alle Systeme einwandfrei laufen. Wir nutzen beides, System-Logs und Applikations-Logs, um Fehler zu analysieren und den Zustand des entsprechenden Systems zu überwachen. Hier sind Logs nützlich und essenziell, aber nicht der Kern unserer Systeme.

Anders sieht es bei Datenbanksystemen aus: Hier ist der Commit-Log zwar tief im System versteckt, aber eine der Hauptkomponenten des Systems. Im Commit-Log speichern Datenbanken jegliche Veränderungen an Daten. Werden Daten hinzugefügt, geändert oder gelöscht, schreibt die Datenbank diese Veränderung zuerst in den Commit Log und erst dann in die entsprechenden Tabellen. Sollte die Datenbank ausfallen, garantiert uns der Commit-Log, dass wir den letzten sauberen Zustand wiederherstellen können, ohne Daten zu verlieren. Ähnliches gilt auch für die Replikation, also den Cluster-Betrieb, von Datenbanken. Oft basiert diese Replikation darauf, das Commit-Log zwischen den Cluster-Einheiten auf dem aktuellen Stand zu halten, und jeder Server baut dann die Tabellen unabhängig voneinander basierend auf den Daten im replizierten Commit-Log auf.

So unterschiedlich diese Logs auch sein mögen, geben alle die Antwort auf die Frage „*Was ist passiert?*“. Die System- und Applikations-Logs beantworten diese Fragen dem Betreiber der Systeme, und die Commit Logs in Datenbanken beantworten diese Frage den anderen Servern im Datenbankcluster oder auch sich selbst, wenn das System abstürzt oder sonstige Fehler auftreten.

Wenn wir zurück an unser Beispiel aus dem ersten Kapitel denken, können wir auch logähnliche Datenstrukturen erkennen. Hier folgt zum Beispiel der Inhalt unseres Topics `flugdaten`:

```
Countdown beendet
Liftoff
Atmosphäre verlassen
Vorbereitung Wiedereintritt
Wiedereintritt erfolgreich
Landung erfolgreich
```

Der Grund für die Ähnlichkeiten ist einfach. Kafka basiert auf Logs als zentrale Datenstruktur. Im Gegensatz zu Datenbanken versteckt Kafka das Log nicht, sondern das Log wird zum Kernelement unseres Systems. Unsere Daten werden in Kafka in Logs gespeichert, und wir können die gleichen Muster und Erkenntnisse aus der Welt der System-, Applikations- und Datenbank-Logs nehmen und auf Kafka anwenden!

2.2.1.2 Grundlegende Eigenschaften eines Logs

Aber auch abseits der IT nutzen wir Logs sehr häufig. Das wohl einfachste Beispiel sind Tagebücher. Wenn wir im Tagebuch vorne anfangen und zum Beispiel jeden Tag eine Seite schreiben, ohne dazwischen Platz zu lassen, und dabei einen Kugelschreiber statt eines Bleistifts benutzen, erkennen wir die Eigenschaften eines Logs wieder:

Reihenfolge und Sortierung: Nachrichten in einem Log sind nach Zeit sortiert. Am Anfang des Logs steht die älteste Nachricht und am Ende die neueste Nachricht. Genauso ist es auch in unserem Tagebuch. Wir wissen (wenn wir keine Seiten frei lassen), dass der Eintrag auf Seite 10 älter ist als der Eintrag auf Seite 11.

Schreib- und Leserichtung: Neue Einträge schreiben wir immer ans Ende des Logs. Wenn wir das Log lesen, fangen wir meist auf einer Seite an und lesen erst die älteren Einträge und dann die neueren. Natürlich können wir im Tagebuch auch zurückblättern, aber dennoch ist die natürliche Leserichtung von alt nach neu.

Unveränderbarkeit: Sobald wir einen Eintrag geschrieben haben, können wir diesen nicht mehr ohne Weiteres ändern oder entfernen. Im Tagebuch ist dies noch möglich, indem wir auf den Seiten Anmerkungen hinzufügen, Wörter durchstreichen oder Seiten ausreißen. Aber zumindest fallen diese Änderungen meistens auf.

Eine weitere interessante Eigenschaft von Logs ist, dass wir sehr gut nachvollziehen können, wie der vom Log beschriebene Teil der Welt sich über die Zeit verändert hat, und damit können wir auch herausfinden, wie dieser Teil der Welt zu einem bestimmten Zeitpunkt ausgesehen hat.

Oft schauen wir uns System- oder Applikations-Logs mit der Frage „*Wie konnte es zu diesem Fehler kommen?*“ an. Wenn wir die Einträge im Log chronologisch lesen, können wir oft nachvollziehen, was sich in der Zeit verändert hat, und oft können wir dadurch einen Hinweis finden, was schiefgelaufen ist. Dasselbe gilt auch für Datenbanken. Um herauszu-

finden, wie eine Tabelle zu einem bestimmten Zeitpunkt ausgesehen hat, fangen wir im Commit-Log ganz am Anfang an, arbeiten uns Eintrag für Eintrag vor, sehen so, wie sich die Tabelle über die Zeit verändert hat und warum wir genau die Einträge in der Tabelle sehen, die sie uns anzeigt.

Aber nicht nur das, Logs erlauben es uns, in der Zeit rückwärts zu reisen! Wenn wir den Zustand unserer Datenbank auf den Stand von vorgestern wiederherstellen möchten, können wir das Log von Anfang an bis zu den Einträgen von vorgestern durchgehen und sehen, wie die Datenbankwelt damals aussah.

Mit diesem Wissen können wir das Log etwas formaler definieren. Ein Log ist eine Liste von Ereignissen, auf der wir folgende zwei Operationen definieren:

Schreiboperation: Wir fügen neue Einträge an das Ende der Liste an.

Leseoperation: Wir beginnen bei einem bestimmten Eintrag und lesen von alt nach neu, üblicherweise bis zum Ende des Logs.

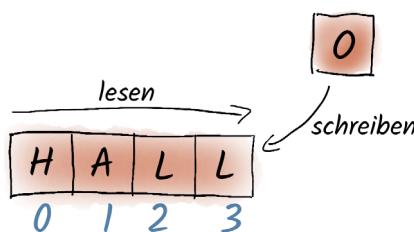


Bild 2.2 Ein Log ist eine Liste, an die wir ausschließlich Elemente hinten anfügen und sequenziell von einer Position (Offset) auslesen können. Wir können zum Beispiel zuerst alle Elemente ab Offset 0 lesen und uns merken, dass wir als Nächstes die Nachricht ab Offset 4 lesen möchten. Solange es keine neuen Einträge gibt, bekommen wir kein Ergebnis, aber wenn wir ein Element anfügen und es erneut versuchen, erhalten wir die neuen Einträge und merken uns wieder den Offset des Eintrages, den wir als Nächstes lesen möchten (hier in dem Beispiel dann die 4).

Meist sind unsere Logs zu groß und zu lang, um sie am Stück lesen zu können. Applikations-Logs werden gerne mehrere Gigabyte groß. Wie merken wir uns also, welche Einträge wir bereits gelesen haben und welche noch nicht? Wir nummerieren unsere Einträge durch. Der erste Eintrag bekommt die Position 0, der zweite die Position 1 und so weiter. In Kafka ist diese Position der *Offset*, den wir zuvor schon kennengelernt haben. Der Offset gibt einerseits die Position einer Nachricht im Log an (ähnlich einer Seitenzahl), aber er gibt auch an, welchen Eintrag wir als Nächstes lesen möchten (ähnlich einer Seitenzahl, die wir uns merken). Wie in einem Buch fangen wir meistens ganz vorne an zu lesen (Offset 0), dann lesen wir ein paar Seiten und merken uns die Seite (beziehungsweise den Offset), die wir als Nächstes aufschlagen möchten. Genau wie wir bei Büchern verwendet Kafka Lesezeichen. Die Systeme, die die Logs lesen, können zwar die Offsets im RAM vorhalten, aber dies ist nicht sehr zuverlässig. Das System könnte jederzeit ausfallen und müsste dann von Anfang an lesen. Stattdessen hilft uns Kafka dabei, die Offsets zu merken.

2.2.1.3 Die Rolle des Logs in Kafka

Da Logs in IT-Systemen meistens nicht eine statische Anzahl an Seiten haben, sondern ständig neue Einträge hinzukommen, haben sie üblicherweise auch kein Ende. In Kafka

merken wir uns den Offset, den wir als Nächstes lesen möchten, und fragen kontinuierlich den Kafka-Cluster an, ob es neue Einträge gibt. Gibt es neue Einträge, bekommen wir diese Einträge zurück und merken uns den nächsten Offset, den wir lesen möchten. Gibt es keine neuen Einträge, dann bekommen wir auch keine Daten und müssen unseren Offset auch nicht anpassen.

Das Interessante an einem Offset ist, dass er lediglich die Position einer Nachricht beschreibt, aber nicht deren Inhalt. Das ist genauso wie mit den Seitenzahlen im Buch. Es ist eine fortlaufende Nummerierung ohne weitere Bedeutung. Das ist wichtig zu verstehen, denn in Logs ist der Offset die einzige Möglichkeit, Daten zu adressieren. Wir können in einem Log nicht auf konkrete Elemente zugreifen.

Dies hat enorme Auswirkungen auf die Architektur unserer Services, die Kafka benutzen. Wir sollten Kafka nicht nutzen, um Anfragen über die Daten darüber ad hoc zu beantworten. Wir sollten Kafka auch nicht nutzen, um ähnlich wie in einem Key-Value Store auf Daten mit einem bestimmten Key zuzugreifen. Jede dieser Operationen würde unter Umständen eine Suche über alle Daten im Log provozieren.

Logs, beziehungsweise Kafka, sind nicht der Heilbringer, um alle unsere Datenbanken, Caches und Analytics-Werkzeuge durch ein System zu ersetzen. Aber Kafka kann uns dabei helfen, die Daten in unserem Unternehmen effektiver zu organisieren und zwischen den Systemen auszutauschen.

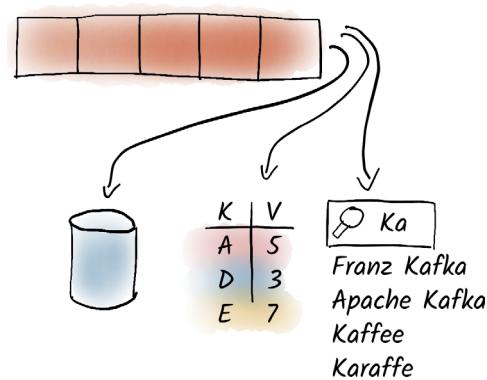


Bild 2.3 Ein Log ist eine perfekte Datenstruktur, um Daten zwischen Systemen auszutauschen. Üblicherweise arbeiten wir nicht direkt mit den Daten im Log, sondern speichern diese in einem Datenformat, welches am besten für unseren jeweiligen Anwendungsfall geeignet ist. Zum Beispiel können wir relationale Datenbanken nutzen, um über unsere Daten komplexe Abfragen auszuführen. Wenn wir schnell auf vorbereitete Daten zugreifen möchten, können wir zum Beispiel einen In-Memory Key-Value Store wie Redis benutzen. Sollten wir eine Suchfunktion über den Daten im Log anbieten wollen, können wir dafür eine Suchmaschine benutzen.

Statt zu versuchen, Kafka als zentrale Datenbank für unsere Services zu missbrauchen, nutzen wir Kafka als zentrale Datendrehscheibe und nutzen in unseren Services die beste Technologie für unseren Anwendungsfall. Wenn wir zum Beispiel einen Suchservice über unsere Astronauten-Stammdaten erstellen möchten, schreiben wir die Daten aus Kafka in eine Suchmaschine wie zum Beispiel *ElasticSearch*. Wenn wir Daten ad hoc auswerten möchten, könnten wir dafür eine relationale Datenbank wie PostgreSQL benutzen und Daten aus Kafka in die Datenbank schreiben.

Wir haben auf den letzten Seiten Kafka als Log kennengelernt. Wir wissen, dass ein Log eine Liste ist, in der wir zum Schreiben Daten hinten anhängen und von alt nach neu lesen. Sobald die Daten im Log geschrieben sind, verändern oder löschen wir die Daten nicht. Ein Grund, warum Logs an vielen Stellen benutzt werden, ist, dass das eine der einfachsten Datenstrukturen ist und sie keine Magie enthalten. Statt wie in einer Datenbank den aktuellen Zustand der Welt abzuspeichern, speichern wir in Logs üblicherweise die Historie von Ereignissen von Anfang an. Wichtig ist es, nicht in Versuchung zu kommen, Logs als Allheilmittel zu betrachten und Kafka als Datenbank zu missbrauchen. Wir werden in einem späteren Kapitel Kafka als Kern einer Streaming-Plattform kennenlernen und sehen, wie Kafka mit anderen Systemen in unserer IT-Landschaft bestmöglich zusammenspielen kann.

2.2.2 Verteilte Systeme

Kafka nur als Log zu betrachten, bringt uns schon einige Erkenntnisse, aber dies reicht nicht aus, um Kafka wirklich zu verstehen. Wir müssen einen Schritt weiter gehen und Kafka als *verteilten Log* verstehen. Denn die Eigenschaften wie Geschwindigkeit, Skalierbarkeit und Ausfallsicherheit können wir allein mit einem Log nur schwierig erreichen, sondern müssen lernen, Logs effizient auf mehrere Server zu verteilen.

Obwohl die Konzepte, auf denen Kafka basiert, seit vielen Jahrzehnten bekannt sind und in etlichen anderen Produkten intensiv genutzt werden, haben viele Entwicklungen der letzten Jahre Kafka erst ermöglicht. Statt wie zum Beispiel bei Datenbanksystemen üblich, die Systeme mit sehr großer Sorgfalt zu pflegen und sich intensiv um den einen oder die wenigen Server einzeln zu kümmern, damit sie nicht ausfallen, gingen Kafkas Entwickler von anderen Annahmen aus. Computersysteme sind relativ unzuverlässig, aber Server aus Massenproduktion relativ billig. Statt Kafka also auf teurer, spezialisierter Hardware (wie Mainframes oder Spezialhardware) zu betreiben, lassen wir Kafka üblicherweise auf handelsüblicher Hardware laufen und gleichen die erhöhte Schadensanfälligkeit dadurch aus, dass wir Kafka nicht nur auf einer einzigen Maschine laufen lassen. Anstatt Fehler um jeden Preis zu vermeiden, lernen wir, Fehler in Kauf zu nehmen und unsere Systeme entsprechend zu planen. Die Praxis gibt uns damit recht. Wenn wir ein einziges System betreiben, ist die Zuverlässigkeit meist gut genug. Aber spätestens im Rechenzentrum stehen Hardwareausfälle auf der Tagesordnung. Zum Beispiel führt die Firma Backblaze Statistik über die Zuverlässigkeit ihrer eingesetzten Festplatten⁸. Diese besagt, dass die eingesetzten Festplatten eine jährliche Ausfallwahrscheinlichkeit von etwa 1 % haben.

In der Informatik können wir Systeme entweder vertikal oder horizontal skalieren. Wenn wir ein System vertikal skalieren, beschaffen wir uns leistungsfähigere Systeme. Statt den Datenbankserver mit acht CPU-Kernen und 16 GB RAM besorgen wir uns den Server mit 64 Kernen und 256 GB RAM, in der Hoffnung, dadurch eine bessere Performance zu erreichen. Das ist eine sehr einfache Möglichkeit, die Performance unseres Systems zu steigern, und funktioniert eine Zeit lang sehr gut. Leider wird unsere Zuverlässigkeit nicht besser. Wenn wir einen Server durch einen stärkeren ersetzen, sinkt meist die Wahrscheinlichkeit eines Ausfalls nicht nennenswert. Horizontale Skalierung bedeutet, dass wir, anstatt immer größere und leistungsfähigere Server zu kaufen, nicht mehr einen, sondern mehrere

⁸⁾ Hard Drive Stats <https://www.backblaze.com/b2/hard-drive-test-data.html>

Server einsetzen. Dies ist sehr verlockend. Wir können scheinbar endlos skalieren. Wenn unsere Server ausgelastet sind, stellen wir einfach einen weiteren hinzu. Wir erhoffen uns durch diese Parallelisierung einen deutlichen Leistungszuwachs und durch die erhöhte Redundanz auch eine verbesserte Zuverlässigkeit und Datenhaltbarkeit.

Leider ist dies oft zu einfach gedacht. Ein System auf mehrere Server zu verteilen ist alles andere als einfach. Die Software muss in der Lage sein, sich zu koordinieren. Irgendjemand muss entscheiden, welcher Server welche Anfrage bearbeitet. Wir müssen uns überlegen, was passiert, wenn ein Server ausfällt. Wie stellen wir fest, dass dieser nicht mehr funktioniert oder, schlimmer noch, Anfragen fehlerhaft beantwortet? Wie ersetzen wir die ausgefallenen Server?

Dass dies nicht nur theoretische Probleme sind, sondern dass wir uns als Entwickler und Betreiber verteilter Systeme darüber ernsthaft Gedanken machen müssen, zeigen auch die regelmäßigen Ausfälle selbst bei den großen IT-Konzernen wie Google, Amazon, Netflix und so weiter. Mit verteilten Systemen zu arbeiten ist oft insbesondere für Menschen, die vorher vor allem die vertikale Art der Skalierung benutzt haben, herausfordernd und bedarf einer anderen Mentalität. Ein prominentes Beispiel ist der *Netflix Chaos Monkey*⁹⁾. Statt davon auszugehen, dass Systeme gut genug funktionieren und zuverlässig sind, um dann schlimmstenfalls an Weihnachten einen Ausfall zu haben, provoziert der Chaos Monkey Ausfälle von Servern zu Bürozeiten. Durch das Einschleusen von Fehlern in Zeiten, in denen die Entwickler und Administratoren am Arbeitsplatz sind, werden solche Fehler zu normalen Fehlern. Dadurch haben wir eine größere Motivation und Dringlichkeit, unsere Systeme so zu entwickeln, dass sie Fehler überstehen, ohne dass unsere Kunden Schaden davon nehmen.

Dies ist nur ein Beispiel davon, wie wir unseren Umgang mit verteilten Systemen verbessern können. Statt anzunehmen, dass unsere Systeme zuverlässig sind, akzeptieren wir nicht nur Fehler, sondern bauen sie in unseren Arbeitsalltag ein, um zu lernen, damit besser umzugehen. Wir akzeptieren diese Annahmen nicht erst zum Zeitpunkt des Betriebs, sondern fangen beim Systemdesign damit an, darüber nachzudenken, wie sich Fehler auswirken. Ein ganz wichtiger Punkt dabei ist es, die verteilten Systeme so einfach wie möglich zu designen und zu entwickeln. Kafka ist zwar ein sehr komplexes Softwaresystem, welches alles andere als einfach ist, aber die Grundkonzepte und die Basistechnologien sind so einfach gehalten, wie es nur möglich ist. Das Log selbst ist zum Beispiel eine der einfachsten Datenstrukturen zum Speichern von größeren Datenmengen. Ein Log ist sehr einfach zu replizieren (Nachricht für Nachricht kopieren) und auch sehr einfach aufzuteilen (wir nehmen mehrere Logs statt einem einzigen).

2.2.2.1 Partitionierung

Zu Kafkas Design-Zielen gehörte es von Anfang an, in der Lage zu sein, mit riesigen Datenmengen umzugehen und diese Datenmengen mit einer sehr hohen Geschwindigkeit verarbeiten zu können. Zusätzlich ist es uns in vielen Anwendungsfällen wichtig, dass unsere Daten weder verloren gehen noch durcheinanderkommen. Deshalb können wir Kafka so konfigurieren, dass die Nachrichten zuverlässig geschrieben werden und die Reihenfolge der Nachrichten eingehalten wird. In unserem Kafka-Raketenmodell haben wir neben der

⁹⁾ <https://netflix.github.io/chaosmonkey/>

ersten Stufe *Grundlagen* zwei Treibstofftanks, *Performance* und *Zuverlässigkeit*. Im Rest dieses Abschnitts lernen wir die Grundlagen dafür kennen, wie Kafka genau diese Performance und Zuverlässigkeit erreicht.

Wir haben soeben besprochen, dass eine der Hoffnungen, ein System auf mehrere Server zu verteilen, eine verbesserte Performance ist. Einfach nur von besserter Performance bei der horizontalen Skalierung zu sprechen, ist zu kurz gedacht. Meistens verringert sich anfangs sogar die Geschwindigkeit, in der Nachrichten verarbeitet werden, wenn wir anfangen, horizontal zu skalieren. Das ist wie bei Mehrkernprozessoren. Wenn eine Aufgabe nicht parallelisierbar ist, nützt es nichts, mehr Kerne zur Verfügung zu haben. Deshalb sprechen wir nicht einfach von besserter Performance bei horizontaler Skalierung, sondern von Parallelisierung. Wir können damit mehr Nachrichten pro Zeiteinheit verarbeiten, als es ein einziger Server könnte. Dafür muss aber die Art und Weise, wie wir die Nachrichten verarbeiten, parallelisierbar sein.

Die einfachste Art und Weise, Daten auf mehrere Subsysteme aufzuteilen, ist die Partitionierung der Daten, auch bekannt als Sharding in Datenbanksystemen. Stellen wir uns zum Beispiel vor, dass wir eine weltweite Taxi-Vermittlungsstelle aufbauen möchten. Wir könnten nun alle Daten zu allen Fahrten in einer zentralen Datenbank speichern. Dies bringt uns allerlei Vorteile. Wir können übliche Bewegungsmuster erkennen, unsere Betriebsabläufe optimieren und so unsere Dienste verbessern. Aber wenn unser Dienst erfolgreich sein sollte, dann werden wir viel mehr Daten empfangen und verarbeiten, als ein einziger Server in der Lage ist zu bearbeiten. Die Folge ist, dass alles langsamer wird und unsere Kunden zur Konkurrenz gehen. Wenn wir uns aber die Daten genauer anschauen, stellen wir fest, dass es uns gar nicht so wichtig ist, alle Daten zu allen Fahrten in allen Städten in einer zentralen Datenbank zu halten. Es reicht uns vollständig aus, Daten aus einer Stadt beisammenzuhalten, aber es ist uns nicht wichtig, dass zum Beispiel die Reihenfolge der Daten über Städte hinweg korrekt ist. So können wir auch einfacher horizontaler skalieren. Wir können mit einem Server starten, und sobald wir mehr und mehr Kunden haben, können wir mehr Server dazustellen und einige Städte darauf auslagern.

Auch in unserem Raketen-Beispiel aus dem ersten Kapitel können wir ähnlich vorgehen. Wir benötigen nicht wirklich alle Daten für alle Raketen in einem einzigen Log. Uns reicht es eigentlich aus, die Daten zu einer Rakete in einem Log zu halten. Wenn die Reihenfolge der Daten von unterschiedlichen Raketen nicht stimmt, ist das nicht dramatisch. Das Wichtigste ist, dass die Daten zu einer einzelnen Rakete korrekt sortiert sind.

In Kafka gehen wir genauso vor. Statt alle Daten eines Topics in ein Log zu schreiben, teilen wir dieses Topic in mehrere Partitionen auf. Wenn wir ein Topic erstellen, müssen wir angeben, wie viele Partitionen wir haben möchten. Erstellen wir nun ein Topic mit mehreren Partitionen:

```
$ kafka-topics.sh \
--create \
--topic flugdaten-partitionen \
--partitions 2 \
--replication-factor 1 \
--bootstrap-server localhost:9092
Created topic partitionen-test.
```

Wir nutzen dafür unser bekanntes `kafka-topics.sh`-Kommando und erhöhen ausschließlich die Anzahl der Partitionen auf 2 (`--partitions`). Produzieren wir jetzt einige Daten in

dieses Topic. Dazu nutzen wir, wie schon eingangs erwähnt, den `kafka-console-producer.sh`:

```
$ kafka-console-producer.sh \
  --topic flugdaten-partitionen \
  --bootstrap-server localhost:9092
> Countdown beendet
> Liftoff
> Atmosphäre verlassen
> Vorbereitung Wiedereintritt
> Wiedereintritt erfolgreich
> Landung erfolgreich
# STRG+D drücken zum Abbrechen
```

Mit dem `kafka-console-consumer.sh` können wir die Daten wieder konsumieren:

```
$ kafka-console-consumer.sh \
  --topic flugdaten-partitionen \
  --from-beginning \
  --bootstrap-server localhost:9092
Vorbereitung Wiedereintritt
Landung erfolgreich
Countdown beendet
Atmosphäre verlassen
Wiedereintritt erfolgreich
Liftoff
# STRG+C drücken, um abzubrechen
Processed a total of 6 messages
```

Wir stellen fest, dass plötzlich die Reihenfolge der Nachrichten nicht mehr stimmt! Dies mag in vielen Fällen nicht das Problem sein, aber in unserem Raketen-Beispiel könnte dies katastrophale Folgen haben. Der Grund dafür ist, dass der `kafka-console-producer.sh` die Daten automatisch auf mehrere Partitionen verteilt und Kafka die Reihenfolge der Nachrichten nur innerhalb einer Partition garantiert. Um die korrekte Reihenfolge zwischen bestimmten Nachrichten zu garantieren, müssen wir sicherstellen, dass diese Nachrichten in einer Partition landen. Dafür nutzen wir in Kafka *Keys*.

In einem der vorherigen Kapitel haben wir gesehen, dass in Kafka Nachrichten aus einem (optionalen) Key und einem Value bestehen. Um Keys mit anzugeben, nutzen wir das Kommandozeilen-Flag `--parse-keys` im `kafka-console-producer.sh`. Erstellen wir wie oben ein neues Topic mit drei Partitionen und produzieren für vier Keys (k1 bis k4) jeweils vier Nachrichten (1 bis 4):

```
# Erst das Topic `flugdaten-partitionen-keys` wie oben erstellen!
$ kafka-console-producer.sh \
  --topic flugdaten-partitionen-keys \
  --property parse.key=true \
  --property key.separator=: \
  --bootstrap-server=localhost:9092
>rakete0:Countdown beendet
>rakete1:Countdown beendet
>rakete1:Liftoff
>rakete1:Atmosphäre verlassen
>rakete0:Liftoff
>rakete0:Atmosphäre verlassen
>rakete0:Vorbereitung Wiedereintritt
>rakete1:Vorbereitung Wiedereintritt
```

```
>rakete0:Wiedereintritt erfolgreich
>rakete1:Wiedereintritt erfolgreich
>rakete1:Landung erfolgreich
>rakete0:Landung erfolgreich
# STRG+D drücken zum abbrechen
```

Im Consumer können wir die Eigenschaft `print.key` aktivieren, um Keys anzuzeigen:

```
$ kafka-console-consumer.sh \
  --from-beginning \
  --topic flugdaten-partitionen-keys \
  --property print.key=true \
  --bootstrap-server=localhost:9092
rakete0    Countdown beendet
rakete0    Liftoff
rakete0    Atmosphäre verlassen
rakete0    Vorbereitung Wiedereintritt
rakete0    Wiedereintritt erfolgreich
rakete1    Countdown beendet
rakete0    Landung erfolgreich
rakete1    Liftoff
rakete1    Atmosphäre verlassen
rakete1    Vorbereitung Wiedereintritt
rakete1    Wiedereintritt erfolgreich
rakete1    Landung erfolgreich
```

Wir sehen hier, dass die Daten in einer ganz anderen Reihenfolge sind als beim Produzieren, aber wir sehen, dass Nachrichten mit dem gleichen Key in der richtigen Reihenfolge auftauchen, auch wenn zwischendurch andere Keys zu sehen sind.

2.2.2 Consumer Groups

Diese Partitionen können wir nun auf unterschiedliche Broker verteilen und somit die Last zwischen den Brokern balancieren. Die Producer entscheiden selbstständig, welche Daten auf welche Partitionen verteilt werden sollten. Wenn wir keine Keys benutzen, verteilen unsere Producer die Daten per Round-Robin-Verfahren auf die Partitionen, also erst auf Partition 0, dann Partition 1 und so weiter. Wenn wir Keys benutzen, dann verteilen wir die Nachrichten so auf die Partitionen, dass Nachrichten mit demselben Key auf derselben Partition landen.

Wir haben aber weiterhin nur einen Consumer. Der Consumer muss nun alle Daten aus allen Partitionen verarbeiten. Hier kennen wir bisher noch keine Möglichkeit, um unsere Consumer zu skalieren. So kann es schnell dazu führen, dass unser Consumer überlastet wird und unser System die Vorteile von Kafka nicht ausreizen kann.

Meistens haben wir auch nicht nur eine Art Consumer, die dieselben Daten konsumieren möchten. In unserem Raketen-Beispiel können wir uns vorstellen, dass wir einen Service haben, der Daten aus dem *flugdaten*-Topic liest und diese dann auf einem großen Bildschirm anzeigt. Für diesen Service würde es vollkommen ausreichen, nur einen Consumer zu benutzen, da dies eine sehr einfache und ressourcenschonende Aufgabe ist. Zusätzlich haben wir einen anderen Service, der die Daten aus demselben *flugdaten*-Topic wissenschaftlich auswerten möchte. Dafür reicht uns ein Consumer wahrscheinlich nicht aus. Es würde viel zu lange dauern, diese Daten mit nur einem Consumer auszuwerten.

Für dieses Szenario benötigen wir zwei Dinge. Erstens benötigen wir eine Möglichkeit, damit diese unterschiedlichen Services unabhängig voneinander Daten aus den Partitionen lesen können, ohne dass sie sich dabei gegenseitig beeinflussen. Zum Beispiel sollte der wissenschaftliche Analytics-Service in der Lage sein, Daten zu lesen, obwohl unser Metriken-Service sie schon gelesen hat. Dies ist zum Glück in Kafka trivial möglich, da die Consumer selbstständig mithilfe der Offsets entscheiden, welche Daten sie lesen möchten.

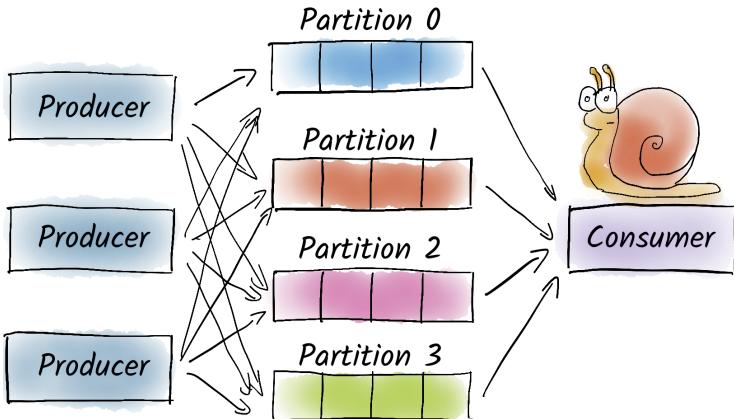


Bild 2.4 Topics bestehen aus einer oder mehreren Partitionen, um unsere Last besser zu verteilen und unsere Verarbeitung zu parallelisieren. Producer entscheiden selbstständig anhand der Daten, in welche Partition sie schreiben möchten. Oft schreibt jeder Producer in jede Partition eines Topics. Dies funktioniert meistens sehr performant. Haben wir aber nur einen Consumer, der Daten aus allen Partitionen lesen muss, kann es sein, dass dieser nicht hinterherkommt und wir die Daten nicht zeitnah verarbeiten können. Da Kafka Daten nicht pusht, sondern der Consumer sich die Daten selbst abholt, kann bei korrekter Programmierung der Consumer nicht überlastet werden. Er verarbeitet die Daten nur entsprechend langsamer.

Zweitens benötigen wir eine Möglichkeit, unsere Services horizontal zu skalieren, sodass die Daten der Partitionen auf die unterschiedlichen Instanzen des Service so aufgeteilt werden, dass alle Daten gelesen werden, aber auch, dass die Reihenfolgen korrekt eingehalten werden. Um dies sicherzustellen, darf jede Partition nur von genau einer Instanz pro Service konsumiert werden. Es ist nicht möglich, dass eine Partition von zwei Instanzen desselben Service konsumiert wird und dabei noch die Reihenfolge-Garantien eingehalten werden.

Um beides zu lösen, nutzen wir in Kafka sogenannte *Consumer Groups*. Verschiedene Consumer bilden eine Consumer Group, wenn deren group.id gleich ist. Consumer in einer Consumer Group verteilen die Last gleichmäßig über die Mitglieder der Gruppe und schreiben ihre Offsets nach Kafka. Das heißt, wenn wir einen Consumer mit einer Consumer Group starten, wieder stoppen und danach wieder starten, merkt sich der Consumer, welche Daten er schon gelesen hat.

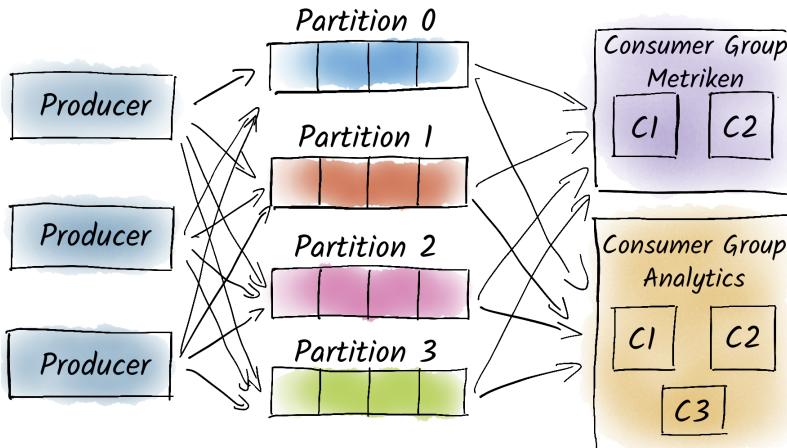


Bild 2.5 Consumer-Groups ermöglichen es uns, die Verarbeitung von mehreren Partitionen zwischen unterschiedlichen Instanzen desselben Service aufzuteilen. Oft konsumiert nicht eine Consumer-Group die Daten aus einem Topic, sondern mehrere. Consumer-Groups sind voneinander isoliert und beeinflussen sich nicht gegenseitig.

Probieren wir es für unser *flugdaten-partitionen-keys*-Topic aus dem letzten Kapitel aus:

```
$ kafka-console-consumer.sh \
  --from-beginning \
  --topic flugdaten-partitionen-keys \
  --property print.key=true \
  --group=metriken \
  --bootstrap-server=localhost:9092
rakete0  Countdown beendet
rakete0  Liftoff
rakete0  Atmosphäre verlassen
rakete0  Vorbereitung Wiedereintritt
rakete0  Wiedereintritt erfolgreich
rakete1  Countdown beendet
rakete0  Landung erfolgreich
rakete1  Liftoff
rakete1  Atmosphäre verlassen
rakete1  Vorbereitung Wiedereintritt
rakete1  Wiedereintritt erfolgreich
rakete1  Landung erfolgreich
Processed a total of 12 messages
```

Das Einzige, was wir hier an dem Befehl verändert haben, ist es, die Consumer Group anzugeben (`--group=metriken`). Wenn wir den Befehl erneut ausführen, sehen wir keine Nachrichten und bekommen, nachdem wir **Strg+C** gedrückt haben, folgende Information:

```
Processed a total of 0 messages
```

Um zu sehen, wie die Consumer einer Consumer Group die Last untereinander aufteilen, starten wir mehrere Consumer gleichzeitig mit der gleichen Group-ID. Dafür können wir den gerade eben verwendeten Befehl in zwei Terminals ausführen und beide Terminal-Fenster nebeneinander anzeigen, damit wir die Ausgabe gleichzeitig sehen. Starten wir zusätzlich dazu einen Producer, der noch ein paar Daten produziert:

```
$ kafka-console-producer.sh \
  --topic flugdaten-partitionen-keys \
  --property parse.key=true \
  --property key.separator=: \
  --bootstrap-server=localhost:9092
>rakete2:Countdown beendet
>rakete3:Countdown beendet
>rakete2:Liftoff
>rakete2:Atmosphäre verlassen
>rakete3:Liftoff
>rakete3:Atmosphäre verlassen
# STRG+D drücken zum abbrechen
```

Nun sollten wir bei unseren Consumern ein Bild ähnlich dem folgenden sehen:

Consumer 1	Consumer 2
rakete2 Countdown beendet	rakete3 Countdown beendet
rakete2 Liftoff	rakete3 Liftoff
rakete2 Atmosphäre verlassen	rakete3 Atmosphäre verlassen

Das heißt, dass die Consumer der Consumer Group erfolgreich die Arbeit untereinander aufgeteilt haben. Consumer 1 war für die Partition, auf der die Nachrichten für rakete2 liegen, zuständig. Consumer 2 hat sich um die andere Partition gekümmert. Da wir unsere Consumer einer Consumer Group nur so weit skalieren können, wie wir Partitionen haben, ist es nicht sinnvoll, mehr als zwei Consumer in einer Gruppe zu haben, da wir das Topic mit nur zwei Partitionen angelegt haben. Wir werden uns im Kapitel „Performance“ nochmals ausgiebiger mit Consumer Groups beschäftigen.

2.2.2.3 Replikation

Nachdem wir Partitionierung als Möglichkeit kennengelernt haben, um Daten zuverlässig auf mehrere Systeme zu verteilen, möchten wir als Nächstes die Zuverlässigkeit unseres Systems verbessern. Vielleicht kennen wir es aus eigener Erfahrung, dass Computersysteme ohne ersichtlichen Grund ausfallen, Festplatten sind plötzlich defekt, jemand hat den falschen Ordner gelöscht und so weiter. Es gibt viele Gründe, warum wir Daten verlieren können. Die zuverlässigste Methode zur Vermeidung von Datenverlusten ist die Replikation. Im Kleinen machen wir dies (hoffentlich) im Privaten. Wir richten Backups ein, um im Falle eines Systemausfalls nicht alle Daten zu verlieren. Backups setzen wir auch im Unternehmenskontext ein. Der Nachteil von Backups ist, dass es Zeit kostet, diese wieder einzuspielen. Wir können mit Backups keinen unterbrechungsfreien Betrieb garantieren, sondern haben für Katastrophenfälle die Möglichkeit, Daten wiederherzustellen. Deswegen bezeichnen wir Backups auch als *kalte Replikation*. Die Daten sind repliziert, aber unter Umständen kann es lange dauern, bis wir auf sie zugreifen können. Kafka selbst hat keine Unterstützung für eine solche kalte Replikation. Wenn wir Backups wünschen, müssen wir uns selbst darum kümmern.

Stattdessen können wir Kafkas Replikationsstrategie als *warm* bezeichnen. Statt Daten nur auf einem einzigen Server zu speichern, legt Kafka die Daten auf mehrere laufende Server ab. Falls ein Kafka-Broker ausfällt, kann ein anderer Broker die Arbeit übernehmen und weiterhin sowohl Anfragen von Producern als auch Consumern entgegennehmen.

In Kafka konfigurieren wir die Replikation auf Topic-Ebene. Für jedes Topic können wir unabhängig von anderen Topics den *Replication Factor* einstellen. Dies tun wir üblicher-

weise beim Anlegen eines Topics. Legen wir beispielsweise das Topic *replication-test* mit drei Partitionen und einem *Replication Factor* von drei an:

```
$ kafka-topics.sh \
  --create \
  --topic replication-test \
  --partitions 3 \
  --replication-factor 3 \
  --bootstrap-server localhost:9092
Created topic replication-test.
```

Logischerweise kann unser *Replication Factor* nicht größer sein als die Anzahl an Brokern, die wir zur Verfügung haben. In unserer Testumgebung, die wir im Anhang beschrieben haben, haben wir drei Kafka-Broker. Wenn wir versuchen, ein Topic mit vier Replicas anzulegen, bekommen wir eine Fehlermeldung:

```
Error while executing topic command : Replication factor: 4 larger than available
brokers: 3.
[2021-01-15 11:08:43,750] ERROR org.apache.kafka.common.errors.
InvalidReplicationFactorException: Replication factor: 4 larger than available
brokers: 3.
(kafka.admin.TopicCommand$)
```

Schauen wir uns doch nun genauer an, was nach dem Anlegen des Topics geschehen ist. Dazu nutzen wir wieder das Kommandozeilenwerkzeug `kafka-topics.sh` mit dem Flag `--describe`:

```
$ kafka-topics.sh \
  --describe \
  --topic replication-test \
  --bootstrap-server localhost:9092
Topic: replication-test PartitionCount: 3 ReplicationFactor: 3 Configs:
Topic: replication-test Partition: 0 Leader: 2 Replicas: 2,1,3 Isr: 2,1,3
Topic: replication-test Partition: 1 Leader: 3 Replicas: 3,2,1 Isr: 3,2,1
Topic: replication-test Partition: 2 Leader: 1 Replicas: 1,3,2 Isr: 1,3,2
```

In Zeile 5 zeigt uns der Befehl an, dass wir wie gewollt drei Partitionen und drei Replicas haben. In Zeile 6 bis 8 zeigt uns der Befehl pro Zeile Informationen zu einer Partition an. Wir sehen, dass für Partition 0 der sogenannte Leader 2 ist. Was bedeutet dies aber?

Kafkas Replikationsstrategie folgt dem *Ein Leader - mehrere Follower*-Prinzip. Das heißt im Konkreten, dass es für jede Partition einen Broker, der Leader ist, gibt. In unserer Beispiel-ausgabe bedeutet dies, dass Broker 2 der Leader für die Partition 0 des Topics *replication-test* ist. Abhängig vom *Replication Factor* kann es weitere Broker geben, die Follower sind. Wir bezeichnen alle diese Broker, auf denen diese Partition abgelegt ist, als Replicas. Das heißt, bei einem *Replication Factor* von eins haben wir im Normalzustand ein Replica und einen Leader und keinen Follower. Bei einem *Replication Factor* von drei, wie in unserem Beispiel, haben wir im Normalzustand drei Replicas, einen Leader und zwei Follower. Mit diesem Wissen können wir auch die weiteren Angaben des `kafka-topics.sh`-Befehls entschlüsseln. **Replicas: 1,3,2** besagt lediglich, dass die Replicas auf den Brokern 1, 3 und 2 liegen. Standardmäßig ist der Broker, dessen ID an erster Stelle in der Liste der Replicas steht, Leader für diese Partition. Wir können an der Ausgabe sehen, dass sich dies auch für die beiden anderen beiden Partitionen deckt. Der Broker 3 steht bei Partition 1 an erster Stelle und ist auch Leader. Bei Partition 2 stimmt dies mit Broker 1 als Leader auch überein.

Zu guter Letzt spricht die Aussage von sogenannten *ISR* (*Isr*: 2,1,3). *ISR* steht für *In-Sync Replicas*. Das sind also die Replicas, die in-sync sind, also auf dem aktuellen Stand. Wenn die *ISR*-Liste nicht mit der Liste der Replicas übereinstimmt, dann wissen wir, dass etwas im Argen liegt und unter Umständen ein Broker ausgefallen ist.

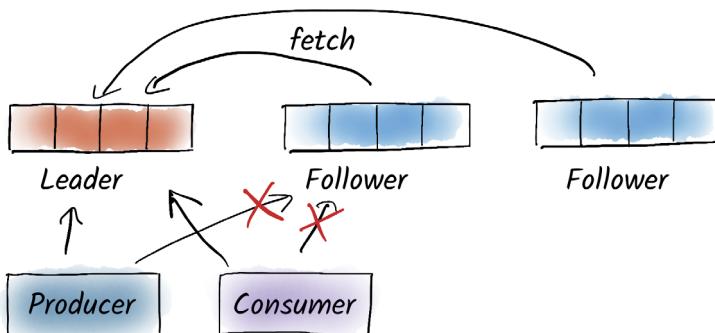


Bild 2.6 Consumer und Producer kommunizieren ausschließlich mit dem Leader (siehe Fußnote für Ausnahmen). Follower sind lediglich dazu da, kontinuierlich neue Nachrichten vom Leader zu replizieren. Fällt der Leader aus, übernimmt einer der Follower.

Wir sprachen bereits davon, dass Kafka einer *Ein Leader – mehrere Follower*-Architektur folgt. Das heißt konkret, dass es pro Partition einen Leader gibt und alle Lese und Schreibvorgänge¹⁰ nur über den Leader erfolgen. Die Follower sind einzig und allein dafür da, die Daten so schnell wie möglich vom Leader zu sich zu replizieren und verfügbar zu sein, falls der Leader ausfällt. Wenn der Leader ausfällt, übernimmt einer der Follower, die in-sync sind (dafür gibt es die *ISR*-Liste), die Arbeit des Leaders und wird selbst zum neuen Leader. Die Producer und Consumer schwenken dann automatisch auf den neuen Leader um.

Der große Vorteil daran, dass wir in Kafka Logs benutzen, ist, dass die Replikation sehr einfach umgesetzt werden kann. Die Follower fragen ähnlich wie die Consumer den Leader an, ob dieser neue Nachrichten ab einem gewissen Offset hat, und schreiben diese Daten dann zu sich in ein lokales Log. Somit erreichen wir nicht nur eine hohe Performance und relativ einfachen Code, sondern können ohne Weiteres unsere Reihenfolge weiterhin garantieren.

Nun stellt sich die Frage, wenn wir nur einen Leader haben, dann ist die Last doch ungleichmäßig über die Broker verteilt. Das wäre ein Problem, falls wir in unserem Kafka-Cluster nur eine Partition hätten. Aber üblicherweise haben wir viele Partitionen. Es gibt nämlich nicht den einen Broker, der Leader für alle Partitionen ist, sondern es gibt für jede Partition einen Leader. Kafka versucht beim Erstellen von Partitionen, die Leader so gleichmäßig wie möglich über die Broker zu verteilen. Wir müssen während des Betriebs von Produktionsystemen sehr auf diese gleichmäßige Verteilung der Last achten, denn sie ist ein Garant für Kafkas Performance.

In diesem Abschnitt haben wir Kafka als verteilten Log kennengelernt. Wir benutzen Replikation, um einerseits Datenverlust vorzubeugen und andererseits auch unsere Verfügbar-

¹⁰⁾ Dies ist nicht ganz korrekt: Mit KIP-392 (<https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica>) wurde eine Möglichkeit geschaffen, in Multi-Rechenzentren-Szenarien von Followern im gleichen Rechenzentrum zu lesen. Dazu aber später mehr.

keit zu erhöhen. Dank Replikation sind wir in der Lage, den Ausfall eines oder sogar mehrerer Broker ohne Einschränkungen des Betriebs zu überstehen. Eine der Hauptmotivationen dafür, den Log zu verteilen, war es, die Performance unseres Gesamtsystems zu erhöhen. Wir erreichen dies durch Partitionierung: Statt eines großen Logs, in das wir alles schreiben und das zwangsläufig auf einem einzigen Rechner liegen muss, partitionieren wir das Log, sodass Daten, die zusammengehören müssen, weiterhin zusammenbleiben, aber Daten, die unabhängig voneinander sind, auch auf unterschiedlichen Brokern liegen können. Diese Partitionierung bringt uns eine Einschränkung: Wir können die Reihenfolge der Nachrichten nur in einer Partition garantieren, aber nicht über Partitionsgrenzen hinweg. In der Praxis bereitet dies sehr wenige Probleme, solange wir sinnvolle Keys wählen. Damit können wir die Last viel besser über die Broker verteilen und damit auch eine sehr hohe Performance erreichen.

2.2.3 Komponenten

Nachdem wir in den letzten Kapiteln bereits sehr viel über die Konzepte hinter Kafka gelernt haben, widmen wir uns in diesem Kapitel nochmals gesondert den Komponenten von Kafka. Im Prinzip besteht ein Kafka-Cluster nur aus drei verschiedenen Komponenten: Koordinationscluster, Broker und Clients.

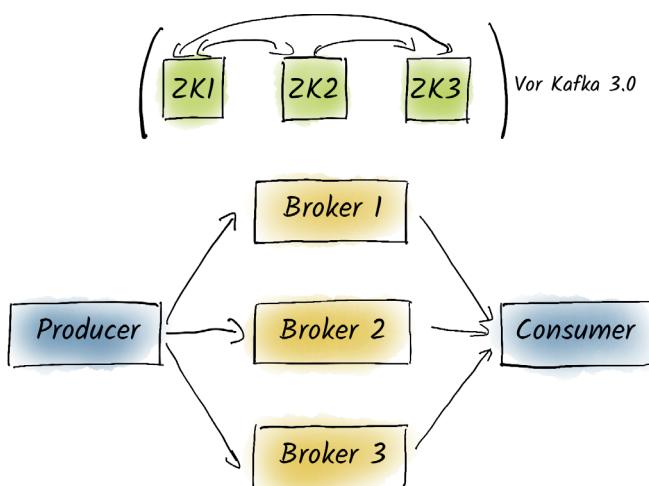


Bild 2.7 Eine typische Kafka-Umgebung besteht aus dem Kafka-Cluster selbst und den Clients, die Daten nach Kafka schreiben und lesen. Vor Kafka 3.0 benötigten wir noch zusätzlich ein Zookeeper-Ensemble als Koordinationscluster. Ab Kafka 3.0 können die Broker die Aufgabe des Koordinationsclusters selbst übernehmen oder an einen eigenständigen Koordinationscluster auslagern.

Das Koordinationscluster und die Broker sind uns in den vergangenen Kapiteln schon mehrfach begegnet, aber was hat es mit Clients auf sich und welche Rolle spielen Consumer und Producer? Die Antwort auf diese Frage ist recht einfach, denn unter Clients verstehen wir alle Anwendungen, die sich mit unserem Kafka-Cluster verbinden, und dazu zählen eben auch die uns bereits bekannten Consumer und Producer. Ein typisches Kafka-Cluster

besteht in der Regel aus einem Koordinationscluster, zusammengesetzt aus mindestens drei Brokern, und drei weiteren Brokern für unser eigentliches Cluster. Den Grund dafür werden wir im Verlauf des Kapitels näher erläutern.

2.2.3.1 Koordinationscluster

Ein verteiltes System wie Kafka benötigt zur Koordination ein gewisses Maß an Orchestrierung. Diese Aufgabe wird in Kafka von dem Koordinationscluster übernommen.

Der Koordinationscluster überwacht laufend den aktuellen Zustand des Clusters und überprüft, ob alle Broker noch erreichbar und funktional sind. Im Rahmen dieser Aufgabe verwaltet er auch die Liste aller aktiven Broker und ist dafür verantwortlich, neue Broker hinzuzufügen und bestehende Broker aus dem Cluster zu entfernen. Der Koordinationscluster speichert Metadaten zu Topics und verwaltet den Zugriff auf Topics. Im Rahmen der Topic-Verwaltung speichert er auch die Zuweisung von Partitionen zu Brokern und den Leader-Broker für die jeweiligen Partitionen. Mithilfe des Koordinationsclusters wird einer der Broker zum sogenannten Controller bestimmt. Der Controller ist dafür verantwortlich, den Zustand der einzelnen Partitionen und Replicas zu verwalten. In dieser Tätigkeit kümmert er sich zum Beispiel darum, Partitionen Broker zuzuweisen und einen Partitions-Leader zu bestimmen.

Vor Kafka 3.0 wurde die Rolle des Koordinationsclusters durch ein *Zookeeper*-Ensemble ausgefüllt. Zookeeper¹¹ selbst ist kein Teil von Kafka, sondern ein eigenständiges Projekt der Apache Software Foundation.

Ab Kafka 3.0 nutzt Kafka stattdessen ein auf Kafka basierendes Koordinationscluster. Dies ermöglicht es uns sogar, auf ein eigenständiges Koordinationscluster zu verzichten, da wir es in unserem eigentlichen Kafka-Cluster betreiben können. Hierbei übernehmen einige unserer normalen Broker zusätzlich die Aufgaben eines Koordinationsclusters. Dies kann insbesondere bei kleinen Kafka-Clustern von Vorteil sein. Bei größeren Clustern mit einer großen Anzahl an Brokern empfiehlt es sich dennoch, ein eigenständiges Koordinationscluster zu erstellen.

Zur Orchestrierung unseres Clusters muss sich die Mehrheit unserer einzelnen Instanzen (Kafka-Broker beziehungsweise Zookeeper) im Koordinationscluster einigen. Die Anzahl an Instanzen in unserem Koordinationscluster hat daher Auswirkungen auf die Ausfallsicherheit unseres Kafka-Clusters. Bei drei Instanzen können wir noch den Ausfall eines Brokers beziehungsweise Zookeepers verkraften, bei fünf Instanzen können wir sogar den Ausfall von zwei Systemen überstehen. In jedem Fall sollte die Anzahl an Instanzen in unserem Koordinationscluster ungerade sein, da wir bei einer geraden Anzahl an Instanzen gegenüber der nächstniedrigeren ungeraden Anzahl keinen Vorteil bezüglich Ausfallsicherheit erzielen.



Tipp

Für kleinere Kafka-Cluster empfiehlt sich ein Koordinationscluster bestehend aus drei Brokern beziehungsweise Zookeeper-Instanzen. Für größere Cluster können wir zur Erhöhung der Ausfallsicherheit auch fünf Instanzen starten.

¹¹⁾ <https://zookeeper.apache.org/>

2.2.3.2 Broker

Broker bilden in Kafka das eigentliche Cluster, da sie verantwortlich für das Empfangen, die Speicherung und das Versenden von Nachrichten beziehungsweise Daten sind. Ein Kafka-Cluster besteht in der Regel aus mehreren Brokern, weil wir ansonsten die Daten nicht replizieren können, was eines der Kernkonzepte von Kafka ist. Wenn unser Kafka-Cluster aus zwei Brokern bestünde, könnten wir zwar bereits replizieren, bekämen aber im Zweifel relativ schnell Probleme. Stellen wir uns zum Beispiel vor, dass wir unser Cluster warten wollen und deshalb einer der Broker offline ist. Fiele der zweite und finale Broker zeitgleich aus, könnte dies schnell zu Datenverlusten führen. Wir sollten daher immer mindestens mit drei Brokern starten. Sollte die Last größer werden, können wir weitere Broker hinzufügen, um die Performance zu erhöhen.

2.2.3.3 Clients

Unter Clients verstehen wir in Kafka jegliche Anwendungen, die sich von außen mit unserem Cluster verbinden. Demzufolge sind sie genau genommen auch keine Komponenten von Kafka. Unsere mittlerweile altbekannten `kafka-console-consumer.sh` und `kafka-console-producer.sh` sind auch nichts anderes als Kafka-Clients. Beide bestehen letzten Endes aus einer Java-Anwendung, die auf die Kafka-API zugreift. Clients, die nicht in Java geschrieben sind, nutzen meist die Bibliothek `librdkafka`¹². Wir gehen in Kapitel 3 „Kafka Deep Dive“ näher auf die Funktionsweise der Clients ein.

Die einfachste Form von Kafka-Clients sind Producer und Consumer, welche wir bereits kennengelernt haben. Producer dienen dazu, Kafka-Cluster mit Daten zu füllen. Consumer wiederum sind dazu da, Daten aus Kafka zu lesen. Wir können es auch so beschreiben, dass Producer Daten produzieren und Consumer Daten konsumieren, womit sich auch die Namensgebung erklärt. *Kafka Streams*¹³ ist ein weiterer wichtiger Vertreter der Kafka-Clients. *Kafka Streams* dient dazu, Daten von einem Kafka-Cluster zu lesen und sie in ein anderes Kafka-Cluster zu schreiben. *Kafka Streams* übernimmt im Prinzip also gleichzeitig die Funktion eines Consumers und Producers. *Kafka Connect*¹⁴ ist ein Client, der dazu dient, Kafka mit anderen Systemen zu verbinden. So können zum Beispiel ganze Datenbanken via Kafka Connect aus einem Kafka-Cluster befüllt werden. Wie genau sich externe Systeme mit Kafka verbinden lassen und insbesondere wie *Kafka Streams* und *Kafka Connect* im Detail funktionieren, werden wir uns im Kapitel 4.1 „Kafka-Ökosystem“ ausführlich ansehen.

2.2.4 Kafka im Unternehmenseinsatz

Wir haben auf den letzten Seiten Kafka als verteilten Log kennengelernt. Im Einsatz finden wir Kafka als verteiltes System mit mehreren Brokern, einem Koordinationscluster und natürlich den ganzen Producern und Consumern, die wir benötigen, um mit Kafka Daten auszutauschen und somit unsere Business-Probleme zu lösen. Oft bringt allein diese Architektur immensen Wert in unser Unternehmen. Aber oft benötigen wir mehr als nur einen

¹²⁾ <https://github.com/edenhill/librdkafka/>

¹³⁾ <https://kafka.apache.org/documentation/streams/>

¹⁴⁾ <https://kafka.apache.org/documentation/#connect>

zentralen Ort, um Daten abzulegen und weiterzuleiten. Kafka als Zentrum einer Streaming-Plattform kann uns helfen, unser Unternehmen hin zu einem datengetriebenen Unternehmen zu bewegen und damit in Nahe-Echtzeit Entscheidungen zu treffen. Dazu reicht aber Kafka allein nicht aus.

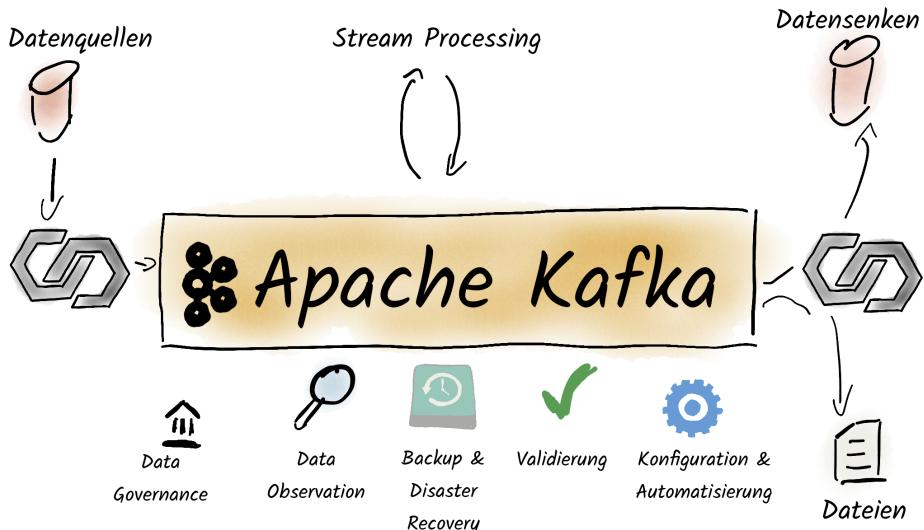


Bild 2.8 Kafka allein reicht meistens nicht. Das Kafka-Ökosystem bietet zahlreiche Komponenten, um Kafka in unsere Unternehmenslandschaft einzubinden und somit eine Streaming-Plattform aufzubauen.

Unser Kafka-Cluster lebt nicht in Isolation und sollte deshalb gut an unsere vorhandenen Systeme im Unternehmen angebunden sein. Die zahlreichen Datenbanksysteme, die unsere Kerndaten speichern, Messaging-Lösungen, mit denen proprietäre Dienste und externe Dienstleister angeschlossen sind, Alt-Applikationen, die niemand mehr anzufassen wagt, und vieles mehr. Natürlich könnten wir für jeden dieser Dienste einen eigenen Kafka Consumer oder Producer schreiben, um sie mit Kafka zu verbinden. Aber glücklicherweise bietet Kafka mit *Kafka Connect* ein Werkzeug und Framework, um Drittsysteme effizient und zuverlässig an Kafka anzubinden. Kafka Connect wird direkt mit Apache Kafka ausgeliefert und steht unter derselben Apache 2.0-Lizenz. Vielleicht haben wir noch Mainframes im Unternehmenseinsatz, für die es keine Kafka-Bibliotheken gibt und wo es sehr mühselig wäre, eigene Connectoren für Kafka Connect zu schreiben. Um diese Systeme anzubinden, können wir zum Beispiel einen der auf dem Markt vorhandenen REST Proxys¹⁵ anbinden. Damit können alle Systeme, die per HTTP/REST kommunizieren, mit Kafka interagieren.

Jetzt haben wir unter Umständen zahlreiche Systeme an Kafka angebunden und können zwischen den Systemen Daten austauschen. Oft möchten wir in Nahe-Echtzeit nicht nur die Daten versenden, sondern auch auswerten und manipulieren. Da wir die Daten in Kafka als Datenstrom ansehen können, eignet sich dazu ein Stream-Processing-System, also ein Sys-

¹⁵⁾ Confluent bietet mit dem REST Proxy einen proprietären REST Proxy für seine Confluent Enterprise-Plattform an. Alternativ gibt es von Aiven eine Open-Source-Variante des REST Proxys: <https://github.com/aiven/karapace>

tem, welches direkt auf den Datenströmen arbeitet. Ursprünglich gab es eine große Menge an konkurrierenden Frameworks, um Stream Processing mit Kafka umzusetzen, wie zum Beispiel Apache Flink, Scala's Akka und Ähnliche. Inzwischen hat sich das mit Kafka ausgelieferte *Kafka Streams* als De-facto-Standard etabliert. Mithilfe von Kafka Streams ist es relativ einfach, auch auf großen Datenmengen skalierende und robuste Stream-Processing-Operationen zu implementieren, also Operation wie das Filtern von Daten, der Manipulation von einzelnen Datensätzen und dem Join, also dem Verbinden von unterschiedlichen Datenströmen. Es gibt auch einige Angebote auf dem Markt, um dieses Stream Processing auch ohne Programmierkenntnisse zu benutzen. Populäre Produkte sind zum Beispiel Confluent's *ksqlDB* oder *lenses.io*. Beide kommerziellen Produkte werden mit einem SQL-ähnlichen Dialekt bedient.

Je mehr Daten wir in Kafka abspeichern und umso länger wir diese Daten aufbewahren wollen, desto wichtiger ist es, dass die Daten in einem einheitlichen Datenformat abgespeichert werden. Insbesondere wenn mehr als ein Team auf den Kafka-Cluster zugreift, ist Schema-Management von sehr hoher Relevanz. Mit Apache Kafka wird allerdings kein Schema-Management ausgeliefert, da sich Kafka selbst nicht für das Datenformat interessiert. Es gibt einige mögliche Ansätze für das Management von Schemas. Angefangen von Schemadefinitionen in einem Git Repository bis hin zu eigenen Komponenten wie Confluent's Schema Registry oder der in Karapace implementierten Schema Registry¹⁶.

Je nachdem, wie wichtig die Daten sind, die wir in Kafka speichern, sollten wir darüber nachdenken, ob es Sinn ergibt, diese in weitere Rechenzentren zu spiegeln. Mit Kafka *MirrorMaker 2* gibt es inzwischen ein mit Kafka mitgeliefertes Werkzeug, welches den Multi-Rechenzentrumsbetrieb von Kafka stark vereinfacht. In einigen Fällen lohnt es sich weiterhin, sich Gedanken über ein Backup und Wiederherstellungskonzept zu machen. Dies ist insbesondere bei größeren Kafka-Installationen anspruchsvoller und umfangreicher, als es auf den ersten Blick scheint.

Darüber hinaus benötigt jede Kafka-Installation, die mehr sein soll als eine Spielwiese zum Ausprobieren der Technologie, weitere Komponenten. Die Streaming-Plattform muss umfangreich mithilfe eines Monitor-Werkzeugs überwacht werden. Kafka sollte bestenfalls nicht manuell aufgesetzt und konfiguriert werden, sondern mithilfe von Automatisierungswerkzeugen. Darüber hinaus ist es insbesondere in größeren Unternehmen wichtig, auf bestimmte Compliance-Regeln zu achten und diese auch durchzusetzen.

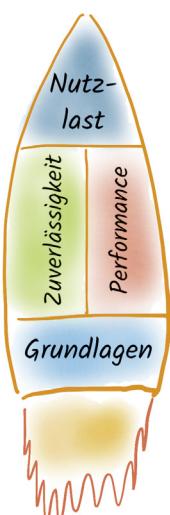
Kafka als Streaming-Plattform in einer hohen Detailtiefe zu betrachten, würde den Umfang dieses Buches bei Weitem sprengen, und dieser Bereich entwickelt sich zum Zeitpunkt des Schreibens sehr dynamisch. Wir gehen im Kapitel „Kafka im Unternehmenseinsatz“ auf einige dieser Punkte in größerem Umfang als hier ein, aber empfehlen den Lesern bei weiteren Fragen, sich bei einem Kafka-Experten über den aktuellen Stand der Dinge zu erkundigen.

¹⁶⁾ Confluent's Schema Registry ist Teil der Confluent Enterprise-Plattform. Karapace, bestehend aus dem oben besprochenen REST Proxy und der Schema Registry, ist auf GitHub zu finden: <https://github.com/aiven/karapace>

2.2.5 Zusammenfassung

Wir haben in diesem Kapitel Kafka als verteilten Log betrachtet. Hierfür haben wir zunächst erläutert, was genau ein Log ist und wo uns außerhalb der Kafka-Welt überall Logs begegnen. Wir haben die grundlegenden Eigenschaften eines Logs kennengelernt und basierend darauf über die Rolle des Logs in Kafka gesprochen. Im Anschluss haben wir uns mit verteilten Systemen beschäftigt und in diesem Zusammenhang bereits erste Einblicke gewinnen können, wie Kafka versucht, die Komplexität von verteilten Systemen zu meistern. Wir haben erfahren, dass wir in Kafka durch Partitionierung viele Aufgaben parallelisieren können, was sich natürlich positiv auf die Leistungsfähigkeit unseres Systems auswirkt. Gleichzeitig haben wir mit den Consumer Groups ein Konzept kennengelernt, welches von Kafka verwendet wird, um das hierbei entstehende Chaos zu verwalten. Außerdem haben wir einen ersten Einblick bekommen, wie Kafka durch seine Replikationsstrategie sowohl die Zuverlässigkeit als auch die Verfügbarkeit verbessert. Im Anschluss haben wir uns mit den verschiedenen Komponenten von Kafka beschäftigt. Wir haben gelernt, dass es einen Koordinationscluster gibt, welcher unseren eigentlichen Kafka-Cluster verwaltet. Ein Kafka-Cluster besteht wiederum aus einer Anzahl von Brokern, welche sich um unsere Topics und die darin enthaltenen Nachrichten kümmern. Als letzte Komponenten haben wir noch die Clients kennengelernt, welche dazu dienen, Nachrichten zu produzieren, zu konsumieren oder sogar beides gleichzeitig. Zum Abschluss dieses Kapitels haben wir noch einen Blick darauf geworfen, wie Kafka sich in Unternehmen einfügt, und kurz die verschiedenen Anwendungsfälle beleuchtet.

■ 2.3 Zuverlässigkeit



In den letzten Kapiteln haben wir bereits sehr viel über Kafka gelernt. Erinnern wir uns kurz zurück an unsere Metapher der Kafka-Rakete aus der Einleitung.

Wir wissen, aus welchen Komponenten Kafka besteht und wie Kafka grob funktioniert. Außerdem haben wir in Erfahrung gebracht, wie Nachrichten in Kafka strukturiert sind. Wir wissen nun also bereits, wie der Payload unserer Kafka-Rakete aussieht und wie in etwa der Antrieb funktioniert. Aber ohne Treibstoff fliegt selbst die beste Rakete nicht. In den nächsten zwei Kapiteln werden wir uns daher mit den beiden Treibstofftanks unserer Rakete beschäftigen. Bevor wir uns im nächsten Kapitel genauer ansehen, wie wir die Performance unseres Kafka-Clusters beeinflussen können, schauen wir uns in diesem Kapitel an, welche Parameter wir haben, um die Zuverlässigkeit zu verbessern.

Was meinen wir eigentlich, wenn wir von Zuverlässigkeit im Zusammenhang mit Kafka reden? Wenn wir das Thema Zuverlässigkeit betrachten, können wir es grob in drei Unterkategorien einteilen. Die erste Kategorie ist Datenhaltbarkeit und beschäftigt sich mit der Frage, wie Kafka sicherstellt, dass Daten auf Dauer korrekt gespeichert werden. Die zweite Kategorie ist Erreichbarkeit und bezieht

sich zum einen darauf, dass Consumer möglichst jederzeit auf unsere geschriebenen Daten zugreifen können, und zum anderen darauf, dass auch Producer jederzeit Daten schreiben können. Sowohl Datenhaltbarkeit und Erreichbarkeit lassen sich in Kafka relativ gut mit Replikation erreichen. Etwas problematischer wird es allerdings, wenn wir Konsistenz, die dritte Kategorie in Kafkas Zuverlässigkeit konzept, ins Spiel bringen. Konsistenz selbst hat auch verschiedene Ausprägungen, die wir im Laufe dieses Kapitels noch genauer untersuchen werden, aber letztlich geht es darum, dass die Daten im Cluster genausopersistiert werden, wie es der Producer wünscht. Das Erreichen von Konsistenz in einem verteilten System wie Kafka ist natürlich eine Herausforderung, und Kafkas Datenreplikation macht das Ganze auch nicht leichter. Zur Lösung dieses Problems bedient sich Kafka des Konzepts der *Acknowledgements* (ACKs).

Bevor wir uns aber in den nächsten beiden Abschnitten mit den Details von ACKs und Replikation in Kafka beschäftigen, betrachten wir kurz an einem einfachen Beispiel, welche Probleme in Bezug auf Datenhaltbarkeit, Erreichbarkeit und Konsistenz überhaupt entstehen können. Stellen wir uns vor, wir haben, wie in unseren bisherigen Beispielen, ein Kafka-Cluster bestehend aus drei Brokern. Solange nichts schiefgeht, gibt es natürlich auch keine Probleme mit Datenhaltbarkeit, Erreichbarkeit und Konsistenz. Was passiert aber, wenn einer der Broker plötzlich ausfällt? Sind unsere Topics dann noch lesbar? Was bedeutet das für die Konsistenz unserer Daten? Können wir weiterhin Daten produzieren und schreiben? Was passiert, wenn der Broker auf einmal wieder online ist oder sogar noch ein weiterer Broker ausfällt? Ist dann immer noch die persistente Speicherung unserer Daten gewährleistet? Können wir weiterhin Daten lesen, oder führt das zu Problemen in der Konsistenz? Hier sehen wir bereits, dass wir manchmal auch Prioritäten in Bezug auf die einzelnen Eigenschaften setzen müssen. Was ist uns wichtiger? Möglichst immer Daten schreiben und lesen zu können oder sicherzustellen, dass die Daten korrekt sind? Alle diese Fragen werden wir in den nächsten beiden Unterkapiteln beantworten.

2.3.1 Acknowledgements

Acknowledgements werden von vielen Netzwerkprotokollen, wie zum Beispiel *TCP* oder auch *IEEE802.11* (WiFi), verwendet, um eine erfolgreiche Datenübertragung zu bestätigen. Doch nicht immer reichen die Mechanismen von niedrigeren Kommunikationsschichten aus, um eine erfolgreiche Übermittlung von Daten zu garantieren. Ist zum Beispiel die Datenleitung stark überlastet oder sogar komplett gestört, kann auch TCP keine erfolgreiche Übertragung mehr garantieren. TCP würde in diesem Fall nach einigen erfolglosen Übertragungsversuchen die Datenübertragung abbrechen. Viele Anwendungen bedienen sich daher eigenen ACKs, so auch Kafka.

ACKs werden von Producern in Kafka verwendet. Consumer verwenden eine andere Methode, um sicherzustellen, dass alle Daten erfolgreich gelesen wurden, auf die wir in einem späteren Kapitel ebenfalls ausführlich eingehen werden. In Kafka werden ACKs nicht nur verwendet, um sicherzustellen, dass der Broker die Daten vom Producer erhalten hat, sondern dienen als Teil der Replikationsstrategie auch einem anderen Zweck. ACKs sind in Kafka komplett unabhängig von der Konfiguration eines Topics und werden im Producer konfiguriert. Allerdings hat die Topic-Konfiguration in Zusammenhang mit ACKs durchaus große Auswirkungen auf die Zuverlässigkeit und auch die Performance.

2.3.1.1 Acknowledgement-Strategien in Kafka

Bevor wir uns genauer mit den Folgen von ACKs in Kafka beschäftigen, werfen wir einen Blick auf die verschiedenen Möglichkeiten, wie ACKs in Kafka überhaupt konfiguriert werden können und welche Wahlmöglichkeiten wir haben. ACKs werden direkt über den Producer gesteuert. Hierbei gibt es drei Optionen, die in Bild 2.9 illustriert sind. Wir behandeln sie in absteigender Reihenfolge mit Blick auf die Zuverlässigkeit des Gesamtsystems. Unser kafka-console-producer nutzt hierzu das Argument `producer-property acks=all`, über das wir zusätzlich Eigenschaften wie ACKs festlegen können.

```
$ kafka-console-producer.sh \
--topic flugdaten \
--bootstrap-server=localhost:9092
--producer-property acks=all
```

Im ersten Fall haben wir unsere Acknowledgement-Strategie auf `all` eingestellt. Das bedeutet, dass der Broker beziehungsweise der Leader der Partition nach dem erfolgreichen Empfang einer Nachricht erst dann eine Bestätigung an den Producer zurückschickt, wenn die Nachricht erfolgreich an alle In-Sync-Replicas (ISR) verteilt wurde. Replicas sind in-sync, wenn sie innerhalb der letzten 30 Sekunden die aktuellen Daten abgeholt haben. Die Einstellung `acks=-1` ist übrigens äquivalent zu `acks=all`.

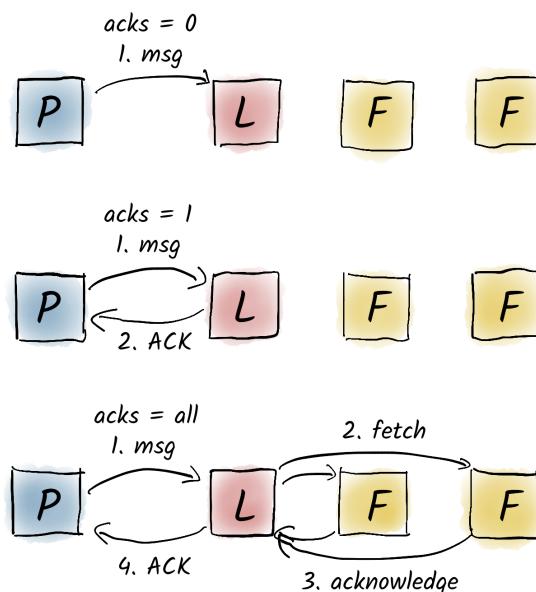


Bild 2.9 Unterschiedliche ACK-Konfigurationen. Mit `acks=0` versendet der Producer Nachrichten und wartet nicht auf Bestätigung, ob sie wirklich angekommen sind. Mit `acks=1` bekommt der Producer eine Rückmeldung, sobald der Leader die Nachricht erhalten hat. Fällt der Leader aus, bevor die Nachricht repliziert wurde, verlieren wir dennoch die Nachricht. Bei `acks=all` (oder `acks=-1`) bekommt der Producer erst eine Rückmeldung, wenn alle In-Sync-Replicas (im Normalfall alle Follower) die Nachricht repliziert haben. In-Sync-Replicas besprechen wir im Verlauf des Buches genauer.

Eine weitere Möglichkeit ist es, `acks=1` zu setzen.

```
$ kafka-console-producer.sh \
  --topic flugdaten \
  --bootstrap-server=localhost:9092
  --producer-property acks=1
```

Diese Einstellung lässt sich sehr gut mit TCP vergleichen. Sobald der Leader die Nachricht empfangen hat, sendet er das ACK an den Producer. Dies ist bis Kafka 3.0 auch die Standardeinstellung von Apache Kafka. Im Vergleich zu `acks=all` erhält der Producer das ACK schneller, was wiederum die Latenz leicht verbessert. Fällt jedoch der Leader unmittelbar nach dem Senden des ACK und vor dem Committen der Nachricht oder der Verteilung der Nachrichten an die Follower aus, muss mit Datenverlust gerechnet werden.

Die letzte Variante ist `acks=0`.

```
$ kafka-console-producer.sh \
  --topic flugdaten \
  --bootstrap-server=localhost:9092
  --producer-property acks=0
```

Hierbei wird keinerlei ACK vom Broker an den Producer gesendet. Diese Konfiguration ist vergleichbar mit UDP. Der Producer sendet seine Daten, aber wenn sie nicht ankommen, ist es ihm egal. Nachrichten werden vom Producer also nur genau einmal gesendet, und Kafka garantiert nicht, dass die Nachricht erfolgreichpersistiert wird. Das bedeutet nicht, dass Nachrichten nicht ankommen, denn auch mit UDP werden je nach Zustand des Netzwerks die meisten Daten erfolgreich übertragen, und außerdem verwendet Kafka in der Transportschicht noch TCP. Dennoch bietet diese ACK-Strategie offensichtlich die geringste Zuverlässigkeit und sollte nur verwendet werden, wenn Datenverlust toleriert werden kann. Im Gegenzug bietet diese Konfiguration eine höhere Performance im Vergleich zu den anderen beiden Strategien. Stellen wir uns ein Szenario vor, in dem wir einen Temperatursensor haben, der jede Sekunde die Temperatur misst und sie an ein Kafka-Cluster sendet. Hier würde es wahrscheinlich nichts ausmachen, wenn wir einige Werte verlieren, da wir uns ohnehin nur für den aktuellen Wert interessieren, und die Wiederholung alter, fehlgeschlagener Übertragungen würde unser Netzwerk und unsere Broker unnötig belasten.

An diesem Punkt scheint es so, dass `acks=0` unter den Acknowledgement-Strategien die beste Leistung ergibt, wenn der Datenverlust tolerierbar ist und `acks=all` die beste Datenkonsistenz ergibt. Warum sollten wir also überhaupt `acks=1` wählen? Um diese Frage zu beantworten, müssen wir einen Blick auf die Topic-Konfiguration werfen, genauer gesagt auf die Eigenschaft `min.insync.replicas`.

2.3.1.2 Acknowledgements und ISR

Die Eigenschaft `min.insync.replicas` entfaltet zusammen mit `acks=all` ihre Wirkung. Wie wir uns erinnern, sendet der Leader bei `acks=all` erst dann ein ACK an den Producer, wenn alle ISR die Daten erfolgreichpersistiert haben. Mit `min.insync.replicas` legen wir fest, dass mindestens eine gewisse Anzahl an Replicas in-sync sein muss, bevor bei `acks=all` ein ACK an den Producer gesendet wird. Sind nicht genug Replicas erreichbar oder in-sync, erhält der Producer eine entsprechende Fehlermeldung und versucht, die Nachricht erneut zu senden. Erstellen wir hierfür testweise ein neues Topic namens `flugdaten-min-insync-replicas-3` und versuchen, testweise Daten auf dem Topic zu schreiben.

```
$ kafka-topics.sh \
  --create \
  --topic "flugdaten-min-insync-replicas-3" \
  --replication-factor 3 \
  --partitions 3 \
  --config min.insync.replicas=3 \
  --bootstrap-server=localhost:9092
```

Mit dem Argument `--config` können wir das Topic genauer konfigurieren, und die minimale Anzahl an ISR legen wir mit dem Parameter `min.insync.replicas=3` auf 3 fest. Schauen wir uns das neu erstellte Topic kurz an.

```
$ kafka-topics.sh \
  --describe \
  --topic "flugdaten-min-insync-replicas-3" \
  --bootstrap-server localhost:9092
Topic: flugdaten-min-insync-replicas-3 PartitionCount: 3      ReplicationFactor: 3
Configs: min.insync.replicas=3
  Topic: flugdaten-min-insync-replicas-3 Partition: 0
    Leader: 1      Replicas: 1,2,3 Isr: 1,2,3
  Topic: flugdaten-min-insync-replicas-3 Partition: 1
    Leader: 2      Replicas: 2,3,1 Isr: 2,1,3
  Topic: flugdaten-min-insync-replicas-3 Partition: 2
    Leader: 3      Replicas: 3,1,2 Isr: 1,2,3
```

Wie erwartet, haben wir drei Partitionen mit jeweils drei Replicas, die jeweils in-sync sind. Solange unsere drei Broker erreichbar sind, werden wir keine Unterschiede bezüglich der ACK-Strategien feststellen, daher schalten wir an dieser Stelle einen unserer Broker ab. Im Anhang befindet sich das `kafka-broker-stop.sh`-Skript, das wir an dieser Stelle verwenden, um unsere Broker mit der ID 3 abzuschalten. Anschließend werfen wir einen kurzen Blick auf unser Topic.

```
$ kafka-broker-stop.sh kafka3
$ kafka-topics.sh \
  --describe \
  --topic "flugdaten-min-insync-replicas-3" \
  --bootstrap-server localhost:9092
Topic: flugdaten-min-insync-replicas-3 PartitionCount: 3      ReplicationFactor: 3
Configs: min.insync.replicas=3
  Topic: flugdaten-min-insync-replicas-3 Partition: 0
    Leader: 1      Replicas: 1,2,3 Isr: 1,2
  Topic: flugdaten-min-insync-replicas-3 Partition: 1
    Leader: 2      Replicas: 2,3,1 Isr: 2,1
  Topic: flugdaten-min-insync-replicas-3 Partition: 2
    Leader: 1      Replicas: 3,1,2 Isr: 1,2
```

Wie wir sehen, hat Broker 1 die Aufgabe des Leaders für Partition 2 von Broker 3 übernommen, und Broker 3 taucht auch nicht mehr in der Liste der ISR auf. Versuchen wir nun mit `acks=0` Nachrichten an das Topic zu senden. Zunächst starten wir aber in einem weiteren Terminal einen `kafka-console-consumer.sh`, um die Nachrichten direkt zu lesen.

```
$ kafka-console-consumer.sh \
  --topic "flugdaten-min-insync-replicas-3" \
  --from-beginning \
  --bootstrap-server=localhost:9092
```

Anschließend starten wir unseren Producer.

```
$ kafka-console-producer.sh \
--topic "flugdaten-min-insync-replicas-3" \
--bootstrap-server=localhost:9092 \
--producer-property acks=0
>Nachricht mit acks=0
```

Die Nachricht sollte nach kurzer Zeit in unserem `kafka-console-consumer.sh` auftaucht sein. Nachrichten kommen also wie versprochen auch bei `acks=0` in der Regel an. Außerdem haben wir keinerlei Probleme mit dem nicht erreichbaren Broker, der offline ist. Ähnlich verhält es sich, wenn wir versuchen, mit `acks=1` Nachrichten zu senden.

```
$ kafka-console-producer.sh \
--topic "flugdaten-min-insync-replicas-3" \
--bootstrap-server=localhost:9092 \
--producer-property acks=1
>Nachricht mit acks=1
```

Auch diesmal wird unsere Nachricht erfolgreich persistiert und kann von unserem Consumer gelesen werden. Schauen wir uns nun an, was passiert, wenn wir versuchen, Nachrichten mit `acks=all` zu produzieren.

```
$ kafka-console-producer.sh \
--topic "flugdaten-min-insync-replicas-3" \
--bootstrap-server=localhost:9092 \
--producer-property acks=all
>Nachricht mit acks=all
>WARN [Producer clientId=console-producer] Got error produce response with
correlation id 5 on topic-partition flugdaten-min-insync-replicas-3-1, retrying
(2 attempts left). Error: NOT_ENOUGH_REPLICAS (org.apache.kafka.clients.producer.
internals.Sender)
...
ERROR when sending message to topic flugdaten-min-insync-replicas-3 with key: null,
value: 22 bytes with error: (org.apache.kafka.clients.producer.internals.
ErrorLoggingCallback)
org.apache.kafka.common.errors.NotEnoughReplicasException: Messages are rejected
since there are fewer in-sync replicas than required.
```

Wir erhalten zunächst Warnungen, dass nicht genügend Replikate vorhanden sind (`NOT_ENOUGH_REPLICAS`) und der Producer versucht, die Nachricht erneut zu senden. Nach einigen fehlgeschlagenen Versuchen erhalten wir die Fehlermeldung, dass die Nachricht abgelehnt wurde, weil es nicht genügend ISR gibt. Auch unser Consumer kann die Nachricht nicht lesen, da sie nicht erfolgreich persistiert werden konnte, obwohl wir eigentlich zwei gesunde Replicas haben. Dieses Beispiel zeigt deutlich, welchen enormen Einfluss die ACK-Strategie in Verbindung mit der Eigenschaft `min.insync.replicas` von Topics auf unser Kafka-Cluster haben kann. Wie genau ein Producer Daten sendet und wie er sich in einem solchen Fall verhält, werden wir uns in einem späteren Kapitel ausführlich ansehen. Wenn wir `min.insync.replicas` nicht oder auf 1 setzen und die Anzahl der ISR 1 beträgt, kann es trotz `acks=all` zu Datenverlusten kommen, da sich `acks=all` in diesem Fall äquivalent zu `acks=1` verhält. Daher ist es wichtig, einen sinnvollen Wert für `min.insync.replicas` zu setzen.

**Tipp**

Ein guter Richtwert für `min.insync.replicas` ist normalerweise der *Replication Factor* minus 1. Wenn der *Replication Factor* zum Beispiel 3 ist, empfiehlt es sich, `min.insync.replicas` auf 2 zu setzen.

2.3.1.3 Nachrichten-Zustellungsgarantien in Kafka

Es gibt jedoch ein Problem, das wir bisher noch gar nicht berücksichtigt haben. Was passiert, wenn eine Nachricht erfolgreich persistiert wurde, aber das ACK nicht beim Producer angekommen ist? In diesem Fall würde der Producer die Nachricht erneut senden, da er davon ausgehen muss, dass die Nachricht nicht erfolgreichpersistiert werden konnte. Dies würde dazu führen, dass wir unwissentlich eine Nachricht doppelt in unserem Cluster speichern. Wenn wir wieder das Beispiel eines Temperatursensors betrachten, wäre das wahrscheinlich kein großes Problem. Aber stellen wir uns nun vor, dass es sich bei der Nachricht um eine Banktransaktion handelt. Dies würde im Zweifelsfall dazu führen, dass Beträge doppelt gutgeschrieben oder belastet werden, was natürlich kein erwünschtes Verhalten ist.

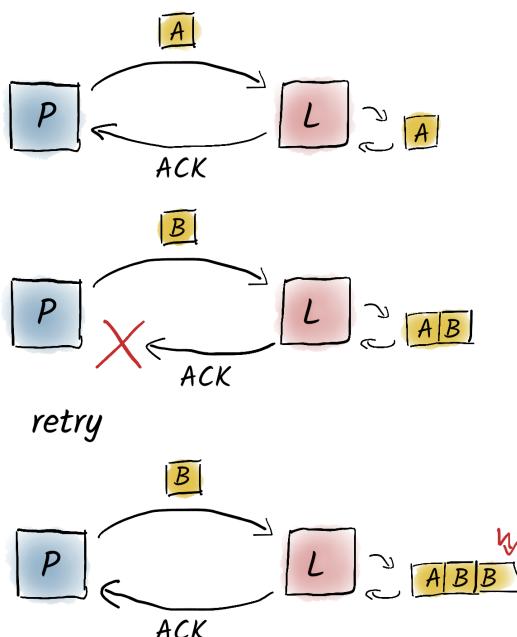


Bild 2.10 Probleme entstehen, wenn der Producer ACKs erwartet und diese auf dem Weg vom Leader zum Producer verloren gehen, obwohl die Daten erfolgreich geschrieben wurden. Der Producer schickt die Nachricht in diesem Fall erneut, und die Nachricht steht doppelt im Log, womöglich sogar in einer falschen Reihenfolge (nicht abgebildet). Um das zu verhindern, müssen wir `acks=all` und `enable.idempotence=true` im Producer setzen. Ab Kafka 3.0 ist dies standardmäßig bereits der Fall.

An dieser Stelle ist es sinnvoll, dass wir uns zunächst die verschiedenen Nachrichtenzustellungsgarantien von Kafka ansehen. Grundsätzlich gibt es in verteilten Systemen drei Zustellungsgarantien: *at most once*, *at least once*, und *exactly once*.

At most once garantiert, dass eine Nachricht höchstens einmal zugestellt wird und es somit keine Duplikate geben kann. Sie garantiert aber nicht, dass die Nachricht tatsächlich ankommt. Dies ist bei `acks=0` der Fall.

Die *at least once*-Garantie wird erreicht, indem `min.insync.replicas` auf einen sinnvollen Wert und `acks` auf `all` gesetzt wird. In diesem Fall wird garantiert, dass eine Nachricht in jedem Fall ankommt. Allerdings können Duplikate auftreten, was zu dem gerade beschriebenen Problem führen kann.

Dies führt uns zu *exactly once*. Hierbei wird garantiert, dass eine Nachricht genau einmal persistiert wird und somit die mathematische Eigenschaft der *Idempotenz* hergestellt ist. Idempotenz bedeutet, dass, wenn wir eine Funktion mehrmals mit den gleichen Eingaben aufrufen, wir immer das gleiche Ergebnis bekommen. Kafka realisiert dies, indem der Producer einer Nachricht eine Sequenz-ID zuweist. Standardmäßig war diese Eigenschaft bis vor Kafka 3.0 deaktiviert, und Nachrichten wurden ohne Sequenz-IDs versendet.

```
$ kafka-console-producer.sh \
  --topic "flugdaten-min-insync-replicas-3" \
  --bootstrap-server=localhost:9092 \
  --producer-property acks=all \
  --producer-property enable.idempotence=true
```

Äquivalent zu Acknowledgements können wir mit dem Argument `--producer-property enable.idempotence=true`.

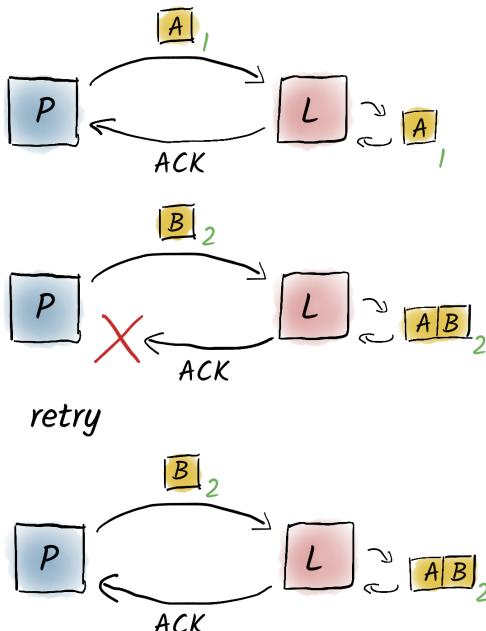


Bild 2.11 Ist Idempotenz aktiviert, verschickt der Producer mit jeder Nachricht seine ID und zusätzlich eine Sequenz-ID zu jeder Nachricht. So kann der Broker erkennen, falls Nachrichten fehlen, in der falschen Reihenfolge auftauchen oder mehrmals verschickt wurden, und kann entsprechend reagieren. In diesem Beispiel ist das ACK des Leaders beim Producer nicht angekommen, und deshalb hat der Producer die Nachricht erneut gesendet. Da der Leader diese Sequenz-ID für den Producer bereits kennt, kann der Leader mit einem ACK antworten, aber schreibt die Nachricht nicht in den Log.

Wenn Idempotenz aktiviert ist, prüft der Leader einer Partition beim Empfang von Nachrichten, ob sie in der richtigen Reihenfolge ankommen. Wurde die Nachricht bereitspersistiert, wird lediglich das ACK erneut versendet. Wenn die Nachrichten in der falschen Reihenfolge ankommen, sendet der Leader einen NACK an den Producer und dieser versucht

dann noch einmal, die Nachricht zu senden. Der Grund, warum Idempotenz bis Kafka 3.0 standardmäßig deaktiviert war, lag darin, der Community Zeit zu geben, ihre Clients zu aktualisieren. Natürlich ist mit Idempotenz auch ein minimaler Performanceverlust verbunden. Dieser Performanceverlust ist jedoch vernachlässigbar.



Tipp

Wir empfehlen an dieser Stelle ausdrücklich, Idempotenz zu aktivieren.

2.3.2 Replikation

Nachdem wir uns im letzten Abschnitt ausführlich mit ACKs beschäftigt und dabei gelernt haben, wie Zustellgarantien in Kafka durch ACKs beeinflusst werden, widmen wir uns nun der Replikation in Kafka.

Wir setzen Replikation in vielen Bereichen der IT in verschiedenen Ausprägungen ein, um Datenhaltbarkeit und auch die Erreichbarkeit zu verbessern, was insgesamt zu einer gesteigerten Zuverlässigkeit der jeweiligen Systeme führt. Bevor wir nun damit beginnen, uns das Konzept hinter der Replikation in Kafka im Detail anzusehen, betrachten wir zunächst allgemein das Konzept der Replikation etwas genauer, um klar abzugrenzen, was Replikation ist und was nicht.

2.3.2.1 Replikation vs. Backup

Im Kern bezeichnet Replikation die Vervielfältigung von Daten in Kombination mit einem regelmäßigen Datenabgleich. Dies führt in der Praxis schnell dazu, dass die Begriffe Replikation und Backup fälschlicherweise oft synonym verwendet werden. Es gibt aber durchaus erhebliche Unterschiede zwischen beiden Konzepten. Unter Replikation verstehen wir in der Regel die kontinuierliche Synchronisation von Daten zwischen einem Original und mehreren Kopien. Backups dagegen werden normalerweise zu fest definierten Zeitpunkten, oft einmal täglich in der Nacht, automatisch erstellt und basieren in der Regel auf Snapshots des gesamten Systems.

Fällt das System aus, sodass ein Backup eingespielt werden muss, ist auch mit einer nicht unerheblichen Ausfallzeit zu rechnen, da das ganze System neu eingespielt werden muss, was je nach System einiges an Zeit in Anspruch nehmen kann. Außerdem ist, je nachdem wie lange das letzte Backup zurückliegt, mit einem mehr oder weniger großen Datenverlust zu rechnen. Hier zeigt sich auch der wesentliche Unterschied zwischen Replikation und Backups. Bei einem Ausfall ist ein Replica schnell in der Lage, die Aufgaben des ausgefallenen Systems zu übernehmen, wodurch es nur zu sehr geringen bis keinen Ausfallzeiten kommt. Außerdem gibt es je nach Replikationsstrategie keinen oder nur einen geringen Datenverlust. Der Nachteil von Replikation gegenüber Backups ist, dass Replikation nicht vor Totalausfällen, Bugs oder menschlichen Fehlern schützt. Kafka selbst hat übrigens kein eingebautes Backup, sondern verlässt sich komplett auf Replikation. Im nächsten Abschnitt werden wir lernen, wie genau Replikation in Kafka umgesetzt ist und warum Kafka damit auch ohne Backups eine hohe Zuverlässigkeit erreicht.

2.3.2.2 Leader-Follower-Prinzip

In den vergangenen Kapiteln haben wir bereits einiges über Replikation und Partitionierung in Kafka gelernt. Wir wissen bereits, dass Kafka für das Management seiner Replicas das sogenannte *Leader-Follower-Prinzip* verwendet. In diesem Abschnitt werden wir tiefer hinter die Kulissen des Leader-Follower-Prinzips in Kafka schauen. Wir beschäftigen uns damit, wie ein Broker zu einem Partitions-Leader wird und welche Aufgaben damit einhergehen. Außerdem sehen wir uns an, wie Follower und Leader miteinander interagieren und was passiert, falls ein Leader ausfällt.

Wir wissen bereits, dass Nachrichten in Kafka in Topics organisiert sind, dass Topics wiederum in Partitionen unterteilt werden können und dass diese Partitionen repliziert werden können. Bei der Erstellung eines neuen Topics geben wir die Anzahl an Partitionen und den *Replication Factor* an. Kafka erstellt dann entsprechende Partitionen und Replicas für das Topic. Aber was heißt das eigentlich? Sämtliche Anfragen, egal ob es um das Erstellen oder Anzeigen von Topics oder das Produzieren in Topics oder das Konsumieren von Topics geht, können an einen beliebigen Broker gesendet werden. In unseren Beispielen verwenden wir hierzu bisher immer unseren Broker mit der ID 1, welcher auf dem Port 9092 lauscht. Wir könnten die Anfragen aber genauso an einen anderen Broker senden. Die eigentliche Erstellung von Topics wird vom Controller übernommen. Die Rolle des Controllers besprechen wir ausführlicher im Kapitel 3.5 „Cluster-Management“.

Betrachten wir das Ganze nun anhand eines Beispiels. Zunächst erstellen wir das Topic *replication-test*:

```
$ kafka-topics.sh --create \
  --topic replication-test \
  --replication-factor 3 \
  --partitions 3 \
  --config min.insync.replicas=2 \
  --bootstrap-server=localhost:9092
Created topic replication-test.
```

Anschließend schauen wir uns an, wie die Partitionen auf die einzelnen Broker verteilt worden sind:

```
$ kafka-topics.sh --describe \
  --topic replication-test \
  --bootstrap-server=localhost:9092 \
Topic: replication-test PartitionCount: 3
  ReplicationFactor: 3  Configs: min.insync.replicas=2
Topic: replication-test Partition: 0
  Leader: 2  Replicas: 2,3,1 Isr: 2,3,1
Topic: replication-test Partition: 1
  Leader: 3  Replicas: 3,1,2 Isr: 3,1,2
Topic: replication-test Partition: 2
  Leader: 1  Replicas: 1,2,3 Isr: 1,2,3
```

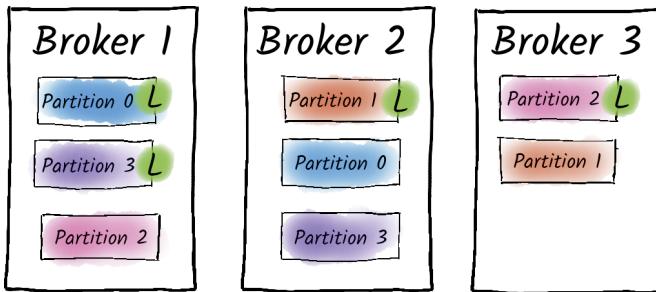


Bild 2.12 Die Partitionen unseres Topics sind auf unsere drei Broker verteilt. Dabei ist Broker 2 der Leader für Partition 0, Broker 3 Leader für Partition 1 und Broker 1 Leader für Partition 2. Da wir einen Replication Factor von 3 und auch drei Broker haben, findet sich jede Partition auf jedem Broker.

Da wir einen *Replication Factor* von 3 haben und insgesamt auch nur drei Broker, wird jede unserer drei Partitionen auf jeden unserer drei Broker repliziert. Die Partitions-Leader werden, um die Last ausgeglichen zu verteilen, gleichmäßig auf die Broker verteilt. Broker 1 ist Leader von Partition 2, Broker 2 ist Leader von Partition 0, und Broker 3 ist Leader von Partition 1.

Im letzten Abschnitt über ACKs haben wir bereits gelernt, dass ein Producer Nachrichten immer an den Leader einer Partition sendet und dass der Leader, je nach ACK-Strategie, erst wartet, bis auch die Follower die Nachricht erhalten haben, bevor er den Erhalt der Nachricht gegenüber dem Producer bestätigt. Wie aber gelangen die Nachrichten vom Leader zu den Followern? Die Kafka-Broker lagern so viel Arbeit wie möglich an die Clients aus. Aus Sicht von Kafka (beziehungsweise eines Partitions-Leaders) sind Follower nichts anderes als einfache Kafka-Clients, genauer gesagt Consumer. Consumer erhalten Nachrichten, indem sie mittels eines speziellen *fetch-Requests*-Topics konsumieren.

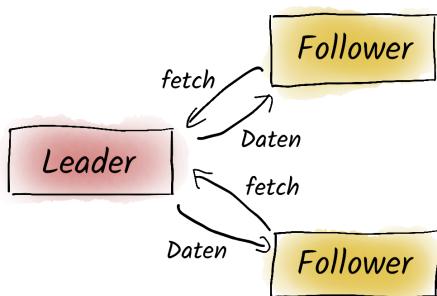


Bild 2.13 Follower fetschen kontinuierlich Daten vom Leader, um auf dem aktuellen Stand zu bleiben und die Rolle des Leaders übernehmen zu können, falls dieser ausfällt.

Wie normale Consumer teilen auch die Follower dem Leader ihren aktuellen Offset, das heißt die aktuelle Leseposition im Log mit. Dadurch weiß der Leader, wann ein Follower eine Nachricht konsumiert hat, und kann dann entsprechend die Bestätigung an den Producer senden.

Spannend wird es, wenn wir uns anschauen, was im Falle eines Ausfalls eines Brokers passiert, und der Broker nicht mehr erreichbar ist. Hierfür stoppen wir zunächst unseren

Broker 3 mit unserem `kafka-broker-stop.sh`-Skript aus dem Anhang und zeigen uns anschließend wieder das Topic an:

```
$ kafka-broker-stop.sh kafka3
$ kafka-topics.sh --describe \
  --topic replication-test \
  --bootstrap-server=localhost:9092 \
Topic: replication-test PartitionCount: 3
  ReplicationFactor: 3    Configs: min.insync.replicas=2
Topic: replication-test Partition: 0
  Leader: 2    Replicas: 2,3,1 Isr: 2,1
Topic: replication-test Partition: 1
  Leader: 1    Replicas: 3,1,2 Isr: 1,2
Topic: replication-test Partition: 2
  Leader: 1    Replicas: 1,2,3 Isr: 1,2
```

Wir sehen, dass Broker 3 zwar immer noch in der Liste der Replicas ist, aber nicht mehr in der Liste der ISR auftaucht. Wenn ein Broker ausfällt, der Leader von Partitionen ist, muss Kafka (beziehungsweise der Controller) einen neuen Leader für die entsprechenden Partitionen bestimmen, da das Topic ansonsten nicht mehr erreichbar wäre. Deshalb hat Broker 1 die Leader-Rolle für Partition 1 übernommen. Weiterhin kommt auch nicht jedes Replica als neuer Leader infrage, sondern lediglich eine Replica aus der Liste der ISR. Warum dies so ist und wie genau bestimmt wird, ob ein Replica in-sync ist, werden wir uns im Kapitel 3.2 „Nachrichten produzieren und persistieren“ noch ausführlich anschauen.

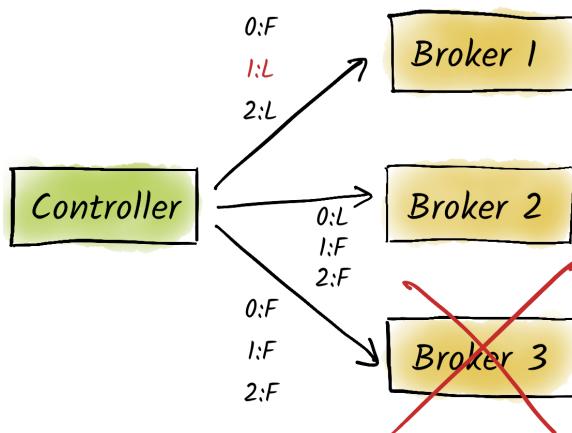


Bild 2.14 Der Controller hat den Ausfall des Broker 3 festgestellt. Da nun der Leader für die Partition 1 ausgefallen ist, sucht sich der Controller einen anderen Broker, der für diese Partition in-sync ist, aus, der Leader werden soll. In unserem Fall ist dies Broker 1. Um dies dem Broker zuzuordnen, sendet der Controller keine direkte Nachricht an den Broker, sondern speichert das im Metadaten-Topic (oder Zookeeper vor Kafka 3.0), aus dem die anderen Broker kontinuierlich Daten holen.

Wir können uns in Kafka mittels `kafka-topics.sh` alle Topics beziehungsweise Partitionen anzeigen, welche aktuell nicht die gewünschte Zahl an Replicas haben. Hierfür übergeben wir die Argumente `describe` und `under-replicated-partitions`:

```
$ kafka-topics.sh --describe \
--bootstrap-server=localhost:9092 \
--under-replicated-partitions
Topic: replication-test Partition: 0
Leader: 2    Replicas: 2,3,1 Isr: 2,1
Topic: replication-test Partition: 1
Leader: 1    Replicas: 3,1,2 Isr: 1,2
Topic: replication-test Partition: 2
Leader: 1    Replicas: 1,2,3 Isr: 1,2
```

Wenig überraschend sehen wir, dass alle Partitionen unseres Topics *replication-test* unter-repliziert sind. Äquivalent dazu können wir uns via *--under-min-isr-partitions* auch alle Partitionen anzeigen lassen, die aktuell zu wenig ISR haben, also weniger ISR als durch *min.insync.replicas* gefordert sind. In diesem Fall bekämen wir allerdings kein Ergebnis (beziehungsweise bekämen keine Partitionen angezeigt), da wir immer noch zwei ISR haben, was unserer *min.insync.replicas*-Anforderung entspricht. Beide Befehle können für das Debugging unseres Kafka-Clusters extrem hilfreich sein, da wir so topic-übergreifend leicht feststellen können, ob etwas nicht stimmt.

Nachdem wir gesehen haben, was passiert, wenn ein Broker ausfällt, werfen wir nun einen Blick darauf, was geschieht, wenn der Broker wieder erreichbar ist. Um den Broker wieder zu starten, können wir einfach *kafka-server-start.sh* erneut ausführen. Anschließend betrachten wir wieder den aktuellen Status des Topics:

```
$ kafka-server-start.sh ~/kafka/config/kafka3.properties
$ kafka-topics.sh --describe \
--topic replication-test \
--bootstrap-server=localhost:9092
Topic: replication-test PartitionCount: 3
ReplicationFactor: 3    Configs: min.insync.replicas=2
Topic: replication-test Partition: 0
Leader: 2    Replicas: 2,3,1 Isr: 2,1,3
Topic: replication-test Partition: 1
Leader: 1    Replicas: 3,1,2 Isr: 1,2,3
Topic: replication-test Partition: 2
Leader: 1    Replicas: 1,2,3 Isr: 1,2,3
```

Broker 3 taucht wieder in der Liste der ISR auf, allerdings bleibt Broker 1 weiterhin Leader von Partition 1, und Broker 3 muss sich mit der Rolle des Followers begnügen. Je nachdem, wie wir unser Cluster konfiguriert haben, findet entweder nach einiger Zeit automatisch ein Leader Rebalancing statt, oder wir müssen das Rebalancing der Leader manuell durchführen. Mit dem Skript *kafka-leader-election.sh* können wir dieses Rebalancing der Leader manuell auslösen:

```
$ kafka-leader-election.sh \
--election-type=preferred \
--all-topic-partitions \
--bootstrap-server=localhost:9092
Successfully completed preferred leader election for partitions replication-test-1
```

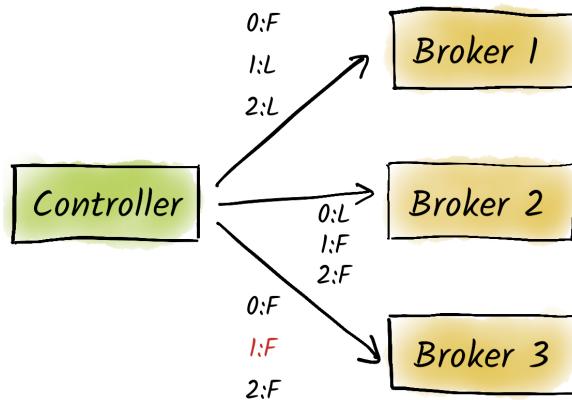


Bild 2.15 Obwohl Broker 3 wieder erreichbar ist, wird dieser nicht sofort Leader für die Partition 1. Broker 3 muss zuerst selbst in-sync werden und wird standardmäßig nach einigen Minuten wieder automatisch Leader für die Partition, für die dieser ein *preferred Leader* ist.

Besonders vorsichtig müssen wir mit dem Argument `election-type` sein. Hier können wir zwischen *preferred* und *unclean* wählen. Erstes bedeutet, dass wir versuchen, den ursprünglichen und damit bevorzugten Leader einer Partition wieder als Leader einzusetzen. Den bevorzugten Leader erkennen wir daran, dass er in der Liste der Replicas an erster Stelle steht. Eine *unclean-Leader-Election* sollten wir nur im äußersten Notfall vornehmen. Was genau ist jetzt aber eine solche Notfallsituation? Wie bereits erwähnt, bestimmt Kafka beim Ausfall eines Partitions-Leaders den neuen Leader aus der Liste der ISR. Was aber passiert, wenn es kein weiteres ISR gibt? In diesem Fall könnte Kafka standardmäßig keinen neuen Leader bestimmen, wodurch keine neuen Nachrichten auf die Partition geschrieben und auch keine Nachrichten aus der Partition konsumiert werden können. Mit der *unclean-Leader-Election* können auch Replicas, die nicht in-sync sind, zum neuen Leader ernannt werden. Da diese Replicas höchstwahrscheinlich nicht alle Nachrichten haben, ist mit Datenverlust zu rechnen, daher wird diese Methode auch als *unclean* bezeichnet. Wir können die *unclean-Leader-Election* auch im Cluster selbst in der Broker Config konfigurieren (`unclean.leader.election.enable`). Davon raten wir allerdings stark ab. Des Weiteren können wir den `kafka-leader-election`-Befehl entweder nur auf ein bestimmtes Topic (`--topic replication-test`) oder aber auf alle Topics anwenden (`--all-topic-partitions`). Insbesondere im Fall einer *unclean-Leader-Election* sollte unbedingt ein bestimmtes Topic ausgewählt werden!



Tipp

Eine *unclean-Leader-Election* sollten wir nur im äußersten Notfall vornehmen.

Schauen wir uns zum Schluss das Topic noch mal an:

```
$ kafka-topics.sh \
--describe \
--topic replication-test \
--bootstrap-server=localhost:9092
Topic: replication-test PartitionCount: 3
    ReplicationFactor: 3    Configs: min.insync.replicas=2
Topic: replication-test Partition: 0
    Leader: 2    Replicas: 2,3,1 Isr: 2,1,3
Topic: replication-test Partition: 1
    Leader: 3    Replicas: 3,1,2 Isr: 1,2,3
Topic: replication-test Partition: 2
    Leader: 1    Replicas: 1,2,3 Isr: 1,2,3
```

Wie wir sehen, ist Broker 3 wieder Leader von Partition 1, und Broker 1 wurde zum normalen Follower ernannt.

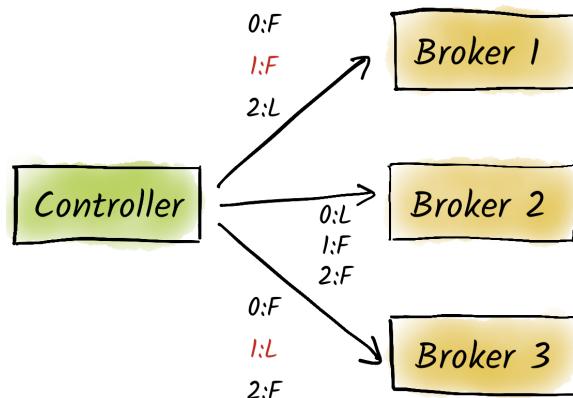
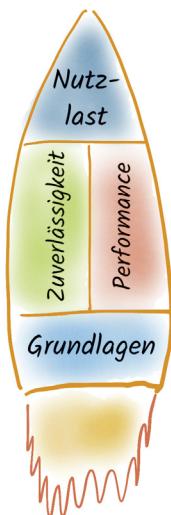


Bild 2.16 Nachdem die preferred Leader-Election (manuell oder automatisch) durchgeführt wurde, wird Broker 3 wieder Leader für Partition 1, und Broker 1 wird Follower für diese Partition.

2.3.3 Zusammenfassung

In diesem Kapitel haben wir uns damit beschäftigt, wie Kafka mittels ACKs und Replikation seine Zuverlässigkeit erreicht. Wir haben uns die verschiedenen ACK-Strategien sowie die Schwächen und Stärken der einzelnen Strategien angesehen. In diesem Zusammenhang haben wir untersucht, welche Zustellgarantien Kafka bietet und wie Nachrichtenduplicate verhindert werden können. Außerdem haben wir uns die Bedeutung von ISR beim Produzieren, insbesondere im Zusammenspiel mit ACKs, und beim Konsumieren von Daten anschaut. Wir haben die Vor- und Nachteile von Backups und Replikation verglichen und gelernt, wie Kafka seine Daten repliziert. Wir haben gelernt, dass Kafka seine Partitionen und Replicas nach dem Leader-Follower-Prinzip organisiert und wie dieses Konzept in Kafka implementiert ist.

■ 2.4 Performance



Im letzten Kapitel haben wir gesehen, wie wir mit Kafka zuverlässig Nachrichten produzieren können. In diesem Kapitel gehen wir auf den noch offenen Teil der Kafka-Rakete ein. Wie erreichen wir mit Kafka die versprochene Performance?

Ähnlich wie bei der Zuverlässigkeit müssen wir, wenn wir über Performance sprechen, erst klären, was dies für uns bedeutet. Wir können Performance unterschiedlich definieren. Geht es uns um die Bandbreite? Die Bandbreite, die uns das System anbietet, ist oft sehr entscheidend. Wir messen die Bandbreite in Bytes pro Sekunde. Wie viele KB/MB/GB pro Sekunde Datendurchsatz erreichen wir mit unserem System? Wenn wir nur über Bandbreite sprechen, vernachlässigen wir oft die Latenz, obwohl sie für uns meist von größerer Bedeutung ist. Wir sollten niemals die Bandbreite eines Lkw vollgeladen mit microSD-Karten unterschätzen:

Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway.

– Andrew Tanenbaum, 1981

Die Bandbreite eines solchen Lkw ist enorm. (XKCD spricht von 160 Terrabyte pro Kilogramm!¹⁷⁾ Mit 1-TB-microSD-Karten kommen wir sogar auf 220 Terrabyte pro Kilogramm. Ein vollbeladener Lkw kommt so auf 480 000 bis 660 000 Terrabyte.) Aber meist ist es nicht das, was wir wollen. Am Ende interessiert uns die Ende-zu-Ende-Latenz eines Systems viel mehr als die theoretische maximale Bandbreite. Insbesondere heutzutage erwarten unsere Nutzer von uns sofortige Antworten. Amazon geht davon aus, dass pro 100 ms zusätzlicher Ladezeiten bis zu 1 % vom Umsatz verloren gehen.¹⁸⁾

Bandbreite und Latenzen sind zwar enorm wichtig für IT-Systeme, aber es gibt noch einen Aspekt, den wir nicht vernachlässigen sollten: die Ressourcenfreundlichkeit eines Systems. Normalerweise stehen uns nicht unbegrenzte Ressourcen zur Verfügung. Wir wollen so genügsam wie möglich mit unseren Ressourcen umgehen; einerseits, um unseren Finanzbedarf überschaubar zu halten, andererseits natürlich auch, um den ökologischen Fußabdruck unserer IT-Systeme so gering wie möglich zu halten.

Insbesondere in einem so umfangreich konfigurierbaren System wie Kafka ist es wichtig zu verstehen, wie wir unsere Systeme optimieren. In unseren Kafka-Schulungen haben die Teilnehmer viel Spaß daran, Kafkas Grenzen auszutesten. Die Teilnehmer optimieren ihre Systeme gerne von 0.2 MB/s Datendurchsatz bis hin zu über 200 MB/s auf einer einzigen moderat großen Maschine.

¹⁷⁾ <https://what-if.xkcd.com/31/>

¹⁸⁾ <http://radar.oreilly.com/2008/08/radar-theme-web-ops.html>

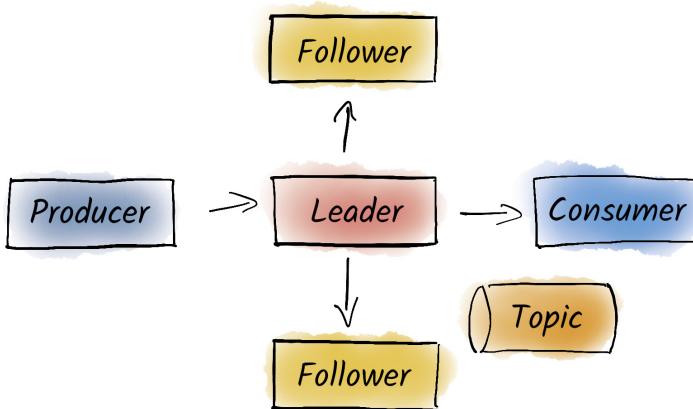


Bild 2.17 Wir können die Performance in Kafka an vielen Stellen konfigurieren: bei der Topic-Konfiguration, den Brokern und allen Clients.

Wo konfigurieren wir nun aber in Kafka überall die Performance? Die kurze Antwort darauf ist: Kafka ist von sich aus an vielen Stellen auf Performance getrimmt, aber wir müssen dennoch in allen Komponenten auf die Konfigurationseinstellungen achten.

Kafkas Performanceoptimierung fängt bei den Grundannahmen an, die es trifft. Kafka ist ein Kind des 21.Jahrhunderts. Wir gehen davon aus, dass Festplatten relativ billig sind. Wir können auch ohne großen Aufwand größere Mengen von Hauptspeicher installieren. Aber viel wichtiger ist eine andere Annahme: Wir wissen, dass einzelne IT-Systeme nicht besonders zuverlässig sind. Anstatt zu versuchen, um jeden Preis Hardwareausfälle zu vermeiden, verfolgt Kafka einen anderen Ansatz. Es ist okay, wenn Subsysteme ausfallen, Kafka kann dennoch weiter funktionieren.

Kafka versucht auch den Ansatz *one-size-fits-all* zu vermeiden, und wir können in einem einzelnen Kafka-Cluster auch unterschiedliche Garantien bezüglich Performance und Zuverlässigkeit treffen.

Architekturell ist Kafka von Grund auf für Performance getrimmt. Es fängt beim Log als zentrale Datenstruktur an. Ein Log ist, wie wir in den ersten Kapiteln gelernt haben, eine der einfachsten Datenstrukturen. Die Zugriffsmuster von Logs sind sehr vorhersehbar, was dazu führt, dass Kafka selbst auf klassischen, rotierenden Festplatten sehr performant läuft. Denn obwohl SSDs klassische Festplatten in fast allen Kategorien schlagen, ist der Preis Letzterer immer noch unschlagbar.

Wir erinnern uns, dass Kafka in der Lage ist, beliebige Datenformate zu transportieren. Dies gibt uns nicht nur größere Flexibilität, sondern zwingt Kafka auch, auf übliche Funktionen anderer Messaging-Systeme zu verzichten. Dadurch, dass Kafka gar nicht erst versucht, Nachrichten zu interpretieren, gewinnen wir weitere Performance. Der gesamte Ansatz Kafkas, dass das Messaging-System möglichst wenige Aufgaben erfüllt und die eigentliche Arbeit an die Clients ausgelagert ist, dient dazu, die Performance des Gesamtsystems weiter zu verbessern.

2.4.1 Topics

Bevor wir anfangen, an den Konfigurationseinstellungen unserer Broker und Clients zu feilen, müssen wir sicherstellen, dass unsere Topics korrekt konfiguriert sind. Wir haben bereits in den ersten Kapiteln gelernt, dass wir überhaupt nur zwecks horizontaler Skalierung unsere Topics partitionieren. Damit können wir sowohl mehr Daten speichern, als auf einen einzelnen Server passen, als auch unser System parallelisieren. Partitionieren bringt allerdings auch einige Feinheiten, die wir vorher bedenken müssen, mit sich. Ein wichtiger Punkt, den wir hierbei zum Beispiel nicht außer Acht lassen dürfen, ist, dass Kafka die korrekte Reihenfolge von Nachrichten nur innerhalb einer Partition garantiert. Dies bedeutet im Umkehrschluss, dass wir bei mehreren Partitionen innerhalb eines Topics im Nachhinein nicht mehr exakt nachvollziehen können, ob eine Nachricht in Partition 0 vor oder nach einer Nachricht in Partition 1 geschrieben worden ist. Wenn uns die Reihenfolge zwischen den Nachrichten nicht so wichtig ist, müssen wir hier auf nichts achten. Sollte uns die Reihenfolge zwischen den Nachrichten allerdings wichtig sein, müssen wir uns überlegen, wie wir garantieren, dass Nachrichten, die zusammengehören, auf der gleichen Partition landen. Glücklicherweise lässt uns auch hier Kafka viel Flexibilität, wie wir dies anstellen möchten.

2.4.1.1 Partitionierung und Keys

In Kafka entscheiden die Producer, in welche Partition eine Nachricht produziert wird. Wir können garantieren, dass zwei Nachrichten in die gleiche Partition produziert werden, indem wir beiden Nachrichten den gleichen Key zuordnen. Keys sind optional, aber sind genau wie Values einfache Byte-Arrays. Wenn wir Keys benutzen, entscheidet die Kafka-Library im Producer anhand des Hash-Wertes des Keys, in welche Partition produziert wird:

```
partition = hash(key) % number_of_partitions;
```

Dazu errechnet die Kafka Library zuerst den Hash-Wert des Keys und rechnet ihn dann als Modulo der Anzahl an Partitionen. Sollten wir die Anzahl der Partitionen eines Topics zwischendurch ändern, ist die Reihenfolgen-Garantie von Kafka nicht mehr gegeben.

Sollte die Reihenfolge der Nachrichten für uns nicht von Belang sein, können wir den Key weglassen. In diesem Fall produzieren die Producer mithilfe des Round-Robin-Verfahrens die Nachrichten in die Partitionen. Das heißt, die erste Nachricht wird an Partition 0 geschickt, die zweite Nachricht an Partition 1 und so weiter. Um die Anzahl der Netzwerk-anfragen zu reduzieren und damit den Datendurchsatz zu erhöhen, benutzt Kafka seit Version 2.4 ein besser optimiertes Round-Robin-Verfahren. Statt nach jeder Nachricht eine andere Partition auszuwählen, akkumuliert Kafka die Nachrichten in die sowieso vorhandenen Batches und wählt erst dann eine andere Partition, nachdem ein Batch abgeschickt worden ist. Somit minimieren wir die Anzahl der Netzwerk-Requests und können gleichzeitig auch unsere Batches besser ausfüllen. Dadurch profitieren wir von besserem Batching und, wenn aktiviert, besserer Komprimierung.

Unsere Producer verteilen nun also unsere Nachrichten auf die unterschiedlichen Partitionen. Im Idealfall sind unsere Partitions-Leader gleichmäßig auf alle Broker verteilt, und somit verteilt sich auch die Last halbwegs gleichmäßig auf diese Broker.

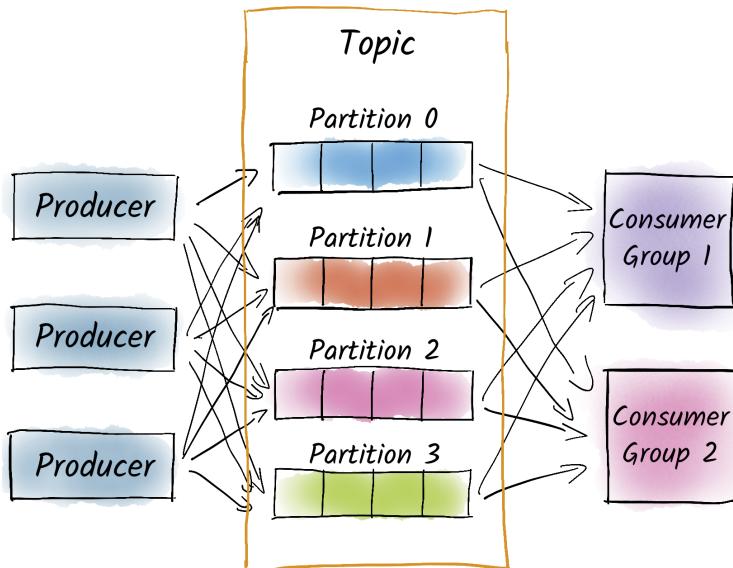


Bild 2.18 In Kafka partitionieren wir Topics für bessere Lastverteilung und Parallelisierung. Wir teilen die Partitionen auf unterschiedliche Broker auf, um die Last besser zu verteilen. Producer entscheiden anhand des Keys oder des Round-Robin-Verfahrens, auf welche Partition die Nachricht produziert werden soll. Consumerseitig nutzen wir Consumer Groups, um die Partitionen parallelisiert verarbeiten zu können. Dabei sind unterschiedliche Consumer Groups voneinander isoliert.

2.4.1.2 Skalierung und Lastverteilung

Die Anzahl der Partitionen hat nicht nur Auswirkungen darauf, wie unsere Producer produzieren und wie sich die Last über die Broker verteilt, sondern vor allem, wie wir unsere Consumer skalieren. Grundsätzlich können viele unabhängige Consumer von ein und demselben Topic Nachrichten konsumieren. Zum Beispiel können wir uns vorstellen, dass es einen Consumer in unserem Kontrollzentrum gibt, welcher das *flugdaten*-Topic konsumiert. Zusätzlich dazu haben wir einen weiteren Consumer, der für unsere TV-Liveübertragungen dasselbe Topic konsumiert, um unseren Zuschauern die aktuellen Daten anzugeben. Auf der einen Seite möchten wir nicht, dass sich diese unabhängigen Consumer in die Quere kommen. Das können wir ohne Weiteres sicherstellen, da die Consumer Kafka explizit nach allen neuen Nachrichten ab einem bestimmten Offset anfragen. Wenn jeder unabhängige Consumer die Offsets für sich selbst verwaltet, kommen sich diese auch nicht in die Quere. Nun haben wir die Situation, dass wir zwar viele Partitionen haben, aber nur einen Consumer, der alle Nachrichten konsumiert. Um unsere Consumer zu skalieren und auch die Offsets zu verwalten, benutzen wir in Kafka sogenannte *Consumer Groups*, die wir schon kennengelernt haben. Wir starten hierzu mehrere Instanzen desselben Consumers, und Kafka unterstützt diese Consumer dabei, die Arbeit untereinander aufzuteilen. Allerdings können wir die Nachrichten nicht beliebig auf Consumer einer Consumer Group verteilen, sondern wir verteilen jeweils ganze Partitionen an Consumer, um die Reihenfolge unserer Nachrichten weiterhin garantieren zu können.

Partitionen helfen uns also nicht nur, die Last auf unterschiedliche Broker zu verteilen, sondern auch, unsere Consumer Groups zu skalieren, um Daten parallelisiert verarbeiten zu können. Diese Partitionierung bringt die Limitierung mit sich, ist, dass wir uns mehr Gedanken über die Reihenfolge von Nachrichten machen müssen, da wir diese nur innerhalb einer Partition sicherstellen können. Und nicht nur das! Zusätzlich müssen wir auch sicherstellen, dass die Nachrichten vom selben Producer produziert wurden, da es bei mehreren Producern ebenfalls keinerlei Garantien bezüglich der Reihenfolge von Nachrichten seitens Apache Kafka gibt. Außerdem können wir die Reihenfolge nur garantieren, wenn wir, wie im Kapitel 2.3 „Zuverlässigkeit“ besprochen, `enable.idempotence=true` setzen. Sonst kann es passieren, dass Kafka beim Empfangen und Verarbeiten der Nachrichten durcheinanderkommt.

Im Zusammenhang mit Partitionen ist es auch immer wichtig, darüber nachzudenken, wie viele Partitionen für den jeweiligen Anwendungsfall richtig sind. Grundlegend könnten wir auf die Idee kommen, dass es besser ist, viele Partitionen zu haben, weil wir dann besser skalieren und somit auch unsere Arbeit besser parallelisieren können. Aber die Frage nach der richtigen Anzahl der Partitionen ist leider keine einfache, und es gibt meistens keine perfekte Antwort darauf.

2.4.1.3 Wie viele Partitionen sollten wir haben?

Prinzipiell führen mehr Partitionen erst einmal zu einem höheren Datendurchsatz. Um zu bestimmen, wie viele Partitionen wir in einem Topic benötigen, müssen wir wissen, mit welchem Datendurchsatz wir zu rechnen haben. Oft fangen hier schon die Schwierigkeiten an, da wir oft nicht wissen, was unsere Anforderungen sind. Dann müssen wir schätzen und ausprobieren. Wenn wir die Anforderungen kennen, müssen wir nun herausfinden, was unser Kafka-Cluster an Datendurchsatz leisten kann. Sobald wir ein neues Kafka-Cluster aufsetzen, sollten wir stets als Erstes Performance-Tests durchführen, um festzustellen, wie viele Nachrichten die Broker im sauberen Zustand empfangen und versenden können. Dazu nutzen wir üblicherweise die Werkzeuge `kafka-producer-perf-test.sh` und `kafka-consumer-perf-test.sh`. Mit diesen Werkzeugen können wir nach einigen Tests herausfinden, welchen Datendurchsatz und welche Latenz unsere Umgebung bietet. Uns interessiert vor allem der Datendurchsatz pro Partition, sowohl auf Producer- als auch auf der Consumer-Seite.

Um eine erste Schätzung für die benötigte Anzahl an Partitionen zu bekommen, benutzen wir die zuvor ermittelten Werte und teilen unsere Bandbreitenanforderung t (wie Throughput) jeweils durch die gemessene Producer-Bandbreite p und Consumer-Bandbreite c und nehmen das Maximum der beiden Divisionen als benötigte Mindestanzahl an Partitionen:

$$\text{partitionCount} = \max\left(\frac{t}{c}, \frac{t}{p}\right)$$

Aber dies ist nicht der einzige Faktor, den wir beachten sollten, wenn wir uns Gedanken um die Anzahl der Partitionen machen. Mehr Partitionen führen nicht nur zu einem erhöhten Datendurchsatz, sondern bringen auch Probleme mit sich. Die Clients legen pro Partition Buffer an, und wenn ein Client mit sehr vielen Partitionen, möglicherweise verteilt auf viele Topics, interagiert (als Producer oder Consumer), summiert sich der RAM-Verbrauch stark auf. Eine große Anzahl an Partitionen erhöht nicht nur den RAM-Bedarf bei den Clients, sondern führte vor Kafka 3.0 auch zu längeren Phasen der Nichterreichbarkeit bei Aus-

fällen. Fällt ein Broker aus, müssen die Leader neu verteilt werden, damit Producer und Consumer weiter mit Kafka interagieren können. Vor Kafka 3.0 musste Kafka einige Schreiboperationen in Zookeeper vornehmen. Dies ließe sich nicht parallelisieren, und jeder Vorgang dauerte etwa 10 ms pro Leader. Dies mag wenig erscheinen, aber wenn wir auf einem Broker Hunderte oder gar Tausende von Partitionen haben, summiert sich dies auf viele Sekunden oder gar Minuten auf. In dieser Zeit konnten wir mit einigen Partitionen nicht interagieren.

Wir wissen inzwischen, dass wir die Partitionen nutzen, um unsere Operationen zu parallelisieren. Um die Last gleichmäßig auf die vorhandenen Systeme aufzuteilen, sollte sich die Anzahl der Partitionen gut durch die Anzahl der Systeme teilen lassen. Auf den Brokern ist dies nicht allzu problematisch, da wir meistens eine Vielzahl an Topics mit mehreren Partitionen haben und sich die Last so im Idealfall sehr gleichmäßig verteilt. Wenn wir über die Lastverteilung bei Partitionen sprechen, geht es meistens um die Lastverteilung unserer Consumer. Hier ist es wichtig, dass sich die Anzahl an Partitionen gut durch die Anzahl an Consumern in einer Consumer Group teilen lässt. Dies hat zur Folge, dass wir es vermeiden sollten, Primzahlen als Partitionsanzahl zu benutzen, sondern stattdessen Zahlen, die gut teilbar sind. In unseren Beratungsaufträgen sehen wir deshalb die Zahl 12 als guten Richtwert, falls wir sehr wenige oder keine Anhaltspunkte dazu haben, mit welchen Datenmengen zu rechnen ist. Natürlich sind zwölf Partitionen nicht für alle Anwendungsfälle geeignet. Für manche Anwendungen ist diese Zahl zu groß, für andere viel zu klein. Aber zwölf Partitionen pro Topic führen auch bei sehr wenigen Daten zu keinem erheblichen Mehraufwand und reichen auch für viele größere Anwendungsfälle aus. 12 ist teilbar durch 1, 2, 3, 4, 6 und 12, was uns bei dieser Anzahl an Consumern eine gute Lastverteilung ermöglicht.



Tipp

Falls wir sehr wenige oder keine Anhaltspunkte haben, ist zwölf ein guter Richtwert für die Anzahl an Partitionen.

2.4.1.4 Verändern der Anzahl an Partitionen

Wir besprechen die Anzahl an Partitionen in dieser Ausführlichkeit hier, weil sie sich nicht ohne Weiteres ändern lässt, zumindest wenn wir uns auf die korrekte Reihenfolge der Nachrichten verlassen möchten. In Kafka lässt sich die Anzahl der Partitionen nur erhöhen. Es ist nicht möglich, sie im Nachhinein für ein Topic zu verkleinern, denn dafür müsste Kafka Partitionen löschen. Dies würde Kafka vor die unlösbare Aufgabe stellen, was es mit den Nachrichten in den gelöschten Partitionen machen soll, denn wir können Nachrichten nicht einfach im Nachhinein in andere Partitionen schreiben, da hierdurch Kafkas Garantie der Reihenfolge von Nachrichten nicht mehr eingehalten werden könnte. Wenn wir aber die Anzahl der Partitionen erhöhen, dann sind die neu erstellten Partitionen zunächst leer. Wir haben somit sofort ein Ungleichgewicht. Dies mag kein Problem sein, wenn wir die Daten in den Partitionen nur für eine kurze Zeit aufbewahren (zum Beispiel nur wenige Tage). Spätestens nach Ablauf der *Retention-Time* sind die Datenmengen in den Partitionen wieder gleich. Ein viel größeres Problem ist allerdings, dass wir bei Änderung der Anzahl an Partitionen die korrekte Reihenfolge der Nachrichten nicht mehr garantieren können, zumindest bis zum Ablauf der Retention Time. Denn um zu bestimmen, in welche Partition unser

Producer Daten produziert, nutzen wir den Hash des Keys und rechnen ihn Modulo der Anzahl an Partitionen. Erhöht sich die Anzahl, so kommen bei den meisten Hashwerten andere Zahlen heraus, und die Nachrichten können nun auf anderen Partitionen landen. Es gibt natürlich Anwendungsfälle, in denen das kein Problem ist, aber dennoch sollten wir sehr vorsichtig sein und es uns sehr genau überlegen, ob wir die Anzahl der Partitionen im Nachhinein ändern möchten. Wenn wir keine Keys benutzen, weil uns die Reihenfolge der Daten im Topic allgemein egal ist, ist es, abgesehen von der Ungleichverteilung der Nachrichten auf die Partitionen, ansonsten kein großes Problem, die Anzahl an Partitionen zu erhöhen.

Ist es für uns keine Option, dass die Reihenfolge der Nachrichten durcheinanderkommt und wir dennoch die Anzahl der Partitionen erhöhen möchten, oder für den Fall, dass wir uns verkalkuliert haben und die Anzahl der Partitionen reduzieren möchten, bleibt uns nichts anderes übrig, als ein neues Topic zu erstellen und das alte Topic zu löschen. Wie wir das am besten angehen, hängt vom Anwendungsfall an. Im einfachsten Fall verarbeitet unser System Nachrichten sofort, nachdem sie geschrieben wurden, und wir haben den Luxus, uns Wartungsfenster einrichten zu können. In diesem Fall könnten wir zu Beginn des Wartungsfensters unsere Producer stoppen, anschließend warten, bis alle Nachrichten verarbeitet wurden, und dann die betreffenden Topics löschen und wieder neu erstellen. In vielen Anwendungsfällen, in denen wir einen reibungslosen Betrieb zu jeder Tages- und Nachtzeit sicherstellen wollen, ist so eine Migration aufwendiger. Üblicherweise erstellen wir zuerst ein neues Topic mit der gewünschten Konfiguration. Dann nutzen wir zum Beispiel *Kafka Streams*, um die Daten aus dem alten Topic in das neue Topic zu kopieren. Somit können wir garantieren, dass die Reihenfolge der Nachrichten eingehalten wird, wenn wir Keys benutzen. Nach dem Erstellen des Topics und dem Kopieren der Daten müssen wir unsere Consumer vom alten Topic auf das neue Topic umziehen. Dies allein ist kein trivialer Prozess, denn wir müssen uns überlegen, wie wir die Offsets übersetzen, da die Offsets für die neuen Partitionen anders sein werden. Zuletzt migrieren wir alle Producer zum neuen Topic und löschen das alte Topic.



Tipp

Wie wir sehen, ist das nachträgliche Ändern der Anzahl der Partitionen sehr zeitaufwendig und fehleranfällig. Deshalb empfiehlt es sich, dass wir uns im Vorfeld gut überlegen, welche Anforderungen für unsere Topics bestehen.

In diesem Abschnitt haben wir uns erschlossen, wie Kafkas Partitionierung nicht nur für die gute Skalierbarkeit sorgt, sondern wie wir mithilfe der Partitionen die Parallelisierbarkeit unserer Operationen sicherstellen können und somit die Performance verbessern.

2.4.2 Producer Performance

Nachdem wir nun wissen, wie wir die Performance unserer Topics optimieren, können wir dazu übergehen, unsere Strecke vom Producer über die Broker bis hin zum Consumer näher zu betrachten. Da der Producer der Punkt ist, an dem Nachrichten in das Kafka-Cluster

produziert werden, können wir hier großen Einfluss auf die Performance und die Ressourcenauslastung des Gesamtsystems nehmen.

2.4.2.1 Producer-Konfiguration

Um Nachrichten zu produzieren, nutzen Producer üblicherweise entweder die Java-Kafka-Bibliothek oder, wenn eine andere Programmiersprache eingesetzt wird, eine Bibliothek basierend auf librdkafka¹⁹⁾. Der Programmcode übergibt üblicherweise nur die Nachricht mit Value und optionalem Key. Die Bibliothek entscheidet dann, in welche Partition diese Nachricht produziert werden soll. Kafka nutzt Batching, also gruppert mehrere Nachrichten in einem Batch, um den Datendurchsatz zu verbessern. Mithilfe des Batchings können wir durch größere Netzwerknachrichten den Datendurchsatz erhöhen, aber verschlechtern unter Umständen dabei die Latenz, da die Kafka-Bibliothek warten muss, bis der Batch voll ist. Batching können wir auf jedem Producer unabhängig voneinander konfigurieren. Dafür gibt es zwei Konfigurationsoptionen, die `batch.size` und die `linger.ms`. Die `batch.size` gibt an, wie groß ein Batch maximal sein darf. Standardmäßig ist diese Einstellung auf 16 384 Bytes, also 16 KB gesetzt. Die `linger.ms` ermöglicht es uns einzustellen, wie lange die Kafka-Bibliothek maximal auf weitere Nachrichten warten soll, bis der Batch abgeschickt wird. Standardmäßig ist diese Zeit auf 0 gesetzt. Das bedeutet, dass der Producer die Nachrichten so schnell wie möglich produziert. Wir verzichten hier nicht komplett auf Batching, denn oft ist es der Fall, dass wir nicht so schnell Nachrichten verschicken können, wie sie produziert werden, und wir können so trotzdem von Batching profitieren. Wenn uns die sofortige Zustellung der Nachrichten nicht so wichtig ist, aber wir den Datendurchsatz erhöhen möchten, dann können wir sowohl die `batch.size` auf einen größeren Wert setzen als auch die `linger.ms`. Wir sollten dabei darauf achten, dass die `batch.size` in Bytes und die `linger.ms` in Millisekunden angegeben wird. Das heißt, wir brauchen keine Angst vor großen Zahlen zu haben. Ist die Latenz für uns wichtiger als der Datendurchsatz, können wir die Standardeinstellungen beibehalten.

Wir haben uns bereits im Kapitel 2.3 „Zuverlässigkeit“ mit den Acknowledgements, den ACKs, beschäftigt und gesehen, dass wir mithilfe der ACKs steuern können, wie zuverlässig oder performant Nachrichten vom Producer an Kafka zugestellt werden. Wir können die Performance auf Kosten der Zuverlässigkeit mit ACK-Werten von 1 oder gar 0 verbessern, aber meist hängt der ACK-Wert nicht von unseren Performance-, sondern von unseren Zuverlässigkeitssanforderungen ab.

Dafür gibt es mittels Komprimierung eine andere Stellschraube, die nicht nur die Performance unserer Kafka-Umgebung deutlich verbessern, sondern auch deren Ressourcenbedarf reduzieren kann. Wir können mithilfe der Einstellung `compression.type` die Art der Komprimierung beeinflussen. Standardmäßig ist Komprimierung zwar ausgestellt, aber Kafka unterstützt einige Algorithmen, so können wir zwischen `none` (keine), `gzip`, `snappy`, `lz4` oder `zstd` wählen. Je nach Anforderung kann ein Algorithmus besser funktionieren als die anderen. Kafka komprimiert dabei nicht einzelne Nachrichten, sondern gesamte Batches auf der gesamten Strecke vom Producer zum Consumer. Dies ist möglich, da ein Producer für das Versenden von Nachrichten pro Partition einen separaten Batch verwendet. Das heißt, alle Nachrichten für eine Partition werden in einem Batch gruppiert, und Nachrichten für eine andere Partition werden in einem separaten Batch gesammelt.

¹⁹⁾ <https://github.com/edenhill/librdkafka/>

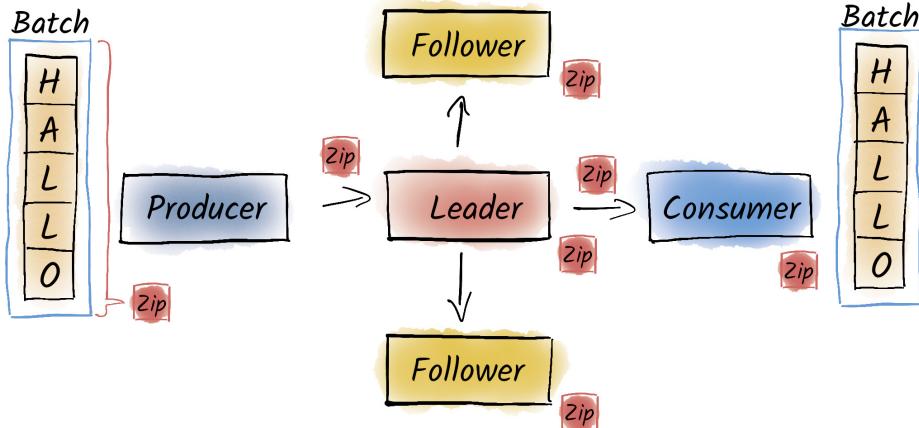


Bild 2.19 Wenn wir Komprimierung anschalten, komprimiert der Producer ganze Batches. Diese komprimierten Batches schicken wir nicht nur über das Netzwerk, sondern sie werden auch komprimiert auf den Brokern abgelegt. Erst der Consumer muss den Batch final entpacken, um die einzelnen Nachrichten zu extrahieren.

Das heißt, dass aktiveres Batching auch zu verbesserter Komprimierung führen kann. Der Producer komprimiert dabei einmalig den gesamten Batch und sendet diesen Batch komprimiert an den Leader. Der Leader packt den Batch nicht aus, sondern speichert ihn unverändert auf der lokalen Festplatte und überträgt diesen komprimierten Batch ebenfalls unverändert an die Follower. Erst wenn der Consumer den Batch erhält, wird dieser entpackt und in die einzelnen Nachrichten aufgeteilt. Das bedeutet, dass Komprimierung nicht nur einmalig Vorteile hat, sondern uns an vielen Stellen sowohl die Netzwerklast reduziert als auch den benötigten Festplattenspeicher.

2.4.2.2 Producer-Performance-Test

Wie testen wir aber die Performance unserer Producer am besten? Kafka wird mit dem Werkzeug `kafka-producer-perf-test.sh` ausgeliefert, mit dem wir unterschiedliche Konfigurationseinstellungen und Nachrichtenarten auf die Performance-Auswirkungen testen können. Wir sollten bei diesen Tests immer beachten, dass dies kein Ersatz für Performance-Tests mit echten Producern ist. Für solche Ende-zu-Ende-Performance-Tests sind Werkzeuge wie Apache JMeter²⁰ besser geeignet.

Am einfachsten starten wir den `kafka-producer-perf-test.sh` mit folgendem Befehl. Dafür nehmen wir an, dass wir bereits ein Topic `performance-test` erstellt haben:

```
$ kafka-producer-perf-test.sh \
--topic performance-test \
--num-records 1000000 \
--record-size 10000 \
--throughput -1 \
--producer-props bootstrap.servers=localhost:9092
```

²⁰⁾ <https://jmeter.apache.org/>

Hier produzieren wir 1 000 000 Nachrichten (`--num-records`), die jeweils 10 000 Bytes groß sind (`--record-size`), ohne den Datendurchsatz zu limitieren (`--throughput`), also so schnell wie möglich. Wir produzieren die Daten in das Topic *performance-test* (`--topic`). Nachdem wir diesen Befehl ausgeführt haben, erhalten wir eine Auswertung, die ähnlich wie die folgende aussieht:

```
39481 records sent, 7896,2 records/sec (75,30 MB/sec), 238,1 ms avg latency, 399,0 ms max latency.
62616 records sent, 12523,2 records/sec (119,43 MB/sec), 164,1 ms avg latency, 253,0 ms max latency.
...
1000000 records sent, 12317,546345 records/sec (117,47 MB/sec), 165,43 ms avg latency, 399,00 ms max latency, 157 ms 50th, 211 ms 95th, 287 ms 99th, 353 ms 99.9th.
```

Schauen wir uns die Ausgabe genauer an. Der Befehl gibt uns alle paar Sekunden die aktuellen Statistiken aus. Es ist nicht unüblich, dass es am Anfang langsamer losgeht und sich der Datendurchsatz und die Latenz nach einigen Ausgaben verbessern. Meistens lässt sich das auf das sogenannte Warmwerden der *Java Virtual Machine* zurückführen. Nachdem die gewünschte Anzahl der Nachrichten produziert ist, können wir die Ergebnisse betrachten und sehen in unserem Beispiel, dass Kafka ungefähr 12317 Nachrichten pro Sekunde mit insgesamt 117,47 MB/s verschickt hat. Unsere durchschnittliche Latenz lag bei 165 ms.

Wenn wir den Performance-Test nur einmalig ausführen, sollten wir vorsichtig sein, zu viel in das Ergebnis zu interpretieren. Um Performance-Tests korrekt durchzuführen, müssen wir mehrere Wiederholungen durchführen und diese dann auswerten. Aber es gibt uns einen ersten Anhaltspunkt, wie wir die Performance weiter verbessern könnten.

Wir wissen zum Beispiel, dass die Vergrößerung der Batches und die gleichzeitige Erhöhung der Linger-Zeit zu einem verbesserten Datendurchsatz führen. Probieren wir es doch mal aus:

```
$ kafka-producer-perf-test.sh \
  --topic performance-test \
  --num-records 1000000 \
  --record-size 10000 \
  --throughput -1 \
  --producer-props bootstrap.servers=localhost:9092 \
    batch.size=100000 linger.ms=100
...
1000000 records sent, 45879,977978 records/sec (437,55 MB/sec), 64,61 ms avg latency, 279,00 ms max latency, 58 ms 50th, 101 ms 95th, 126 ms 99th, 178 ms 99.9th.
```

Wir sehen hier, dass wir, statt wie zuerst etwa 117 MB/s, ganze 437,55 MB/s Datendurchsatz erreichen. Interessanterweise sinkt sogar die Latenz von ursprünglich 165 ms auf 65 ms. Der Grund für die Verbesserung der Latenz liegt hier darin, dass wohl die Batch Size für den Anwendungsfall zu gering gewählt wurde. Wir erinnern uns, dass die Batch Size standardmäßig auf 16 KB gesetzt ist. Wenn wir also Nachrichten mit je etwa 10 KB generieren, nutzen wir das Batching gar nicht aus, und die Nachrichten stauen sich auf. Das führt zu verschlechterter Latenz.

Konfigurationseinstellung	Beschreibung
acks	0: Höherer Durchsatz, Datenverlust 1: Etwas geringerer Durchsatz, Datenverlust möglich -1: Noch etwas geringerer Durchsatz, höhere Zuverlässigkeit
enable.idempotence	false: Vernachlässigbarer Performanceunterschied true: Garantie der Reihenfolge und Eindeutigkeit von Nachrichten
compression.type	none: Keine Komprimierung gzip: Bessere Komprimierung, mehr CPU-Last zstd, snappy, lz4: Kann performanter als gzip sein
batch.size.linger.ms	Größere Werte: eher höhere Latenz, eher höherer Durchsatz

Dieses Beispiel soll nochmals verdeutlichen, dass es essenziell ist, die Performance im eigenen Cluster mit den gegebenen Anwendungsfällen zu testen. Die Tabelle gibt zwar Hinweise, was beim Performance-Tuning eines Producers zu beachten ist, aber kann einen echten Performance-Test nicht ersetzen. Denn jeder Anwendungsfall ist unterschiedlich, und wir sollten die Einstellungen stets mit Bedacht wählen.

2.4.3 Broker Performance

Die Broker sind die zentrale Entität in Kafka. Alle Daten werden auf den Brokern gespeichert, die Clients kommunizieren mit den Brokern, und die Broker verwalten sich mithilfe des Koordinationsclusters selbst. Wir haben schon in der Einleitung dieses Kapitels gesehen, dass Kafka sehr viel Wert auf Performance legt und die Performance-Optimierungen bei der Systemarchitektur anfangen. Dies wird besonders bei unseren Brokern deutlich. Konzeptionell unterscheidet sich Kafka grundlegend von klassischen Messaging-Systemen, denn wo viele klassische Messaging-Systeme und vor allem Enterprise-Service-Busse versuchten, den Clients möglichst viel Arbeit abzunehmen, geht Kafka einen anderen Weg. Die Broker versuchen, möglichst wenig selbst zu machen und möglichst alle performance-kritischen Operationen an die Clients auszulagern. Deswegen sind die Producer dafür zuständig, sich für eine Partition zu entscheiden. Kafka interpretiert deshalb auch keine Nachrichten, sondern unterstützt ausschließlich Byte-Arrays als Nachrichteninhalt.

2.4.3.1 Die Aufgaben eines Brokers

Kafka-Broker schieben am Ende des Tages nur Bytes von Netzwerk-Sockets auf die Festplatte (beim Produzieren) und von der Festplatte auf die Netzwerk-Sockets (beim Konsumieren). Aber auch hier versucht die Kafka-Software, möglichst wenig selbst zu erledigen. Viele Datenbanksysteme versuchen zum Beispiel, das Betriebssystem, wo es geht, auszutricksen und zu umgehen, um bessere Zugriffsgeschwindigkeiten auf die benötigten Daten zu bekommen. Kafka dagegen arbeitet mit dem Betriebssystem und nicht dagegen an. Kafka versucht, es dem Betriebssystem möglichst einfach zu machen, die Daten, die Kafka benötigt, vorzuhalten. Die Kernidee ist das sequenzielle Lesen und Schreiben. Kafkas Zu-

griffsmuster sind sehr vorhersehbar. Wir schreiben entweder an das Ende einer Datei oder lesen sequenziell in einer Datei. Selbst auf klassischen Festplatten mit rotierenden Scheiben sind diese Zugriffe performant. Kafka geht aber bei der Optimierung noch weiter. Wir haben bereits besprochen, dass Kafka die geschriebenen Daten nicht auf das Dateisystem committet. Das bedeutet, dass Kafka Daten nur in den Page Cache im Hauptspeicher schreibt und das Betriebssystem nicht anweist, diese Daten auf Festplatte zu schreiben. Kafka verlässt sich darauf, dass das Betriebssystem dies später im Hintergrund selbstständig macht. Dennoch schickt Kafka nach dem Schreiben der Daten in den Hauptspeicher ein ACK an den Producer, obwohl bei einem Stromausfall die Daten verloren gehen würden. Das mag sich auf den ersten Blick katastrophal anhören, aber Kafkas Zuverlässigkeit kommt nicht daher, dass die Daten sofort auf Festplatte geschrieben werden, sondern dass die Daten über mehrere Broker hinweg repliziert werden. Dies setzt voraus, dass nicht alle Broker auf einmal abstürzen und die Daten verlieren. Wenn die Broker auf unterschiedlichen Rechnern in unterschiedlichen Teilen eines Rechenzentrums betrieben werden, ist dies meist eine legitime Annahme.

Da Kafkas Datenformat überall gleich ist, also die Daten, die der Producer produziert, genauso über das Netzwerk übertragen, auf Festplatte geschrieben und so vom Consumer empfangen werden, können die Broker eine Funktion des Linux Kernels, den sogenannten Zero-Copy Transfer, benutzen. Eigentlich müssten die Daten, die ein Producer produziert, vom Netzwerk-Socket im Hauptspeicher in den Page Cache im Hauptspeicher geschrieben werden. Aber da die Daten identisch sind, muss der Linux Kernel nur ein paar Zeiger verändern, und die Daten landen, ohne dass sie bewegt wurden, im Page Cache.

2.4.3.2 Broker-Konfiguration und -Optimierung

Wie können wir aber selbst Einfluss auf die Broker Performance nehmen? Grundsätzlich gilt, dass Kafkas Voreinstellungen auf der Broker-Seite schon von sich aus auf Performance getrimmt sind. Bevor wir Kafka betreiben, ist es empfehlenswert, einige Betriebssystemeinstellungen anzupassen. Von besonderer Wichtigkeit ist es, die maximale Anzahl der offenen File *Descriptors* zu erhöhen, da die Broker jedes Segment in jeder Partition als offene Datei halten und sich das schnell addiert. Weiterhin sollten wir auch einige Einstellungen bezüglich des virtuellen Speichers tätigen, damit das Betriebssystem früh anfängt, im Hintergrund Daten auf Festplatte zu schreiben und unsere Broker-Prozesse möglichst nicht blockiert. Auch möchten wir die Swappiness des Betriebssystems ausschalten oder auf ein Minimum reduzieren und einige Einstellungen auf der Netzwerkebene tätigen. Es würde dieses Kapitel sprengen, wenn wir jede Änderung im Detail besprechen, deshalb verweisen wir hier auf weitere (Online-)Literatur²¹⁾, die im Zweifel aktueller ist als dieses Buch. Wir sollten uns auch bewusst sein, dass es nicht in jeder Umgebung möglich ist, diese Parameter zu beeinflussen. Wenn wir zum Beispiel Kafka in einer Kubernetes-Umgebung laufen lassen möchten, haben wir meistens keinen Einfluss auf diese Werte. Wir können stattdessen ein ausführliches Monitoring betreiben, um wenigstens festzustellen, wenn wir an gewisse Limits kommen, um Zeit zu haben, uns zu überlegen, wie wir damit umgehen möchten.

²¹⁾ Zum Beispiel bietet Cloudera zum Zeitpunkt des Schreibens eine gute Übersicht über die empfohlenen Parameter an: https://docs.cloudera.com/documentation/enterprise/latest/topics/kafka_system_level_broker_tuning.html

Grundsätzlich gilt bei jeder Art der Optimierung: erst verstehen, dann messen, dann optimieren. Dies gilt insbesondere für Kafka, da die Broker-Konfiguration standardmäßig bereits recht performant ist. Aber die Anzahl der Broker ist etwas, über das wir uns schon frühzeitig Gedanken machen sollten. Die meisten kleineren Umgebungen starten mit drei Brokern. Damit können wir für unsere Partitionen einen *Replication Factor* von 3 benutzen und können im Falle einer Broker-Wartung auch den Ausfall eines weiteren Brokers überstehen, ohne dass es zu einem Ausfall kommt. Wir müssen die Anzahl der Broker anhand dessen abschätzen, welche Rechnerarten uns zur Verfügung stehen. Wie viel Hauptspeicher und Festplattenplatz benötigen wir? Mit welchen Datendurchsätzen rechnen wir? Welche Netzwerkanbindung liegt vor? Wie lange möchten wir Daten aufbewahren? Grundsätzlich müssen wir eine gute Balance zwischen zu großen und zu kleinen Brokern finden. Je mehr Broker wir haben, desto weniger stört uns der Ausfall eines einzelnen Brokers. Aber desto mehr Verwaltungsaufwand haben wir, und wir sollten die Größe der Rechner nicht zu klein wählen, da Kafka selber einen gewissen CPU und Speicherverbrauch mit sich bringt, der mit wachsenden Rechnern weniger relevant wird.

Wenn wir zum Beispiel Rechner mit 2 TB Festplattspeicher verwenden und wissen, dass wir am Tag etwa 1 TB an Daten verarbeiten, und diese für sieben Tage aufheben möchten, dann bräuchten wir mindestens vier Broker, um die 7 TB Daten zu speichern. Aber diese Daten haben wir noch nicht repliziert. Bei einem *Replication Factor* von 3 bräuchten wir nun Broker, die insgesamt 21 TB an Daten speichern, also elf Broker. Bei diesen Datenmengen sollten wir uns auch überlegen, mit welchen Datenraten wir zu rechnen haben und was unsere Netzwerkschnittstelle leisten kann.

Bei kleineren Installationen ist es üblich, erst mal mit drei Brokern zu starten und für den Fall, dass die Broker mit den Datenmengen überlastet sind, mehr Broker hinzuzufügen.

2.4.4 Consumer Performance

Im Gegensatz zu vielen klassischen Messaging-Systemen sind die Consumer in Kafka fetch-basiert. Das heißt, die Kafka Broker senden Daten nicht proaktiv an die Consumer, sondern die Consumer holen sich die Daten von Kafka selbstständig ab. Die Consumer entscheiden selbstständig, wann sie fetchen, welche Daten sie bekommen möchten und wie viele. Der Vorteil dieses Ansatzes ist es, dass unsere Consumer nicht überlastet werden können, außer es hat sich ein Fehler in den Programmcode eingeschlichen.

2.4.4.1 Consumer-Konfiguration

Ähnlich wie beim Producer haben wir bei den Consumern die Möglichkeit, eher die Bandbreite oder eher die Latenz zu verbessern. Dafür haben wir beim Consumer zwei Konfigurationseinstellungen. Da ist einmal die `fetch.min.bytes`-Einstellung, die den Broker anweist, mindestens auf diese Summe an Bytes zu warten, bevor er dem Consumer antwortet. Standardmäßig ist `fetch.min.bytes=1`, der Broker schickt dem Consumer also eine Antwort, sobald es mindestens 1 Byte an neuen Nachrichten gibt. Der Broker schickt auch keine einzelnen Nachrichten an den Consumer, sondern immer ganze Batches. Es ist sogar möglich, mehrere Batches in einer Antwort zu erhalten. Natürlich wartet der Broker nicht ewig, bis er dem Consumer eine Antwort schickt, sondern der Consumer kann zudem mit

der Einstellung `fetch.max.wait.ms` konfigurieren, wie lange der Broker maximal warten darf, bis er eine Antwort schickt, selbst wenn weniger als `fetch.min.bytes` Daten vorliegen. Standardmäßig ist der Wert auf 500 ms gesetzt. Das heißt, auch wenn es keine neuen Nachrichten gibt, bekommt der Consumer nach 500 ms eine Antwort auf seine Anfrage. Nachdem der Consumer die Antwort erhalten hat und die Nachrichten verarbeitet wurden, bittet der Consumer um die nächsten Nachrichten. Dies erfolgt natürlich mit einem höheren Offset, wenn wir Nachrichten verarbeitet haben.

2.4.4.2 Consumer-Performance-Test

Ähnlich wie beim Producer können wir auch die Performance eines Consumers testen. Nehmen wir dazu am besten das Topic, in welches wir mit dem `kafka-producer-perf-test.sh` produziert haben. Das Werkzeug für Consumer-Performance-Tests heißt `kafka-consumer-perf-test.sh`:

```
$ kafka-consumer-perf-test.sh \
  --topic performance-test \
  --messages 1000000 \
  --bootstrap-server localhost:9092
```

Leider hat dieses Kommando ein leicht anderes Verhalten als der `kafka-producer-perf-test.sh`. Wir können nicht einfach per Kommandozeile die Consumer-Konfiguration beeinflussen, sondern müssen eine Konfigurationsdatei übergeben. Wenn wir zum Beispiel die `fetch.min.bytes` hoch auf 1 MB und das `fetch.max.wait.ms` explizit auf 500 ms setzen möchten, erstellen wir zum Beispiel die Datei `consumer.properties` mit folgendem Inhalt:

```
fetch.min.bytes: 1000000
fetch.max.wait.ms: 500
```

Nun starten wir den `kafka-consumer-perf-test.sh` noch einmal unter Einbeziehung der Konfigurationsdatei:

```
$ kafka-consumer-perf-test.sh \
  --topic performance-test \
  --consumer.config ./consumer.properties \
  --messages 1000000 \
  --bootstrap-server localhost:9092
```

Die Ausgabe ist leider sehr unübersichtlich:

```
start.time, end.time, data.consumed.in.MB, MB.sec, data.consumed.in.nMsg, nMsg.sec,
rebalance.time.ms, fetch.time.ms, fetch.MB.sec, fetch.nMsg.sec
2021-03-17 10:59:37:208,...
```

Wir unterdrücken die gesamte Ausgabe, sie ist lang, und es ist einfacher, wenn wir uns die Ausgaben der beiden Consumer-Tests in einer Tabelle anschauen:

Name	Ergebnisse mit Standardwerten	Ergebnisse mit unserer consumer.properties
start.time	2021-03-17 10:44:18:309	2021-03-17 10:59:37:208
end.time	2021-03-17 10:44:35:742	2021-03-17 10:59:50:200
data.consumed.in.MB	9536,7432	9536,7432
MB.sec	547,0512	734,0473
data.consumed.in.nMsg	1000000	1000000
nMsg.sec	57362,4735	76970,4433
rebalance.time.ms	1615974258616	1615975177627
fetch.time.ms	-1615974241183	-1615975164635,
fetch.MB.sec	-0,0000	-0,0000
fetch.nMsg.sec	-0,0006	-0,0006

Auch in Tabellenform müssen wir genau hinsehen, um zu verstehen, was uns der Performance-Test sagen möchte. Die Zeilen mit `start.time` und `end.time` zeigen uns, wie lange die Befehle gebraucht haben. Mit unserer angepassten `consumer.properties` braucht der Befehl nur 13 Sekunden statt 17 Sekunden. Beide Befehle verarbeiten die gleiche Anzahl an Nachrichten (`data.consumed.in.nMsg=1000000`) und dieselbe Datenmenge (`data.consumed.in.MB=9536,7432`), also etwa 9,5 GB. Wir erreichen so mit unserer Konfigurationsdatei etwa 734 MB/s lesend und mit den Standardwerten 547 MB/s (`MB.sec`). Die untersten vier Zeilen sind nur relevant, wenn wir Consumer Groups benutzen. Wenn wir sie nicht benutzen (wie in unserem Fall), dann können wir sie ignorieren, da sie wenig nützliche Zahlen anzeigen.

Wir sehen also, dass wir mit einigen wenigen Konfigurationseinstellungen unsere Consumer-Performance deutlich verbessern können. Wir sollten aber stets beachten, dass beim Consumer die Performance meistens nicht durch Kafka limitiert ist, sondern dadurch, wie schnell der Consumer die Daten dann weiterverarbeitet. Wir können hier auch sehen, dass unser Consumer deutlich schneller ist als unsere Producer in unseren Producer-Performance-Tests. Unser Producer erreichte bis zu 437 MB/s und unsere Consumer nun bis zu 734 MB/s.

Diese Tests, die wir hier durchgeführt haben, sind nicht repräsentativ und zeigen nur beispielhafte Werte und sind mit Vorsicht zu genießen. Für echte Performance-Tests sollten wir auch echt aussehende Daten benutzen und Kafka in einer produktionsähnlichen Umgebung laufen lassen.

Wir haben unsere Tests hier jeweils für ein Topic mit einer Partition und ohne Replikation durchgeführt. Damit haben wir zwar keinen zusätzlichen Aufwand Daten zu replizieren, aber wir können unsere Operationen nicht parallelisieren, denn dazu bräuchten wir mehr als eine Partition.

Wir haben gesehen, dass einzelne Consumer die Daten oft sehr performant empfangen können. Der Flaschenhals ist meistens nicht Kafka selbst, sondern entweder die Netzwerk-anbindung oder die Weiterverarbeitung in unserem eigenen Code. Um unsere Consumer

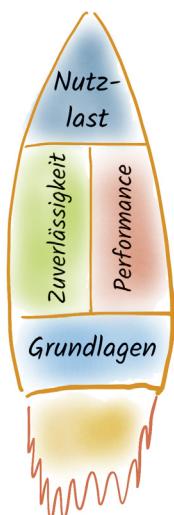
wirklich performant arbeiten zu lassen, reicht ein einziger Consumer oft nicht aus. Statt dessen nutzen wir Consumer Groups, um unsere Consumer zu skalieren und die Datenverarbeitung zu parallelisieren. Zusätzlich benötigen wir Consumer Groups, um mit Kafkas Bordmitteln unsere Offsets zu speichern. Üblicherweise heißt das, dass wir selbst dann Consumer Groups benutzen, wenn wir nur einen Consumer haben, wodurch wir uns keine Gedanken um unsere Offsets machen müssen.

2.4.5 Zusammenfassung

In diesem Kapitel haben wir uns einen guten Überblick darüber verschafft, wie Kafka die versprochene Performance erreicht und wie wir sie selbst weiter für unseren konkreten Anwendungsfall optimieren können. Wir haben uns angeschaut, welchen Einfluss die Konfiguration unserer Topics auf die Performance hat, und sind dabei insbesondere auf Partitionierung eingegangen. In diesem Zusammenhang haben wir erfahren, dass es nicht ohne Weiteres möglich ist, die Anzahl an Partitionen im Nachhinein zu ändern. Wir sollten uns deshalb von vornherein gut überlegen, wie viele Partitionen wir benötigen. Des Weiteren haben wir uns sowohl für Producer als auch Consumer angesehen, welche Konfigurationsparameter jeweils Einfluss auf die Performance nehmen, und haben auch entsprechende Performance-Tests illustriert. Außerdem haben wir einen Blick auf die Aufgaben eines Brokers geworfen und auch darauf, welche Möglichkeiten wir hierbei haben, die Performance unseres Kafka-Clusters zu beeinflussen.

3

Kafka Deep Dive



In den vorherigen Kapiteln haben wir die Grundlagen Apache Kafkas ergründet. Wir haben uns Kafka als eine Rakete vorgestellt, eine Rakete mit hoher Zuverlässigkeit und Performance. Zur besseren Veranschaulichung haben wir diese Rakete in mehrere Teile aufgeteilt und einen nach dem anderen besprochen. Welche Nutzlasten können wir überhaupt mit Kafka transportieren? Wie und warum funktioniert Kafka? Was ist ein Log? Wie verteilen wir ein Log? Zu guter Letzt haben wir uns mit den zwei Treibstofftanks beschäftigt. Wie erreicht Kafka die Zuverlässigkeit, und wie können wir sie weiter verbessern? Warum ist Kafka so performant, und welche Möglichkeiten haben wir, die Performance von Kafka weiter zu verbessern?

In diesem Kapitel möchten wir eine Ebene tiefer eintauchen, um zu verstehen, was wirklich in Kafka passiert. Was passiert genau in den Producern, Brokern und Consumern? Wie werden Nachrichten persistiert, und wie können wir sie auch wieder löschen? Wie können wir *Exactly-Once*-Semantiken mithilfe von Kafka erreichen?

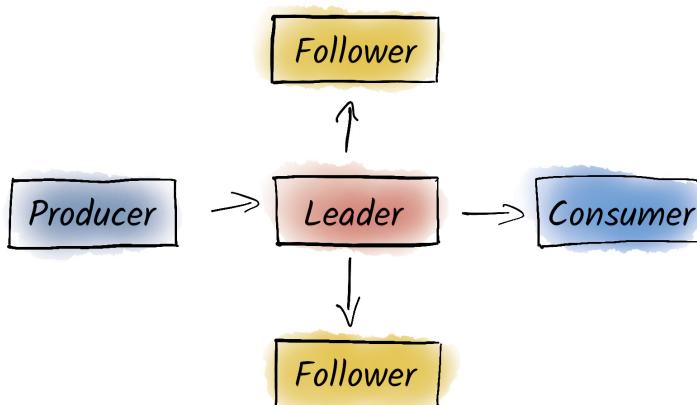


Bild 3.1 Was passiert eigentlich wirklich? Wie senden Producer Nachrichten an die Broker? Wie replizieren wir Daten zwischen Leader und Followern? Wie holen sich Consumer Daten ab? Alles das sind Themen für dieses Kapitel.

■ 3.1 Verbindung zu Kafka

Bevor wir als Client in der Lage sind, Nachrichten zu konsumieren oder zu produzieren, müssen wir uns als Erstes mit Kafka verbinden. Bei einem klassischen Datenbanksystem ist dies einfach möglich. Wir geben die Verbindungsdaten zur Datenbank an und sind verbunden. Kafka ist aber ein verteiltes System, bestehend aus mehreren Kafka-Brokern und einem Koordinationscluster. Außerdem liegen normalerweise nicht alle Leader aller Partitionen auf einem Broker. Die Informationen, welcher Leader für welche Partition auf welchem Broker zu finden ist, speichert Kafka im Koordinationscluster, und jeder Broker speichert diese Informationen in einem Cache zwischen. In den ersten Kafka-Versionen haben die Clients diese Informationen direkt beim Koordinationscluster beziehungsweise Zookeeper-Ensemble abgefragt. Seit Kafka 0.10.0 können diese Informationen direkt bei einem beliebigen Kafka-Broker in einem sogenannten *Metadata-Request* abgefragt werden.

Schauen wir uns beispielsweise folgenden Befehl an, der eine Liste aller Topics ausgibt:

```
$ kafka-topics.sh \
  --list \
  --bootstrap-server localhost:9092
```

Wir übergeben mit dem Parameter `--bootstrap-server` die URL zu einem unserer Kafka-Broker. Im Falle unserer Testumgebung aus dem Anhang ist das der Broker 1. Wir könnten stattdessen jeden beliebigen anderen Broker eintragen, und der Befehl würde weiterhin funktionieren. In einer produktiven Umgebung wäre dieser Ansatz, nur einen einzelnen Broker direkt anzusprechen, nicht empfehlenswert, da unsere Clients nicht mehr starten, sobald der Broker, den wir eingetragen haben, ausfällt. Stattdessen ist der empfohlene Ansatz alle Broker, im Parameter `--bootstrap-server` anzugeben:

```
$ kafka-topics.sh \
  --list \
  --bootstrap-server \
  localhost:9092,localhost:9093,localhost:9094
```

Da dies mit einer wachsenden Anzahl an Brokern umständlich wird und es egal ist, mit welchem Broker sich ein Client zuerst verbindet, können wir stattdessen einen Loadbalancer für diesen Metadata-Request benutzen oder dieses Loadbalancing über unser DNS bewerkstelligen. Dabei ist es aber wichtig, dass die Broker weiterhin eine eigene, erreichbare IP-Adresse oder einen DNS-Namen haben.

Unser Client schickt also an einen der Bootstrap-Server einen Metadata-Request, indem der Client um die Informationen für eine Menge an Topics bittet. Wenn wir zum Beispiel in einem Consumer das Topic *flugdaten* konsumieren möchten, würde die Antwort für den Metadata-Request die Informationen über alle Broker in unserem Cluster zurückliefern und zusätzlich dazu, welcher Broker für welche Partition dieses Topics der Leader ist.

Im nächsten Schritt kann sich unser Client mit den entsprechenden Leadern unserer Partitionen verbinden und Nachrichten abfragen. Der Client muss die Metadaten nicht mehr aktualisieren. Falls sich etwas Kritisches ändert, zum Beispiel ein Ausfall eines Brokers, der Leader für eine der konsumierten Partitionen ist, schickt der Client einen neuen Metadata-Request an einen der Bootstrap-Server.

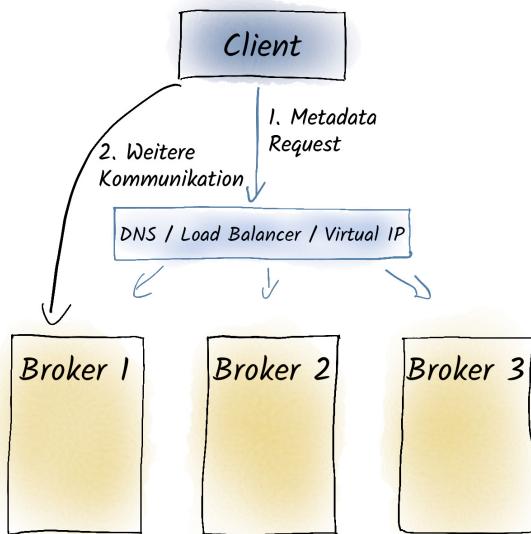


Bild 3.2 Bevor unsere Clients Nachrichten produzieren oder konsumieren können, müssen sie wissen, welcher Broker für was zuständig ist. Dafür gibt es den Metadata-Request. Da es egal ist an welchen Broker wir diesen schicken, können wir per DNS, Loadbalancer oder virtueller IP-Adressen den Verbindungsaufbau erleichtern. Ansonsten müssten wir in allen unseren Clients möglichst alle Broker-Adressen angeben. Nach dem Metadata-Request kommunizieren die Clients dann aber direkt mit den entsprechenden Brokern.

■ 3.2 Nachrichten produzieren und persistieren

Nachdem wir uns verbunden haben, können unsere Clients anfangen, ihre eigentliche Arbeit zu erledigen. Im Gegensatz zu klassischen Messaging-Systemen ist es Kafkas Philosophie, so viel Arbeit wie möglich an die Clients auszulagern; so auch beim Produzieren der Nachrichten. Bei vielen Messaging-Systemen muss der Producer nur eine Nachricht an ein Topic schicken, und das Messaging-System kümmert sich um die Zustellung der Nachricht. Bei Kafka sind die Producer und Consumer selbst für die Zustellung und die Abholung der Nachrichten verantwortlich. Die Kafka-Broker sind am Ende nur dafür verantwortlich, die Nachrichten korrekt zu empfangen und an die richtige Stelle des Hauptspeichers, beziehungsweise der Festplatte, zu schreiben. Beim Konsumieren von Nachrichten muss der Broker nur diese Nachrichten wieder korrekt lesen und an den Consumer senden.

3.2.1 Producer

Üblicherweise nutzen unsere Producer entweder die offizielle Kafka-Java-Bibliothek oder, wenn unser Producer nicht in der Java Virtual Machine läuft, eine Bibliothek, die auf der offiziellen C-Bibliothek `librdkafka`¹ aufbaut.



Tipp

Wir raten unseren Kunden tendenziell eher davon ab, andere Bibliotheken zu benutzen, da, obwohl sie manchmal einfacher in der Benutzung sind, oft viele Funktionen und Optimierungen fehlen.

Es würde ein ganzes Buch füllen, wenn wir uns tiefer mit den unterschiedlichen Aspekten der Entwicklung für Kafka befassen würden. Da dieses Buch kein Entwicklerhandbuch ist, möchten wir nur kurz auf die wichtigsten Details eingehen. Der Übersichtlichkeit halber setzen wir in unseren Code-Beispielen meistens Python ein. In Java und anderen Programmiersprachen sind die Grundkonzepte bei der Kommunikation mit Kafka identisch oder zumindest sehr ähnlich.

3.2.1.1 Nachrichten produzieren

Ein typischer Producer-Code in Python sieht meistens folgendermaßen aus:

```

1. from confluent_kafka import Producer
2.
3. producer = Producer({
4.     'bootstrap.servers': 'localhost:9092',
5.     'partitioner': 'murmur2_random',
6.     'acks': -1,
7. })
8. producer.produce(
9.     "flugdaten",
10.    key="rakete1",
11.    value="Countdown gestartet",
12.    on_delivery=delivery_report)

```

Wir importieren unsere Kafka-Bibliothek für unsere Programmiersprache. `confluent-kafka` ist die von Confluent offiziell unterstützte Python-Bibliothek. Bevor wir in der Lage sind, mit Kafka zu kommunizieren, erstellen wir einen Producer. Diesem übergeben wir als wichtigstem Parameter die `bootstrap.servers`. Wenn wir sowohl Java-Bibliotheken als auch Bibliotheken, die auf `librdkafka` aufbauen, benutzen, dann sollten wir in den `librdkafka`-Bibliotheken die Hash-Funktion des Partitioners manuell auf `murmur2_random`, den die Java-Bibliothek benutzt, setzen. `Librdkafka` setzt nämlich standardmäßig eine andere Hash-Funktion ein. Unterschiedliche Partitioner würden im schlimmsten Fall dazu führen, dass wir überhaupt keine Garantien über Reihenfolgen mehr geben könnten, da die Nachrichten für die gleichen Keys auf ganz unterschiedliche Partitionen verteilt werden würden.

¹⁾ <https://github.com/edenhill/librdkafka/>

Sobald wir unseren Producer initialisiert haben, können wir Nachrichten produzieren. In Python nutzen wir dafür die `produce()`-Methode. In Java würden wir erst die Nachricht, die wir verschicken möchten, manuell erstellen und dann an die `send()` Methode übergeben.

Wir übergeben dem Producer die Nachricht, die wir produzieren möchten und in welches Topic sie geschrieben werden soll. Optional können wir einen Callback (hier `delivery_report`) übergeben, der aufgerufen wird, sobald wir ein ACK für die Nachricht bekommen haben. Unsere Nachricht besteht aus dem Value und optional dem Key.

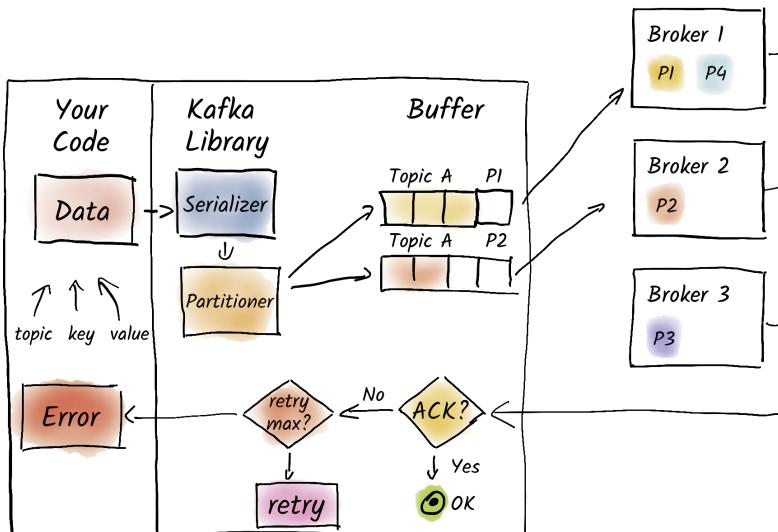


Bild 3.3 Was passiert beim Produzieren einer Nachricht? Nach dem Aufruf der `produce()`-Methode müssen die Daten zuerst mithilfe des *Serializers* serialisiert werden. Der *Partitioner* entscheidet dann, in welche Partition geschrieben werden soll, und legt die Nachricht in einen Puffer für eben diese Partition ab. Nachdem genug Nachrichten vorhanden sind, schickt der Producer ganze Batches an die entsprechenden Broker und wartet, je nach ACK-Einstellung, auf ein ACK. Wenn wir kein ACK bekommen, versuchen wir es eine Zeit lang erneut. Im Fehlerfall wirft die Kafka-Bibliothek eine Exception.

Was passiert aber zwischen dem Aufruf der `produce()`-Methode und dem Aufrufen des Callbacks? Eine ganze Menge, aber glücklicherweise übernimmt die Kafka-Bibliothek die meiste Arbeit für uns. Um Daten produzieren zu können, müssen wir der `produce()`-Methode mindestens das Topic mitgeben, wohin die Nachricht geschrieben werden soll, und einen Value.

3.2.1.2 Serialisierung und Partitionierung

Wir haben bereits ausführlich besprochen, dass Kafka den Inhalt von Nachrichten nicht interpretiert, sondern nur mit Byte-Arrays umgehen kann. Das heißt, dass unser Producer dafür zuständig ist, die Daten zu serialisieren. In unserer Python-Implementierung sehen wir die Magie, die passiert, nicht, da die Python-Bibliothek als Eingabewert einen String oder ein Byte-Array erwartet. Strings werden dann mit dem String-Serializer in Byte-Arrays umgewandelt. Unsere Kafka-Bibliotheken unterstützen von Haus aus meistens mehrere un-

terschiedliche Serializer-Klassen, wie zum Beispiel JSON, *Protobuf*² oder *Avro*³. Sollten wir ein anderes Datenformat benutzen, können wir auch mit überschaubarem Aufwand unsere eigenen Serializer (und Deserializer) implementieren.

Sobald unsere Keys und Values als Byte-Arrays vorliegen, kann der Partitioner entscheiden, in welche Partition die Nachricht produziert werden soll. Wir können der `produce()`-Methode natürlich auch eine feste Partition mitgeben, aber wenn wir keine handfesten Gründe dafür haben, sollten wir uns auf die mitgelieferten Partitionierungsmethoden verlassen. Wir haben bereits mehrfach besprochen, dass der Partitioner anhand des Hash-Wertes des Keys eine Partition bestimmt oder, wenn kein Key vorhanden ist, einen Round-Robin-Mechanismus benutzt.

Für jede Partition und für jedes Topic, in welches wir produzieren möchten, gibt es einen Buffer, in welchen der Partitioner die Nachrichten schreibt. Standardmäßig ist der gesamte Buffer 32 MiB groß (`buffer.memory`-Einstellung im Producer). Aus diesem Buffer werden dann Batches erstellt, die maximal `batch.size` (standardmäßig 16 KiB) groß sind. Obwohl der Producer standardmäßig nicht wartet, sondern die Nachrichten, die er produzieren soll, so schnell wie möglich an die Broker schickt, können wir Batching nutzen, wenn wir schneller Daten produzieren, als wir einzelne Nachrichten an die Broker senden können.

Sobald die Nachrichten im Buffer angekommen sind, können wir sie an die Broker schicken. Wir erinnern uns, dass wir Nachrichten nur an die Leader-Broker der Partitionen schicken. In vielen Fällen haben wir mehr Partitionen als Broker. Statt nun pro Batch eine Nachricht an den Broker zu schicken, erstellt der Producer aus den Partitions-Batches größere Batches für die Nachrichten, die an den gleichen Broker gehen sollen. Dann schickt der Producer diese Nachrichten ab.

3.2.1.3 ACKs und deren Auswirkungen

Was nun geschieht, hängt von den ACK-Einstellungen ab. Haben wir im Producer `acks=0` gesetzt, schickt der Producer die Nachricht ab und interessiert sich nicht weiter, was passiert. Die Nachricht wird in den meisten Fällen ankommen, aber ähnlich wie bei UDP wissen wir es nicht. In den meisten Fällen, insbesondere wenn uns die Daten wichtig sind, setzen wir `acks=all`, um sicherzustellen, dass die Daten nicht nur beim Leader ankommen (wie bei `acks=1`), sondern auch erfolgreich repliziert worden sind. Wir haben die ACKs sehr ausführlich im Kapitel „Zuverlässigkeit“ betrachtet.

Wenn wir den Producer anweisen, auf ein ACK zu warten, dann wartet dieser so lange auf das ACK, bis der abgeschickte Request zu einem Timeout führt, also standardmäßig 30 s (`request.timeout.ms`). Falls in dieser Zeit kein ACK kommt oder wir einen Fehler vom Broker bekommen, nimmt der Producer an, dass die Zustellung nicht erfolgreich war. Dann versucht unser Producer, die Nachricht erneut zu senden.

²⁾ Google Protocol Buffers <https://developers.google.com/protocol-buffers/>

³⁾ Apache Avro <http://avro.apache.org/>

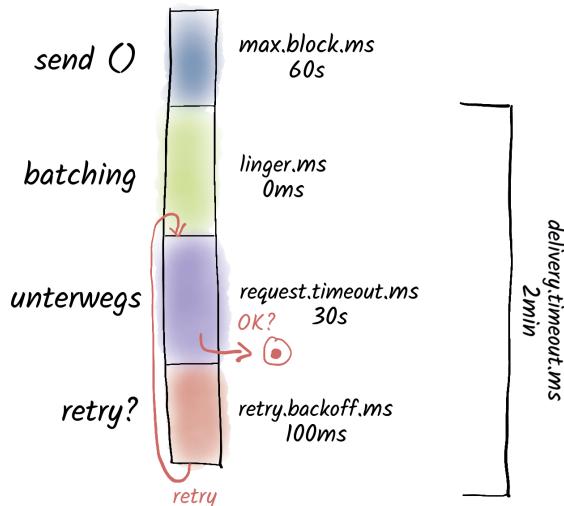


Bild 3.4 Wir können im Producer zahlreiche Timeouts definieren. Die wichtigsten sind die `linger.ms`, mit der wir das Batching konfigurieren können, und das `delivery.timeout.ms`, mit dem wir konfigurieren, wann die Kafka-Bibliothek aufgibt, Nachrichten erneut zuzustellen, und einen Fehler wirft.

Schauen wir uns die Timeouts genauer an. Die `produce()`- (oder `send()`-)Methode ist dafür zuständig, die Nachrichten in den Buffer zu schreiben. Ist der Buffer voll, blockiert die Methode maximal `max.block.ms` (also standardmäßig 60 s). Der Batching-Mechanismus wartet `linger.ms` Zeit (0 s) auf weitere Nachrichten für den Batch und schickt den Batch danach ab. Auf eine Antwort warten wir dann `request.timeout.ms`. Sollten wir keine Antwort in dieser Zeit oder einen Fehler erhalten, warten wir `retry.backoff.ms` (100 ms), bis wir es erneut versuchen. Nach maximal `delivery.timeout.ms` (2 min) brechen wir den Versuch ab, die Nachrichten zu senden, und die Bibliothek wirft eine Exception, mit der wir dann umgehen müssen.

Was unternehmen wir aber, wenn wir eine solche Exception bekommen? Wir sollten die Nachrichten nicht blind noch einmal senden, da wahrscheinlich unser Kafka-Cluster in einem schlechten Zustand ist. Den `retry`-Mechanismus sollten wir der Kafka-Bibliothek überlassen, da sie sich zum Beispiel auch um die korrekte Reihenfolge der Nachrichten kümmert, wenn wir `enable.idempotence` aktiviert haben. Im Fall einer Exception sollten wir zum Beispiel einen Circuit Breaker⁴ oder ein anderes Entwurfsmuster zum Auffangen dieses Fehlerfalls einsetzen.

3.2.2 Broker

Wir haben uns im letzten Abschnitt genauer angesehen, wie unsere Kafka-Producer-Bibliothek Nachrichten verschickt und mit Fehlern umgeht. Kafka lagert viel Arbeit an die Clients aus, damit unsere Broker so wenig wie möglich selbst erledigen müssen. Im Fall, dass wir

⁴⁾ <https://www.martinfowler.com/bliki/CircuitBreaker.html> Martin Fowlers Webseite ist eine gute Quelle für Entwurfsmuster beim Design von verteilten Systemen.

Nachrichten produzieren, muss unser Leader die Nachrichten korrekt entgegennehmen, eventuell überprüfen, ob wir überhaupt dazu autorisiert sind, in die Partitionen zu schreiben, und im Anschluss die Nachrichten so schnell wie möglich auf Platte zu schreiben. Dann müssen diese Nachrichten auf die Follower verteilt werden, aber dafür sind zum Glück die Follower selbst zuständig. Wenn wir unseren Producer so konfiguriert haben, dass uns die Zuverlässigkeit der Nachrichten wichtig ist, schickt unser Leader am Ende ein ACK zurück an den Producer.

3.2.2.1 Empfang und Persistierung von Nachrichten

Obwohl dies nicht nach viel Aufwand klingt, sind die Kafka-Broker dennoch sehr komplexe Systeme, um diese Aufgaben so performant und zuverlässig wie möglich zu erfüllen.

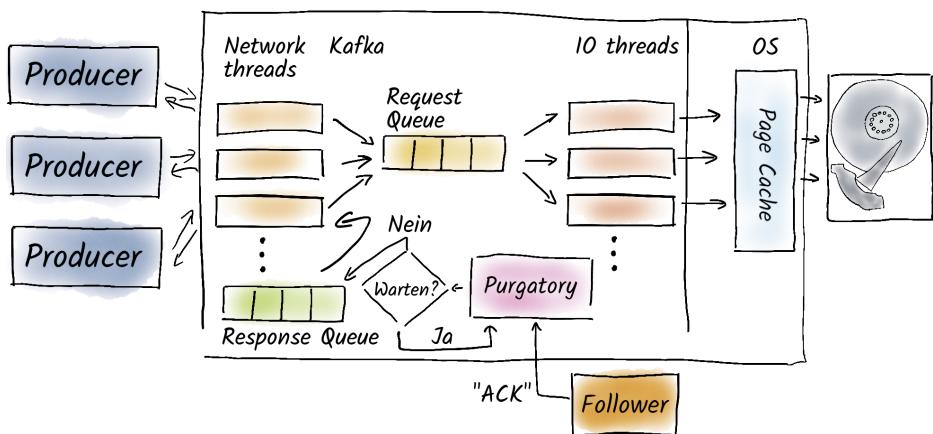


Bild 3.5 Wenn ein Broker einen Produce-Request erhält, muss der Broker abstrakt betrachtet die erhaltenen Daten lediglich in den Page Cache des Betriebssystems schreiben und, falls der Producer um ACKs bittet, möglicherweise auf die Replikation der Nachrichten durch die Follower warten und ein ACK an den Producer senden. Die Queues entkoppeln die Systeme im Broker voneinander, und die Threads sind für die parallele Verarbeitung von Requests zuständig. Der Broker committed die Nachrichten nicht selbstständig auf die Festplatte, sondern überlässt dies dem Betriebssystem.

Als Erstes kümmern sich die Network Threads darum, die Nachrichten der Producer entgegenzunehmen, um diese nach erfolgreicher Autorisierung auf die Request-Queue zu schreiben. Wenn unsere IO-Threads nicht überlastet sind, dann verbleiben die Nachrichten nur sehr kurz in der Request-Queue und werden dann von den IO-Threads an die richtige Stelle im Dateisystem, also an das Ende des aktuellen Log-Segments für die entsprechende Partition, geschrieben. Kafka kümmert sich nicht darum, dass diese Nachrichten auch erfolgreich auf Platte persistiert werden, sondern das ist Aufgabe des Betriebssystems.



Tipp

Wir möchten hier, soweit es geht, blockierende Synchronisation vermeiden. Stattdessen soll das Betriebssystem im Hintergrund die Daten selbstständig aus dem Page Cache auf die Platte schreiben.

Übrigens ist die Tatsache, dass Kafka sich überhaupt nicht um die Persistierung von Nachrichten auf der Festplatte kümmert, nicht zu 100% korrekt. Wir haben in Kafka nämlich durchaus die Möglichkeit, das Betriebssystem in dieser Hinsicht zu beeinflussen. Hierfür gibt es in der Konfiguration zwei Einstellungsmöglichkeiten. Zum einen können wir mit `flush.ms` angeben, nach wie vielen Millisekunden Kafka einen manuellen `fsync` auslösen soll, und zum anderen können wir mit `flush.messages` angeben, nach wie vielen Nachrichten ein `fsync` ausgelöst wird. Standardmäßig sind beide Einstellungen auf die größtmögliche Zahl gesetzt. Wenn wir unser Cluster also lang genug betreiben würden, würde Kafka einen `fsync` auslösen. Im Falle von `flush.ms` wären dies zum Beispiel nur läppische 300 Millionen Jahre! Auch wenn wir mit diesen beiden Konfigurationsmöglichkeiten theoretisch die Ausfallsicherheit unseres Clusters verbessern könnten, raten wir an dieser Stelle dringend davon ab, da dies mit erheblichen Performanceverlusten einhergehen würde. Außerdem erreichen wir in Kafka die Ausfallsicherheit durch Replikation.

Die Performance-Optimierung, dass wir Daten nicht selbstständig auf Platte committen, kommt demzufolge mit dem Preis, dass wir uns Gedanken machen müssen, wo und wie wir die Broker deployen. Wenn wir alle Replicas einer Partition zum Beispiel auf dem gleichen VM-Host deployen und dieser Host dann abstürzt, dann verlieren wir definitiv Daten. Wir sollten unsere Broker deshalb möglichst gleichmäßig auf die Systeme verteilen, die uns zur Verfügung stehen.

3.2.2.2 Broker und ACKs

Sobald wir die Nachrichten im Dateisystem abgelegt haben, muss unser Broker schauen, ob wir dem Producer überhaupt antworten müssen und, wenn ja, ob wir noch auf andere Broker warten müssen, damit sie den Erhalt der Nachricht bestätigen. Für das Warten nutzen wir die sogenannte *Purgatory*. Dies ist ein Cache-Mechanismus, in dem wir Responses an die Clients aufheben, wo wir noch auf weitere Informationen warten. Im Fall des Produzirens warten wir auf die Follower, damit sie den Erhalt der Nachrichten bestätigen. Sobald wir die Bestätigung der Follower erhalten haben, schreiben wir die Response in die Response-Queue, und Network Threads schicken ein ACK an den Producer.

3.2.2.3 Optimierung

Wir können alle Komponenten in dem Schaubild optimieren, konfigurieren und auch überwachen. Für viele Anwendungsfälle sind die Standardeinstellungen ausreichend, und wir müssen sie nicht anpassen. Falls wir zum Beispiel den Datenverkehr per TLS absichern wollen, sollten wir die Anzahl der Network Threads (`num.network.threads`) von drei auf sechs verdoppeln. Auch die Anzahl der IO-Threads sollten wir anpassen, wenn wir viele Festplatten einsetzen. Standardmäßig haben wir acht IO-Threads, und wenn wir mehr Festplatten haben, sollte `num.io.threads` der Anzahl der benutzten Festplatten entsprechen. Wenn wir sehr viele Producer und Consumer haben, ist es ratsam, die maximale Anzahl von Requests in der Request-Queue von 500 (`queued.max.requests`) zu erhöhen, sodass sie der Anzahl der Clients, die sich mit dem Broker verbinden, entspricht. Wir sehen hier, dass wir die Konfigurationen nur anpassen, wenn wir große Datenmengen in Produktionsumgebungen mit Kafka verarbeiten möchten. In solchen Fällen raten wir, sich professionelle Unterstützung zu holen, da wir in diesem Buch nicht alle Anwendungsfälle abdecken können.

3.2.3 Daten- & Dateistrukturen

Nachdem wir uns in den letzten beiden Abschnitten ausführlich damit beschäftigt haben, wie Nachrichten produziert und anschließend vom Broker verarbeitet werden, widmen wir uns in diesem Abschnitt ganz den Daten- und Dateistrukturen von Kafka. Wir werden uns genauer ansehen, welche verschiedenen Daten Kafka speichert und wie diese Daten in unseren Brokern strukturiert sind.

3.2.3.1 Metadaten, Checkpoints und Topics

Beim Aufsetzen unserer Testumgebung haben wir für unsere drei Broker jeweils ein Verzeichnis unter `~/kafka/data/kafka<Broker id>` angelegt. Genau dort speichern unsere Broker auch alle ihre Daten. Wir haben zur besseren Übersicht unser Kafka-Cluster für dieses Kapitel komplett neu aufgesetzt, indem wir einfach die entsprechenden Verzeichnisse gelöscht und neu erstellt haben. Werfen wir nun zunächst mit `ls` einen Blick in das für unseren Broker mit der ID 1 entsprechende Verzeichnis:

```
$ ls -1 ~/kafka/data/kafka1/
cleaner-offset-checkpoint
log-start-offset-checkpoint
meta.properties
recovery-point-offset-checkpoint
replication-offset-checkpoint
```

In unserem frischen Kafka-Cluster finden sich hier insgesamt fünf Dateien. Die Datei `meta.properties` enthält die ID unseres Brokers, die ID unseres Kafka-Clusters und die Metadaten Version, sodass Kafka weiß, wie die entsprechenden Dateien zu interpretieren sind. Alle anderen Dateien enthalten jeweils nur zwei Nullen. Die erste Null steht hier für die Metadaten-Version, und die zweite Nummer gibt an, wie viele Zeilen mit eigentlichen Informationen folgen. Da unser Cluster frisch aufgesetzt ist und wir keinerlei Topics haben, ist dieser Wert logischerweise überall null. Bevor wir dazu kommen, welche Informationen genau in diesen Checkpoint-Dateien gespeichert werden, erstellen wir zunächst ein Topic namens `datei-struktur`. Wir wählen hierfür unseren standardmäßigen *Replication Factor* von 3 und verteilen das Topic auf drei Partitionen:

```
$ kafka-topics.sh \
--create \
--topic datei-struktur \
--partitions 3 \
--replication-factor 3 \
--bootstrap-server localhost:9092
Created topic datei-struktur.
```

Nachdem wir das Topic erstellt haben, schauen wir uns den Inhalt der Datei `replication-offset-checkpoint` etwas genauer an:

```
$ less ~/kafka/data/kafka1/replication-offset-checkpoint
0
3
datei-struktur 0 0
datei-struktur 2 0
datei-struktur 1 0
```

Die Metadaten-Version ist immer noch 0 (Zeile 1 unserer Datei), allerdings enthält unsere Datei jetzt drei Zeilen mit Informationen (Zeile 2 in unserer Datei). Die weiteren Zeilen unserer Datei beginnen jeweils mit dem Namen unseres eben erstellten Topics. Die zweite Spalte bezieht sich hierbei auf die Partition, und die dritte Spalte enthält einen Offset. Aber was bedeuten diese Offsets? Ganz einfach, sie verweisen auf die Position im Log der jeweiligen Partition, bis zu der unsere Nachrichten bereits erfolgreich auf unsere Follower-Replicas repliziert wurden. Was genau dies wiederum bedeutet, werden wir uns im nächsten Abschnitt dieses Kapitels genauer ansehen. Da wir noch keine Nachrichten produziert haben, ist sowohl die aktuelle Position in unserem Log 0 als auch logischerweise die Position im Log, bis zu der repliziert wurde.

Die Datei `recovery-offset-checkpoint` ist ähnlich aufgebaut, nur beziehen sich hier die Offsets auf die Position im Log, bis zu der unsere Nachrichten erfolgreichpersistiert worden sind, das heißt, vom Betriebssystem aus dem Hauptspeicher auf die Festplatte geschrieben worden sind. Die Datei `log-start-offset-checkpoint` enthält Informationen zum Offset der ersten Nachricht in unserem Log beziehungsweise zu unseren Partitionen, denn die erste Nachricht in unserem Log muss keinesfalls den Offset 0 haben. Dies hängt damit zusammen, dass Kafka kontinuierlich unser Log aufräumen kann und so zum Beispiel alte und nicht mehr benötigte Nachrichten automatisch löscht, da ansonsten unser Log bis ins Unendliche wachsen würde. Wie genau Kafka unser Log aufräumt und welche Möglichkeiten wir hierbei haben, behandeln wir ausführlich im Kapitel „Nachrichten aufräumen“. Die letzte noch verbliebene Datei `cleaner-offset-checkpoint` enthält die Offsets unserer Partitionen, bis zu denen der Kafka-Log Cleaner Nachrichten compactet hat. Was genau das wiederum bedeutet, werden wir ebenfalls im Kapitel „Nachrichten aufräumen“ lernen.

3.2.3.2 Partitionen

Mit der Erstellung unseres `datei-struktur`-Topics wurden allerdings nicht nur Metadaten für dieses Topic in unsere bestehenden Dateien hinzugefügt, sondern es wurde auch für jede Partition ein Unterverzeichnis im Datenverzeichnis unseres Brokers erstellt (`datei-struktur-0`, `datei-struktur-1`, `datei-struktur-2`):

```
$ ls -1 ~/kafka/data/kafka1/
cleaner-offset-checkpoint
datei-struktur-0/
datei-struktur-1/
datei-struktur-2/
log-start-offset-checkpoint
meta.properties
recovery-point-offset-checkpoint
replication-offset-checkpoint
```

Genau genommen wird nicht für jede Partition in jedem Broker eine Datei erstellt, sondern lediglich für jede Partition, die unserem Broker zugewiesen wurde. Da wir einen *Replication Factor* von 3 haben und unser Cluster aus insgesamt drei Brokern besteht, wurde in jedem unserer Broker beziehungsweise in jedem unserer Broker-Dateipfade ein Unterverzeichnis für alle drei Partitionen unseres `datei-struktur`-Topics erstellt. Werfen wir nun einen Blick in die Dateien unserer Partition 0:

```
$ ls -1 ~/kafka/data/kafka1/data-struktur-0
00000000000000000000.index
00000000000000000000.log
00000000000000000000.timeindex
leader-epoch-checkpoint
```

Insgesamt befinden sich vier Dateien in unserer Partition. Die Datei `leader-epoch-checkpoint` ist ähnlich aufgebaut wie unsere anderen Checkpoint-Dateien. Sie enthält in der ersten Zeile die Metadata-Version, und in Zeile 2 folgt die Anzahl an Zeilen mit Informationen. Je nachdem, ob der Broker, in dessen Dateiverzeichnis wir uns gerade befinden, der Leader der Partition ist, sieht die Datei etwas anders aus. Schauen wir uns daher zunächst kurz an, welcher Broker Leader von welcher Partition ist:

```
$ kafka-topics.sh --describe \
--topic datei-struktur \
--bootstrap-server=localhost:9092
Topic: datei-struktur PartitionCount: 3
ReplicationFactor: 3 Configs:
Topic: datei-struktur Partition: 0
Leader: 1 Replicas: 1,3,2 Isr: 1,3,2
Topic: datei-struktur Partition: 1
Leader: 2 Replicas: 2,1,3 Isr: 2,1,3
Topic: datei-struktur Partition: 2
Leader: 3 Replicas: 3,2,1 Isr: 3,2,1
```

Broker 1 ist Leader von Partition 0, daher schauen wir uns die Datei `leader-epoch-checkpoint` in Partition 0 von Broker 1 an:

```
$ less \
~/kafka/data/kafka1/datei-struktur-0/leader-epoch-checkpoint
0
1
0 0
```

Die Informationen dieser Datei sind lediglich zwei Nullen. Die erste Null steht hierfür für die Anzahl vorheriger Partitions-Leader. Die zweite Null steht für den zum Zeitpunkt der Leader-Election aktuellen Log-Offset. Die anderen Partitionen in Broker 1 enthalten an dieser Stelle lediglich zwei Nullen. Die Leader Epoch wird bei jedem Leader-Wechsel hochgezählt und dient dazu sicherzustellen, dass nicht aus Versehen zwei Broker die Leader-Rolle für eine Partition beanspruchen. Unter welchen Umständen dies genau passieren könnte und wie die Leader Epoch dies verhindert, schauen wir uns anhand des folgenden Beispiels an. Mit dem `kafka-broker-stop.sh`-Skript aus dem Anhang stoppen wir zunächst unseren Broker mit der ID 3. Dieser ist Leader von Partition 2. Anschließend schauen wir uns wieder unser `leader-epoch-checkpoint` an:

```
$ kafka-broker-stop.sh kafka3
$ less \
~/kafka/data/kafka1/datei-struktur-0/leader-epoch-checkpoint
0
1
1 0
```

Kafka stellt fest, dass einer der Broker nicht mehr erreichbar ist, und startet eine *Leader-Election*, um einen neuen Leader für Partition 2 zu bestimmen. Die Leader-Epoch wird in jeder Partition um 1 inkrementiert. Mit `kafka-topics.sh --describe` könnten wir uns

anschauen, welcher Broker die Leader-Rolle für Partition 2 übernommen hat, für unser Beispiel spielt dies aber keine große Rolle.

Starten wir nun unseren Broker mit der ID 3 wieder. Genau an dieser Stelle wird unsere Leader-Epoch wichtig. Die Leader-Epoch stellt sicher, dass nur der aktuelle Leader einer Partition Nachrichten verteilen kann. Stellen wir uns zum Beispiel vor, dass Broker 3 gar nicht bemerkt hat, dass er nicht erreichbar war. Er würde dann immer noch denken, dass er Leader von Partition 2 ist. Beim Replizieren der Nachrichten sendet ein Broker auch immer die aktuelle Leader Epoch. Sendet ein Broker eine veraltete Leader Epoch, so ignorieren die anderen Broker seine Nachrichten. Da aus Sicht von Broker 3 die Leader Epoch immer noch 0 ist, werden seine Nachrichten abgewiesen, da die aktuelle, tatsächliche Leader Epoch den Wert 1 hat. Broker 3 ist jetzt natürlich nicht dazu verdammt, auf ewig Nachrichten zu senden, welche abgelehnt werden. Nach kurzer Zeit bekommt er die aktuellen Partitions Leader und Leader-Epochen mitgeteilt, außerdem wird Broker 3 nach einiger Zeit selbst wieder zum Leader von Partition 2, da Kafka standardmäßig alle fünf Minuten versucht, die präferierten Leader wieder als tatsächliche Leader einzusetzen. Werfen wir zum Abschluss dieses Beispiels einen erneuten Blick auf unseren `leader-epoch-checkpoint`:

```
$ kafka-server-start.sh ~/kafka/config/kafka3.properties
$ less \
~/kafka/data/kafka3/datei-struktur-2/leader-epoch-checkpoint
0
1
2 0
```

Für Partition 2 hat sich der Wert erneut inkrementiert, da Broker 3 die Leader-Rolle wieder von Broker 2 übernommen hat. Für Partition 1 und 2 haben sich die Werte übrigens nicht verändert, da es weder einen neuen Leader gibt, noch eine neue Leader-Election durchgeführt wurde.

Produzieren wir nun zunächst ein paar Nachrichten in unser `datei-struktur`-Topic, bevor wir uns die restlichen Dateien im Partitionsverzeichnis ansehen:

```
$ kafka-console-producer.sh \
--topic datei-struktur \
--bootstrap-server localhost:9092
> M1
> M2
...
> M9
```

Werfen wir zunächst aber kurz einen erneuten Blick auf unsere Replikations-Offsets:

```
$ less ~/kafka/data/kafka1/replication-offset-checkpoint
0
3
datei-struktur 0 3
datei-struktur 2 5
datei-struktur 1 1
```

Wie wir sehen können, wurden die Nachrichten bereits erfolgreich repliziert. Dass die Nachrichten nicht gleichmäßig auf die Partitionen verteilt worden sind, hängt damit zusammen, dass Nachrichten vom Producer in Batches versendet werden, und je nachdem, wie schnell wir beim Tippen unserer Nachrichten waren, sind so mehrere Nachrichten in einem

Batch und somit einer Partition gelandet, wodurch es ein kleines Ungleichgewicht bezüglich der Lastverteilung gibt.

3.2.3.3 Log-Dateien und -Indizes

Wir wissen nun, dass unsere Daten repliziert worden sind, aber wo genau und wie werden sie eigentlich gespeichert? Hier kommen unsere restlichen Dateien im Partitionsverzeichnis ins Spiel. Die Datei mit der Endung `.log` enthält hierbei wenig überraschend unsere Nachrichten. Mit dem `kafka-dump-log.sh`-Skript liefert uns Kafka auch direkt ein Werkzeug, mit dem wir einen genaueren Blick in unser Log werfen können. Hierzu müssen wir lediglich noch über das Argument `--files` das zu inspizierende Log-Segment übergeben. Schauen wir uns nun die Datei `00000000000000000000000000000000.log` in unserer Partition 2 des `datei-struktur`-Topics an. Der Grund, warum wir diese Partition ausgewählt haben, ist, dass dort die meisten Nachrichten gelandet sind und wir somit am meisten sehen können.

```
$ kafka-dump-log.sh \
--files ~/kafka/data/kafka1/datei-struktur-2/0[....]000.log
Dumping ~/kafka/data/kafka1/datei-struktur-2/0[....]000.log
Starting offset: 0
baseOffset: 0 lastOffset: 1 count: 2 [...] position: 0 CreateTime: 1617493014867 size:
80 [...] crc: 1926995389 baseOffset: 2 lastOffset: 3 count: 2 [...] position: 80
CreateTime: 1617493017460 size: 80 [...] crc: 2995813756
baseOffset: 4 lastOffset: 4 count: 1 [...] position: 160
CreateTime: 1617493021932 size: 70 [...] crc: 1450778056
```

Die Ausgabe unseres Befehls zeigt uns einige Metadaten zu unserem Logfile an. Zur besseren Übersicht haben wir die Ausgabe dieses Befehls etwas gekürzt und beschränken uns auf die Erläuterung der für uns an dieser Stelle interessantesten Daten. Wir sehen, mit welchem Offset das Log startet (`Starting offset`), und anschließend erhalten wir eine Übersicht über die einzelnen Batches. Wir können sehen, welchen Offset die erste Nachricht und die letzte Nachricht des jeweiligen Batches haben (`baseOffset` und `Last Offset`) und wie viele Nachrichten der jeweilige Batch insgesamt enthält (`count`). Wir sehen ebenfalls den Timestamp (`CreateTime`) und die Checksumme des Batches (`crc`). Die Größe des Batches in Byte verbirgt sich hinter dem Attribut `size`, und `position` gibt an, an welcher Byte-Position im Segment der jeweilige Batch beginnt. Wir sehen ebenfalls, dass ein Batch bestehend aus zwei Nachrichten eine Größe von 80 Byte besitzt, und ein Batch mit nur einer Nachricht mit 70 Byte gar nicht so viel kleiner ist. Dies liegt darin, dass ein Batch auch einige Metadaten enthält, insgesamt 60 Byte, was bei der Fülle an Informationen, die wir durch `kafka-dump-log.sh` erhalten, auch nicht verwunderlich ist. Hierbei sehen wir auch, wie durch Batches die Performance von Kafka, insbesondere bei vielen kleinen Nachrichten, erheblich verbessert wird!

Der Payload der Nachrichten selbst ist jeweils sogar nur zwei Bytes groß (wir haben jeweils nur zwei Zeichen gesendet), denn die Nachrichten in den einzelnen Batches enthalten wiederum weitere Metadaten. Wenn wir unserem `kafka-dump-log.sh` zusätzlich das Argument `--print-data-log` übergeben, enthalten wir sämtliche Informationen zu unseren einzelnen Nachrichten, inklusive der eigentlichen Nachricht! Die Ausgabe sparen wir uns an dieser Stelle allerdings.

Die Datei mit der Endung `.index` dient dazu, unsere Nachrichten im Log schneller zu finden, denn Kafka speichert alle Nachrichten beziehungsweise Batches einfach nacheinander

im Log ab. Ohne den Index müssten wir jedes Mal, wenn ein Consumer Nachrichten konsumieren möchte, das gesamte Segment dekodieren und dann nach dem entsprechenden Offset suchen. Da dies nicht sehr effizient wäre, speichert Kafka stattdessen die zu den jeweiligen Offsets zugehörige Byte-Position im Log. Dies geschieht allerdings auch nicht für jeden Offset, sondern nur, falls seit dem letzten Eintrag im Index mindestens 4096 weitere Bytes in unserem Log hinzugefügt worden sind. Dieser Wert kann auch in der Topic-Konfiguration durch die Einstellung `index.interval.bytes` angepasst werden. Alternativ können wir den Standardwert für unser Kafka-Cluster auch über die Konfiguration unserer Broker durch die Einstellung `log.index.interval.bytes` anpassen. Wenn wir diesen Wert verringern, wären wir zwar in der Lage, dichter an die gewünschte Position im Log zu springen, aber der Index würde schneller wachsen. Der Standardwert von 4 KiB bietet uns an dieser Stelle in den allermeisten Fällen einen sehr guten Trade-off. Den Inhalt der Indexdatei können wir uns ebenfalls mit `kafka-dump-log.sh` anzeigen lassen.

Die Datei mit der Endung `.timeindex` hat eine ähnliche Aufgabe und Funktionsweise wie unsere Indexdatei, nur findet hier anstelle des Matchings zwischen Offset und Byte-Position ein Matching zwischen Timestamp und Offset statt. Dadurch können wir einfach Nachrichten eines bestimmten Tages konsumieren. Ein häufiger Anwendungsfall hierfür ist zum Beispiel das Zurücksetzen der Consumer-Offsets auf den Anfang des Tages und damit das erneute Konsumieren aller Nachrichten dieses Tages.

3.2.3.4 Segmente

Wir haben im Verlauf dieses Kapitels schon mehrfach von Segmenten gesprochen. Wir wissen bereits, dass Nachrichten in Kafka-Topics zugewiesen werden und dass Topics wiederum in Partitionen unterteilt sind. Partitionen sind wiederum in Segmente unterteilt, welche jeweils aus den drei Log- und Indexdateien, die wir gerade kennengelernt haben, bestehen. Der Grund dieser Unterteilung ist, dass Partitionen mit der Zeit natürlich wachsen und manchmal Dutzende von Gigabyte oder sogar Terabyte an Daten enthalten können. Eine einzelne Log- oder Index-Datei wären dann schnell sehr ineffizient. Dies bezieht sich jedoch weniger auf das Konsumieren von Nachrichten als vielmehr darauf, wie Nachrichten in Kafka aufgeräumt werden. Dies werden wir uns im Kapitel 3.4 „Nachrichten aufräumen“ noch genauer ansehen. An dieser Stelle bietet es sich außerdem an, dass wir einen genaueren Blick auf die Dateinamen der Log- und Index-Dateien werfen, denn der Dateiname ist nichts anderes als der erste Offset im jeweiligen Segment. Wann genau erstellt Kafka nun aber ein neues Segment? Hierfür gibt es zwei wichtige Kenngrößen. Kafka erstellt ein neues Segment, wenn entweder das Segment eine bestimmte Größe überschritten (standardmäßig 1 GiB) oder ein gewisses Alter erreicht hat (standardmäßig sieben Tage). Wir können diese Einstellungen wieder über die Topic-Konfiguration anpassen. Mit `segment.ms` geben wir die Zeit in Millisekunden an, nach der ein neues Segment erstellt wird. Die maximale Segmentgröße in Byte können wir durch `segment.bytes` beeinflussen. Auch hier haben wir natürlich wieder die Möglichkeit, die entsprechenden Standardeinstellungen für Topics in unserer Broker-Konfiguration anzupassen (`log.segment.bytes` beziehungsweise `log.roll.ms`).

Testen wir dies doch direkt aus, indem wir das maximale Alter eines Segmentes auf 60 s verringern. Hierfür können wir mit dem `kafka-configs.sh`-Skript die Config unseres `datei-struktur` Topics entsprechend anpassen:

```
$ kafka-configs.sh \
  --alter \
  --topic datei-struktur \
  --add-config segment.ms=60000 \
  --bootstrap-server=localhost:9092
```

Nachdem wir unsere Topic-Konfiguration angepasst haben, müssen wir noch ein paar Nachrichten produzieren, damit Kafka ein neues Segment erstellt. Hierfür benutzen wir wieder unseren `kafka-console-producer.sh`:

```
$ kafka-console-producer.sh \
  --topic datei-struktur \
  --bootstrap-server localhost:9092
> M10
> M11
...
> M16
```

Kafka sollte nun ein neues Segment erstellt haben. Überprüfen wir das Ganze doch einfach, indem wir einen Blick auf die Dateien in unserer Partition werfen:

```
$ ls -1 ~/kafka/data/kafka1/data-struktur-
00000000000000000000.index
00000000000000000000.log
00000000000000000000.timeindex
00000000000000000003.index
00000000000000000003.log
00000000000000000003.snapshot
00000000000000000003.timeindex
leader-epoch-checkpoint
```

Wie erwartet befinden sich nun weitere Log- und Index-Dateien in unserer Partition. Allerdings ist noch eine weitere Datei mit der Endung `.snapshot` hinzugekommen. Diese Datei bezieht sich in unserem Beispiel lediglich auf den Start-Offset des aktuellen Segments. Abgesehen davon wird diese Datei zur Verwaltung von Transaktionen in Apache Kafka verwendet. Transaktionen werden ausführlich im Kapitel „Verarbeitungsgarantien und Transaktionen“ behandelt.

3.2.3.5 Gelöschte Topics

Zum Abschluss dieses Kapitels löschen wir unser Topic wieder, allerdings nicht um aufzuräumen, sondern um zu sehen, wie Kafka Topics löscht. Hierfür nutzen wir wieder das `kafka-topics.sh` Skript:

```
$ kafka-topics.sh \
  --delete \
  --topic "flugdaten" \
  --bootstrap-server=localhost:9092
```

Kafka löscht Dateien allerdings nicht direkt, sondern markiert sie zunächst zum Löschen. Schauen wir uns das doch direkt an:

```
$ ls -1 ~/kafka/data/kafka1/
cleaner-offset-checkpoint
datei-struktur-0.<UniqueID>-delete/
datei-struktur-1.<UniqueID>-delete/
```

```
datei-struktur-2.<UniqueID>-delete/
log-start-offset-checkpoint
meta.properties
recovery-point-offset-checkpoint
replication-offset-checkpoint
```

Kafka ändert dabei einfach den Namen der Partitionenverzeichnisse, indem es den ursprünglichen Verzeichnissen eine spezielle Erweiterung in der Form .<UniqueID>-delete hinzufügt. Nach einer Minute werden die Verzeichnisse dann endgültig gelöscht. In diesem Zeitraum könnten wir theoretisch das Löschen unseres Topics rückgängig machen und so unsere Daten retten, falls wir versehentlich ein falsches Topic gelöscht haben. Wir haben auch hier wieder die Möglichkeit, diese Zeitspanne anzupassen. Wir können hierfür entweder die Topic-Konfiguration über den Parameter file.delete.delay.ms anpassen oder die Einstellung global für unser gesamtes Cluster über die Broker-Konfiguration log.segment.delete.delay.ms ändern.

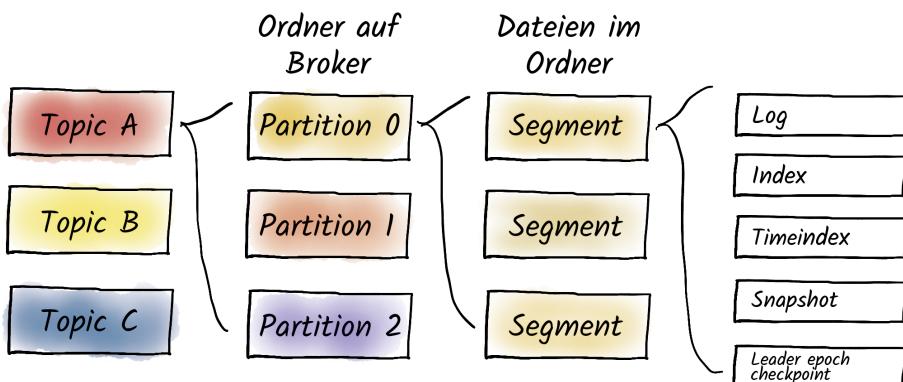


Bild 3.6 Üblicherweise haben wir zahlreiche Topics in unserem Kafka-Cluster. Jedes Topic besteht aus einer oder mehreren Partitionen. Für jede Partition, für die ein Broker zuständig ist, existiert ein Verzeichnis. Wir teilen Partitionen (ähnlich zu Log-Rotation) in mehrere Segmente auf, und jedes Segment besteht wiederum aus mehreren Dateien.

3.2.4 Replikation

Nachdem wir in den letzten Abschnitten ausführlich die Datei- und Ordnerstrukturen von Kafka ergründet haben, werden wir uns in diesem Abschnitt umfassend mit Replikation und insbesondere mit In-Sync-Replicas (ISR) beschäftigen.

Wir replizieren Daten in Kafka, indem Follower in regelmäßigen Abständen vom Leader fetchen, also Daten abholen. Der Leader weiß so, welcher Follower zu welchem Zeitpunkt Daten abgeholt hat. Wir wissen bereits, dass wir bei acks=all warten, bis alle ISR diese Nachricht abgeholt haben. Aber warum warten wir nicht einfach, bis alle Follower die Nachricht abgeholt haben? Wir möchten ein Ausbremsen des gesamten Clusters vermeiden, wenn ein Broker gerade langsam oder nicht erreichbar ist. Dafür definieren wir die sogenannten ISR.

3.2.4.1 In-Sync-Replicas

Ein Follower ist in-sync, wenn er innerhalb der letzten 30 s alle Nachrichten vom Leader abgeholt hat. Diese Zeitspanne können wir mittels `replica.lag.time.max.ms` in der Broker Config anpassen. Leader sind immer in-sync. Standardmäßig fragt ein Replica alle 500 ms den Leader nach neuen Nachrichten. Wir können diese Zeit ebenfalls anpassen, indem wir im Follower-Broker den Wert für `replica.fetch.wait.max.ms` anpassen. Hat es ein Replica in der maximalen Lag-Zeit nicht geschafft, mindestens einmal alle Nachrichten bis zum sogenannten *Log End Offset* (LEO) des Leaders zu konsumieren, wird es vom Leader aus der Liste der ISR der Partition entfernt. Der LEO markiert auf jeder Partition und jedem Replica die Position der zuletzt erhaltenen Nachricht und zeigt damit auf das Ende des entsprechenden Logs. Dies dient dazu, Replicas, welche temporär nicht schnell genug konsumieren können, zu identifizieren und somit zu verhindern, dass ein langsamer Broker die Performance des gesamten Clusters negativ beeinflusst. Dass ein Replica nicht schnell genug konsumieren kann, weil der zugehörige Broker kurzzeitig überlastet ist, kann in einem großen Kafka-Cluster mit Hunderten von Topics und Tausenden von Partitionen und Replicas durchaus von Zeit zu Zeit passieren. Unabhängig davon, wird ein Replica auch sofort aus der Liste der ISR entfernt, wenn es nicht mehr auf Heartbeats antwortet, weil der Broker zum Beispiel komplett ausgefallen ist.

3.2.4.2 High Watermark

Woher weiß der Leader nun aber, dass die Nachrichten auch erfolgreich beim Follower repliziert worden sind, und welche Rolle spielt dabei der LEO? Sobald ein Partitions-Leader oder Follower eine neue Nachricht erhält, passt er den dazugehörigen LEO an. Wenn ein Follower einen *fetch*-Request an den Leader sendet, enthält diese Anfrage auch den aktuellen LEO des Followers, äquivalent dazu, wie normale Consumer ihren aktuellen Offset mitteilen. Dadurch weiß der Leader, ob der Follower bereits alle Nachrichten erhalten hat oder welche Nachrichten ihm noch fehlen und an ihn gesendet werden müssen.

An dieser Stelle kommt noch ein weiterer Offset ins Spiel, und zwar die sogenannte *High Watermark* (HWM). Die HWM ist die Position im Log, bis zu der alle ISR bereits Nachrichten konsumiert haben. Es ist also der unter allen ISR kleinste an den Leader committete LEO. Außerdem gilt in Kafka auch erst dann eine Nachricht als erfolgreich committet, wenn alle ISR die Nachricht erhalten haben, also wenn das HWM größer oder gleich der Position der Nachricht im Log ist, und zwar unabhängig von der ACK-Strategie. Das aktuelle HWM wird vom Leader auch während des *fetch* an seinen Follower propagiert.

Die HWM dient in Kafka allerdings noch einem weiteren Zweck. Consumer können nur Nachrichten lesen, die aus Kafkas Sicht committet sind. Diese Grenze im Log wird eben genau durch das HWM markiert. Da die Replikation in Kafka komplett asynchron abläuft, kann es außerdem je nach Einstellung durchaus einige Sekunden dauern, bis eine neu produzierte Nachricht auch konsumiert werden kann. Im Normalzustand läuft die Kommunikation mit Kafka in Nahe-Echtzeit ab, und wir reden hier von Latenzen im Millisekunden-Bereich.

Schauen wir uns dazu kurz die Schritte an, welche beim Replizieren einer Nachricht ablaufen. Wenn ein Partitions-Leader eine neue Nachricht erhält, so erhöht sich zunächst der LEO des Leaders. Beim nächsten *fetch*-Request der Follower stellt der Leader fest, dass den Followern eine Nachricht fehlt und sendet die Nachricht an die jeweiligen Follower. Beim

nächsten *fetch*-Request stellt der Leader fest, dass alle Follower die Nachrichten erhalten haben, da der LEO der Follower dem LEO des Leaders entspricht, und der Leader inkrementiert das HWM und propagiert den neuen Wert mit der Antwort des *fetch*-Request an die Follower

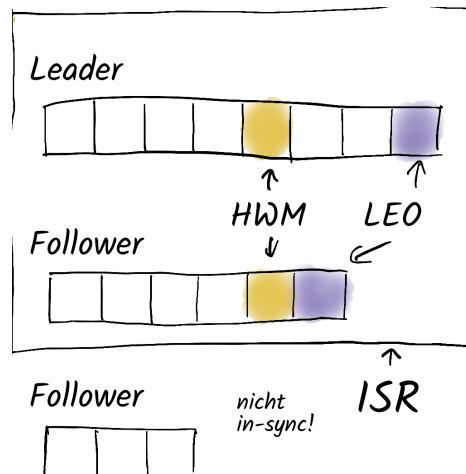


Bild 3.7 Die *High Watermark* (HWM) ist der Offset, den alle ISR repliziert und beim Leader committet haben. Der unterste Follower ist nicht in-sync und wird beim Berechnen der HWM nicht berücksichtigt.

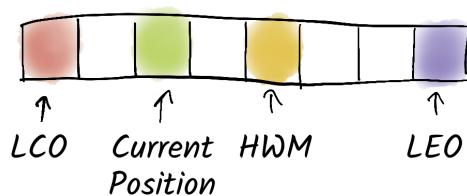


Bild 3.8 Wir betrachten hier das Log eines Leaders aus Sicht eines Followers. Um neue Daten zu erhalten, sendet der Follower einen *fetch*-Request mit der aktuellen Position (Current Position). Nun weiß der Leader, dass der Follower bis zu dieser Position repliziert hat, und kann den *Last Committed Offset* (LCO) des Followers aktualisieren. Da die HWM neuer ist als der LCO des Followers, können wir daraus schließen, dass der Follower nicht in-sync ist, da die HWM der Offset ist, den alle ISR committet haben.

Betrachten wir nun das Ganze aus der Sichtweise eines Followers. Möchte ein Follower Nachrichten lesen, so sendet er einen *fetch*-Request mit seinem aktuellen Offset im Log an den Leader der entsprechenden Partition. Der Leader sendet dann die Nachrichten an den Follower und aktualisiert den zu diesem Follower gespeicherten *Last Committed Offset* (LCO). Zu diesem Zeitpunkt spielt es für den Leader keine Rolle, ob die Nachrichten erfolgreich angekommen sind. Wenn ein Follower eine Nachricht nicht erhalten hat, sendet er einfach eine neue Anfrage an den Broker mit einem unveränderten Offset, woraufhin der Broker die Nachrichten erneut sendet. Den Brokern wird dadurch Komplexität abgenommen, da sie sich an dieser Stelle nicht um das Fehlermanagement kümmern müssen.

3.2.4.3 Auswirkungen von Verzögerungen bei der Replikation

Betrachten wir zum Abschluss dieses Abschnitts noch, welche Auswirkungen es auf das Konsumieren und Produzieren von Daten hat, wenn ein ISR anfängt hinterherzuhinken, es also nicht mehr schnell genug Nachrichten vom Leader replizieren kann.

Solange ein Replica in-sync ist, nimmt es Einfluss auf das HWM und damit darauf, welche Nachrichten konsumiert werden können. Dies hat zur Folge, wenn ein ISR anfängt hinterherzuhinken, werden zwangsweise auch alle Consumer ausgebremst beziehungsweise sind nicht mehr in der Lage, die produzierten Nachrichten auch direkt zu konsumieren. Nach 30 Sekunden, oder was wir als maximalen *Lag* konfiguriert haben, würde das Replica aus der Liste der ISR verschwinden, das HWM vermutlich einen großen Sprung machen, und die Consumer können wieder aufholen. Abgesehen davon, können wir so lange Nachrichten konsumieren, wie es einen Partitions-Leader gibt. Dies bedeutet auch, dass, falls der bisherige Leader einer Partition ausfällt, wir für einen kurzen Moment (bis ein neuer Leader bestimmt wurde) keine Nachrichten konsumieren können.

Falls wir seitens Producer als ACK-Strategie `acks=all` gewählt haben, kann es je nach weiteren Einstellungen im Producer auch zu einer Verzögerung bei der Produktion von Nachrichten durch hinterherhinkende ISR kommen, da Kafka beziehungsweise der Partition-Leader das ACK für eine empfangene Nachricht erst dann an den Producer sendet, wenn die Nachricht an alle ISR repliziert wurde. Warum also nicht einfach den maximalen Lag auf zwei Sekunden reduzieren? Das würde wiederum neue Probleme mit sich bringen. Durch das asynchrone Replizieren kann es relativ schnell zu einem temporären kleinen Lag zwischen Followern und Leader kommen, was dazu führen würde, dass wir Replicas unnötigerweise dauernd aus der Liste der ISR entfernen und kurze Zeit später wieder hinzufügen würden. Dies würde nebenbei bemerkt auch einen beachtlichen Overhead verursachen, da die ISR vom Koordinationscluster getrackt werden. Wenn damit die Anzahl unserer ISR aber unter die konfigurierte minimale Anzahl an ISR fällt, würde auch hier das Produzieren von neuen Nachrichten unterbunden werden, da nicht genügend ISR vorhanden sind. Auch wenn es an dieser Stelle verlockend erscheinen mag, die minimale Anzahl der ISR aus Gründen des Performance-Tunings nicht zu konfigurieren, raten wir dringend davon ab und empfehlen an dieser Stelle nochmals ausdrücklich, die minimale Anzahl der ISR auf einen sinnvollen Wert zu setzen! Kafka committet die erhaltenen Nachrichten nämlich nicht selbst auf das Dateisystem, sondern überlässt dies dem Betriebssystem. Falls der Leader das einzige ISR ist und dann plötzlich ausfällt, könnte dies dazu führen, dass Daten bereits konsumiert worden sind, die noch nicht im Dateisystempersistiert worden sind.



Tipp

Die minimale Anzahl der ISR sollte immer auf einen sinnvollen Wert gesetzt werden.

3.2.5 Zusammenfassung

In diesem Kapitel haben wir uns alles, was mit dem Produzieren und Persistieren von Nachrichten zusammenhängt, ausführlich angesehen. Wir haben die Prozesse im Producer

näher beleuchtet und sind dabei auch auf die Serialisierung und die Partitionierung unserer Nachrichten genauer eingegangen. Außerdem haben wir uns damit beschäftigt, wie Producer ACKs verwenden, und insbesondere, wie sie deren Timeouts oder Exceptions behandeln. Im Anschluss haben wir die Empfängerseite der Nachrichten, das heißt unsere Broker, näher betrachtet. Wir haben erfahren, wie genau Nachrichten persistiert werden und auch wie brokerseitig mit ACKs umgegangen wird. Des Weiteren haben wir uns angesehen, welche Möglichkeiten wir haben, die Performance unserer Broker zu verbessern. Außerdem haben wir einen ausführlichen Blick auf die Daten- & Dateistrukturen unserer Broker geworfen. Hierbei haben wir uns unter anderem angesehen, welche Dateien ein Broker zur Verwaltung seiner Topics beziehungsweise Partitionen anlegt und wie diese Dateien strukturiert sind. Wir haben auch erfahren, dass Kafka für jede Partition ein eigenes Verzeichnis anlegt und dass es in diesem Verzeichnis neben dem eigentlichen Log noch verschiedene andere Dateien gibt. Wir haben gelernt, wie genau das Log strukturiert ist und welche Metadaten Kafka für unsere Nachrichten speichert. Wir wissen nun auch, dass Kafka verschiedene Indexdateien zur Performanceoptimierung anlegt und wie wir die Häufigkeit der Indizierung beeinflussen können. Wir haben auch erfahren, dass Partitionen in Segmente unterteilt werden und dass diese Unterteilung besonders wichtig für das Aufräumen unseres Logs ist. Außerdem haben wir noch gelernt, dass Kafka ein Topic nicht sofort löscht, sondern es zunächst nur zum Löschen markiert. Anschließend haben wir uns noch ausführlich mit der Replikation in Kafka beschäftigt. Hierbei haben wir erfahren, welche Bedeutung ISR in Kafka haben und wie genau Kafka definiert, ob ein Replica in-sync ist. In diesem Zusammenhang haben wir auch die Bedeutung von ISR für das Produzieren und auch Konsumieren von Nachrichten erörtert. Zum Abschluss haben wir uns noch angesehen, welche Folgen es für unsere Kafka-Cluster hat, wenn ein ISR anfängt hinterherzuinken, und auch welche Konfigurationsmöglichkeiten wir in diesem Kontext haben.

■ 3.3 Nachrichten konsumieren

Nehmen wir also an, dass unsere Nachrichten erfolgreich vom Producer an den Kafka-Cluster geschickt und dort erfolgreich auf alle Follower verteilt wurden. Nun ist die Frage, wie wir eigentlich die Nachrichten konsumieren. Wir haben bereits an einigen Stellen festgestellt, dass Kafka immer möglichst viel Arbeit an die Clients auslagert, damit sich die Broker auf ihre Kernarbeit fokussieren können. Wir wissen auch bereits, dass die Kafka-Broker Nachrichten nicht proaktiv an die Consumer pushen, sondern dass Kafka-Consumer pull-basiert sind.

3.3.1 Consumer

Im einfachsten Fall haben wir einen Consumer, der Daten aus einem oder mehreren Topics konsumieren möchte. Der Consumer hat sich bereits erfolgreich zum Kafka-Cluster verbunden und dank der Antwort auf seinen *Metadata-Request* weiß unser Consumer, welche Broker Leader für welche Partitionen sind.

3.3.1.1 Fetch-Request

Um nun Nachrichten zu konsumieren, schickt der Consumer an die Leader einer Partition, die er konsumieren möchte, und dem Offset, ab welchem der Broker die Nachrichten zurückliefern soll. Das können wir mit dem `kafka-console-consumer.sh` folgendermaßen simulieren:

```
$ kafka-console-consumer.sh \
--topic flugdaten \
--offset 0 \
--partition 0 \
--bootstrap-server localhost:9092
Countdown gestartet
```

Dafür müssen wir explizit die gewünschte Partition und auch den Offset angeben, ab dem wir Nachrichten erhalten möchten. Üblicherweise geben wir weder die Partition noch den Offset explizit an, sondern verlassen uns darauf, dass sich die Kafka-Bibliothek darum kümmert. Wir haben bereits im ersten Kapitel das Flag `--from-beginning` kennengelernt. Standardmäßig beginnt der `kafka-console-consumer.sh` am Ende einer Partition zu lesen, und wir nutzen dieses Flag, um von Beginn an zu lesen. Standardmäßig nutzt die Kafka-Library dieselbe Einstellung. Wenn wir einen Consumer starten, fangen wir bei den neuesten Nachrichten zu lesen an. Um am Anfang der Partitionen zu beginnen, müssen wir die Konfigurationseinstellung `auto.offset.reset` auf den Wert `earliest` (Standard `latest`) setzen.

Wenn wir aber diesen Consumer stoppen und wieder erneut starten, dann haben wir uns keine Offsets gemerkt, sondern starten da, wie es uns die Einstellung `auto.offset.reset` vorschreibt, zum Beispiel immer wieder vom Anfang.

3.3.1.2 Fetch vom nächsten Replica

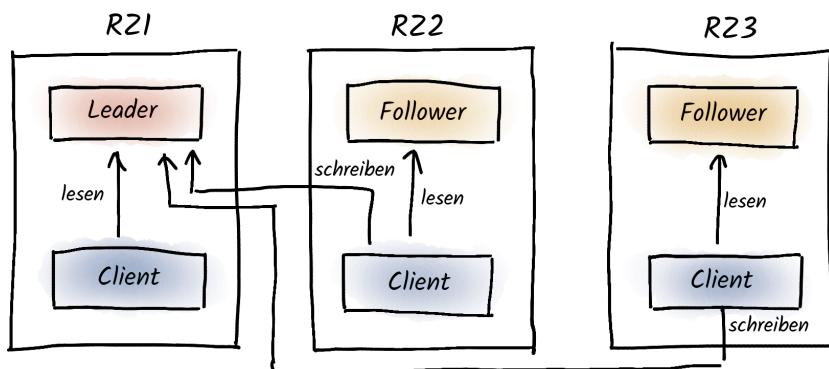


Bild 3.9 Fetch vom nächsten Replica

Bisher haben wir immer davon gesprochen, dass ein Consumer vom Leader einer Partition Nachrichten abfragt. Das ist so nicht korrekt, denn seit KIP-392⁵ gibt es die Möglichkeit,

⁵⁾ KIP-392: Allow consumers to fetch from closest replica

vom nächsten Replica zu konsumieren. Woher weiß Kafka nun aber, welches das nächste Replica ist? In Kafka gibt es hierfür die Möglichkeit, Brokern eine Rack-ID beziehungsweise einen Ort zuzuweisen (`broker.rack`). Ursprünglich diente diese Konfiguration nur dazu, Kafka mitzuteilen, wo sich die Broker befinden, damit Kafka die Partitionen gleichmäßig auf die Fehlerdomänen verteilen kann, um die Ausfallsicherheit zu erhöhen. Mit KIP-392 wurde eine äquivalente Konfigurationsmöglichkeit für Clients hinzugefügt (`client.rack`) und den Clients die Möglichkeit gegeben, ein spezielles Replica für das Abrufen der Nachrichten auszuwählen.

Bevor wir uns im Detail anschauen, wie Consumer Groups uns dabei helfen, die Offsets zu persistieren, schauen wir uns gemeinsam an, was der Broker leisten muss, um unserem Consumer eine zufriedenstellende Antwort zu liefern.

3.3.2 Broker

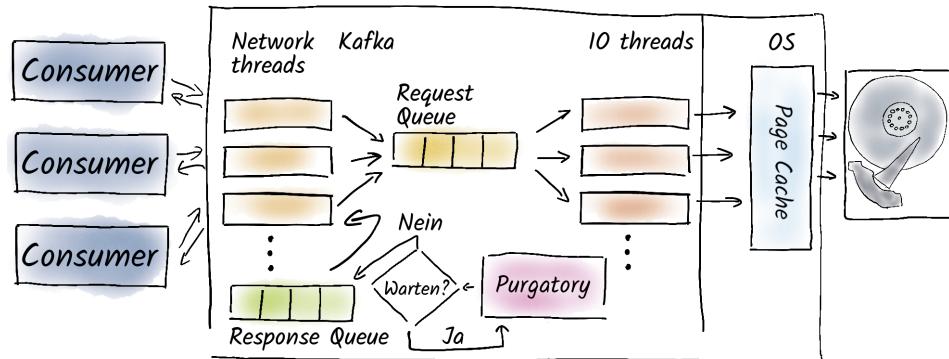


Bild 3.10 Der Prozess des Brokers beim Empfangen eines `fetch`-Requests ähnelt sehr stark dem beim Empfang eines `produce`-Requests. Statt aber Daten auf die Festplatte zu schreiben, müssen wir lesen. Auch müssen wir nicht auf weitere Broker warten, sondern abhängig von den `fetch.max.wait.ms`- und `fetch.min.bytes`-Einstellungen auf weitere Nachrichten.

Im Grunde ist der Prozess, den ein Broker bei einem Fetch-Request abarbeiten muss, sehr ähnlich dem eines Produce-Requests. Statt Daten ins Dateisystem zu schreiben, müssen wir davon lesen. Wir starten im Broker auch hier mit denselben Network Threads, die der Producer nutzt, um den Request zu empfangen, und legen diesen auf der Request-Queue zum Puffern ab. Die IO-Threads schreiben nun keine Batches ins Dateisystem, sondern lesen diese von dort. Im Optimalfall müssen wir dafür nicht einmal die Festplatte bemühen, da die Nachrichten noch im Page-Cache liegen und von dort bezogen werden. Auch beim Konsumieren nutzen wir die Purgatory. Hier nutzen wir sie nicht, um auf andere Broker zu warten, sondern auf weitere Nachrichten, wenn der Consumer dies wünscht. Dies konfigurieren wir auf Consumer-Seite, da unterschiedliche Consumer unterschiedliche Anforderungen an Latenzen und Bandbreiten haben können. Standardmäßig schickt der Broker sofort eine Antwort an den Consumer, wenn mindestens eine neue Nachricht vorliegt, da die Einstellung `fetch.min.bytes` auf den Wert 1 gesetzt ist. Aber auch wenn es keine

neuen Nachrichten gibt, wartet der Broker nicht unendlich lange, bis er dem Consumer eine Antwort schickt, sondern maximal `fetch.max.wait.ms`. Standardmäßig ist dieser Wert auf 500 ms eingestellt. Wie im Kapitel „Performance“ besprochen, können wir über diese Parameter die Consumer-Performance optimieren.

Nachdem sich unser Broker dafür entschieden hat, eine Antwort zu schicken, legt dieser die Antwort auf die Response-Queue, von wo aus ein Network Thread die Antwort an unseren Consumer schicken kann.

3.3.3 Offsets

Während klassische Messaging-Systeme sich selbst darum kümmern, welcher Consumer welche Nachrichten bekommt, ist es bei Kafka die Aufgabe der Consumer zu bestimmen, welche Nachrichten in einer Partition der Broker schicken soll. Das heißt, der Consumer muss sich selbst darum kümmern, sich zu merken, welche Nachrichten er schon gelesen hat und welche noch nicht. Wir haben dafür in Kafka die Offsets kennengelernt. Die erste Nachricht in einer Partition hat den Offset 0, die zweite den Offset 1 und so weiter. Der Kafka-Broker inkrementiert den Offset für jede neue Nachricht. Auch wenn eine Nachricht später gelöscht werden sollte, verändern sich die Offsets der anderen Nachrichten nicht. Das heißt, wir können unter Umständen auch Lücken in unseren Offsets haben, aber dann schickt der Kafka-Broker uns einfach die nächste Nachricht.

3.3.3.1 Verwaltung von Offsets

Für den Consumer ist der Offset die Information, welche Nachricht als Nächstes gelesen werden soll. Wir haben im Abschnitt „Consumer“ die Einstellung `auto.offset.reset` kennengelernt, mit der wir einstellen können, was der Consumer tun soll, wenn er noch keinen Offset für eine Partition kennt; ob er bei der ältesten Nachricht (`auto.offset.reset=earliest`), also der mit dem geringsten Offset, oder am Ende der Partition (`auto.offset.reset=latest`) zu lesen beginnt.

Wo aber können unsere Consumer die Offsets speichern, sodass wir auch nach einem Neustart wieder da weiterlesen können, wo der Consumer aufgehört hat? Es gibt sogar Fälle, wo wir unsere Offsets gar nicht speichern möchten. Nehmen wir zum Beispiel den Fall an, dass unser Consumer einen In-Memory-Cache mit Daten aus Kafka befüllt. Wenn dieser Consumer abstürzt, dann sind die Daten weg, und wir müssen diesen Cache wieder von vorne befüllen. Wir müssen hier keine Offsets speichern. Oder wir stellen uns vor, dass wir in unserem Raketen-Kontrollzentrum einen großen Bildschirm haben, der aktuelle Werte der Rakete anzeigt (zum Beispiel die aktuelle Entfernung von der Erde). Dieser Bildschirm soll einfach immer den neuesten Wert anzeigen, und wir wissen, dass dieser ständig aktualisiert wird. Wir müssen auch hier keine Offsets zwischenspeichern, wenn der Service nämlich neu startet, soll er einfach am Ende der Partition weiterlesen, um die neuen, aktuellen Werte anzuzeigen.

Aber in den meisten Fällen möchten wir die Daten aus Kafka irgendwo persistieren, verarbeiten oder in ein Drittsystem weiterleiten. Das heißt, wir müssen unsere Offsets speichern. Dafür gibt es unterschiedliche Möglichkeiten. Wir könnten zum Beispiel den Offset lokal im Service auf der Festplatte abspeichern, und jedes Mal, wenn wir den Service neustarten,

lesen wir den Offset von der Festplatte und fragen die neuen Nachrichten an. Dies ist nützlich, wenn wir zum Beispiel Daten aus Kafka in eine lokale Datenbank oder einen lokalen Cache speichern möchten. Oft möchten wir aber keinen Zustand in unseren Services abspeichern. Sie sollen *stateless* sein. Auch hindert uns dieser Ansatz daran, unsere Services horizontal zu skalieren, da wir dann irgendwie auf magische Art und Weise diese Offsets verteilen müssten.

Eine andere Möglichkeit ist es, die Offsets in einem externen System abzuspeichern. Wenn wir zum Beispiel die Daten aus Kafka in eine Datenbank schreiben, können wir eine zusätzliche Tabelle für Offsets ablegen und sogar die Daten mit den Offsets in einer Transaktion schreiben. Damit gewinnen wir sogar *Exactly-Once*-Garantien! Denn entweder wird die gelesene Nachricht mit dem Offset in die Datenbank geschrieben oder weder das eine noch das andere. Dies ist übrigens genau der Ansatz, den viele Kafka Connect Connectoren benutzen, um Exactly-Once-Garantien zu bieten.

Aber eigentlich haben wir bereits ein System, in dem wir Daten speichern. Offsets sind auch nichts anderes als Daten. Üblicherweise kümmern wir uns gar nicht selbstständig um die Offsets, sondern lassen uns von Kafka und der Kafka-Bibliothek unterstützen. In Kafka nutzen wir das `__consumer_offsets`-Topic, um unsere Offsets abzuspeichern. Dieses Topic ist ein ganz normales Topic mit ganz wenig eingebauter Magie. Wie also weiß Kafka, wem welche Offsets gehören? Kafka selbst kann es gar nicht wissen, sondern wir müssen es Kafka sagen. Wir speichern nicht für jeden Consumer eigene Offsets, sondern um Offsets in Kafka zu speichern, müssen diese immer einer Consumer Group zugeordnet sein. In diesem Kapitel schauen wir uns nur die Offsets genauer an, aber wir benutzen Consumer Groups auch, um unsere Consumer horizontal zu skalieren. Dies ist das Thema des nächsten Abschnitts „Consumer Groups“.

3.3.3.2 Praxisbeispiel Offsets

Um dies zu veranschaulichen, erstellen wir ein Topic `flugdaten-2partitions` mit zwei Partitionen und produzieren in dieses Topic einige Nachrichten:

```
> rakete0:Countdown beendet
> rakete1:Countdown beendet
> rakete1:Liftoff
```

Starten wir nun einen Kafka Consumer und geben eine Gruppe an:

```
$ kafka-console-consumer.sh \
  --topic flugdaten-2partitions \
  --bootstrap-server=localhost:9092 \
  --group flugueberwachung \
  --from-beginning \
  --property print.key=true
rakete0    Countdown beendet
rakete1    Countdown beendet
rakete1    Liftoff
```

Wenn wir diesen Befehl mit **STRG+C** abbrechen und noch mal starten, dann sehen wir keine neuen Nachrichten. Das heißt, dass wir uns irgendwo die korrekten Offsets gemerkt haben. Um das zu überprüfen, können wir das Werkzeug `kafka-consumer-groups.sh` benutzen, das uns die Informationen über diese Gruppe anzeigt. Starten wir es dazu am besten in einem neuen Terminal:

```
$ kafka-consumer-groups.sh \
--group flugueberwachung \
--describe \
--bootstrap-server localhost:9092
```

Da wir im Buch nicht unendlich viel Platz in der Breite haben, haben wir die Tabelle der Übersichtlichkeit halber gedreht:

GROUP	flugueberwachung	flugueberwachung
TOPIC	flugdaten-2partitions	flugdaten-2partitions
PARTITION	0	1
CURRENT-OFFSET	2	1
LOG-END-OFFSET	2	1
LAG	0	0
CONSUMER-ID	consumer-flugueberwa- chung-1-[...]a1d0	consumer-flugueberwa- chung-1-[...]a1d0
HOST	/192.168.0.110	/192.168.0.110
CLIENT-ID	consumer-flugueberwachung-1	consumer-flugueberwachung-1

In der Tabelle sind die Informationen zu unserer Consumer Group `flugueberwachung` abgebildet. Wir sehen, dass unsere Consumer Group wie erwartet das Topic `flugdaten-2partitions` konsumiert und dort für jede Partition einen Eintrag hat. Partition 0 hat den Log-End-Offset 2, also wird der Offset der nächsten geschriebenen Nachricht 2 sein. Dieser Log-End-Offset ist identisch zum Current-Offset unserer Gruppe. Das heißt, die Gruppe hat alle Nachrichten dieser Partition gelesen. Daraus resultiert der Lag von 0. Für unsere Consumer ist die wichtigste Metrik dieser sogenannte Lag. Dieser besagt, wie viele Nachrichten eine Consumer Group hinterherhinkt. Ist dieser 0, wissen wir, dass alle Nachrichten gelesen wurden. Für die Partition 1 sind die Werte ähnlich, nur haben wir erst eine Nachricht in der Partition stehen.

Die Consumer-ID wird automatisch generiert, um Mitglieder einer Consumer Group identifizieren zu können. Da wir nur einen Consumer gestartet haben, ist die ID Identisch. Der Host entspricht der IP-Adresse des Consumers, und die Client-ID wird vom `kafka-console-consumer.sh` automatisch generiert.

Wenn wir nun unseren Consumer stoppen, dann einige Nachrichten schreiben und den Befehl `kafka-consumer-groups.sh` noch einmal aufrufen, sehen wir, dass sich der Log-End-Offset und der Lag verändert haben:

CURRENT-OFFSET	2	1
LOG-END-OFFSET	3	5
LAG	1	4

Wenn wir den Consumer wieder starten würden, könnten wir die neuen Nachrichten lesen, und der Lag würde sich wieder auf 0 reduzieren.

Wir können sehen, selbst wenn wir es nicht nötig haben, unsere Consumer horizontal zu skalieren, bieten uns Consumer Groups eine sehr komfortable und zuverlässige Methode,Offsets zu speichern.



Tipp

Wenn es keine stichhaltigen Argumente dagegen gibt, sollten immer Consumer Groups benutzt werden.

Wichtig ist dabei, dass allein die Group-ID darüber bestimmt, wer welche Offsets bekommt und committet. Es passiert leicht, dass wir den Code eines anderen Consumers kopieren und dabei vergessen, die Group ID zu ändern. Dies führt dann dazu, dass wir einem anderen Consumer seine Offsets stehlen. In Entwicklungsumgebungen mag dies unangenehm sein, aber in Produktionsumgebungen kann das katastrophale Folgen haben.

3.3.4 Consumer Groups

In diesem Buch sind uns Consumer Groups schon mehrfach begegnet. Wir erinnern uns, dass wir Consumer Groups nutzen, um unsere Consumer horizontal zu skalieren, und wir haben im letzten Abschnitt erfahren, dass wir Consumer Groups benötigen, um Offsets bequem in Kafka zu verwalten.

3.3.4.1 Funktionsweise von Consumer Groups

Um Consumer Groups zu benutzen, benötigen wir keine weiteren Komponenten, wir müssen lediglich in unseren Clients die Group-ID setzen. Ein oder mehrere Consumer mit der gleichen Group-ID bilden eine Consumer Group.

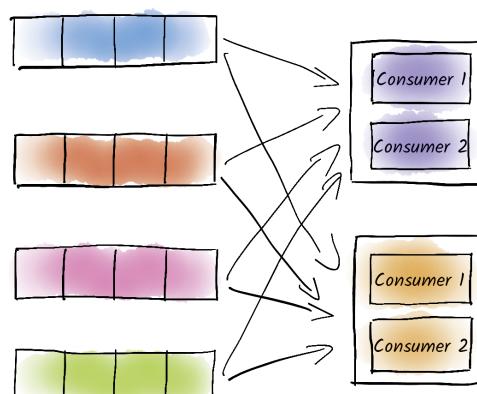


Bild 3.11 Eine Consumer Group besteht aus beliebig vielen Consumern, die die gewünschten Partitionen untereinander aufteilen. Unterschiedliche Consumer Groups sind voneinander isoliert und wissen nichts voneinander. Die Kafka-Broker unterstützen die Consumer bei der Koordination der Aufgabenverteilung.

Nun ist unser Ziel in Kafka, dass die gleichen Nachrichten auch mehrmals von unterschiedlichen Consumern gelesen werden können. Zum Beispiel haben wir in der Einführung das Topic *flugdaten* erstellt und hatten zwei Services, einen Analytics-Service und einen

Metrik-Service, laufen lassen. Wir haben gesehen, dass diese beiden Consumer Groups unabhängig voneinander das Topic konsumiert und innerhalb der Gruppe die Partitionen aufgeteilt haben.

Wie wissen aber die Consumer in einer Gruppe voneinander? Wie teilen sie die Partitionen untereinander auf? Die Consumer müssen die Arbeit irgendwie miteinander koordinieren. Wir erinnern uns aus unserer Einführung in verteilte Systeme, dass Koordination ein sehr schwieriges Problem ist. Aber wir haben ja bereits ein ausfallsicheres, verteiltes System zur Hand, nämlich die Kafka-Broker selbst. Statt nun also für jede Consumer Group ein eigenes Koordinationscluster zu betreiben, unterstützen uns die Kafka-Broker dabei, die Consumer zu koordinieren.

Wir werden später feststellen, dass Consumer Groups nicht die einzige Komponente ist, die sich von Kafka unterstützen lässt. Genauso unterstützen die Kafka-Broker unsere Kafka Connect Cluster und Kafka Streams-Anwendungen bei deren Koordination.

3.3.4.2 Kafka Rebalance Protocol

Dafür gibt es sogar ein eigenes Protokoll, nämlich das Kafka Rebalance-Protokoll. Damit ist es möglich, Ressourcen (zum Beispiel Partitionen, Kafka Streams Tasks oder Kafka Connect Tasks) auf Mitglieder einer Gruppe aufzuteilen. Auch hier verfolgt Kafka die Philosophie, so viel Arbeit wie möglich auf die Clients auszulagern. Kafka selbst interessiert sich nicht dafür, was genau koordiniert wird, sondern kümmert sich nur um das Allernötigste.

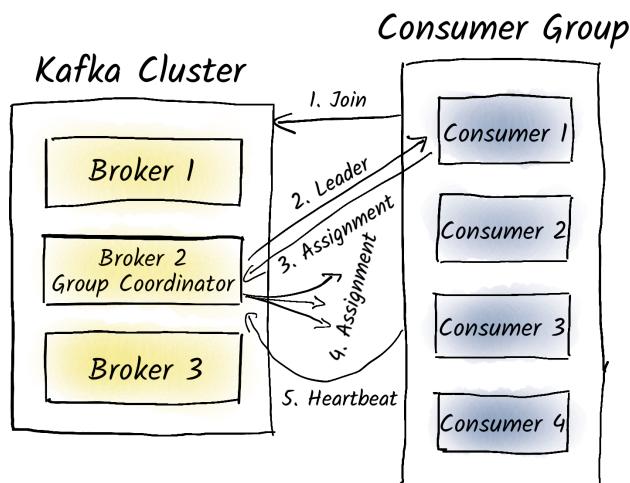


Bild 3.12 Das Kafka Rebalance-Protokoll ermöglicht es, dass eine Gruppe gewisse Ressourcen untereinander aufteilt. Zum Beispiel Consumer-Partitionen. Dabei fungiert ein Broker als *Group Coordinator* und einer der Consumer als *Group Leader*.

Wie funktioniert aber dieses Kafka Rebalance-Protokoll? Dafür benötigen wir als Erstes auf Kafka-Seite eine zentrale Instanz, die für eine bestimmte Gruppe verantwortlich ist. Diesen Broker nennen wir *Group Coordinator*. Für eine bessere Lastverteilung gibt es natürlich nicht den einen Broker, der Group Coordinator ist, sondern wir verteilen die Last so gleich-

mäßig wie nötig. Dafür nutzen wir ein uns bekanntes Muster, abhängig von der Group ID verteilen wir die Group Coordinators auf unsere Broker.

Als Erstes melden unsere Gruppenmitglieder an, dass sie gerne Teil der Gruppe werden möchten. Dafür schicken sie einen *Join-Request* (1) an den Group Coordinator der Gruppe, der sie beitreten möchten. Join-Requests sind eine Synchronisationsbarriere in diesem Protokoll. Es gibt erst eine Antwort auf den Join-Request, sobald sich alle potenziellen Gruppenmitglieder beim Group Coordinator gemeldet haben. Das heißt, falls die Gruppe Mitglieder hat, werden sie vom Group Coordinator als Erstes aus der Gruppe geworfen, um sich erneut mit einem Join-Request zu melden. Ab diesem Zeitpunkt und bis zum Ende des Protokolls sind die Consumer nicht Mitglied der Gruppe und dürfen demzufolge keine Daten konsumieren. Der Group Coordinator wartet, bis alle Gruppenmitglieder die Chance hatten, sich zu melden, und bestimmt das erste Mitglied, das den Join-Request gesendet hat, als *Group Leader* (2).

Wir haben bereits mehrmals besprochen, dass die Kafka-Broker möglichst viel Arbeit an die Clients auslagern. So auch hier. Der Group Coordinator koordiniert nur die Gruppe, die restliche Arbeit wird an die Clients ausgelagert. Im Falle von Consumer Groups ist der Group Leader, also einer der Consumer, dafür zuständig zu bestimmen, welches Mitglied der Gruppe für welche Partition zuständig ist. Wir schauen uns die unterschiedlichen Möglichkeiten, die wir haben, später im Verlauf des Kapitels an.

Der Group Leader bestimmt also die Arbeitsverteilung und schickt diese an den *Group Coordinator* (3). Der Group Coordinator verteilt dann die Aufgaben auf die *Mitglieder der Gruppe* (4). Ab diesem Zeitpunkt dürfen die Gruppenmitglieder zum Beispiel mit dem Konsumieren der Nachrichten fortfahren.

Wie weiß nun aber der Group Coordinator, ob noch alle Mitglieder leben und Teil der Gruppe sind? Die Mitglieder schicken in regelmäßigen Abständen *Heartbeat-Signale* (5). Wie in Netzwerkprotokollen üblich, gibt der Group Coordinator den Mitgliedern drei Heartbeats Zeit, sich zu melden, und wirft sie danach aus der Gruppe. Wir können sowohl die Intervalle zwischen den Heartbeats (`heartbeat.interval.ms=3s`) als auch die Zeit, nachdem der Group Coordinator ein Mitglied, das sich nicht mehr meldet, aus der Gruppe wirft (`session.timeout.ms=10s`), anpassen. Für viele Anwendungsfälle sind aber die Standardeinstellungen ausreichend und empfehlenswert.

Wir betrachten dieses Protokoll vor allem mit dem Interesse, es bei unseren Consumer Groups einzusetzen, um unsere Partitionen auf mehrere Consumer aufzuteilen, um so unsere Verarbeitung der Daten zu parallelisieren. Kafka bietet mehrere Möglichkeiten, Partitionen auf die Consumer zu verteilen.

3.3.4.3 Verteilung der Partitionen auf die Consumer

Wir konfigurieren dies über die Einstellung `partition.assignment.strategy` im Consumer. Standardmäßig nutzt Kafka den *RangeAssignor*. Wir gehen für den *RangeAssignordass* alle unsere Consumer dieselben Topics konsumieren möchten, die dieselbe Anzahl an Partitionen haben.

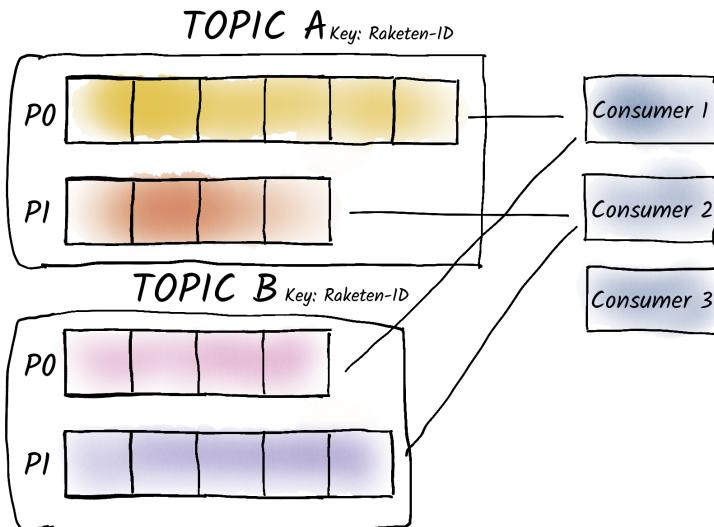


Bild 3.13 Standardmäßig werden die Partitionen auf die Consumer einer Consumer Group so aufgeteilt, dass die gleiche Partition unterschiedlicher Topics immer vom selben Consumer konsumiert wurde. Es ist zwar ressourcenechnisch keine optimale Aufteilung, aber so können wir einfach Joins über mehrere Topics durchführen.

Der **RangeAssignor** garantiert uns nun, dass Partition 0 aller Topics demselben Consumer zugeordnet wird, dass Partition 1 aller Topics demselben Consumer zugeordnet wird und so weiter. Haben wir, wie im Beispielbild, zwei Topics mit jeweils zwei Partitionen und drei Consumern, ordnen wir Partition 0 beider Topics dem Consumer 1 zu und Partition 1 beider Topics dem Consumer 2. Der Consumer 3 bekommt keine Partitionen zugeordnet. Dieser Assignor ist nützlich, wenn wir Daten aus mehreren Topics gemeinsam verarbeiten möchten. Zum Beispiel könnten wir die Daten aus dem Topic *flugdaten* mit Informationen über unsere Raketen aus dem Topic *raketen* anreichern. Wir benutzen in beiden Topics die ID der Rakete als Key und wissen somit, dass die Nachrichten für dieselbe Rakete immer auf denselben Partition landen, egal ob im Topic *flugdaten* oder dem Topic *raketen*. Das heißt, wenn wir Daten zu einer Rakete aus dem Topic *flugdaten* in Partition 0 lesen, dann finden wir die Daten zur Rakete im Topic *raketen* auch in Partition 0. Das nennen wir einen sogenannten *Stream-Join*, und insbesondere Kafka Streams macht sehr regen Gebrauch davon.

Manchmal möchten wir nur von unterschiedlichen Topics konsumieren, ohne die Daten miteinander in Verbindung zu setzen. Vielleicht nutzen wir gar nicht erst Keys. Dann müssen wir uns auch nicht auf die Einschränkungen des *RangeAssignors* einlassen, sondern können den *RoundRobinAssignor* nutzen, um die Partitionen aller Topics so gleichmäßig wie möglich auf unsere Consumer aufzuteilen.

Wir erinnern uns, dass wir bei einem Rebalance aufhören müssen zu konsumieren und unser Consumer nach einem Rebalance unter Umständen andere Partitionen bekommt. Das heißt, unsere Consumer müssen vor einem Rebalance die Caches leeren und die Offsets committen. Wenn in unserem Beispielbild Consumer 1 ausfällt, dann bekommt jeder Consumer ganz andere Partitionen zugewiesen. Wir können dies vermeiden, indem wir den

StickyAssignor benutzen. Statt jedes Mal das Partitions-Assignment von Neuem zu berechnen, versucht der *StickyAssignor*, so wenig wie möglich zu verändern. Ansonsten funktioniert dieser genauso wie der *RoundRobinAssignor*.

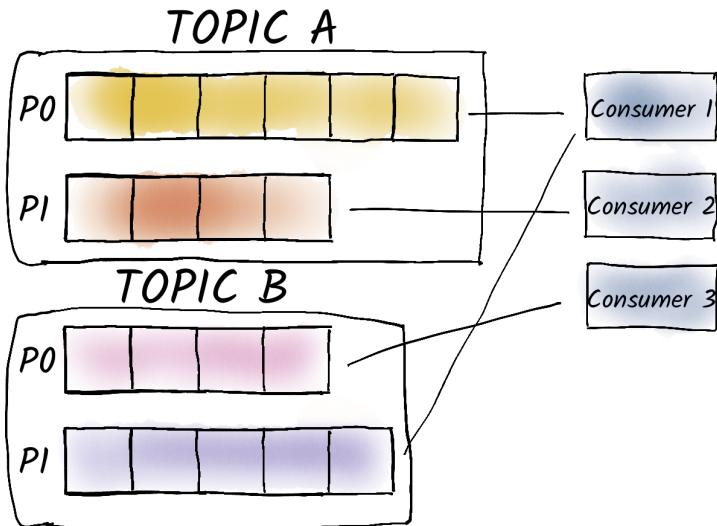


Bild 3.14 Wenn wir keine Joins benötigen, setzen wir den *RoundRobinAssignor* ein, um die Partitionen gleichmäßig auf unsere Consumer zu verteilen.

Das Kafka Rebalance-Protokoll ermöglicht es uns also, Ressourcen wie zum Beispiel Partitionen oder Connectoren auf Mitglieder einer Gruppe zu verteilen. Da wir aber während der Ausführung des Protokolls alle Prozesse anhalten müssen, sind Rebalances sehr teuer. Rebalances werden ausgelöst, wenn neue Mitglieder der Gruppe beitreten möchten oder Mitglieder, gewollt oder ungewollt, die Gruppe verlassen, aber auch wenn sich zum Beispiel die Anzahl der Partitionen eines konsumierten Topics ändert. Wenn wir beispielhaft in unserer vollautomatisierten Umgebung ein Rolling-Restart von drei Consumern einer Consumer Group durchführen, würden wir damit sechs Rebalances auslösen, je einmal beim Herunterfahren eines Consumers und wieder beim Hochfahren.

3.3.4.4 Static Memberships

Um solche exzessiven Rebalances zu vermeiden, wurden mit Kafka 2.3 die sogenannten *Static Memberships* eingeführt. Wir gehen dabei davon aus, dass unsere Infrastruktur voll automatisiert ist, und falls ein Consumer ausfällt, wird dieser sofort wieder vollautomatisch neu aufgesetzt. Dies ist zum Beispiel in Cloud- und Kubernetes-Umgebungen oft der Fall. Anstelle jedes Mal ein Rebalance auszulösen, wenn ein Consumer neu gestartet wird, erhöhen wir den `session.timeout.ms` auf einen deutlich größeren Wert (zum Beispiel mehrere Minuten). Zusätzlich dazu müssen wir jedem Consumer seine Identität mitgeben. Dafür setzen wir für jeden Consumer die `group.instance.id` auf einen pro Consumer eindeutigen Wert. Wir müssen dabei nur garantieren, dass der Consumer nach einem Neustart dieselbe ID bekommt. Das können wir zum Beispiel sicherstellen, indem wir in Kubernetes

StatefulSets benutzen oder in der Cloud die ID auf den Hostnamen setzen, wenn dieser über Neustarts hinweg konstant bleibt. Mit diesen Einstellungen muss der Rebalance nur noch ausgeführt werden, wenn neue Consumer hinzukommen, sich ein Consumer für mehrere Minuten nicht meldet oder die Anzahl der Partitionen verändert wird. Damit erreichen wir eine deutlich bessere Auslastung unserer Consumer und vermeiden größere Ausfälle in der Verarbeitung der Daten.

Kafka 2.4 und Kafka 2.5 optimieren dieses Verhalten noch weiter, indem sie den *Cooperative-StickyAssignor* neu einführen. Statt im Falle eines Rebalances die Welt anzuhalten, optimiert dieser Assignor das Verhalten so, dass die Consumer sich iterativ dem Wunschzustand annähern und in Kooperation das gewünschte Assignment finden. Dieses Verhalten hat vor allem Auswirkungen auf Kafka Streams und Kafka Connect, wo ansonsten alle Connectoren und Applikationen einer Gruppe angehalten werden mussten. Voraussichtlich wird Kafka 3.1 diesen Assignor als Standarmäßig verwenden.

3.3.5 Zusammenfassung

In diesem Kapitel haben wir uns genau angesehen, wie Nachrichten in Kafka eigentlich konsumiert werden. Wir haben sehr viel über Consumer und wie sie Nachrichten abrufen gelernt. Natürlich haben wir in diesem Zusammenhang auch wieder einen Blick auf unsere Broker geworfen und uns angesehen, wie die Broker eben diese Anfragen unserer Consumer bearbeiten. Wir haben einiges über Offsets und insbesondere deren Verwaltung gelernt. Das Offset-Thema haben wir außerdem noch ausführlich in einem Praxisbeispiel vertieft. Den Abschluss dieses Kapitels bildete ein Abschnitt über Consumer Groups. Wir haben hierbei erfahren, was genau Consumer Groups sind und wozu sie dienen. Wir wissen nun auch, wie die Arbeit, das heißt das Konsumieren der einzelnen Partitionen, in Consumer Groups auf die einzelnen Consumer aufgeteilt wird. In diesem Zusammenhang haben wir uns auch damit beschäftigt, was passiert, wenn ein neuer Consumer zur Consumer Group hinzugefügt wird oder ein bestehender Consumer die Consumer Group verlässt. Mit *Static Memberships* haben wir zum Abschluss auch ein Konzept kennengelernt, wodurch unnötige Rebalances vermieden werden können.

■ 3.4 Nachrichten aufräumen

In den letzten Kapiteln haben wir uns ausführlich damit beschäftigt, wie Nachrichten in Apache Kafka produziert, verarbeitet, gespeichert und auch konsumiert werden. Allerdings haben wir bisher einen sehr wichtigen Punkt, nämlich wie Nachrichten in Apache Kafka aufgeräumt werden, komplett ausgelassen. Genau hier setzt dieses Kapitel an. Wir werden lernen, welche Möglichkeiten Kafka zum automatischen Aufräumen von Nachrichten bietet und was wir dabei beachten müssen.

3.4.1 Wieso müssen wir Nachrichten aufräumen?

Bevor wir uns aber im Detail ansehen, wie Kafka Nachrichten aufräumt, sollten wir kurz darüber nachdenken, wieso wir eigentlich Nachrichten in Kafka aufräumen sollten und welche Konsequenzen damit einhergehen würden, wenn wir Nachrichten nie aufräumen würden. Ein Grund ist unsere Speicherkapazität. Theoretisch könnten wir zwar einfach alle Nachrichten für immer speichern, allerdings würde so auch unser Log bis ins Unendliche wachsen und somit auch schnell an die Grenzen unseres verfügbaren Speichers stoßen. Ein anderer Grund ist Performance. Umso größer unser Log ist, desto länger brauchen wir im Zweifel für die Verarbeitung aller Nachrichten, und je nach Anwendungsfall verarbeiten wir vermutlich auch eine große Anzahl von irrelevanten Nachrichten. Der vermutlich wichtigste Grund, um Daten zu löschen, ist es, dass wir die Daten gar nicht mehr benötigen oder aus rechtlichen Gründen gar nicht mehr behalten dürfen. Wie stellen wir aber sicher, dass wir nicht versehentlich Nachrichten löschen, die wir noch brauchen?

3.4.2 Kafkas Aufräum-Methoden

Kafka verfolgt hierzu zwei verschiedene Ansätze, *Log Retention* und *Log Compaction*. Bei der Log Retention werden einfach alle Nachrichten gelöscht, die ein gewisses Alter erreicht haben, das heißt Nachrichten, die vor einer bestimmten Zeit produziert worden sind. Log Compaction auf der anderen Seite löscht veraltete Daten. Um festzustellen, ob eine Nachricht veraltet ist, nutzt Kafka die Keys unserer Nachrichten. Hierbei wird lediglich für jeden Key die neueste Nachricht behalten. Log Compaction ist demzufolge auch nur möglich, wenn wir Keys für unsere Nachrichten vergeben.

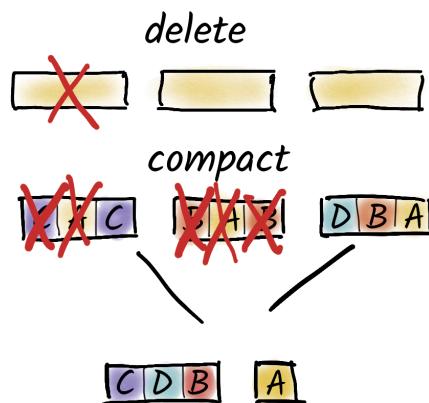


Bild 3.15 Der Log Cleaner geht immer segmentweise vor. Bei der *delete*-Policy wird ein gesamtes Segment gelöscht, sobald die *neuste* Nachricht des Segments älter ist als ein vorgegebener Wert. Nutzen wir die *compact*-Policy, löscht der Log Cleaner lediglich Nachrichten mit einem Key, für den es eine neuere Nachricht mit demselben Key gibt. Es ist auch möglich, beide Verfahren einzusetzen.

Beide Varianten haben jeweils Vor- und Nachteile. Es ist auch möglich, sie miteinander zu kombinieren. Log Retention hat den Vorteil, dass sie relativ einfach umzusetzen ist und

wenig Arbeit für unsere Broker verursacht, da wir zum Löschen unserer Nachrichten lediglich auf das Alter unserer Nachrichten schauen müssen. Gleichzeitig bedient Log Retention auch auf diese sehr einfache Art und Weise wichtige Anwendungsfälle. Wir können so zum Beispiel sicherstellen, dass wir Nachrichten, die aus datenschutzrechtlichen Gründen nach einer gewissen Zeit gelöscht werden müssen, automatisch löschen. Eine weitere wichtige Klasse von Anwendungsfällen für Log Retention sind Sensordaten, welche nach einer gewissen Zeit vermutlich irrelevant sind und gelöscht werden können, oder Logs von Programmen, die wir auch nicht unbegrenzt vorhalten müssen. Vielleicht wird Kafka auch nur als Message-System genutzt, dann können wir ebenfalls ohne Bedenken Nachrichten nach kurzer Zeit löschen. Allerdings können wir mit Log Retention nicht gezielt Daten löschen und müssen hier aufpassen, dass wir nicht versehentlich wichtige Daten löschen, nur weil sie alt sind.

An dieser Stelle kommt Log Compaction ins Spiel. Stellen wir uns zum Beispiel vor, dass wir ein Log haben, in dem wir Kundendaten, wie zum Beispiel die Kundenadresse, speichern. Jedes Mal, wenn sich die Adresse ändert, wird eine neue Nachricht mit dem zum Kunden gehörenden Key erzeugt. Die alte Adresse wird von da an wahrscheinlich irrelevant, aber gleichzeitig müssen wir darauf achten, dass wir die aktuelle Adresse nicht löschen, nur weil die Nachricht ein bestimmtes Alter erreicht hat. Log Compaction ist genau für solche Anwendungsfälle prädestiniert, da es sicherstellt, dass wir immer die neueste Nachricht zu einem Key behalten und veraltete Informationen beziehungsweise Nachrichten automatisch löschen. Der Nachteil von Log Compaction gegenüber Log Retention ist der vergleichsweise große Aufwand, da wir bei Log Compaction das gesamte Log durchsuchen müssen, um festzustellen, welche Nachrichten wir löschen können.

Ob wir nun Log Retention, Log Compaction oder sogar beides gleichzeitig zum Aufräumen von Nachrichten verwenden, können wir entweder je Topic mittels `cleanup.policy` festlegen oder via `log.cleanup.policy` einen Standardwert für alle unsere Topics festlegen. Standardmäßig ist in Kafka übrigens Log Retention aktiviert (`log.cleanup.policy=delete`). In den folgenden beiden Abschnitten werden wir uns näher ansehen, wie genau Log Retention und Log Compaction in Kafka funktionieren.

3.4.3 Log Retention

In diesem Abschnitt werden wir uns anhand von Beispielen anschauen, wann genau Kafka Daten löscht und wie wir Log Retention nach unseren Bedürfnissen konfigurieren können.

Erstellen wir hierfür zunächst ein Topic namens `retention-test`:

```
$ kafka-topics.sh \
  --create \
  --topic retention-test \
  --partitions 3 \
  --replication-factor 3 \
  --bootstrap-server localhost:9092
Created topic retention-test.
```

Anschließend produzieren wir ein paar Nachrichten in das eben erstellte Topic, damit wir gleich auch Nachrichten zum Löschen haben:

```
$ kafka-console-producer.sh \
--topic retention-test \
--bootstrap-server localhost:9092
> M1
> M2
...
> M9
```

Wenn wir einen Blick in eine unserer Partitionen werfen, sehen wir die uns aus dem Kapitel „Nachrichten produzieren und persistieren“ bekannten Dateien.

```
$ ls -1 ~/kafka/data/kafka1/retention-test-0
000000000000000000000000.index
000000000000000000000000.log
000000000000000000000000.timeindex
leader-epoch-checkpoint
```

3.4.3.1 Wann wird ein Log via Retention aufgeräumt?

Wir können in Kafka Log Retention auf zwei verschiedene Arten konfigurieren. Zum einen können wir die Größe einer Partition festlegen, bei der Log Retention getriggert wird. Sollte die Partitionsgröße über diesen Wert anwachsen, wird das älteste Segment gelöscht. Der Parameter zur Konfiguration der Partitionsgröße, bei der dies passiert, ist `retention.bytes` und definiert die Größe in Bytes. Standardmäßig ist dieser Wert auf `-1` gesetzt, was bedeutet, dass Log Retention über die Partitionsgröße deaktiviert ist. Der Grund dafür, dass es standardmäßig deaktiviert ist, ist, dass dies nur in seltenen Fällen sinnvoll ist. Wenn wir zum Beispiel Massendaten verarbeiten und es uns wichtiger ist, dass Kafka stets erreichbar ist, auch wenn wir unter Umständen Daten verlieren, können wir diese Einstellung aktivieren.

Die zweite Möglichkeit ist es, Nachrichten beziehungsweise Segmente nach einer festgelegten Zeitspanne (Aufbewahrungszeit) zu löschen. Der Parameter hierfür ist `retention.ms`, die Zeitspanne wird also in Millisekunden definiert. Standardmäßig löscht Kafka Nachrichten beziehungsweise Segmente, wenn die neueste Nachricht in einem Segment älter als sieben Tage ist. Mit `log.retention.bytes` und `log.retention.ms` können wir die Standardwerte für unser Cluster anpassen. Übrigens lassen sich beide Optionen auch wunderbar kombinieren. So können wir sicherstellen, dass wir sowohl alte Nachrichten löschen und auch dass unser Topic beziehungsweise unsere Partition nicht ins Unendliche wächst.

Probieren wir nun Log Retention aus, indem wir die Konfiguration unseres `retention-test`-Topics mit dem `kafka-configs.sh`-Skript anpassen und den Wert für `retention.ms` auf 60 Sekunden setzen:

```
$ kafka-configs.sh --alter \
--topic retention-test \
--add-config retention.ms=60000 \
--bootstrap-server=localhost:9092
```

Wenn wir anschließend einen Blick in unsere Partition werfen, werden wir feststellen, dass sich einiges verändert hat:

```
$ ls -1 ~/kafka/data/kafka1/retention-test-0
00000000000000000004.index
00000000000000000004.log
00000000000000000004.snapshot
00000000000000000004.timeindex
leader-epoch-checkpoint
```

Wie wir sehen können, sind unsere ursprünglichen Dateien, die die Nachrichten enthielten, die wir gerade produziert haben, verschwunden. Das liegt daran, dass wir unsere letzte Nachricht vor mehr als 60 Sekunden produziert haben und daher die letzte Nachricht in unserem Segment und damit das Segment selbst älter war als unsere Aufbewahrungszeit von 60 Sekunden, was dazu führte, dass das gesamte Segment gelöscht wurde. Da eine Partition immer aus mindestens einem Segment besteht, wurde ein neues Segment mit dem aktuellen Offset als Dateinamen angelegt. Außerdem wurden die *Leader Epoch* und *Recovery Checkpoints* auf die aktuellen Offsets angepasst. Wenn wir anschließend Nachrichten produzieren, landen sie in unserem neuen Segment. Nach 60 Sekunden wird dann unser Segment wieder gelöscht und ein neues Segment erstellt.

Log Retention ist übrigens sehr stark abhängig davon, wie oft wir ein neues Segment erstellen. Diese Abhängigkeit lässt sich am besten anhand eines Beispiels zeigen. Stellen wir uns vor, wir dürfen aus regulatorischen Gründen eine Nachricht maximal sieben Tage vorhalten. Es wäre jetzt naheliegend, einfach `retention.ms` auf einen Wert zu setzen, der sieben Tage entspricht. Doch das reicht nicht aus, da sich die Aufbewahrungszeit auf die neueste Nachricht in einem Segment bezieht. Solange wir also regelmäßig Nachrichten in ein Segment schreiben, kann es nicht gelöscht werden, und damit bleiben alle Nachrichten in diesem Segment bestehen. Wir müssen also zum einen sicherstellen, dass wir regelmäßig ein neues Segment ausrollen. Dies können wir, wie wir im Kapitel „Nachrichten produzieren und persistieren“ gelernt haben, einfach über `segment.ms` realisieren. Zum anderen müssen wir sicherstellen, dass `retention.ms` so gewählt ist, dass anschließend auch die älteste Nachricht in einem Segment spätestens bis zur maximal erlaubten Aufbewahrungszeit gelöscht wird. Wichtig ist letzten Endes, dass `segment.ms` und `retention.ms` in Summe nicht größer als sieben Tage sind, wobei das auch noch nicht zu 100 Prozent korrekt ist, da Kafka standardmäßig nur alle fünf Minuten überprüft, ob Segmente gelöscht werden können und wir somit weitere fünf Minuten abziehen müssen. Diese Eigenschaft kann via `log.retention.check.interval.ms` angepasst werden. Je nachdem, wie wir `segment.ms` und `retention.ms` auf diese sieben Tage bzw. sieben Tage minus fünf Minuten aufteilen, hat das Auswirkungen darauf, wie lange unsere Nachrichten tatsächlich vorgehalten werden. Rollen wir zum Beispiel nur alle sechs Tage ein neues Segment aus, dürften wir `retention.ms` auf maximal 23 Stunden und 55 Minuten setzen. Dies hätte zur Folge, dass unser Segment zum Zeitpunkt der Löschung Nachrichten enthält, die zwischen einem und sieben Tagen alt sein können. Wenn wir stattdessen jeden Tag ein neues Segment ausrollen und dementsprechend `retention.ms` auf fünf Tage 23 Stunden 55 Minuten anpassen, löschen wir Nachrichten, die zwischen sechs und sieben Tage alt sind. Umso häufiger wir also ein neues Segment ausrollen, desto feingranularer können wir unsere Nachrichten löschen und desto höher können wir `retention.ms` setzen, was dazu führt, dass wir Nachrichten länger vorhalten können.



Tipp

Über die Aufbewahrungszeit oder auch die Größe können wir übrigens auf eine einfache Art sämtliche Nachrichten in einem Topic löschen, indem wir einen der Werte einfach auf 0 setzen. Anschließend dürfen wir natürlich nicht vergessen, den Wert wieder auf den ursprünglichen Wert zu setzen, da wir ansonsten alle neuen Nachrichten direkt wieder löschen würden.

3.4.3.2 Offset Retention

Log Retention dient außerdem dazu, die zu Consumer-Gruppen gespeicherten Offsets aufzuräumen. Mit dem Parameter `offsets.retention.minutes` können wir festlegen, nach welcher Zeitspanne in Minuten der letzte von einem Consumer committete Offset gelöscht wird. Sollte eine Consumer Group alle ihre Consumer verlieren (gewollt oder ungewollt), so werden die zu einer Consumer Group zugehörigen Offsets nach dieser Zeitspanne gelöscht. Dieser Wert ist standardmäßig auf sieben Tage gesetzt, und wir empfehlen an dieser Stelle, auch diesen Wert nicht zu verändern, da eine kürzere Zeit dazu führen könnte, dass eine Consumer Group, falls sie temporär alle Consumer verloren hat, unnötigerweise wieder von vorne anfangen muss zu konsumieren. Andererseits wird durch die Consumer Offsets auch kaum Speicherplatz verbraucht. Äquivalent zum `log.retention.check.interval.ms` gibt es auch für die Offsets mit `offsets.retention.check.interval.ms` einen Parameter, durch den festgelegt wird, wie oft überprüft wird, ob Consumer Offsets gelöscht werden können. Der Standardwert ist auch hier auf fünf Minuten gesetzt.

3.4.4 Log Compaction

Nachdem wir im letzten Abschnitt gelernt haben, wie Log Retention in Kafka funktioniert, werden wir uns nun der zweiten Möglichkeit des Aufräumens von Nachrichten in Kafka zuwenden. Log Compaction ermöglicht es uns, veraltete Daten anhand ihrer Keys zu identifizieren und zu löschen. Im Gegensatz zur Log Retention garantiert die Log Compaction, dass wir immer mindestens den aktuellen Eintrag zu einem Schlüssel in unserem Log haben. Am besten lässt sich Log Compaction anhand eines praktischen Beispiels erklären. Erstellen wir hierfür zunächst ein neues Topic namens `compaction-test`:

```
$ kafka-topics.sh \
  --create \
  --topic compaction-test \
  --partitions 3 \
  --replication-factor 3 \
  --config cleanup.policy=compact \
  --bootstrap-server localhost:9092
Created topic compaction-test.
```

Diesmal haben wir explizit über den `--config`-Parameter `compact` als Aufräum-Richtlinie festgelegt (`--config cleanup.policy=compact`), da Topics ansonsten standardmäßig Log

Retention als Aufräum-Richtlinie gesetzt haben. Probieren wir nun wie gehabt, eine Nachricht zu produzieren:

```
$ kafka-console-producer.sh \
  --topic compaction-test \
  --bootstrap-server localhost:9092
> M1
> ERROR when sending message to topic compaction-test with \
key: null, value: 2 bytes with error: (...) \
org.apache.kafka.common.InvalidRecordException: One or more records have been
rejected
```

Kafka erlaubt es uns diesmal nicht, einfach Nachrichten zu senden. Da wir als Aufräum-Richtlinie Log Compaction gewählt haben und Log Compaction zwangsweise Schlüssel benötigt, werden sämtliche Nachrichten ohne Schlüssel von unseren Brokern abgelehnt. Probieren wir also erneut, Nachrichten zu senden, indem wir die aus Kapitel 2.1 „Payload“ bekannten Producer-Eigenschaften `parse.key` und `key.separator` verwenden und anschließend unsere Nachrichten mit einem Schlüssel versehen:

```
$ kafka-console-producer.sh \
  --topic compaction-test \
  --property parse.key=true \
  --property key.separator=: \
  --bootstrap-server localhost:9092
>K1:M1
>K2:M2
...
>K8:M8
```

Anschließend werfen wir einen kurzen Blick auf das Log für unsere Partition 0, indem wir das `kafka-dump-log.sh`-Skript verwenden. Mit `--print-data-log` zeigt uns dieser Befehl auch die Inhalte der einzelnen Batches an. Wir sind also in der Lage zu sehen, welche Schlüssel und Werte sich in unseren Nachrichten befinden:

```
$ kafka-dump-log.sh \
  --print-data-log \
  --files ~/kafka/data/kafka1/compaction-test-0/0[...]000.log
Dumping ~/kafka/data/kafka1/compaction-test-0/0[...]000.log
Starting offset: 0
baseOffset: 0 lastOffset: 1 count: 1 [...] position: 0 CreateTime: 1621022716474 size:
72 [...] crc: 1371859366
| offset: 0 CreateTime: 1621022716474 keysize: 2 valuesize: 2 sequence: -1
headerKeys: [] key: K7 payload: M7
```

In unserem Beispiel ist lediglich die Nachricht bestehend aus dem Schlüssel K7 und dem Wert M7 in Partition 0 gelandet. Produzieren wir nun eine weitere Nachricht mit dem Schlüssel K7:

```
$ kafka-console-producer.sh \
  --topic compaction-test \
  --property parse.key=true \
  --property key.separator=: \
  --bootstrap-server localhost:9092
>K7:M9
```

Anschließend schauen wir uns wieder das Log an.

```
$ kafka-dump-log.sh \
--print-data-log \
--files ~/kafka/data/kafka1/compaction-test-0/0[....]000.log
Dumping ~/kafka/data/kafka1/compaction-test-0/0[....]000.log
Starting offset: 0
[...]
| offset: 0 CreateTime: 1621022716474 keysize: 2 valuesize: 2 sequence: -1
headerKeys: [] key: K7 payload: M7
[...]
| offset: 1 CreateTime: 1621023111342 keysize: 2 valuesize: 2 sequence: -1
headerKeys: [] key: K7 payload: M9
```

3.4.4.1 Wann wird ein Log via Compaction aufgeräumt?

Erwartungsgemäß landet die neue Nachricht mit demselben Key ebenfalls in Partition 0, allerdings befindet sich trotz Log Compaction auch immer noch die veraltete Nachricht in unserem Log. Warum ist das so? Hätte die ältere Nachricht nicht gelöscht sein müssen? Dies liegt daran, dass Log Compaction, genau wie Log Retention, nicht permanent stattfindet. Zum einen überprüft unser Log Cleaner standardmäßig nur alle 15 Sekunden (`log.cleaner.backoff.ms`), ob Nachrichten gelöscht werden können. Zum anderen wäre es ziemlich ineffizient (und praktisch sogar unmöglich), wirklich alle 15 Sekunden das gesamte Log nach veralteten Nachrichten zu durchsuchen. Daher nutzt der Log Cleaner weitere Parameter, um festzustellen, ob ein Log compactet werden kann. Über `min.cleanable.dirty.ratio` können wir festlegen, wie groß das Verhältnis zwischen dem sogenannten *dirty* Log und dem gesamten Log mindestens sein muss, damit unser Log Cleaner tätig wird. Ein Segment ist *dirty*, wenn es noch nie compactet wurde. Standardmäßig muss das Verhältnis zwischen *dirty* Segmenten und dem gesamten Log mindestens 0,5 betragen, damit eine Partition erneut compactet wird. Über `log.cleaner.min.cleanable.dirty.ratio` können wir diesen Standardwert anpassen. Umso geringer wir diesen Wert wählen, desto häufiger wird unser Log compactet und desto geringer ist der maximale Speicherplatz, der durch veraltete Nachrichten belegt wird, da maximal nur so viele Nachrichten veraltet sein können, wie wir neue Nachrichten in den nicht compacteten Segmenten haben. Bei einem Wert von 0,5 wird zum Beispiel maximal doppelt so viel Speicherplatz verwendet, wie eigentlich nötig wäre, und bei einem Wert von 0,2 beträgt der Overhead an Speicherplatz maximal 25 Prozent.

Warum also nicht einfach den Wert auf null oder fast null setzen? Immerhin würden wir dadurch sehr viel Speicherplatz sparen. Das hängt damit zusammen, dass Log Compaction sehr aufwendig ist, da bei jeder Log Compaction die gesamte Partition gelesen werden muss und unsere Broker dann nur noch mit Log Compaction beschäftigt wären. Die Log-dirty-Ratio ist allerdings nicht der einzige Parameter, anhand dessen der Log Cleaner überprüft, ob ein Log compactet werden kann oder nicht. Über die Parameter `max.compaction.lag.ms` und `min.compaction.lag.ms` können wir festlegen, nach welcher maximalen Zeitspanne eine Nachricht compactet werden soll beziehungsweise wie lange Nachrichten nicht compactet werden dürfen. Erstes dient dazu, auch Logs, welche keinen hohen Daten durchsatz haben, regelmäßig zu compacten. Letzteres dient dazu sicherzustellen, dass Nachrichten mindestens für einen bestimmten Zeitraum vorgehalten werden, und so den Consumern die Möglichkeit zu geben, alle Nachrichten zu lesen, auch wenn sie theoretisch

bereits veraltet sind. Standardmäßig ist die minimale Zeitspanne auf 0 gesetzt (`log.cleaner.min.compaction.lag.ms=0`), und auch die maximale Zeitspanne ist durch den Standardwert von ca. 300 Millionen Jahren praktisch nicht gesetzt (`log.cleaner.max.compaction.lag.ms=9223372036854775807`). Eine Partition wird vom Log Cleaner dann als geeignet für Compaction angesehen, wenn es uncompactet Nachrichten gibt, die älter sind als der maximale Compaction Lag, oder wenn die dirty-Ratio über dem Grenzwert liegt und es uncompactet Nachrichten gibt, die älter als der minimale Compaction Lag sind. Die Log dirty-Ratio erfüllt auch noch einen weiteren Zweck und dient der Priorisierung von Log Compaction, falls mehrere Partitionen gleichzeitig für Log Compaction geeignet sind.

Kehren wir nun zu unserer ursprünglichen Frage zurück, warum unsere Partition noch nicht compactet wurde. Nach dem, was wir eben gelernt haben, hätte unsere Partition ja bereits compactet werden müssen. Dies hängt damit zusammen, dass der Log Cleaner beziehungsweise Log Compaction in Kafka immer das aktuelle Segment ignorieren. Der Grund dafür ist relativ einfach, denn durch Log Compaction verändern wir Segmente beziehungsweise bauen unsere Partition grundsätzlich neu auf. Würde man die Log Compaction auch auf das aktuelle Segment anwenden, also das Segment, in das wir alle neu produzierten Nachrichten schreiben, könnte das leicht zu Inkonsistenzen führen.

Wir müssen also dafür sorgen, dass ein neues Segment erstellt wird. Wir könnten hierfür natürlich wieder unseren bekannten `segment.ms`-Parameter verwenden, allerdings können wir den gleichen Effekt auch über `max.compaction.lag.ms` erreichen. Durch diesen Parameter wird nämlich auch automatisch ein neues Segment erstellt, falls die älteste Nachricht in einem Segment älter als diese maximale Verzögerungszeit ist. Passen wir also die Config unseres `compaction-test`-Topics an und setzen `max.compaction.lag.ms` auf 60 Sekunden:

```
$ kafka-configs.sh \
  --alter \
  --topic compaction-test \
  --add-config max.compaction.lag.ms=60000 \
  --bootstrap-server=localhost:9092
```

Das reicht allerdings noch nicht aus, um ein neues Segment zu erzeugen. Ähnlich zu `segment.ms` müssen wir zunächst noch eine neue Nachricht erzeugen, da erst dann überprüft wird, ob ein neues Segment erzeugt werden muss. Also müssen wir wohl wieder unseren `kafka-console-producer.sh` bemühen und erzeugen eine weitere Nachricht für unsere Partition:

```
$ kafka-console-producer.sh \
  --topic compaction-test \
  --property parse.key=true \
  --property key.separator=: \
  --bootstrap-server localhost:9092
>K7:M10
```

Werfen wir nun wieder einen Blick auf unsere Partition:

```
$ ls -1 ~/kafka/data/kafka1/compaction-test-0
00000000000000000000.index
00000000000000000000.log
00000000000000000000.timeindex
00000000000000000002.index
00000000000000000002.log
```

```
00000000000000000000000000000002.snapshot
00000000000000000000000000000002.timeindex
leader-epoch-checkpoint
```

Wenig überraschend wurde ein neues Segment erstellt.

3.4.4.2 Wie funktioniert der Log Cleaner?

Wer an dieser Stelle besonders schnell war, konnte sogar den Log Cleaner live miterleben und hat gesehen, dass das ursprüngliche Segment mit dem Start Offset 0 zunächst zum Löschen markiert (Dateiendung `.delete` wurde hinzugefügt) und anschließend gelöscht wurde. Dies hängt mit der Funktionsweise unseres Log Cleaners zusammen. Dieser schaut zunächst in unsere nicht-compacteten Segmente und speichert für jeden Schlüssel den neuesten Offset. Anschließend beginnt er von der ältesten Nachricht an zu lesen. Er ignoriert hierbei Schlüssel beziehungsweise Nachrichten, welche veraltet sind, das heißt Schlüssel, zu denen es einen neueren Eintrag gibt, und schreibt die bereinigten Nachrichten in ein neues Segment. Sobald dieses neue Segment voll ist, werden das alte oder die alten Segmente durch das neue saubere Segment ersetzt. Hieraus ergibt sich auch der durch den Log Cleaner maximal zusätzlich nötige Speicherplatz von nur einem Segment. Wie sieht nun aber der Inhalt unseres Logs aus:

```
$ kafka-dump-log.sh \
  --print-data-log \
  --files ~/kafka/data/kafka1/compaction-test-0/0[...]000.log
Dumping ~/kafka/data/kafka1/compaction-test-0/0[...]000.log
Starting offset: 0
[...]
| offset: 1 CreateTime: 1621023111342keysize: 2 valuesize: 2 sequence: -1 headerKeys:
[] key: K7 payload: M9
```

Unsere Nachricht mit dem Wert *M7* wurde tatsächlich gelöscht, die Nachricht *M9* ist aber immer noch da, das hängt damit zusammen, dass das neueste Segment nicht compactet wird und somit komplett außen vor ist. Produzieren wir kurz hintereinander noch zwei weitere Nachrichten, um uns dies noch etwas genauer anzusehen. Anschließend warten wir kurz, damit der Log Cleaner laufen kann:

```
$ kafka-console-producer.sh \
  --topic compaction-test \
  --property parse.key=true \
  --property key.separator=: \
  --bootstrap-server localhost:9092
>K7:M11
>K7:M12
```

Wenn wir einen Blick in unsere Partition werfen, werden wir wenig überraschend feststellen, dass unser Log erneut rotiert wurde:

```
$ ls -1 ~/kafka/data/kafka1/compaction-test-0
00000000000000000000000000000000.index
00000000000000000000000000000000.log
00000000000000000000000000000000.timeindex
00000000000000000000000000000003.index
00000000000000000000000000000003.log
00000000000000000000000000000003.snapshot
00000000000000000000000000000003.timeindex
```

Schauen wir uns noch kurz mit `kafka-dump-log.sh` die Inhalte unserer beiden Segmente an:

```
$ kafka-dump-log.sh \
  --print-data-log \
  --files ~/kafka/data/kafka1/compaction-test-0/0[...]000.log
Dumping ~/kafka/data/kafka1/compaction-test-0/0[...]000.log
Starting offset: 3
[...]
| offset: 3 CreateTime: 1621025578468 keysize: 2 valuesize: 2 sequence: -1
headerKeys: [] key: K7 payload: M11
[...]
| offset: 4 CreateTime: 1621025580522 keysize: 2 valuesize: 2 sequence: -1
headerKeys: [] key: K7 payload: M12
```

Unser aktuelles Segment enthält unsere beiden eben produzierten Nachrichten. Es wurde also nicht compactet, da wir ansonsten lediglich die neueste Nachricht dort sehen würden. In unserem neuesten Segment können sich demzufolge beliebig viele Nachrichten zu einem Key befinden:

```
$ kafka-dump-log.sh \
  --print-data-log \
  --files ~/kafka/data/kafka1/compaction-test-0/0[...]000.log
Dumping ~/kafka/data/kafka1/compaction-test-0/0[...]000.log
Starting offset: 0
[...]
| offset: 2 CreateTime: 1621024938523 keysize: 2 valuesize: 2 sequence: -1
headerKeys: [] key: K7 payload: M10
```

Unser compacted Segment enthält nun die zuvor produzierte Nachricht *M10* und sonst keine weitere Nachricht zu diesem Schlüssel. Auch wenn wir hier noch weitere Segmente hätten, würden wir in allen compacted Segmenten maximal eine Nachricht pro Schlüssel finden.

Eine weitere wichtige Eigenschaft von Log Compaction haben wir bisher noch nicht explizit erwähnt. Obwohl während Log Compaction neue Segmente erstellt werden, werden die Offsets und auch die Reihenfolge unserer Nachrichten nicht verändert. Dies ist insofern zwar logisch, da wir während der Log Compaction im Prinzip lediglich veraltete Einträge löschen und ansonsten die Segmente nur zusammenführen. Allerdings ist diese Eigenschaft aus Konsistenzgründen enorm wichtig! An dieser Stelle müssen wir uns zwangsläufig noch die Frage stellen, was eigentlich passiert, wenn ein Consumer versucht, nicht vorhandene Offsets beziehungsweise Nachrichten zu lesen. Selbst in unserem kleinen Beispiel wäre dies bereits der Fall, da unser erstes Segment als Start-Offset trotz gelöschter Nachrichten immer noch 0 hat. Kafka hat auch hier eine einfache Lösung parat. Wenn ein Offset nicht vorhanden ist, wird einfach zur nächsten vorhandenen Nachricht gesprungen. Wenn ein Consumer also die Nachricht mit dem Offset 0 anfordert, ist diese Anfrage äquivalent zur Anforderung der Nachricht mit Offset 1.

3.4.4.3 Tombstones

Log Compaction bringt noch ein weiteres Feature mit. Wir können nicht nur veraltete Daten überschreiben, sondern sogar gezielt Daten löschen. Hierfür müssen wir lediglich einen sogenannten Tombstone (zu Deutsch Grabstein) erzeugen, indem wir eine Nachricht mit null Payload erzeugen. Log Compaction löscht dann wie gewohnt die zu dem Schlüssel

dazugehörigen veralteten Nachrichten. Die Besonderheit hierbei gegenüber der normalen Log Compaction ist, dass die Tombstone-Nachricht selbst nach einiger Zeit gelöscht wird und wir somit keinerlei Nachrichten zu diesem bestimmten Key mehr in unserem Log finden. Die Zeit, nach der ein Tombstone von Kafka beziehungsweise dem Log Cleaner gelöscht wird, beträgt standardmäßig einen Tag und kann über die Parameter `delete.retention.ms` für ein Topic beziehungsweise über `log.cleaner.delete.retention.ms` für unser gesamtes Cluster verändert werden. Der Grund, warum die Tombstones nicht sofort gelöscht werden, ist, dass wir den Consumern die Möglichkeit geben wollen, von der Löschung unserer Daten zu erfahren und so eine valide Momentaufnahme unseres letzten Standes zu erhalten, damit sie die Löschung in ihren eigenen Prozessen entsprechend verarbeiten können. Der Grund, warum Kafka Tombstones überhaupt löscht, ist, dass unser Cluster ansonsten nach einiger Zeit sehr viele Segmente hätte, welche nur aus Tombstones bestehen, und dies unser Log unnötig aufblättern würde.

3.4.5 Zusammenfassung

In diesem Kapitel haben wir uns ausführlich mit dem Aufräumen von Nachrichten beschäftigt. Wir haben zunächst über die Gründe, warum wir überhaupt Nachrichten aufräumen beziehungsweise löschen müssen, gesprochen. Anschließend haben wir ausführlich die beiden Ansätze in Kafka zum Löschen von Nachrichten behandelt. Wir haben gelernt, was genau Log Retention ist, wie genau diese in Kafka funktioniert und welche Möglichkeiten wir haben, um Log Retention in Kafka zu konfigurieren. Hierbei sind wir nicht nur auf die Löschung von Nachrichten eingegangen, sondern haben uns auch angesehen, wie Offsets aufgeräumt werden. Zum Abschluss haben wir auch einen detaillierten Blick auf Log Compaction geworfen, die zweite Möglichkeit, Nachrichten in Kafka aufzuräumen. Hier haben wir uns ebenfalls ausführlich die Funktionsweise und Konfigurationsmöglichkeiten angesehen. Außerdem haben wir erfahren, dass wir mit Log Compaction durch sogenannte Tombstones auch die Möglichkeit haben, gezielt Daten in Kafka zu löschen.

■ 3.5 Cluster-Management

Wir haben in den ersten Kapiteln Kafka als verteiltes System kennengelernt und auch mehrfach angesprochen, dass Koordination in solchen Systemen immer mit Herausforderungen verbunden ist. Diese Probleme betreffen nicht nur Kafka an sich, sondern auch unsere Consumer, wenn wir Consumer Groups verwenden. Glücklicherweise nimmt uns Kafka sehr viel Arbeit ab, und wir müssen uns keine Gedanken machen, wie wir die Partitionen zuverlässig auf unsere Consumer verteilen.

Aber managen wir Kafka selbst? Und wofür benötigen wir überhaupt Cluster-Management? Die Kernherausforderung, die Cluster-Management-Systeme lösen, ist es, einen Konsens in verteilten Systemen zu finden. Solange wir nur einen Broker haben, ist dies einfach. Wir müssen uns mit niemandem einigen, sondern dieser Broker ist der Leader für alle Partitionen, ist immer in-sync, ist Group Coordinator für alle Consumer Groups und so weiter. Aber

wie stellen wir sicher, dass wir einen konsistenten Zustand haben, wenn wir weitere Broker hinzufügen? Wer verteilt dann die Partitionen auf die Broker, wer beobachtet den Gesundheitszustand der Broker, und wer wirft Broker aus dem Cluster, wenn sie sich nicht melden? Wer speichert die aktuelle Konfiguration des Clusters?

In Kafka gibt es einen Broker, der sich als Controller um diese Metadaten kümmert und allen Brokern Anweisungen gibt. Nun können wir uns wieder die Frage stellen, wie wir diesen Controller auswählen. Wir könnten natürlich beim Starten von Kafka einen Controller festlegen, aber was passiert, wenn dieser ausfällt oder zu langsam ist oder, oder, oder?

Während wir dieses Buch schreiben, befindet sich die Kafka-Welt im Umbruch. Die bisherige Art und Weise, wie Kafka die Metadaten verwaltet und Konsens findet, ändert sich massiv. Der bisherige Ansatz, dafür *Apache Zookeeper*⁶ zu verwenden, wird mit dem inzwischen umgesetzten KIP-500⁷ durch einen neuen Ansatz ersetzt. Da wir glauben, dass ältere, auf Zookeeper basierende Kafka-Versionen noch längere Zeit im Einsatz sein werden, besprechen wir hier beide Ansätze.

3.5.1 Zookeeper-basiertes Cluster-Management

Bis Version 3.0 nutzt Kafka das Zookeeper-Projekt für das Cluster-Management. Apache Zookeeper ist ein eigenständiges Apache-Projekt und wird oder wurde von zahlreichen Softwareprodukten wie Apache Hadoop MapReduce oder *Neo4j* benutzt. Zookeeper selbst fühlt sich in der Benutzung zuerst wie ein hierarchischer Key-Value Store an. Wir können, ähnlich den Verzeichnissen und Dateien bei Betriebssystemen, sogenannte ZNodes (Knoten) anlegen, auslesen und verändern. Aber der Clou bei Zookeeper ist, dass Zookeeper garantiert, dass alle Zookeeper-Knoten immer genau den gleichen Zustand kennen. Das heißt, wenn wir zum Beispiel versuchen, mit mehreren Clients gleichzeitig einen Zookeeper ZNode zu schreiben, wird es nur einen Gewinner geben, und alle unsere Clients sehen genau das gleiche Ergebnis. Das nutzen wir zum Beispiel in Kafka, um den Controller zu bestimmen. Wenn es keinen Controller ZNode im Key-Value Store von Zookeeper gibt, versuchen alle Broker, ihre eigene ID in diesen ZNode zu schreiben, und wer Zookeeper überzeugen konnte, wird der neue Controller. Weiterhin kann Zookeeper auch überwachen, welche Clients aktuell noch verbunden sind, und somit können wir in Kafka feststellen, wenn ein Broker ausfällt.

Um zuverlässig zu erkennen, wer ausgefallen ist, basiert Zookeeper auf dem sogenannten *Paxos- Protokoll*. Um einen Konsens zu finden, muss sich im sogenannten Zookeeper- Ensemble die Mehrheit auf einen Wert einigen, und die unterlegenen Knoten mit anderer oder keiner Meinung akzeptieren die vorherrschende Meinung. Da die Gesamtanzahl der Mitglieder bekannt ist, können wir auch Ausfälle einzelner Mitglieder verkraften. Wenn wir mit drei Zookeeper-Knoten starten, kann ein Knoten ohne Konsequenzen ausfallen, da die anderen zwei einen Konsens finden können. Wenn wir fünf Zookeeper-Knoten haben, dann können sogar zwei ausfallen. Wir empfehlen, nicht mehr als fünf Zookeeper-Knoten anzulegen, da mehr zu Performance-Problemen führen würden. Wichtig ist dabei immer zu

⁶⁾ <https://zookeeper.apache.org/>

⁷⁾ <https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum>

beachten, dass die Anzahl der Zookeeper-Knoten keinen Einfluss auf die Anzahl der Broker hat. Wir können also sowohl drei Zookeeper-Knoten haben und nur einen Broker als auch andersherum.

Wie nutzt aber Kafka Zookeeper? Wir haben schon angedeutet, dass wir Zookeeper zur Bestimmung des Controllers benutzen und auch die auf Kafka-Cluster bezogenen Metadaten in Zookeeper speichern. Konkret speichern wir im Zookeeper ab, welche Partitionen welchen Brokern zugeordnet sind und welcher Broker Leader für welche Partitionen ist. Partitions-Leader schreiben in Zookeeper, welche Follower gerade in-sync sind. Da wir aber auch ACLs und andere sicherheitsrelevante Informationen in Zookeeper ablegen, müssen wir Zookeeper zusätzlich absichern, aber dazu später im Kapitel „Kafka-Referenzarchitektur“ mehr.

In Bild 3.16 haben wir einen beispielhaften Kafka-Cluster mit Zookeeper abgebildet. Wir sehen, dass wir ein Zookeeper-Ensemble aus drei Zookeeper-Knoten und vier Kafka-Brokern haben. Der Controller ist lediglich einer dieser Broker und übernimmt zusätzlich gewisse Management-Aufgaben. Wenn dieser Controller ausfällt, übernimmt ein anderer Broker. Wichtig ist zu erwähnen, dass auch andere Broker mit Zookeeper direkt reden und wir hier mit den Pfeilen nur die Hauptkommunikationspfade eingezeichnet haben.

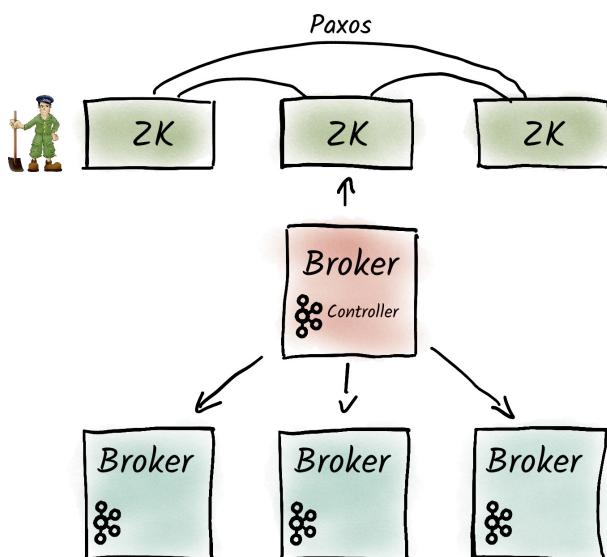


Bild 3.16 Vor Version 3.0 bestand ein Kafka-Cluster aus mehreren Brokern und zusätzlich dazu einem Zookeeper-Ensemble bestehend aus drei oder fünf Zookeeper-Knoten. Der Controller war dabei lediglich einer der Broker. Fiel der Controller aus, übernahm ein anderer Broker diese Aufgaben.

Ohne Zookeeper ist es zweifelhaft, ob Apache Kafka in der jetzigen Form so entstanden wäre, da Zookeeper sehr viele komplizierte Aufgaben übernimmt, die nicht ohne Weiteres selbst zu implementieren sind. Doch es bringt auch viele Probleme, sich auf Zookeeper zu verlassen. Zookeeper ist, dem Grundproblem geschuldet, eher sehr langsam und behindert an vielen Stellen Kafkas Performance. Für Administratoren ist Zookeeper oft eine Last, da

dieses weitere, für die meisten Administratoren unbekannte System für nicht unerheblichen Mehraufwand sorgt. Vor allem propagiert Kafka das Log als gute Datenstruktur, um Nachrichten zwischen Systemen auszutauschen, aber speichert seine eigenen Metadaten in einem anderen System ab. Dies zusammen mit den Performance-Problemen führt dazu, dass die Broker viele Informationen lokal vorhalten und es in seltenen Fällen zu Konsistenzproblemen führen kann, da Broker, insbesondere der Controller, zu spät (oder gar nicht) von Änderungen in Zookeeper erfährt und falsche Entscheidungen trifft. Dies liegt daran, dass es einige Operationen gibt, die am Controller vorbei direkt von den Brokern in Zookeeper vorgenommen werden und aus Performance-Gründen der Controller aktiv nach solchen Änderungen nachfragen muss und so manchmal wichtige Änderungen verpasst.

3.5.2 KRaft-basiertes Cluster-Management

Deswegen wurde mit der Kafka Version 3.0 ein neues System eingeführt, welches Zookeeper ersetzt. Die Idee ist, dass ausgewählte Kafka-Broker oder ein eigenständiger kleiner Kafka-Management-Cluster die Aufgaben des Zookeepers und auch die des Controllers übernimmt. Diese ausgewählten Kafka-Broker wählen untereinander einen aktiven Controller aus, aber stehen gleichzeitig als Stand-by-Controller bereit. Die Metadaten schreibt Kafka nicht mehr in Zookeeper, sondern in ein spezielles internes Metadaten-Topic `__cluster_metadata`. Dieses Topic wird auf alle Broker repliziert, damit daraus alle Broker ihren eigenen Zustand erfahren können und sich nicht über eigenständige Aufrufe über ihre Aufgaben informieren müssen. Sobald der Koordinationscluster feststellt, dass der Controller nicht mehr verfügbar ist, wird ein neuer Controller ausgewählt. Im Gegensatz zu Zookeeper ist dafür keine langsame Synchronisation notwendig, sondern dieser neue Controller, der das Metadaten-Topic lokal repliziert hat, kann sofort übernehmen. Dieser Ansatz wurde im Detail in KIP-500 ausgeführt und im Verlauf von über einem Jahr Stück für Stück in der Kafka-Software umgesetzt. Statt auf dem Paxos-Protokoll zu basieren, nutzt Kafka für die Koordination nun das Raft-Protokoll⁸⁾. Daher kommt der Name *KRaft*.

Mit diesem neuen Ansatz können wir nun in Entwicklungsumgebungen einen einzigen Kafka-Broker benutzen, der gleichzeitig Controller, aber auch normaler Broker ist. Dies vereinfacht insbesondere den Einstieg für neue Nutzer. In Zukunft soll es sogar möglich sein, Produktiv-Cluster in diesem Ein-Broker-Modus zuverlässig zu betreiben. Das ist insbesondere für Umgebungen mit wenig verfügbaren Ressourcen wie zum Beispiel beim Edge-Computing attraktiv.

In kleinen Clustern mit drei Brokern benötigen wir in vielen Fällen keinen Koordinationscluster à la Zookeeper, sondern können die Broker so konfigurieren, dass sie gleichzeitig Controller-Broker als auch normale Broker sind. In größeren Umgebungen gibt es meistens drei oder fünf eigenständige Broker, die ausschließlich die Controller-Aufgaben übernehmen, und die restlichen Broker sind für die normalen Broker-Aufgaben zuständig.

⁸⁾ In Search of an Understandable Consensus Algorithm: <https://raft.github.io/raft.pdf>

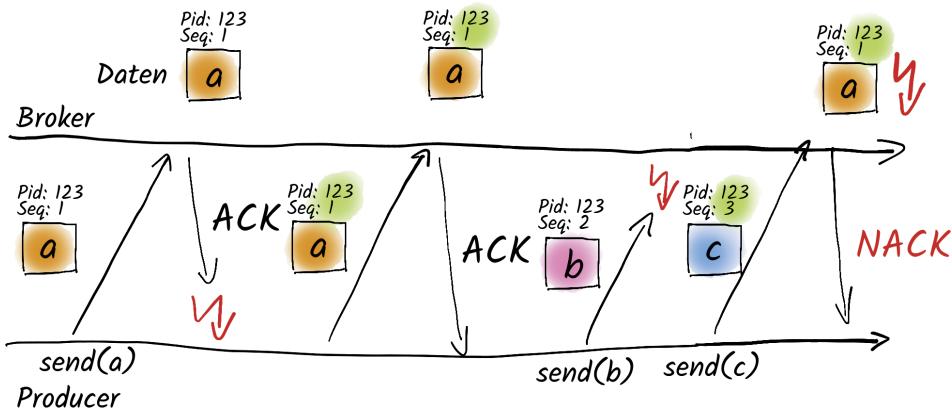


Bild 3.17 Mit Kafka 3.0 wurde Zookeeper durch einen auf Raft basierten Koordinationscluster ersetzt.

Wir können aus eigener Erfahrung sagen, dass dies ein sehr lang ersehntes neues Feature in Kafka ist, und einige unserer Kunden warten seit Jahren darauf, endlich Zookeeper loszuwerden. Mit Kafka 3.0 ist dies nun Realität geworden, und damit wird nicht nur Kafkas Betriebskomplexität reduziert, sondern es ist nun möglich, deutlich größere Kafka-Cluster zu betreiben, da die empfohlene Limitierung auf 200 000 Partitionen pro Cluster wegfällt und wir auch an vielen Stellen deutliche Performance-Verbesserungen erwarten.

3.5.3 Zusammenfassung

In diesem Kapitel haben wir uns mit dem Cluster-Management von Kafka befasst. Zum einen haben wir gelernt, wie das Management von Kafka via Zookeeper funktioniert. Wir haben auch die damit verbundenen Performance-Probleme kurz angerissen, was auch dazu geführt hat, dass Kafka ab Version 3.0 das Cluster-Management umgestellt hat, das seitdem auf KRaft basiert. Diese neue Methode haben wir dann im zweiten Abschnitt dieses Kapitels angerissen und sind dabei auf die Vorteile gegenüber Zookeeper eingegangen.

■ 3.6 Verarbeitungsgarantien und Transaktionen

Wir haben im Kapitel „Zuverlässigkeit“ bereits erfahren, wie wir unsere Verarbeitungsgarantien in Kafka auf unseren Anwendungsfall anpassen können. Im Optimalfall möchten wir, dass unsere Nachrichten genau einmal verarbeitet werden. Aber das ist in verteilten Systemen gar nicht so einfach sicherzustellen, im Gegenteil, im Allgemeinen sogar unmöglich zu garantieren. Aber es ist in einigen Fällen gar nicht so wichtig, dass jede Nachricht verarbeitet wird. Wenn wir zum Beispiel im Sekundentakt Daten von Wetterstationen ab-

greifen, reicht es vielleicht sogar aus, dass die meisten Nachrichten verarbeitet werden. Wenn ein paar Messdaten fehlen, oder mehrfach verarbeitet werden, mag das unangenehm sein, aber nicht schlimm.

Wenn wir stärkere Garantien haben möchten, müssen wir uns klassischerweise zwischen der At-most-once- und der At-least-once-Verarbeitung entscheiden. At-most-once-Verarbeitung garantiert uns, dass jede Nachricht höchstens einmal verarbeitet wird. Das heißt, es kann passieren, dass wir Nachrichten verlieren, aber das System garantiert uns, dass keine Nachrichten doppelt verarbeitet werden. Diese Garantie wird oft in Systemen mit harten Echtzeitanforderungen, wie zum Beispiel beim Verarbeiten von Video-Streams, benutzt. Es ist keine Katastrophe, wenn ein Frame ausgelassen wird, das merken die Zuschauer nicht, aber wenn ein nicht dazugehöriger Frame später erneut auftaucht, ist das ungünstig. In manchen Fällen brauchen wir nicht mal diese harte Garantie, dass wir Duplikate verbieten, sondern es reicht uns aus, davon auszugehen, dass nicht jede Nachricht ankommt.

In der Kafka-Welt haben wir es häufiger mit der sogenannten At-least-once-Garantie zu tun. Wir möchten garantieren, dass jede Nachricht mindestens einmal ankommt, aber unsere nachfolgenden Systeme sind darauf ausgelegt, dass sie mit Duplikaten klarkommen.

Exactly-once ist der Heilige Gral verteilter Systeme. Im Allgemeinen können wir diese Anforderung nicht abbilden, aber wir kennen zum Beispiel von Datenbanken, oder aus dem Bereich der parallelen Programmierung Möglichkeiten, dennoch Daten genau einmal zu verarbeiten.

3.6.1 Idempotenz

Die einfachste Art und Weise, Daten genau einmal zu verarbeiten, ist es, ein System zu benutzen, welches uns eine At-least-once-Garantie anbietet, und basierend darauf Duplikate herauszufiltern. Dabei helfen uns sogenannte idempotente Funktionen. Eine Funktion ist idempotent, wenn sie für beliebige Eingangsparameter auch bei mehrfacher Ausführung ihrer selbst hintereinander gleiche Ausgabewerte liefert. Das mag sich am Anfang trivial anhören, aber spätestens, wenn wir Kreditkartentransaktionen durchführen möchten, wird es komplizierter.

Bis vor Version 3.0 bietet uns Kafka standardmäßig nur die Garantie, dass Nachrichten in eine Partition in derselben Reihenfolge geschrieben werden, wie sie ankommen. Um zu garantieren, dass Nachrichten auch im Fehlerfall korrekt ankommen, haben wir im Producer die Einstellung `acks=all` und einen sinnvollen `min.insync.replicas` Wert benutzt. Damit können wir garantieren, dass jede Nachricht, die ein Producer erfolgreich produziert, mindestens einmal in einer Partition abgelegt wird. Wir können dabei aber weder garantieren, dass Nachrichten in der richtigen Reihenfolge geschrieben werden, noch, dass sie nicht mehrmals ankommen. Um dies zu erreichen, haben wir `enable.idempotence=true` gesetzt. Damit können wir garantieren, dass jede Nachricht, die ein Producer erfolgreich produziert, genau einmal und in der richtigen Reihenfolge in der Partition auftaucht. Natürlich kann uns hier Kafka nicht helfen, wenn in der Zwischenzeit der Producer abstürzt. Mit Kafka 3.0 wurden neue Standardkonfigurationen eingeführt: `acks=all` und `enable.idempotence=true`.

Intern setzt Kafka dies mit sogenannten Sequence-IDs und Producer-IDs um. Zu jeder Nachricht schickt der Producer seine eindeutige ID an den Broker und zusätzlich dazu eine sequenzielle ID für jede Partition. Damit sieht der Broker, ob die Nachricht schon geschrieben wurde, und kann damit ein ACK schicken, ohne die Nachricht erneut zu schreiben. Aber der Broker kann auch feststellen, wenn Nachrichten in der falschen Reihenfolge ankommen (weil die Sequenz-ID eine Lücke aufweist) und diese Nachrichten dann abweisen. Der Performance-Overhead ist vernachlässigbar, und deshalb wird ab Kafka 3.0 dieses Verhalten auch zum Standard erklärt.

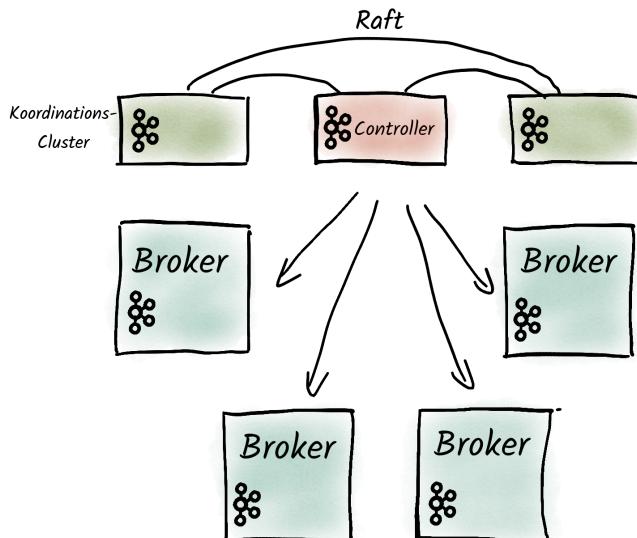


Bild 3.18 Ein häufiger Grund dafür, dass Nachrichten in einer Partition doppelt auftauchten, war, dass das ACK für eine Nachricht zum Beispiel aufgrund eines Netzwerkproblems nicht ankam und der Producer demzufolge die Nachricht erneut abschickte. Ab Kafka 3.0 ist Idempotenz standardmäßig im Producer aktiviert, und so wird dies verhindert. Der Producer verschickt dabei mit jeder Nachricht seine eindeutige Producer-ID und eine Sequenznummer. Dadurch können wir auch die Reihenfolge von Nachrichten sicherstellen und vermeiden, dass zum Beispiel die Nachricht c vor b ankommt.

Mit der Idempotenz können wir nun sicherstellen, dass Nachrichten in einer Partition in der richtigen Reihenfolge geschrieben werden und auch genau einmal vorhanden sind.

3.6.2 Transaktionen

Nun ist es natürlich nett, dass die Nachrichten korrekt in einer Partition abgelegt werden, aber oft möchten wir diese Daten auch weiterverarbeiten. Die Producer-Idempotenz hat keinen Einfluss auf unsere Consumer oder gar Drittanwendungen. Auf dem Weg dorthin können Nachrichtenduplikate oder Nachrichtenverluste dennoch passieren, falls wir nicht aufpassen. Wenn wir uns den Lesevorgang in Kafka genauer anschauen, stellen wir fest, dass jeder Lesevorgang auch gleichzeitig ein Schreibvorgang ist, denn wir müssen unseren

Offset committen. Kafka bietet uns die Möglichkeit an, unsere Offsets in Kafka abzuspeichern, dies ist aber nicht verpflichtend.

3.6.2.1 Transaktionen in Datenbanken

Wenn wir zum Beispiel Daten aus Kafka genau einmal in ein Datenbanksystem übertragen möchten, haben wir zwei Möglichkeiten, um exactly-once zu gewährleisten: Wir können ganz normal unseren Kafka Consumer weiternutzen und committen unsere Offsets nach Kafka, indem wir die Daten zuerst in die Datenbank schreiben und erst danach den Offset committen. Dann müssen wir aber garantieren, dass der Schreibvorgang in die Datenbank idempotent ist. Das bedeutet, wir überprüfen vor jedem Schreibvorgang, ob die Daten bereits vorhanden sind, und schreiben sie nur in dem Fall, dass sie noch nicht da sind. Am einfachsten können wir das über sogenannte *UPSERT*-Statements erreichen. In PostgreSQL könnte dies folgendermaßen aussehen:

```
INSERT INTO flugdaten (id, timestamp, state)
    VALUES (...)

    ON CONFLICT DO NOTHING;
```

Eine Alternative dazu ist es, die Offsets in der Datenbank selbst und nicht in Kafka zu speichern. Wenn wir nun die Daten zusammen mit dem Offset in einer Transaktion schreiben, können wir garantieren, dass jede Nachricht aus Kafka genau einmal in der Datenbank auftaucht, da wir die Offsets mit den Daten zusammen atomar schreiben. In PostgreSQL könnten wir das folgendermaßen umsetzen:

```
BEGIN TRANSACTION;
INSERT INTO flugdaten (id, timestamp, state)
    VALUES (...);

INSERT INTO offsets (topic, partition, offset)
    VALUES ('flugdaten', 0, 123);

COMMIT;
```

Aber was können wir tun, wenn wir Daten aus einem Kafka-Topic lesen möchten, diese verarbeiten und dann in ein anderes Kafka-Topic schreiben möchten?

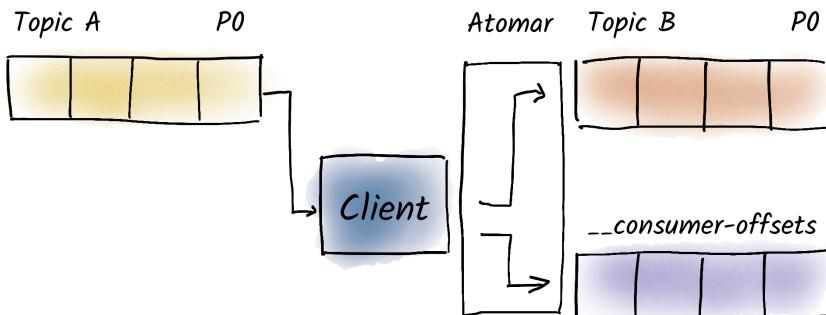


Bild 3.19 Transaktionen in Kafka ermöglichen uns, atomar in zwei oder mehr Partitionen zu schreiben. Entweder werden alle Nachrichten erfolgreich geschrieben oder keine.

Wir bräuchten auch hier eine Möglichkeit, atomar in mehrere Partitionen zu schreiben; und zwar in die Partitionen unserer Zieltopics und die Partitionen des `__consumer_offsets`-Schemas.

Topics. Dieser Prozess des Lesens von Daten, ihrer Verarbeitung und des Zurückschreibens in ein anderes Topic ist so häufig, dass es mit *Kafka Streams* eine Bibliothek gibt, um dies so einfach wie möglich zu machen.

3.6.2.2 Transaktionen in Kafka

Um Transaktionen in Kafka zuverlässig umzusetzen, benötigen wir zwei Zutaten. Erstens brauchen wir eine Methode, um zuverlässig Nachrichten in einzelne Partitionen zu produzieren. Das erreichen wir mithilfe idempotenter Producer. Zweitens benötigen wir eine Möglichkeit, atomar Nachrichten in mehrere Partitionen gleichzeitig zu schreiben. Das heißt, dass entweder alle Nachrichten in einer Transaktion erfolgreich geschrieben werden oder keine. Transaktionen können wir in unserem Code ähnlich wie in einer Datenbank benutzen. Wir müssen zuerst dem Producer mitteilen, dass wir Transaktionen benutzen möchten. Anschließend starten wir die Transaktion, schreiben unsere Nachrichten, und eventuell committen wir unseren Offset. Zuletzt committen wir die Transaktion selbst, um sie erfolgreich abzuschließen. Erst dann dürfen Consumer die geschriebenen Nachrichten verarbeiten. Wenn wir eine Transaktion abbrechen, löschen wir die geschriebenen Nachrichten nicht aus den Partitionen, sondern markieren sie so, dass die Consumer sie ignorieren.

Das klassische Beispiel für Transaktionen ist es, dass ein Benutzer (zum Beispiel Bob) einer anderen Benutzerin (zum Beispiel Alice) einen Geldbetrag (zum Beispiel 10 €) überweisen möchte. Dafür müssen wir Bobs Kontostand um 10 € reduzieren und Alice' Kontostand um 10 € erhöhen. Beides sollte atomar passieren, also entweder die Überweisung ist erfolgreich, und beide Konten wurden angepasst, oder es ist etwas schiefgelaufen, und keines der beiden Konten wurde angepasst. In Kafka könnten wir das zum Beispiel mit folgendem Python-Code umsetzen:

```
producer = Producer({
    'bootstrap.servers': 'localhost:9092',
    'partitioner': 'murmur2_random',
    'transactional.id': 'ueberweisungen-1',
})
producer.init_transactions()
producer.begin_transaction()
producer.produce("transaktionen",
    key="bob", value="-10€")
producer.produce("transaktionen",
    key="alice", value="+10€")
producer.commit_transaction()
```

Wir initialisieren zuerst unseren Producer mit dem uns bekannten `bootstrap.server` und dem Partitioner `murmur2_random`, um kompatibel zu Java zu bleiben. Zusätzlich müssen wir eine eindeutige `transactional.id` für diesen Producer angeben. Wir müssen dafür sicherstellen, dass zu keinem Zeitpunkt zwei Producer mit der gleichen `transactional.id` laufen, da sie sich dann in die Quere kämen. Bevor wir Transaktionen benutzen können, müssen wir den Producer zuerst mit `init_transactions()` initialisieren. Dann starten wir die Transaktion (`begin_transaction()`), schreiben unsere Nachrichten (`produce()`) und können zu guter Letzt unsere Transaktion erfolgreich beenden (`commit_transaction()`). UmOffsets in der Transaktion zu committen, müssen wir im Producer die Funktion `send_offsets_to_transaction()` verwenden. Um eine Transaktion abzubrechen, rufen wir `abort_transaction()` auf. Wichtig ist hier, dass wirklich der Producer diese

Funktion aufrufen muss, nicht der Consumer. Denn der Producer befindet sich in der Transaktion, der Consumer weiß davon nichts.

3.6.2.3 Transaktionen und Consumer

Ganz wichtig ist es, dass wir im Consumer die Einstellung `isolation.level` auf `read_committed` setzen! Sonst würde unser Consumer nicht-committete Nachrichten lesen, und unser Einsatz von Transaktionen wäre ohne Sinn.

Warum müssen wir aber das Isolation-Level setzen, und wie funktionieren diese Transaktionen überhaupt? Kafka erfindet auch hier das Rad nicht neu, sondern benutzt eine Abwandlung des *2-Phase-Commit-Protokolls*⁹⁾, um Nachrichten atomar in mehrere Partitionen zu schreiben. Wir können uns das Vorgehen wie bei einer Reisebuchung vorstellen. Wir möchten ein Hotel und einen Flug gleichzeitig buchen, aber wissen, dass dieser Prozess manchmal fehleranfällig ist. Wir möchten am Ende nicht mit einem gebuchten Hotel, aber ohne Flug dastehen oder andersrum. Dafür rufen wir zuerst beim Hotel an und reservieren uns ein Hotelzimmer. Das Hotel möchte natürlich nicht für immer warten, sondern wir sagen dem Hotel, dass wir innerhalb von zehn Minuten anrufen, um die Reservierung zu bestätigen. Das Hotel geht darauf ein. Dann buchen wir das Flugticket ganz normal. Falls dies nicht funktioniert, rufen wir beim Hotel an und stornieren das Zimmer wieder oder warten einfach zehn Minuten, bis es das Zimmer automatisch storniert. Wenn die Flugbuchung erfolgreich war, rufen wir beim Hotel an und bestätigen die Reservierung.

Im Groben ist dies ein ähnliches Vorgehen, wie es Kafka bei den Transaktionen umsetzt. Zuerst melden wir bei einem der Broker, der unsere Transaktion koordiniert, die Partitionen an, in die wir schreiben möchten. Dann schreiben wir ganz normal in diese Partitionen, aber markieren die Nachrichten als *Unter Vorbehalt*. Sobald wir mit der Transaktion fertig sind, melden wir das dem Transaktionskoordinator und dieser schreibt sogenannte Commit-Marker in die Partitionen, um die Transaktion zu bestätigen. Sollten wir die Transaktion abbrechen, schreibt der Transaktionskoordinator Abort-Marker.

Die Consumer lesen ganz normal alle Nachrichten, aber wenn sie feststellen, dass eine Nachricht zu einer noch nicht abgeschlossenen Transaktion gehört, blockieren sie das Konsumieren, bis die Transaktion abgeschlossen wird. Erst dann kann unser Code diese Nachrichten lesen. Falls die Transaktion abgebrochen wird, ignoriert die Kafka-Bibliothek diese Nachrichten, und unser Code bekommt die Nachrichten nicht zu Gesicht. Hier ist es noch mal wichtig zu erwähnen, dass wir im Consumer unbedingt den `isolation.level` auf `read_committed` setzen müssen, da der Consumer sonst die Marker ignoriert und alle Nachrichten an unseren Code weiterleitet, unabhängig davon, ob die Transaktion noch läuft, abgebrochen ist oder doch erfolgreich war. Der Transaktionskoordinator garantiert uns, dass die Transaktionen zuverlässig funktionieren und dass auch bei Abstürzen unserer Kafka-Broker oder unserer Producer die Garantien nicht verletzt werden.

Dies bringt gewissen Performance-Implikationen mit sich. Da der Transaktionskoordinator pro Partition zusätzliche Nachrichten schreiben muss, hängt der Overhead von der Anzahl der an der Transaktion beteiligten Partitionen ab, aber nicht von der Anzahl der geschriebenen Nachrichten. Das bedeutet, dass bei sehr kurzen, sich im Millisekundenbereich befindlichen Transaktionen der Overhead signifikant ist und bei längeren Transaktionen immer

⁹⁾ <https://de.wikipedia.org/wiki/Commit-Protokoll>

geringer wird. Zum Beispiel hat die Firma Confluent bei der Einführung von Transaktionen gemessen, dass bei Transaktionen, die 100 ms dauern und die Producer mit maximaler Geschwindigkeit produzieren, der Overhead etwa 3% betrug.¹⁰⁾

3.6.3 Zusammenfassung

Wir haben uns in diesem Kapitel ausführlich mit Verarbeitungsgarantien und Transaktionen beschäftigt. Zunächst haben wir uns angesehen, welche verschiedenen Garantien es in Kafka gibt und wie Kafka diese umsetzt. Hierbei sind wir insbesondere auf Idempotenz eingegangen und haben gelernt, wie wir diese in Kafka erreichen können. Anschließend haben wir ausführlich Transaktionen behandelt. Wir haben zunächst erläutert, warum wir überhaupt Transaktionen benötigen. Im Anschluss haben wir uns detailliert angesehen, welche Möglichkeiten es gibt, Transaktionen in Kafka umzusetzen. Wir haben zum einen erfahren, dass Consumer, die in Datenbanken schreiben, sich auch deren Fähigkeiten bezüglich Transaktionen zunutze machen können. Zum anderen haben wir uns genau angesehen, wie Transaktionen in Kafka selbst umgesetzt werden, und auch, was wir hierbei sowohl bei unseren Producern als auch Consumern beachten müssen.

¹⁰⁾ Transactions in Apache Kafka <https://www.confluent.io/blog/transactions-apache-kafka/>
(Artikel vom 17.11.2017)

4

Kafka im Unternehmenseinsatz

Aus unserer Trainings- und Beratungspraxis sehen wir vor allem zwei Anwendungsfälle für Kafka in Unternehmen. Beide sind valide, sehr verbreitet und eignen sich sowohl für kleine als auch große Datenmengen. Insbesondere für sehr kleine Datenmengen sind wir immer vorsichtig damit, Kafka zu empfehlen, denn die Gefahr ist groß, mit Kanonen auf Spatzen zu schießen, und es benötigt einiges Geschick, die Kafka-Broker so zu konfigurieren, dass sie nicht zu viele Ressourcen verschwenden. Aber zum Glück reduziert sich mit Kafka 3.0 der Betriebsaufwand durch den Wegfall von Zookeeper.

Der erste klassische Anwendungsfall ist der von Kafka als Messaging-System. Es geht darum, Kafka dafür einzusetzen, um Nachrichten zwischen Services im Unternehmen auszutauschen. Das kann zwischen neuen Services und bestehenden Applikationen oder Anwendungen, die von null aufgebaut werden, sein. Besonders populär war Kafka in der Anfangszeit, um als Aggregationssystem zu dienen, um Daten aus zahlreichen Quellen weiterzuleiten, zum Beispiel an Big-Data-Systeme.

Diese Anwendungsfälle zeichnen sich dadurch aus, dass die Retention-Zeiten sehr kurz sind. Je nach Anwendungsfall können das wenige Stunden bis wenige Tage sein. Kafkas Zustellgarantien sind nett, werden hier aber oft nicht verwendet, denn die rohe Performance steht meist über der Zuverlässigkeit. Hier hat Kafka auch die größte Konkurrenz und muss sich in der Projektplanung mit klassischen Messaging-Systemen wie *ActiveMQ*, *RabbitMQ* oder gar Enterprise-Service-Bus-Systemen etablierter Anbieter messen. Wir besprechen die Unterschiede zwischen solchen Systemen und Kafka im Kapitel „Vergleich mit anderen Technologien“.

Der zweite Anwendungsfall ist der von Kafka als Kernelement einer Streaming-Plattform. Im deutschen Konzernumfeld bezeichnet man diese Strategie gerne als Datendrehscheibe. Anstatt dass Kafka nur als Messaging-System eingesetzt wird, ist das Ziel dieser Architektur, dass Kafka die Datenhoheit über eine Vielzahl an Datenarten hält. Wir sagen dazu auch, dass Kafka die *Single Source of Truth* für die Daten ist. Das heißt, weder integrieren wir unterschiedliche Systeme miteinander noch bringen wir ihnen bei, miteinander zu reden, und wir synchronisieren Datenbestände auch nicht mehr miteinander. Stattdessen fungiert Kafka als *zentrale Quelle der Wahrheit*, und alle Dienste schicken Änderungen an Daten an Kafka und holen sich den aktuellen Stand der Daten von Kafka ab.

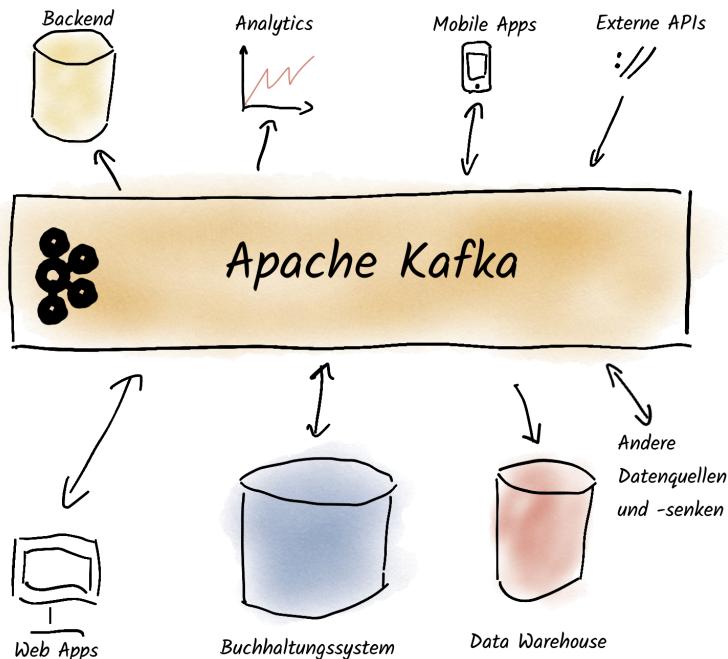


Bild 4.1 Kafka als zentrales Nervensystem für Daten

Da wir oft nicht auf der grünen Wiese starten, sondern schon eine Vielzahl an IT-Systemen im Unternehmen haben, die auch nicht alle mit Kafka kompatibel sind, sehen wir viele Mischformen dieser zwei Anwendungsfälle. Oft sehen wir das Muster der Trennung zwischen *neuer* und *alter Welt*, wo wir bei Letzterer von Integration sprechen und die von großen, monolithischen, sich langsam entwickelnden Systemen geprägt ist. Aber diese Welt ist auch oft der Kern des Unternehmens und bildet wichtige, bestehende Prozesse ab und ist nicht ersetzbar. Auf der anderen Seite haben wir die *neue Welt*, geprägt von Microservices (oder zumindest Systemen, die agiler entwickelt werden als zum Beispiel das SAP-System), agilen Entwicklungsmethoden, aber auch den Wunsch, immer mehr Services immer schneller zu entwickeln und diese in der Cloud zur Verfügung zu stellen. In solchen Umgebungen sehen wir Kafka einerseits als Streaming-Plattformen in der *neuen Welt*, aber auch als Kommunikationsplattform zur *alten Welt*.

Wichtig ist hier zu beachten, dass dies Architekturmuster sind. Das heißt, Kafka ist nicht für alle Anwendungsfälle geeignet, und die Herausforderung für Architekten und Entwickler ist es herauszufinden, ob und wie Kafka im konkreten Unternehmen eingesetzt werden kann, um möglichst viel Nutzen zu schaffen. Insbesondere ist es nicht sinnvoll, ein altes Messaging-System, welches niemand so richtig versteht, durch Kafka zu ersetzen, in der Hoffnung, dass dann alles besser wird. Oft fehlt einfach nur das Wissen, wie Systeme funktionieren. Wenn sich niemand intensiv mit Kafka beschäftigt und es kein Team gibt, welches sich mit Kafka auskennt, dann wiederholen wir lediglich dieselben Fehler.

In solchen Mischarchitekturen wird es auch spannend, wie die unterschiedlichen Systeme miteinander und mit Kafka sprechen. Neue Services in der Entwicklung werden meistens schon so programmiert, dass sie mit Kafka reden. Manche kommerzielle und Open-Source-

Software bieten von Haus aus Connectoren für Kafka an und sind entsprechend einfach zu verbinden. Aber in fast jedem Unternehmen gibt es diese Services, die vor vielen Jahren eingeführt wurden und die sich niemand traut anzufassen. Meistens liegt diesen Services eine Datenbank zugrunde, die mehr oder weniger gut strukturiert ist und sich mithilfe von Kafka Connect relativ einfach mit Kafka verbinden lässt. Insbesondere in der Banken- und Versicherungswelt werden weiterhin Mainframes eingesetzt, die nicht auf die Zugriffsmuster moderner IT ausgelegt sind. Um sie zu entlasten, muss Software geschrieben werden, um die Daten nach Kafka zu schicken.

■ 4.1 Kafka-Ökosystem

Um mit Kafka eine Streaming-Plattform umzusetzen, aber auch um Kafka als vollwertiges Messaging-System zu benutzen, reicht Kafka allein nicht aus. Glücklicherweise gibt es ein immer größer werdendes Ökosystem rund um Kafka. Wir haben mit *Kafka Connect* ein Framework, um Drittsysteme an Kafka anzubinden, das heißtt, um Daten von externen Systemen nach Kafka zu senden, aber auch um Daten aus Kafka zu extrahieren. Es gibt Connectoren für zahlreiche Datenbanken wie zum Beispiel PostgreSQL, MySQL oder Microsoft SQL Server. Aber auch zahlreiche Cloud-Services bei AWS, Azure oder in der Google Cloud werden unterstützt. Allein der *Connect Hub* der Firma Confluent¹ zählt zum Zeitpunkt des Schreibens etwa 200 Connectoren. Darüber hinaus gibt es auch zahlreiche andere Connectoren aus anderen Quellen. Eine weitere Möglichkeit ist es, einen der vorhandenen *REST Proxies*² zu benutzen, um Drittanwendungen, die nur REST sprechen können, an Kafka anzubinden.

Nun können wir Daten nach Kafka bewegen und aus Kafka heraus wieder extrahieren. Für eine Streaming-Plattform reicht dies nicht aus, sondern wir benötigen eine Möglichkeit, die Daten in Kafka zu manipulieren, Verknüpfungen zu finden und Daten aufzubereiten. Natürlich können wir dafür die Kafka-Bibliothek benutzen und übliche Stream-Operationen manuell nachimplementieren. *Kafka Streams* ist aber oft eine bessere Wahl. Diese Java-Bibliothek stellt uns eine Menge an Operationen zur Verfügung, um mit wenig Aufwand performante Streaming-Applikationen zu entwickeln. Dabei bleiben die Applikationen skalierbar und ausfallsicher. Eine Alternative dazu sind Produkte wie *ksqldb*³ oder *lenses.io*⁴, die uns ermöglichen, mithilfe eines SQL-Dialekts Abfragen über Daten in Kafka-Topics zu stellen.

Kafka ermöglicht es uns, beliebige Datenstrukturen zu speichern, aber mit dieser Freiheit kommt große Verantwortung. Denn ganz ohne Kontrolle und Governance riskieren wir großes Chaos in unseren Kafka-Topics. Schema Registries wie die von Confluent (kommerziell)⁵

¹⁾ Confluent Hub: <https://www.confluent.io/hub/>

²⁾ Open Source REST Proxy: <https://github.com/aiven/karapace> oder
Confluent REST Proxy <https://docs.confluent.io/platform/current/kafka-rest/quickstart.html>

³⁾ <https://ksqldb.io/>

⁴⁾ <https://lenses.io/>

⁵⁾ <https://docs.confluent.io/platform/current/schema-registry/index.html>

oder Aiven (Open Source)⁶ helfen uns hierbei, den Überblick zu behalten und sicherzustellen, dass geschriebene Nachrichten korrekt formatiert sind.

Kafka selbst bringt bereits eine gute Ausfallsicherheit mit sich. Aber wie wir bereits beschrieben haben, ist Replikation nicht das Gleiche wie Backups. *Kafka MirrorMaker 2* gibt uns die Möglichkeit, Daten zwischen verschiedenen Kafka-Clustern auszutauschen oder Daten über mehrere Rechenzentren hinweg zu spiegeln. Für Backups gibt es leider zum Zeitpunkt des Schreibens keine gute Lösung. Ansätze wie das *Kafka Backup*⁷ eines der Autoren zeigen Möglichkeiten, aber sind zum Zeitpunkt des Schreibens nicht produktionsreif.

4.1.1 Kafka Connect

In den allermeisten Fällen führen wir Kafka nicht unabhängig von anderen Systemen in unserem Unternehmen ein, sondern möchten weitere Systeme wie Datenbanken, Messaging-Systeme oder Ähnliches an Kafka anbinden. Die meisten solcher Anwendungsfälle sind sehr ähnlich. Wir möchten Daten aus vordefinierten Datenbanktabellen in bestimmte Topics übertragen oder Daten aus bestimmten Topics in eine Datei schreiben. Natürlich haben wir da immer die Möglichkeit, händisch eigene Producer und Consumer zu schreiben, um Daten nach oder aus Kafka zu bewegen. Dies ist aber sehr aufwendig und führt in vielen Fällen zu fehleranfälligen Systemen, die auch noch schlecht skalierbar sind. Auch bei so einfachen Anforderungen wie dem Transport von Daten aus einem System in ein anderes gibt es viele Sonderfälle zu beachten.

4.1.1.1 Wie funktioniert Kafka Connect?

Eine Alternative ist es, *Kafka Connect* einzusetzen. Kafka Connect ist ein Framework, um Daten aus Drittssystemen nach Kafka zu schreiben und auch von Kafka in andere Systeme. Als Framework ist es Teil von Apache Kafka und genauso wie Kafka unter der Apache 2.0-Lizenz als Open-Source-Software verfügbar.

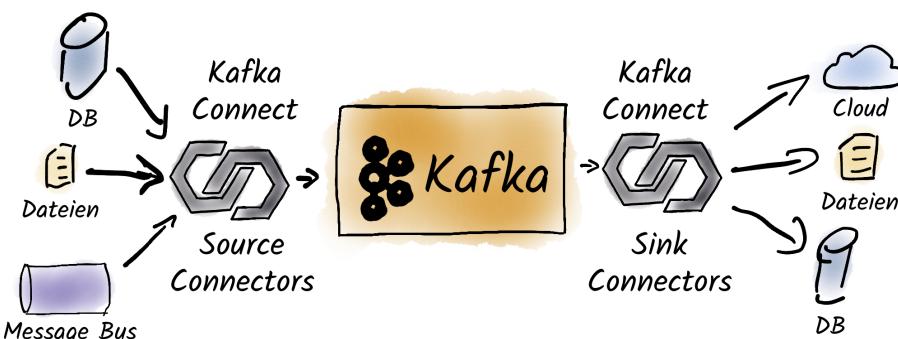


Bild 4.2 Kafka Connect bindet Drittssysteme an Kafka an.

⁶⁾ <https://github.com/aiven/karapace>

⁷⁾ <https://github.com/itadventurer/kafka-backup/>

Ziel von Kafka Connect ist es, ein standardisiertes Werkzeug zu sein, um Daten zu bewegen. Genauso wie Kafka ist Kafka Connect skalierbar, fehlertolerant und legt viel Wert auf Korrektheit und Performance. Es gibt Connectoren für eine Vielzahl an Systemen. Für Datenbanken wie PostgreSQL, SQLite oder MySQL, für Object Storage-Systeme in unterschiedlichen Cloud-Umgebungen wie AWS S3 oder Azure Blob Storage; für Messaging-Protokolle wie MQTT oder JMS oder auch Data-Warehouse-Systeme wie Snowflake oder AWS Redshift. Allein der Confluent Hub⁸ listet etwa 200 Connectoren.

Kafka Connects Grundidee ist es, sich mit Datenquellen und -senken zu verbinden, die sich als sogenannter *Partitioned Stream*, also partitionierter Datenstrom, beschreiben lassen. Die Idee ist dabei, dass wir eine Möglichkeit brauchen, eine Datenquelle in unterschiedliche Datenströme aufzuteilen, um skalieren zu können. Diese Datenströme beinhalten Daten, die zusammengehören und nicht weiter aufgeteilt werden sollen oder können. In Kafka teilen wir Topics in Partitionen auf, um genau das zu erreichen. In Datenbanksystemen können wir einzelne Tabellen als Datenströme betrachten, die wir nicht ohne Weiteres aufteilen können, ohne Reihenfolge-Garantien und andere Eigenschaften zu verlieren. In einem Dateisystem könnten diese Datenströme zum Beispiel einzelne Dateien sein.

Als Zweites benötigen wir für unsere *Partitioned Streams* eine Möglichkeit, uns zu merken, welche Nachrichten wir bereits gelesen haben und welche noch nicht. Da wir davon ausgehen, dass diese Datenströme sehr lang werden, können wir uns nicht einfach merken, welche Nachrichten wir schon gesehen haben. Wir arbeiten stattdessen, wie in Kafka, mit Offsets, also mit Positionen im Datenstrom, um festzustellen, bis zu welcher Position wir Daten bereits gelesen haben. In Datenbanken könnten wir sequenzielle IDs dafür benutzen, oder auch Timestamps, um festzustellen, wann zuletzt eine Zeile geändert wurde. In Dateien könnten wir uns dafür merken, bis zu welcher Zeile oder welcher Position wir bereits gelesen haben.

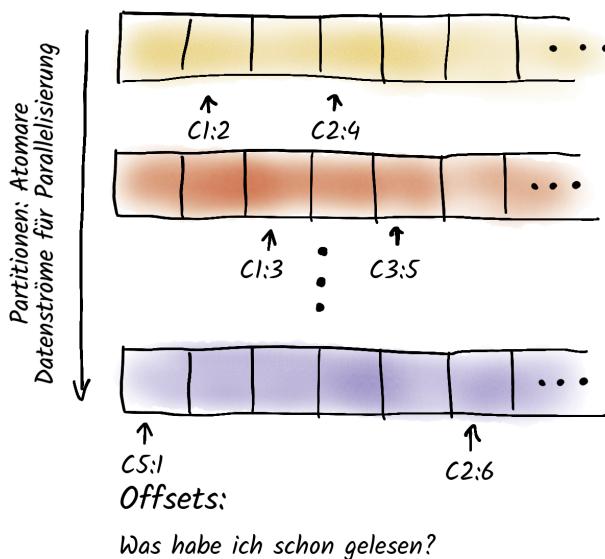


Bild 4.3 Partitioned Streams in Kafka

⁸⁾ <https://www.confluent.io/hub/>

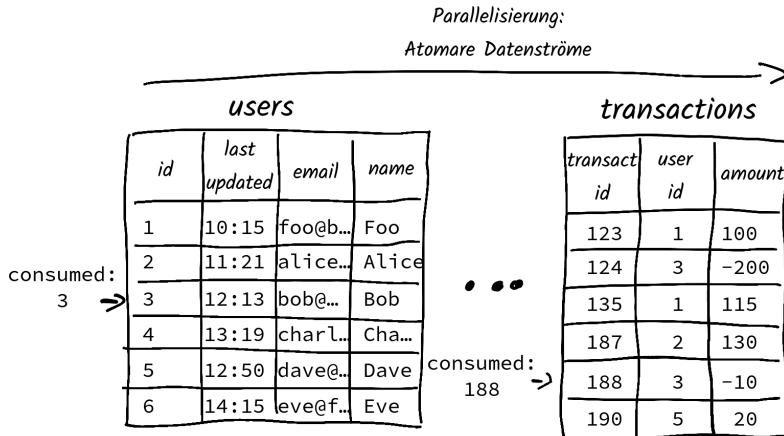


Bild 4.4 Partitioned Streams in Datenbanken

4.1.1.2 Praxisbeispiel Distributed Mode

Um von allen Vorteilen von Kafka Connect zu profitieren, starten wir es meistens im sogenannten Distributed Mode. In diesem Modus speichert Kafka Connect alle Konfigurationsdaten und internen Offsets in Kafka-Topics ab. Dann starten wir so viele Kafka Connect-Instanzen, wie wir benötigen. Alle Kafka Connect-Instanzen mit der gleichen group.id bilden jeweils einen Kafka Connect-Cluster, in dem sie sich koordinieren und die Last untereinander aufteilen.

In Entwicklungsumgebungen und in bestimmten Einzelfällen kann es sinnvoll sein, auf diese Vorteile zu verzichten und stattdessen Kafka Connect im sogenannten Stand-alone Mode zu betreiben. In diesem Modus speichert Kafka Connect keine internen Daten in Kafka ab und merkt sich auch keine Konfigurationen. Wir starten Kafka Connect einfach als Programm auf unserem Rechner, und wenn wir fertig sind, können wir es wieder stoppen. Dieser Modus ist ideal, um auf Entwicklungsrechnern Kafka Connect auszuprobieren, bietet uns aber keine Möglichkeiten, zu skalieren und unsere Kafka Connect-Offsets in Kafka zu speichern.

Als Erstes konfigurieren wir den Kafka Connect Worker für den Distributed Mode. Dafür legen wir eine Datei `worker.properties` mit folgendem Inhalt an:

```
bootstrap.servers=localhost:9092
group.id=connect
config.storage.topic=connect-config
offset.storage.topic=connect-offset
status.storage.topic=connect-status
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=org.apache.kafka.connect.storage.StringConverter
key.converter.schemas.enable=false
value.converter.schemas.enable=false
rest.port=8083
```

Wie bei allen Kafka-Clients müssen wir zuerst unsere Bootstrap-Server angeben. Verschiedene Instanzen von Kafka Connect mit der gleichen group.id bilden hier, wie bereits erwähnt, einen Kafka Connect-Cluster. Hier setzen wir die `group.id=connect`. Wir benötigen

drei Topics, ein Topic für Connector-Konfigurationen (`connect-config`), eines, in dem Kafka Connect die Offsets speichert (`connect-offset`), und eines für den aktuellen Status der Connectoren (`connect-status`). Zusätzlich müssen wir Standardeinstellungen für sogenannte Converter anlegen. Damit legen wir fest, in welchem Datenformat die Daten auf den Kafka-Topics abgelegt werden sollen. In unserem Anwendungsfall möchten wir die Daten lediglich als Strings ohne Schemas ablegen und setzen deshalb sowohl die `key.converter` als auch `value.converter` auf `StringConverter` und deaktivieren die Schemas. Zu guter Letzt setzen wir den REST-Port von Kafka Connect. Wir können nun Kafka Connect mit folgendem Befehl starten:

```
$ connect-distributed.sh worker.properties
```

Wenn die Topics noch nicht existieren, legt Kafka Connect sie für uns an. Wichtig ist, dass der empfohlene *Replication Factor* für die Topics bei mindestens drei liegt. Wenn wir weniger als drei Broker haben, kann Kafka Connect nicht starten. Wir können in Testumgebungen den *Replication Factor* über die Einstellung `config.storage.replication.factor` (und ähnlich für die anderen Topics) reduzieren.

Nun läuft zwar Kafka Connect, macht aber vorerst nichts. Dafür müssen wir erst einen Connector konfigurieren. Wir erstellen und verwalten Connectoren über die REST API. Stellen wir uns beispielhaft vor, dass wir eine Datei `raketen.txt` haben, in der wir Namen unserer Raketen abspeichern. Wir möchten nun diese Daten in Kafka importieren. Dafür können wir den mit Kafka ausgelieferten *FileStreamSource* Connector benutzen. Wir empfehlen, diesen Connector nicht in Produktion einzusetzen, da dieser sehr minimalistisch ist und schlecht mit Fehlern umgehen kann. Erstellen wir dafür eine Datei `source-connector.json`:

```
{"name": "rocket-source",
  "config": {
    "connector.class": "FileStreamSource",
    "tasks.max": "1",
    "file": "/tmp/rockets.txt",
    "topic": "rockets"
}}
```

Hier konfigurieren wir einen Connector `rocket-source`, der die Klasse `FileStreamSource` benutzen soll. Da wir nur eine Datei lesen, können wir diesen Connector nicht skalieren, und deshalb setzen wir `tasks.max` auf 1. Wir möchten Daten aus der Datei `/tmp/rockets.txt` in das Topic `rockets` schreiben.

Mithilfe des folgenden `curl`-Kommandos senden wir diese Konfigurationsdatei via *POST-Request* an Kafka Connect:

```
$ curl -X POST -H "Content-Type: application/json" \
--data @source-connector.json \
http://localhost:8083/connectors
```

Wir können alle Connectoren in unserem Kafka Connect-Cluster unter der URL `http://localhost:8083/connectors` abfragen:

```
$ curl http://localhost:8083/connectors
["rocket-source"]
```

Den aktuellen Status unseres Connectors finden wir mit folgendem Befehl heraus:

```
$ curl http://localhost:8083/connectors/rocket-source/status
{
  "name": "rocket-source",
  "connector": {
    "state": "RUNNING",
    "worker_id": "127.0.0.1:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "RUNNING",
      "worker_id": "127.0.0.1:8083"
    }
  ],
  "type": "source"
}
```

Wir sehen hier, dass der Connector im Zustand RUNNING ist und auf dem Worker 127.0.0.1:8083 läuft. Darüber sehen wir alle Tasks. Da wir nur einen Task definiert haben, ist es derselbe, auf dem der Connector selbst läuft. Wenn wir uns die Log-Ausgabe von Kafka Connect genauer anschauen, stellen wir fest, dass nicht alles so rund läuft, wie uns der /status-Endpunkt sagt:

```
WARN Couldn't find file /tmp/rockets.txt for FileStreamSourceTask
```

Schreiben wir also in einem anderen Terminalfenster ein paar Daten in die Datei /tmp/rockets.txt:

```
$ echo "Death Star" >> /tmp/rockets.txt
$ echo "Falcon" >> /tmp/rockets.txt
```

Wir können nun parallel einen Consumer starten, um zu beobachten, was Kafka Connect in das Topic *rockets* schreibt:

```
$ kafka-console-consumer.sh \
  --topic rockets \
  --from-beginning \
  --bootstrap-server=localhost:9092
Death Star
Falcon
```

Wenn alles funktioniert hat, sehen wir hier unsere beiden Raketen. Wenn nicht, dann können wir den /status-Endpunkt oder Kafka Connect-Logs auf Fehlermeldungen überprüfen, um eine Idee zu bekommen, was falsch gelaufen ist.

Wenn wir nun parallel ein paar weitere Daten in die Datei schreiben, dann kommen sie auch im Consumer an. Wenn wir Kafka Connect neu starten, dann fängt der Connector nicht an, die Daten von Anfang an aus der Datei wieder einzulesen, sondern merkt sich die Position in der Datei und liest nur neue Einträge ein.

4.1.1.3 Skalierbarkeit und Ausfallsicherheit

Wie aber ermöglicht uns Kafka Connect, unsere Connectoren zu skalieren und unsere Ausfallsicherheit zu erhöhen? Zuerst sollten wir anmerken, dass dies mit dem einfachen FileStreamSource nicht ohne Weiteres möglich ist, da wir nur Dateien von der lokalen Fest-

platte lesen und die Dateien höchstwahrscheinlich nicht auf anderen Connect Workern zu finden sind. Wir möchten an dieser Stelle erneut davor warnen, diesen Connector in Produktion einzusetzen, da dieser eher einem Proof of Concept ähnelt als einem ausgereiften Connector.

Ein Kafka Connect-Cluster besteht aus einem oder mehreren Workern, die im Distributed Mode laufen und die gleiche `group.id` haben. Diese Worker koordinieren sich, genauso wie Consumer in einer Consumer Group, mithilfe des Kafka Rebalancing Protocols. Stürzt ein Worker ab oder fügen wir einen neuen Worker hinzu, teilen die restlichen Worker die Aufgaben unter sich auf. In einem Kafka Connect-Cluster können wir mehrere Connectoren gleichzeitig betreiben. Zum Beispiel können wir sowohl Daten aus einer externen Datenbank lesen als auch andere Daten in ein Drittssystem schreiben. Wenn wir unseren Kafka-Cluster skalieren möchten, fügen wir einfach weitere Worker mit der gleichen `group.id` hinzu. Die Connectoren selbst teilen die Aufgaben in unterschiedliche Tasks auf, die unabhängig voneinander laufen. Wenn wir uns zum Beispiel vorstellen, dass wir Daten aus zwölf Datenbanktabellen importieren möchten und vier Tasks zur Verfügung haben, dann ist jeder Task für drei Tabellen zuständig. Diese Tasks verteilt Kafka Connect auf die unterschiedlichen Worker. So erreichen wir eine hoffentlich gleichmäßige Lastverteilung und sind auch in der Lage, größere Datenmengen schnell nach oder aus Kafka zu bewegen.



Tipp

Es gibt bereits eine große Menge an unterschiedlichen Connectoren, und wir empfehlen sehr, diese zu benutzen. Falls es für einen konkreten Fall keinen Connector gibt, aber sich die Datenstrukturen gut als Partitioned Stream darstellen lassen, empfehlen wir im Allgemeinen, es immer zu präferieren, eigene Kafka Connect Connectoren zu schreiben, statt mithilfe von Producern und Consumern dies eigenständig nachzubauen.

4.1.2 Kafka Streams

Mithilfe von Kafka Connect haben wir eine gute Möglichkeit, Daten aus Drittssystemen nach Kafka zu schreiben, aber auch Daten aus Kafka zu exportieren. Zusätzlich dazu haben wir üblicherweise in Unternehmen zahlreiche andere Systeme, die Daten schreiben oder lesen möchten. Nun möchten wir meistens mehr mit den Daten tun, als diese nur zu transportieren. Wir möchten sie verarbeiten.

4.1.2.1 Stream Processing

Nun können wir auch Daten in Kafka mithilfe von Batch-Prozessen jede Nacht verarbeiten, um die Last auf unsere Produktivsysteme zu reduzieren, aber dies ist in unserer heutigen Welt nicht mehr zeitgemäß. Unsere Kunden erwarten die Ergebnisse meistens in Echtzeit, und dafür eignet sich Batch-Processing nicht sonderlich gut. Stattdessen fokussieren wir uns in Systemen, die Kafka benutzen, auf sogenanntes Stream Processing. Statt Ergebnisse zu bestimmten Zeitpunkten, wie zum Beispiel nachts, zu berechnen, aktualisieren wir die

Ergebnisse kontinuierlich in Nahe-Echtzeit. Statt also im schlimmsten Fall einen ganzen Tag auf neue Ergebnisse zu warten, können wir sie im besten Fall bereits wenige Millisekunden nach dem Ereignis sehen.

Statt während der Berechnung von Ergebnissen den aktuellen (diskreten) Gesamtzustand unseres Systems zu betrachten, gehen wir beim Stream Processing davon aus, dass unsere Ergebnisse immer Zwischenstände sind, da wir jederzeit neue Daten erwarten. Wir betrachten unsere Daten nicht als diskrete Zustände, die die Welt im *Jetzt* darstellen, sondern als nie endenden Datenstrom. Auf diesem Datenstrom können wir dann zahlreiche Operationen durchführen. Diese Operationen nehmen einen oder mehrere Datenströme als Eingangsdaten entgegen und schreiben die Ergebnisse in ein oder mehrere ausgehende Datenströme. Dabei können diese Operationen lokale Zustände halten, um sich zum Beispiel die Daten der letzten fünf Minuten zu merken, oder auch gänzlich ohne lokale Zustände auskommen. Leser, die schon Berührungspunkte mit funktionaler Programmierung hatten, könnte diese Konzepte wiedererkennen.

Die *filter*-Operation nimmt einen Datenstrom entgegen und entscheidet für jeden Wert anhand eines Prädikats, ob dieser Wert in den Ausgangsdatenstrom geschrieben werden soll oder nicht. Mit der *map*-Operation können wir eine Funktion auf jeden Datenpunkt im Eingangsdatenstrom anwenden und das Ergebnis dieser Funktion in den Ausgangsdatenstrom schreiben. Mit *join* können wir, ähnlich wie bei relationalen Datenbanken, Daten aus zwei Datenströmen kombinieren und das Ergebnis in einen ausgehenden Datenstrom schreiben. Mit *groupBy* können wir Daten im eingehenden Datenstrom gruppieren (zum Beispiel nach dem Key) und auf diesen Gruppen später Aggregate (wie zum Beispiel *Sum*, *Max*, *Min*, *Count* und andere) anwenden.

Wenn wir in unseren Applikationen das Muster sehen, dass wir Daten aus einem Topic lesen, diese manipulieren und wieder in ein anderes Topic schreiben, wenden wir höchstwahrscheinlich schon Konzepte aus dem Stream Processing ein. Natürlich ist es möglich, dies mithilfe normaler Producer und Consumer selbst zu implementieren. Dabei werden wir wahrscheinlich viele Räder neu erfinden und Konzepte, die Stream Processing-Bibliotheken perfektioniert haben, fehleranfällig und inperformant nachimplementieren. *Kafka Streams* ist so eine Stream Processing-Bibliothek, die sich inzwischen als Mittel der Wahl in der Kafka-Welt etabliert hat.



Tipp

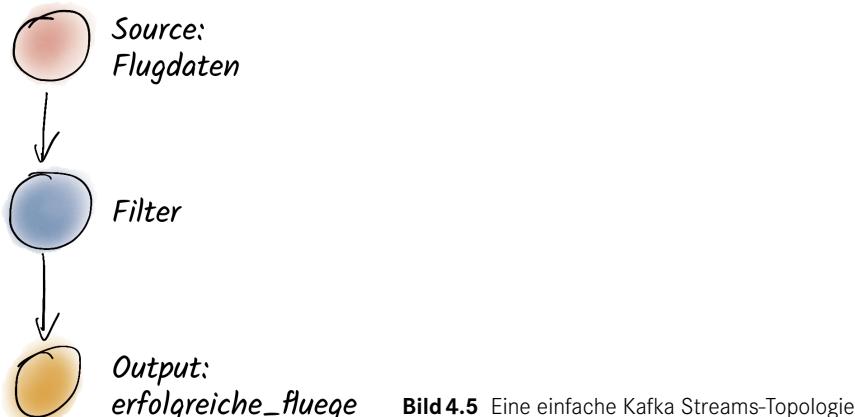
Wenn wir in unseren Applikationen das Muster sehen, dass wir Daten aus einem Topic lesen, diese manipulieren und wieder in ein anderes Topic schreiben, empfehlen wir dies mit Kafka Streams umzusetzen.

4.1.2.2 Wie funktioniert Kafka Streams?

Kafka Streams nutzt die eben beschriebenen Operationen aus der funktionalen Programmierung und wendet diese auf Datenströme, also Topics und Partitionen an. Dabei ist Kafka Streams, wie vieles im Kafka-Ökosystem, sowohl auf Performance als auch Zuverlässigkeit optimiert. Wir können unsere Streams-Applikationen einfach skalieren, indem wir weitere Instanzen desselben Codes zur Verfügung stellen. Diese skalierten Instanzen verteilen die

Aufgaben elegant untereinander auf, und falls eine oder mehrere Instanzen ausfallen, übernehmen die übrig gebliebenen Instanzen die Aufgaben. Leider gibt es Kafka Streams ausschließlich für Java, und Code, der nicht in der Java Virtual Machine läuft, muss auf diese Bibliothek verzichten.

Die einzelnen Operationen, die Kafka Streams anbietet, sind an sich schon nützlich, um gewisse Operationen durchzuführen, aber entfalten erst im Zusammenspiel ihre Wirkung. Dafür kombinieren wir unterschiedliche Operationen in sogenannten Topologien. Eine Kafka Streams-Topologie ist ein Graph, also ein Netzwerk, mit einem oder mehreren Eingangsknoten (den Topics, die wir verarbeiten möchten), einer beliebigen Anzahl von Zwischenknoten (den Operationen, die wir auf den Daten ausführen möchten) und einem oder mehreren Ausgängen (die Topics, in die wir die Ergebnisse schreiben). Wenn wir zum Beispiel aus unserem Topic *flugdaten* nur die erfolgreichen Landungen filtern möchten, können wir folgende Topologie anlegen.



Als Eingangsknoten nutzen wir unser Topic *flugdaten*, dann filtern wir die Daten in diesem Stream nach erfolgreichen Landungen, und zuletzt schreiben wir die Daten in das Topic *erfolgreiche_fluege*. Ähnlich wie Kafka selbst, können wir auch diese Topologie sehr gut parallelisieren, da wir wissen, dass das Eingangstopic in mehrere Partitionen aufgeteilt ist und wir diese Topologie unabhängig voneinander auf mehreren Maschinen parallel verarbeiten können. Diese Topologie können wir direkt so in Code umsetzen:

```

1. final Serde<String> stringSerde = Serdes.String();
2. final StreamsBuilder builder = new StreamsBuilder();
3. builder.stream("flugdaten_mit_keys",
4.                 Consumed.with(stringSerde, stringSerde))
5.         .filter((key, value) ->
6.             value.equals("Landung erfolgreich"))
7.         .to("erfolgreiche_fluege");

```

Mit `.stream()` erstellen wir einen Eingangs-Stream, der aus dem Topic *flugdaten_mit_keys* lesen soll. Sowohl unsere Keys als auch Values sind Strings, und das geben wir mit den beiden `stringSerdess` an. Das Ergebnis dieses Streams filtern wir mit `.filter()` nach den Einträgen Landung erfolgreich und schreiben das Ergebnis mit `.to()` in das Topic *erfolgreiche_fluege*. Wir könnten an dieser Stelle ein eigenes Buch mit den Details zu Kafka Streams

füllen. Deswegen verweisen wir hier auf Kafka Streams-Literatur als auch auf unsere Kafka-Schulungen für Entwickler.

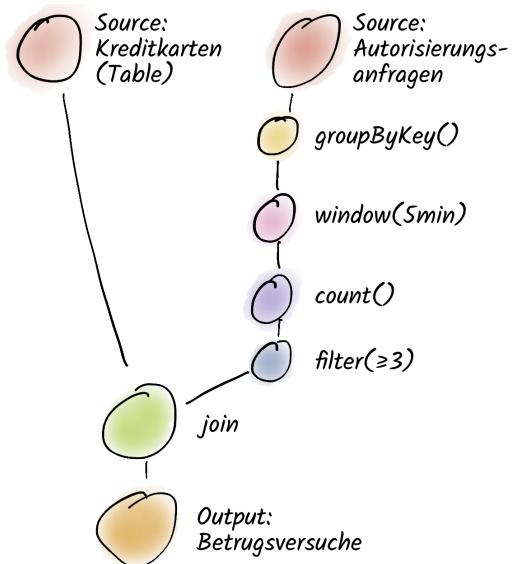


Bild 4.6 Kafka Streams-Topologie mit Join

Natürlich ist Kafka Streams nicht nur darauf beschränkt, einfache Operationen zu verketten, sondern kann auch Streams miteinander verbinden oder aufsplitten. Um im Kreditkartenkontext zum Beispiel die Frage zu beantworten, bei welchen Kreditkarten in den letzten fünf Minuten mindestens drei Autorisierungsversuche durchgeführt wurden, benötigen wir weitere Features von Kafka Streams. Eine beispielhafte Topologie haben wir in Bild 4.6 abgebildet. Statt nur eines Eingangstopics haben wir nun zwei, eins für die Autorisierungsanfragen und eins, um mehr Informationen zu der Kreditkarte anzeigen zu können. Uns interessiert dabei aber nicht die gesamte Historie, sondern immer nur der aktuelle Wert.

4.1.2.3 KTables

Dafür können wir das Konzept von *KTables* nutzen. Im Gegensatz zu einem *KStream*, der uns einen kontinuierlichen Datenstrom zurückgibt, können wir in einer *KTable* immer den aktuellen Wert für einen bestimmten Key abspeichern. Wenn wir zum Beispiel die Namen der Kreditkarteninhaber zu den Kreditkartennummern speichern wollen, können wir das zum Beispiel in einer *KTable* ablegen und bei der Abfrage der *KTable* immer den aktuellen Wert bekommen.

Zuerst gruppieren wir unsere Autorisierungsanfragen nach dem Key (wir nehmen an, dass der Key die Kreditkartennummer ist), dann nutzen wir die sogenannte windowing-Operation, um nur die aktuellen fünf Minuten der Daten beizubehalten. Im nächsten Schritt nutzen wir die *count*-Operation, um aus diesem gruppierten Datenstrom eine *KTable* zu erzeugen und so für die Kreditkartennummern die Anzahl der Autorisierungsversuche vorzuhalten. Diese *KTable* filtern wir, um sie im Anschluss mit der Kreditkarten-*KTable* per *join*-Operation zu verknüpfen, um unsere Kunden zum Beispiel benachrichtigen zu können.

Diese zwei Beispiele zeigen auf der einen Seite die Mächtigkeit von Kafka Streams, auf der anderen Seite aber auch, dass sich diese komplexen Prozesse mit wenigen Zeilen Code umsetzen lassen. In vielen Fällen sind die eigentlichen Topologien sehr überschaubar und lassen sich auch mit Werkzeugen wie dem Kafka Streams Topology Visualizer⁹ im Nachhinein analysieren und verständlicher machen. Auf der anderen Seite haben wir Kafka Streams nur angerissen, und es gibt viele spannende Themen, die wir in diesem Buch nicht besprechen werden. Einer dieser Themen ist die Unterstützung für unterschiedliche Datenformate. Insbesondere beim Einsatz von Kafka Streams ist es essenziell, auf saubere Datenhaltung zu achten. Dies ermöglichen wir in Kafka-basierten Architekturen mit wie auch immer geartetem Schema-Management, welches wir uns im folgenden Abschnitt genauer anschauen werden.

4.1.3 Schema-Management

In diesem Kapitel werden wir uns mit dem Schema-Management in Kafka einem wichtigen Thema widmen, das wir bisher komplett außen vor gelassen haben. Wir werden erfahren, was genau wir unter Schemas und Schema-Management in Kafka verstehen und welche Vorteile Schemas in diesem Kontext mit sich bringen. Wir werden uns ansehen, welche Möglichkeiten es in Kafka gibt, Schemas zu definieren, und uns dabei auch kurz mit Kafka Schema Registrys beschäftigen. Zum Abschluss dieses Abschnitts werden wir noch auf die Feinheiten, die wir bei Anpassungen von bestehenden Schemas beachten müssen, eingehen.

4.1.3.1 Schemas in Apache Kafka

In unseren bisherigen Beispielen haben wir lediglich Nachrichten, welche aus einfachen Strings bestanden, produziert und konsumiert. In der Realität würden wir allerdings mit einfachen Strings schnell auf Probleme stoßen, da Nachrichten wesentlich komplexere Objekte enthalten können. Stellen wir uns zur Veranschaulichung wieder unsere Kafka-Rakete vor, die bereits erfolgreich ins Weltall gestartet ist und nun regelmäßig ihren Status überträgt. Der Status wird vermutlich aus mehr als nur einer Kenngröße bestehen. Interessant wären wahrscheinlich Informationen bezüglich Geschwindigkeit, Position, Energieverbrauch/-produktion, Treibstoffmenge und natürlich noch vieles mehr. Es wäre an dieser Stelle vielleicht naheliegend, für jede Kenngröße ein eigenes Topic anzulegen, aber zum einen würde dies einen unnötigen Overhead für unser Kafka-Cluster bedeuten, und zum anderen würden wir letzten Endes unsere Topics beim Konsumieren vermutlich sowieso wieder aggregieren müssen. Wir könnten diese Daten natürlich auch alle in einen einzelnen String verpacken, aber das würde die automatische Verarbeitung dieser Daten erheblich erschweren. Was wir brauchen, ist eine Datenstruktur, denn in Kafka selbst ist alles nur ein Byte-Array. Dies gibt uns als Anwender zwar viele Freiheiten, aber bringt natürlich auch eine gewisse Verantwortung mit sich. Wir könnten diese Datenstruktur zwar bei jeder einzelnen Nachricht mitliefern, aber das würde nur unnötig Overhead erzeugen.

Genau hier setzen Schemas an. Anstelle bei jeder einzelnen Nachricht auch die Struktur, die sich (fast) nie ändert, mitzuliefern, definieren wir einfach ein Schema, nach dem unsere

⁹⁾ <https://zz85.github.io/kafka-streams-viz/>

Nachrichten aufgebaut sind. Unsere Consumer müssen dieses Schema dann auch nur einmalig abrufen und sind dann in der Lage, unsere Nachrichten korrekt zu interpretieren. Prinzipiell unterliegen übrigens alle Nachrichten einem impliziten Schema, auch wenn wir dies nicht explizit angeben. Für unsere Rakete haben wir zum Beispiel Statusinformationen produziert, und als wir die Nachrichten anschließend konsumiert haben, wussten wir natürlich, wie diese zu interpretieren sind, ohne dass wir extra ein Schema angeben mussten. Solange wir allein arbeiten, mag das unter Umständen zwar noch okay sein, aber spätestens, wenn unsere Anwendung komplexer wird und damit vermutlich auch zahlreiche verschiedene Producer, Topics und Consumer mit einhergehen, welche von einem größeren Team an Programmierern verwaltet werden, sollten wir unbedingt explizit Schemas definieren!

4.1.3.2 Schemaanpassung und -kompatibilität

Wirklich spannend wird es allerdings, wenn wir ein Schema anpassen. Denn ein Topic und die Nachrichten innerhalb eines Topics repräsentieren letzten Endes Informationen aus der realen Welt, und diese ist nun mal lebendig und laufend Änderungen ausgesetzt. Demzufolge können sich auch unsere Daten und die Struktur dieser Daten im Verlauf der Zeit ändern. Vielleicht gibt es ein neues Feature, welches thematisch in ein bestehendes Topic eingegliedert werden soll, oder eine Funktionalität unserer bisherigen Anwendung wird entfernt. Ist dies der Fall, müssen wir natürlich auch unser Schema anpassen. Wenn wir nun aber ein bestehendes Schema anpassen, stehen wir auf einmal vor neuen Problemen. Wie sieht es mit der Rückwärts- oder Vorwärtskompatibilität aus? Ein Schema ist dann rückwärtskompatibel, wenn ein Consumer, welcher das aktuelle Schema benutzt, auch Daten, die mit dem vorherigen Schema produziert worden sind, verarbeiten kann. Vorwärtskompatibilität wiederum bedeutet, dass ein Consumer, welcher das letzte Schema verwendet auch Daten, welche mit dem neuen Schema produziert worden sind, lesen kann. Am besten wäre es natürlich, wenn unser Schema vollständig kompatibel, das heißt, zugleich vorwärts- und rückwärtskompatibel ist. Welche Änderungen dürfen wir aber nun an unseren Schemas vornehmen, um die jeweilige Kompatibilität zu erreichen?

Bei der Rückwärtskompatibilität ist es uns erlaubt, optional Felder hinzuzufügen oder bestehende Felder zu löschen. Außerdem müssen wir das neue Schema zunächst an unsere Consumer verteilen. Würden wir nämlich zuerst unsere Producer updaten und unsere neuen Nachrichten hätten ein nichtoptionales Feld weniger, dann würden unsere Consumer vergeblich nach diesem Feld suchen und könnten die Nachricht nicht parsen.

Bei der Vorwärtskompatibilität dürfen wir Felder hinzufügen oder optionale Felder entfernen. Bezuglich der Reihenfolge, in der wir die Schemas unserer Clients updaten, verhält es sich genau umgekehrt zur Rückwärtskompatibilität. Diesmal müssen wir zuerst unsere Producer updaten. Würden wir unsere Consumer zuerst updaten, würden diese Felder erwarten, die es aber in unseren Nachrichten noch nicht gibt.

Wenn wir vollständige Kompatibilität erreichen wollen, dürfen wir nur optionale Felder hinzufügen oder entfernen. An dieser Stelle ist es auch egal, ob wir zuerst die Consumer oder die Producer updaten.

Natürlich steht es uns auch frei, weder Vorwärts- noch Rückwärtskompatibilität zwischen unseren verschiedenen Versionen zu gewährleisten, und manchmal ist dies auch gar nicht möglich. Deshalb ist es auch wichtig, dass unsere Schemas versioniert sind und die zum

Serialisieren eines Nachrichteninhaltes verwendete Version des Schemas auch in der Nachricht mit angegeben ist, sodass die Consumer wissen, welches Schema sie zum Deserialisieren verwenden müssen.

4.1.3.3 Schemaformate

In Kafka gibt es verschiedene Datenformate, um Schemas zu definieren. Am geläufigsten sind *JSON*, *AVRO* und *PROTOBUF*. *JSON* ist ein sehr verbreitetes Datenformat und wird von vielen Anwendungen verwendet, da es auf eine sehr einfache Art und Weise ein sowohl für Menschen als auch Computer gut lesbares Dateiformat bereitstellt. Hierbei wird allerdings kein normales *JSON* verwendet, sondern das sogenannte *JSON Schema*, da es in *JSON* an sich keine Möglichkeit gibt, Schemas explizit zu definieren. Warum brauchen wir aber überhaupt andere Formate? Der Grund liegt darin, dass Kafka sämtliche Daten serialisiert, das heißt als Byte-Array abspeichert, und genau hier liegt die Schwäche von *JSON* gegenüber den anderen beiden Formaten. Sowohl *AVRO* als auch *PROTOBUF* sind speziell zur Serialisierung entwickelte Binärformate und verbrauchen im Vergleich zu *JSON* weniger Speicherplatz für dieselbe Menge an Informationen. Während *AVRO* zum Repertoire von Apache gehört und im Kern auf *JSON* Objekten basiert, wurde *PROTOBUF* von Google als schnellere, einfachere und effizientere Alternative zu *XML* entwickelt. Auch wenn wir an dieser Stelle nicht weiter auf die Details der einzelnen Serialisierer und Deserialisierer der verschiedenen Schemaformate eingehen, da dies den Rahmen dieses Buches sprengen würde, so wollen wir uns zumindest grob anschauen, wie ein Schema via *AVRO* definiert und benutzt werden kann.

4.1.3.4 Avro

Zunächst erstellen wir die Datei `rocket.avsc`, in der wir unser AVRO-Schema definieren:

```
{"namespace": "rakete.avro",
"type": "record",
"name": "Rakete_Status",
"fields": [
    {"name": "Name", "type": "string"},
    {"name": "Geschwindigkeit_m_s", "type": "int"},
    {"name": "Entfernung_m", "type": "int"}
]}
```

Als Namespace wählen wir `rakete.avro` und geben dem Schema den Namen `Rakete_Status`. Diese beiden Werte müssen zusammen einzigartig sein und dienen gemeinsam zur Identifizierung unseres Schemas. Die Art (`type`) unseres Schemas ist `record`. Dies erlaubt es uns, weitere Felder (`fields`) zu definieren. Felder haben jeweils einen Namen (`name`) und einen Datentyp (`type`). Wir wollen, dass unsere Nachrichten den Namen unserer Rakete, deren Geschwindigkeit in Meter/Sekunde und die Entfernung zur Erde in Metern enthalten.

Anschließend können wir das Schema nutzen und Daten entsprechend unserem Schema serialisieren. Wir nutzen hierfür ein Python-Skript, welches wir in der Datei `rocket.py` abspeichern:

```

import avro.schema
from avro.datafile import DataFileReader, DataFileWriter
from avro.io import DatumReader, DatumWriter

schema = avro.schema.parse(open("rocket.avsc", "rb").read())

writer = DataFileWriter(open("rocket.avro", "we"), DatumWriter(), schema)
writer.append({"Name": "Kafka", "Geschwindigkeit_m_s": 10000, "Entfernung_m": 10000000})
writer.append({"Name": "Kafka", "Geschwindigkeit_m_s": 9999, "Entfernung_m": 10100000})
writer.close()

reader = DataFileReader(open("rocket.avro", "rb"), DatumReader())
for user in reader:
    print(user)
reader.close()

```

Wir importieren zunächst alle nötigen Bibliotheken (vorher müssen wir eventuell noch die Python-Bibliothek *avro* installieren). Anschließend parsen wir unser eben erstelltes Schema (*avro.schema.parse*). Der *DataFileWriter* öffnet eine Datei (*rocket.avro*). Außerdem übergeben wir dem *DataFileWriter* den zu verwendenden Serialisierer (*DatumWriter*) und das eben geparsste Schema (*schema*). Unsere serialisierten Nachrichten speichern wir in der Datei *rocket.avro*. Danach öffnen wir eben diese Datei erneut und deserialisieren die enthaltenen Nachrichten wieder mit dem *DatumWriter*. Zum Schluss zeigen wir unsere Nachrichten auf unserer Shell an. Wenn wir unser Skript ausführen, erhalten wir die folgende Ausgabe:

```

$ python3 rocket.py
{'Name': 'Kafka', 'Geschwindigkeit_m_s': 10000, 'Entfernung_m': 10000000}
{'Name': 'Kafka', 'Geschwindigkeit_m_s': 9999, 'Entfernung_m': 10100000}

```

Die Datei *rocket.avro* enthält übrigens nicht nur unsere serialisierten Nachrichten, sondern auch das gesamte Schema, weshalb wir beim Deserialisieren auch nicht noch mal unser Schema angeben mussten. Genau hier kommen in der Praxis in Kafka Schema Registrys zum Einsatz, da es doch recht umständlich und vor allem ineffizient wäre, jedes Mal das Schema mitzuschicken. Stattdessen wird nur die ID unseres Schemas mit der Nachricht versendet, und anhand dieser ID können die Consumer das eigentliche Schema aus der Schema Registry erhalten und damit die Nachrichten deserialisieren.

4.1.3.5 Schema Registry

Was hat es nun aber mit der Schema Registry auf sich? Die Schema Registry ist nichts anderes als der Ort, an dem unsere Schemas gespeichert und abgerufen werden können. Wenn wir eine Schema Registry verwenden wollen, können wir zum Beispiel auf die von der Firma Confluent zurückgreifen, oder wir stellen mit der auf Python basierenden Open-Source-Alternative *Karapace* unsere eigene Schema Registry bereit. Die von einer Schema Registry verwalteten Schemas werden übrigens selbst in einem speziellen Kafka-Topic gespeichert! Schauen wir uns zum Abschluss dieses Kapitels noch kurz an, wie die Schema Registry von unseren Producern und Consumern verwendet wird.

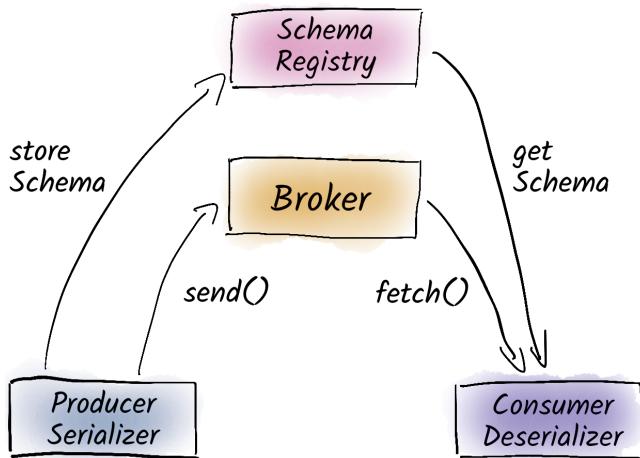


Bild 4.7 Producer und Consumer kommunizieren mit der Schema Registry.

Wir erstellen zunächst einen Producer und definieren unter anderem, welcher Serialisierer für die Werte unserer Nachrichten verwendet werden soll und wie die Adresse unserer Schema Registry lautet. Anschließend müssen wir nur noch unser Schema definieren und es in den entsprechenden Serialisierer laden. Ab jetzt können wir Nachrichten basierend auf unserem definierten Format serialisieren und anschließend produzieren. Jedes neue Schema bekommt hierbei eine eindeutige ID und landet automatisch in unserer Schema Registry. Für unseren Consumer müssen wir ebenfalls den verwendeten Serialisierer und die Adresse der Schema Registry definieren. Anschließend können wir wie gewohnt Nachrichten konsumieren, und nach der Deserialisierung liegen unsere Daten in dem entsprechenden Schema vor und können weiterverarbeitet werden. Standardmäßig verwendet Kafka den `StringSerializer`, um Daten zu serialisieren.

4.1.4 Sicherheit

Immer mehr Unternehmen benutzen Kafka, um kritische Unternehmens- oder gar Kunden-daten zu transportieren. Die Absicherung von Kafka-Clustern wird daher immer wichtiger.

4.1.4.1 Verschlüsselung

Kafka bietet von Haus aus Möglichkeiten, um die Transportschicht mittels *TLS* abzusichern. Darüber hinaus kann mithilfe unterschiedlicher Authentifizierungs- und Autorisierungsmechanismen der Zugriff auf die Daten eingeschränkt werden. Wichtig ist hier zu betonen, dass es in Kafka von Haus aus keine Möglichkeit gibt, Daten auf den Brokern selbst zu verschlüsseln. Das ist aus Kafkas Sicht die Aufgabe des Betriebssystems oder der Entwickler.

Im Auslieferungszustand ist *TLS* deaktiviert, lässt sich aber über die bei Java üblichen Parameter einstellen. Dazu müssen wir für jeden Broker ein Schlüssel-Paar erstellen und auch die Clients korrekt konfigurieren. Wir möchten an dieser Stelle nicht den konkreten

Prozess beschreiben. Die Kafka-Doku erklärt dies aber im Detail¹⁰. Wir empfehlen es, bei diesem Prozess sehr akribisch vorzugehen, da sich unserer Erfahrung nach sehr einfach Fehler einschleichen, die schwierig zu debuggen sind. Es ist auch möglich, einen Kafka-Cluster, der bisher unverschlüsselt kommuniziert, im Nachhinein auf TLS umzustellen, ohne dass hierzu das gesamte System heruntergefahren werden muss. Kafka kann nämlich auf unterschiedlichen Ports Verbindungen mit unterschiedlichen Konfigurationen annehmen. Bei einem Umstieg auf TLS würden wir zuerst einen zusätzlichen Port freischalten und einen Client nach dem anderen auf diesen TLS-Port migrieren. Sobald alle Clients TLS sprechen, schalten wir den unverschlüsselten Port ab.



Tipp

Je nach Java-Version führt TLS-Verschlüsselung zu mehr oder weniger Performance-Einbußen. Wir empfehlen stets die aktuelle Java-Version, aber mindestens Java 11 zu nutzen, um den Performance-Overhead zu reduzieren.

Wollen wir mit Kafka Ende-zu-Ende-Verschlüsselung anbieten, müssen wir dies selbst implementieren. Am einfachsten geht dies, indem wir einen Serializer und Deserializer mit Verschlüsselungsfunktion schreiben. Durch diese Flexibilität ist es auch möglich, lediglich einzelne Felder in unseren Daten zu verschlüsseln und öffentliche Daten unverschlüsselt zu speichern. Das Key-Management müssen wir dafür auch wieder selbst übernehmen.



Tipp

Wir empfehlen stets, das Key-Management professionellen Key-Management-Systemen (KMS) wie zum Beispiel Hashicorp Vault¹¹ oder dem KMS unseres Cloud-Anbieters zu überlassen.

4.1.4.2 Authentifizierung und Autorisierung

Authentifizierung ist der Prozess, bei dem sich eine Entität gegenüber einer anderen Stelle ausweist. Die Verwechslung mit Autorisierung, bei der es darum geht zu überprüfen, zu was diese Entität berechtigt ist, ist allgegenwärtig. Kafka bietet zur Authentifizierung zwei grundlegende Prinzipien an. Wenn wir TLS für die Transportverschlüsselung einsetzen, können wir TLS auch zur Authentifizierung der Clients benutzen. Alternativ können wir das sogenannte SASL-(Simple Authentication and Security Layer-)Framework zur Authentifizierung einsetzen. Dadurch können sich die Clients gegenüber den Brokern über Nutzername und Passwort (SASL-SCRAM) oder über Kerberos authentifizieren. Wir sind der Meinung, dass auf Kafka lediglich über technische Nutzer zugegriffen werden sollte, und favorisieren daher Authentifizierung über TLS oder über automatisch generierte Zugangsdaten per SASL-SCRAM.

¹⁰) https://kafka.apache.org/documentation/#security_ssl

¹¹) <https://www.hashicorp.com/products/vault>

Ist ein Nutzer authentifiziert, so darf dieser im Auslieferungszustand alles tun. Wir benötigen nun einen Autorisierungsmechanismus, der diese Rechte einschränkt. Dafür haben wir in Kafka zwei Möglichkeiten. Klassischerweise vergeben wir Rechte über ACLs. Wir können bestimmten Nutzern den Zugriff auf bestimmte Topics (schreibend oder lesend) gewähren oder verbieten. Genauso können wir mittels ACLs Administratorrechte auf dem Cluster einschränken. Wichtig ist genauso, das Erstellen von Topics einzuschränken.



Tipp

Die Automatisierung von Nutzerverwaltung (und Topic-Verwaltung) ist aus unserer Sicht ein Muss für alle Produktivsysteme.

Kafka Streams erstellt automatisch zahlreiche Topics, deshalb müssen wir Applikationen, die Kafka Streams verwenden, das Recht gewähren, Topics mit einem gewissen Präfix (die *app.id*) zu erstellen. Besonders wichtig ist aus unserer Sicht auch die Einschränkung, welcher Nutzer welche Consumer Group-ID benutzen darf. Es ist einigen unserer Kunden schon mehrfach passiert, dass Code zwischen Consumern kopiert wurde und jemand vergessen hat, die *group.id* zu ändern. Ohne korrekte ACLs würde ein Consumer den anderen Consumern die Offsets stehlen.



Tipp

Auch die Benutzung von Consumer Group-IDs sollte per ACLs eingeschränkt werden.

4.1.4.3 Zookeeper

Zookeeper war vor Kafka 2.5 quasi nicht sinnvoll abzusichern. Erst mit Version 2.5 unterstützt Kafka eine Zookeeper-Version, die Transportverschlüsselung und Authentifizierung über TLS. Vor 2.5 war die einzige Möglichkeit, Zookeeper abzusichern, es in ein privates Netzwerk zu sperren und lediglich den Kafka-Brokern und den Administratoren Zugriff darauf zu gewähren. Es gab auch Ansätze, die Zookeeper-Kommunikation über das Werkzeug *stunnel*¹² per TLS zu tunneln. Zookeepers Sicherheit muss unabhängig von Kafka konfiguriert werden. Mit Kafka 3.0 entfällt dieser Aufwand, da wir kein Zookeeper-Ensemble mehr haben.

4.1.5 Desaster-Management

In diesem letzten Kapitel des Themenblocks *Kafka-Ökosystem* werden wir uns mit dem Desaster-Management in Kafka beschäftigen. Wir werden uns ansehen, welche Optionen wir in einem Worst-Case-Szenario haben, und auch, wie wir die Wahrscheinlichkeit eines solchen Desasters von vornherein minimieren können. Zunächst sollten wir uns im Klaren

¹²⁾ <https://www.stunnel.org/>

darüber werden, was genau wir unter einem solchen Szenario überhaupt verstehen und welche Fälle von vornherein eher weniger kritisch sind.

4.1.5.1 Was kann schon schiefgehen?

Es gibt drei verschiedene Arten von Ausfällen, die uns in einem Kafka-Cluster begegnen können. Dies sind jegliche Art von Problemen mit der Nachrichten- oder Datenübertragung (Netzwerk), Probleme mit unseren Brokern (Compute) oder Ausfälle unseres persistenten Speichers (Storage). Oft werden diese Ausfälle aber einfach durch den Faktor Mensch hervorgerufen.

Betrachten wir zunächst Netzwerkausfälle. Insbesondere in einem verteilten System wie Kafka ist es mit der Zeit fast schon garantiert, dass Netzwerkprobleme ab und zu auftreten. Je nach der genauen Art unserer Netzwerkprobleme hat dies unterschiedliche Konsequenzen für unseren Kafka-Cluster. Am häufigsten wird vermutlich der Fall auftreten, dass einzelne Clients mit Verbindungsproblemen jeglicher Art zu kämpfen haben. Dies kann eine langsame Verbindung sein oder auch ein kompletter Ausfall. Sind dagegen mehrere Clients betroffen, so liegt die Ursache vermutlich im Rechenzentrum unseres Kafka-Clusters, und das Problem ist etwas ernster. Vielleicht ist aber auch die Verbindung zwischen einzelnen Brokern gestört. Im Allgemeinen lässt sich zu Netzwerkproblemen im Zusammenhang mit Kafka sagen, dass sie zwar sehr ärgerlich sind, aber in der Regel auch schnell wieder behoben sind. Mitunter verschwinden sie auch von selbst, weil der Link zum Beispiel nur kurzzeitig überlastet war oder das Netzwerkproblem außerhalb unseres Zuständigkeitsbereichs lag und sich die betreffenden Stellen um die Lösung der Einschränkung gekümmert haben. Wirklich problematisch sind Netzwerkprobleme eigentlich auch nur, wenn wir echtzeitkritische Anwendungen haben. Außerdem werden Netzwerkprobleme auch von unserem Netzwerkstack erkannt, da Kafka auf TCP basiert, und an unsere Clients weitergegeben. Durch die asynchrone Architektur von Kafka können Nachrichten, sobald die Netzwerkprobleme behoben sind, einfach erneut produziert oder konsumiert werden, sodass wir an dieser Stelle weder Angst vor Datenverlust noch vor inkonsistenten Zuständen haben müssen.

Etwas schwieriger wird es allerdings, wenn wir Probleme mit unseren Brokern selbst bekommen. Zwar können wir aufgrund von Kafkas Replikationsstrategie in Abhängigkeit zur Größe unseres Clusters einige Ausfälle verkraften (unter Beachtung der minimalen Anzahl an In-Sync-Replicas), allerdings kann es hier mitunter zu Datenverlust kommen, da je nach ACK-Strategie eine Nachricht aus Sicht eines Producers bereits erfolgreich produziert wurde, auch wenn sie vielleicht noch nicht auf unserer Festplatte persistiert ist. Fällt der entsprechende Broker aus, bevor die Nachrichten tatsächlich committet worden sind, dann sind eben genau diese Daten verloren. Wir möchten an dieser Stelle daher noch mal betonen, dass wir nur mit `acks=all` und einem sinnvollen Wert für die minimale Anzahl an ISR die Zustellung garantieren können. Dass einzelne Broker kurzzeitig nicht erreichbar sind, kommt regelmäßig vor. Hierfür muss der Broker beziehungsweise Server nicht mal selbst einen Fehler aufweisen. Vielleicht besteht gerade auch nur ein temporäres Netzwerkproblem, wodurch unser Broker nicht mehr erreichbar ist. Außerdem wird der zugrundeliegende Server vermutlich regelmäßig neu gestartet, um wichtige Updates einzuspielen. Wie oft ein Broker nicht erreichbar war, lässt sich ganz gut an der Leader-Epoche festmachen, da diese bei jedem Wechsel eines Partitions-Leaders inkrementiert wird. Unsere Erfahrungen bei Kunden haben gezeigt, dass es nicht ungewöhnlich ist, dass die Leader Epoch nach einem Jahr Betrieb bereits im dreistelligen Bereich liegt. Solange nicht alle Broker bezie-

hungsweise Replicas einer Partition gleichzeitig ausfallen, stellt dies allerdings kein Problem für uns dar.

Der Ausfall unseres persistenten Speichers ist eines der schlimmsten Probleme, die uns in Kafka passieren können. Im besten Fall ist auch dieser Ausfall nur temporär, weil zum Beispiel der Netzwerkspeicher kurzzeitig nicht erreichbar ist oder der Zugriff aufgrund einer kurzzeitigen Lastspitze langsam ist. Wie sieht es aber aus, falls unser persistenter Speicher wirklich ausfällt? Im Idealfall ist nur ein Broker betroffen, sodass wir den Ausfall gut kompensieren können, und wir merken gar nicht, wenn eine einzelne Festplatte den Geist aufgibt. Was passiert aber, wenn unser Log tatsächlich und unwiederbringlich korrumptiert ist? Genau diese Frage werden wir in den nächsten Abschnitten ausführlich beantworten und auch zeigen, wie wir ein solches Worst-Case-Szenario von vornherein vermeiden können.

4.1.5.2 Backups in Kafka

Die naheliegendste Lösung sind vermutlich Backups. Allerdings sind diese in Kafka nur bedingt sinnvoll. Dies hängt damit zusammen, dass wir in einem Kafka-Cluster laufend neue Nachrichten produzieren und auch permanent Nachrichten konsumieren. Backups sind allerdings Snapshots unseres Logs zu festgelegten Zeitpunkten. So könnten wir zum Beispiel jeden Tag um drei Uhr nachts ein Backup durchführen und könnten dies natürlich in einem Fehlerfall wieder einspielen. Aber leider ist dies in den meisten Fällen nicht sehr praktikabel und mitunter auch inakzeptabel, da wir hierdurch alle nach dem Backup produzierten Nachrichten definitiv verlieren würden. Außerdem würden wir sehr wahrscheinlich mit inkonsistenten Zuständen zu kämpfen haben, da einige Consumer diese verlorenen Nachrichten wohl schon gelesen haben und andere eben noch nicht. Natürlich könnten wir jetzt auf die Idee kommen, einfach häufiger Backups zu machen, aber selbst wenn wir minütlich Backups erstellen würden, hätten wir mit Datenverlust zu kämpfen, und aus Performance-Sicht wäre es vermutlich selbst bei inkrementellen Backups eine Katastrophe, unsere Daten jede Minute zu sichern. Abgesehen davon, würde es auch Stunden oder noch länger dauern, ein Kafka-Cluster anhand eines Backups wiederherzustellen. All dies führt dazu, dass Backups in den meisten Fällen keine Alternative sind. An dieser Stelle bietet sich ein Zitat eines unserer KundInnen an, der sagte:

„Wenn wir jemals Backups in Kafka einspielen müssten, sind wir sowieso pleite.“

Dies ist allerdings kein Grund, den Kopf in den Sand zu stecken, denn Kafka bietet zum Glück gute Alternativen für die Desaster-Recovery, die wir uns in den nächsten Abschnitten genauer ansehen¹³.

4.1.5.3 Stretched Cluster

Wir können die Wahrscheinlichkeit eines Totalausfalls erheblich verringern, indem wir unser Kafka-Cluster über mehrere Rechenzentren erstrecken und ein sogenanntes *Stretched Cluster* betreiben. Während wir in den Beispielen in unserem Buch ein lokales Cluster mit mehreren Brokern auf nur einem Rechner aufgesetzt haben, versucht ein Stretched Cluster genau das Gegenteil. Das heißt, wir versuchen unsere Broker und damit unser Cluster auf

¹³⁾ Einer der Autoren hatte einen Ansatz, wie Backups in Kafka zuverlässig umgesetzt werden können, mangels Zeitprobleme hat er diesen Ansatz bisher nicht weiterverfolgt. Sollten Sie Interesse daran haben, sprechen Sie uns an: buch@streamcommit.de

verschiedene Rechenzentren zu verteilen. Dies hat den Vorteil, dass, wenn ein Rechenzentrum ausfällt, wir immer noch ein funktionierendes Kafka-Cluster haben, da der Ausfall eines ganzen Rechenzentrums für uns letzten Endes auch nicht mehr ist als der Ausfall eines einzelnen Brokers. Solange also nicht mehrere Rechenzentren gleichzeitig ausfallen, sind wir weiterhin einsatzfähig. Die Wahrscheinlichkeit einer ernsthaften Störung unseres Kafka-Clusters wird dadurch erheblich verringert, da die Wahrscheinlichkeit, dass zwei oder sogar drei und mehr Rechenzentren gleichzeitig ausfallen, natürlich sehr gering ist. Wenn wir dagegen unser Cluster in einem einzigen Rechenzentrum betreiben, so würde auch das Kafka-Cluster im Fehlerfall komplett ausfallen.



Tipp

Stretched Cluster sollten über mindestens drei Rechenzentren verteilt werden. Bei zwei Rechenzentren kann es unter Umständen passieren, dass der Koordinations-Cluster keine Mehrheit findet, da zwei von drei Knoten gleichzeitig ausfallen können.

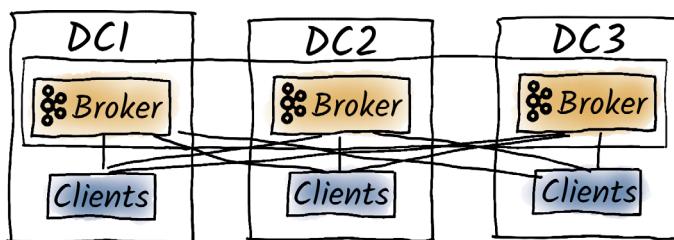


Bild 4.8 Stretched Cluster

Abgesehen von der deutlich erhöhten Ausfallsicherheit, können *Stretched Cluster* sich unter Umständen auch noch positiv auf die Performance unseres Clusters auswirken. Normalerweise konsumieren Consumer immer vom Partitions-Leader, aber seit KIP-392¹⁴ ist es auch möglich, von anderen Replicas zu konsumieren, genauer gesagt vom nächsten Replica. So können wir zum einen die Last der Broker noch besser verteilen und zum anderen trotz verteiltem Kafka-Cluster und verteilten Consumern noch eine möglichst geringe Laufzeitverzögerung zwischen Broker und Consumer erreichen. Woher weiß jetzt aber ein Consumer, welches das nächste Replica ist? Ganz einfach, über die Rack-ID, welche sowohl auf Seiten der Broker als auch auf Seiten der Clients ein gewisses Standort-Bewusstsein schafft.

¹⁴⁾ <https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica>



Tipp

Sollte es uns nicht möglich sein, unsere Kafka-Cluster auf verschiedene Rechenzentren zu verteilen, so sollten wir zumindest versuchen, unsere Broker auf verschiedene Racks in diesem Rechenzentrum zu verteilen, um so innerhalb eines Rechenzentrums die bestmögliche Ausfallsicherheit zu erreichen. Im Idealfall sollten diese Racks komplett unabhängig voneinander sein, das heißt, eine unterschiedliche Netzwerk- oder Stromanbindung haben. Die Rack-ID sollte auch hier gesetzt werden.

4.1.5.4 MirrorMaker

Eine weitere gute Möglichkeit, uns vor Ausfällen und Datenverlust zu schützen, ist das Spiegeln unseres gesamten Clusters. Im Gegensatz zu konventionellen Backups kopieren wir hierbei kontinuierlich unsere Nachrichten von einem Cluster in ein anderes. Und im Gegensatz zu normaler Replikation können wir so Cluster über größere Entfernungn spiegeln. In Kafka gibt es hierfür mit *MirrorMaker*¹⁵ auch bereits ein Tool, welches uns die ganze Arbeit abnimmt. MirrorMaker selbst basiert auf Kafka Connect, welches wir bereits im Verlauf dieses Buches kennengelernt haben. Das Beste an MirrorMaker ist, dass wir nicht nur unsere Topics und Nachrichten spiegeln können, sondern gleichzeitig auch die ACLs der Topics und die Consumer Offsets. MirrorMaker nutzt hierfür drei verschiedene Kafka Connectoren. Der *MirrorSourceConnector* ist dafür zuständig, unsere eigentlichen Topics (und ACLs) zu spiegeln. Der *MirrorCheckpointConnector* kümmert sich um die Offsets. Außerdem gibt es noch den *MirrorHeartbeatConnector*, welcher periodisch die Verbindung zwischen den Clustern überprüft.

Wir wissen jetzt, dass es einen einfachen und effizienten Weg gibt, unsere Cluster mit MirrorMaker zu spiegeln. Schauen wir uns nun an, welche verschiedenen Architekturen wir mit MirrorMaker in der Praxis erstellen können, und vor allem, was dies für unser Desaster-Management bedeutet.

Die einfachste Topologie ist die *Aktiv-Passiv-Paarung*. Hierbei haben wir ein aktives Cluster, in das wir produzieren und von dem wir auch konsumieren. Dieses Cluster spiegeln wir dann in ein zweites Cluster. Abgesehen davon produzieren wir keine Nachrichten in dieses Cluster, daher bezeichnen wir es auch als passiv.

¹⁵⁾ Wir sprechen hier stets von MirrorMaker 2. MirrorMaker 1 ist inzwischen deprecated und taugt nicht als Replikationsmechanismus.

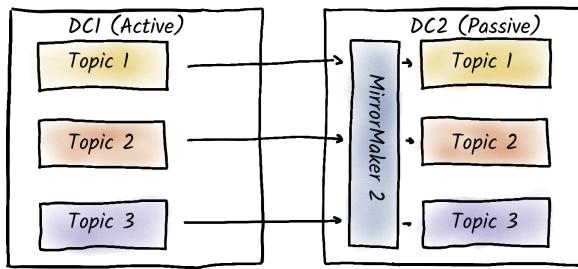


Bild 4.9 Ein Aktiv-Passiv-Cluster mit MirrorMaker

Fällt nun unser aktives Cluster aus, so übernimmt das passive Cluster die Rolle des aktiven Clusters, und unsere Producer fangen an, in dieses Cluster ihre Nachrichten zu produzieren, und auch die Consumer lesen jetzt die Nachrichten aus diesem Cluster. Sollte unser ursprüngliches aktives Cluster wieder erreichbar sein, wird es allerdings nicht wieder zum aktiven Cluster. Stattdessen bleibt unser gespiegeltes Cluster aktiv, und wir bauen ein komplett neues Kafka-Cluster auf, welches uns als neues passives Cluster dient. Der Grund hierfür ist, dass wir ansonsten sehr wahrscheinlich mit inkonsistenten Zuständen zu kämpfen hätten. Da es sehr umständlich ist, bei jedem kleinen Ausfall ein komplett neues Cluster zu bauen und anschließend alle unsere Topics zu spiegeln, empfehlen wir es, diese Architektur zu vermeiden. Stattdessen empfehlen wir die *Aktiv-Aktiv-Paarung*.

Bei der Aktiv-Aktiv-Paarung haben wir zwei komplett gleichwertige Cluster, die sich gegenseitig spiegeln, das heißt, wir haben in jedem Cluster einen MirrorMaker, welcher das jeweils andere Cluster in das eigene Cluster spiegelt. Damit dies möglich ist, wurden mit *MirrorMaker 2* sogenannte *Remote Topics* und *Remote Partitions* eingeführt. Remote Topics und Partitionen sind durch eine spezielle Namenskonvention gekennzeichnet, die das Ursprungs-Cluster als Präfix vor den Topic- beziehungsweise Partitionsnamen hinzufügt. Die Besonderheit bei diesen Topics und Partitionen ist es, dass sie von MirrorMaker nicht gespiegelt werden und somit Endlosschleifen von Topic-Spiegelungen, zu denen es ansonsten kommen würde, verhindert werden.

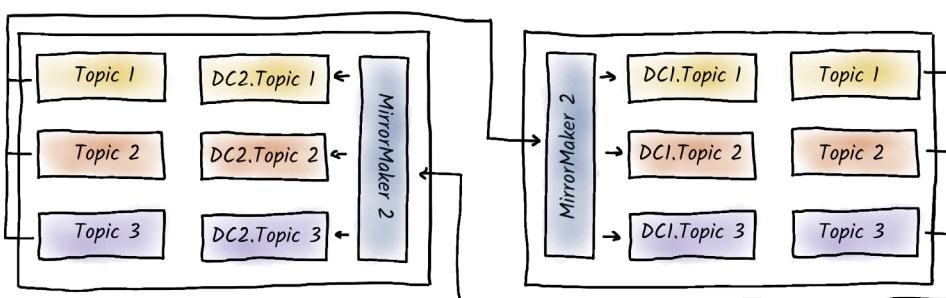


Bild 4.10 Ein Aktiv-Aktiv Cluster mit MirrorMaker

Die Remote Topics und Partitionen sind übrigens genaue Mappings von den originalen Topics und Partitionen und damit auch von den originalen Replicas. Fällt eines der Cluster

aus, so können sämtliche Aufgaben dieses Cluster sofort von dem anderen Cluster übernommen werden. Im Gegensatz zur Aktiv-Passiv-Paarung müssen wir im Fehlerfall auch kein neues Cluster erstellen, sondern können einfach mit unserem bisherigen Cluster weiterarbeiten, vorausgesetzt, das fehlerhafte Cluster ist wieder erreichbar. Dies ist möglich, da wir diesmal in beide Richtungen spiegeln und somit auch wieder einen konsistenten Zustand herstellen können. Producer können sich bei der Aktiv-Aktiv-Paarung aussuchen, in welches Cluster sie produzieren. Bei den Consumern ist es allerdings etwas komplizierter, da hierdurch die Daten auch auf beiden Clustern verteilt sind. Dies lässt sich aber einfach lösen, indem die Consumer einfach ein Topic und alle dazugehörigen Remote Topics konsumieren. Im Anschluss müssen sie lediglich die Daten korrekt aggregieren.

Die letzte Architektur, die wir in diesem Abschnitt erläutern, ist die sogenannte *Hub-and-Spoke*-Topologie. Bei dieser Architektur haben wir ein zentrales Kafka-Cluster und viele kleinere lokale Cluster. Das zentrale Cluster dient als Aggregations-Cluster für alle dezentralen Cluster, das heißt, es besteht im Prinzip aus mehreren MirrorMakern, welche jeweils für die Spiegelung eines Clusters zuständig sind.

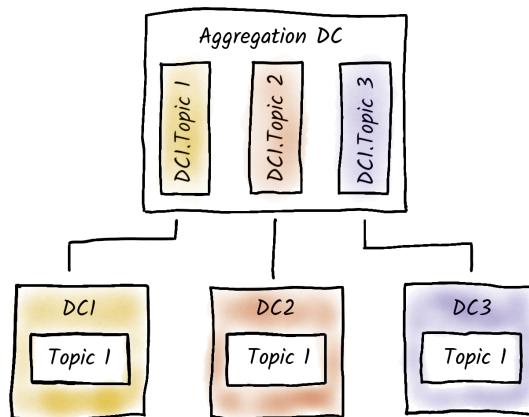


Bild 4.11 Eine Hub-and-Spoke-Topologie mit MirrorMaker 2

Fällt unser zentrales Cluster aus oder ist es nicht erreichbar, können weiterhin Daten in unsere lokalen Cluster produziert oder von unseren lokalen Clustern konsumiert werden. Ist unser zentrales Cluster wieder erreichbar, werden einfach die neu produzierten Nachrichten gespiegelt, und das Cluster ist anschließend wieder auf dem aktuellen Stand. Die lokalen Cluster können dadurch unabhängig vom zentralen Cluster agieren.

Fällt eines unserer lokalen Cluster aus, so sind die Nachrichten weiterhin in unserem zentralen Cluster vorhanden und sind nicht verloren. Allerdings könnten wir vermutlich kurzzeitig keine neuen Nachrichten mehr in die entsprechenden Topics produzieren und hätten an dieser Stelle zumindest mit einem Teilausfall unseres Systems zu kämpfen. Sollte unser Cluster, weil die Probleme gravierender sind, nicht mehr online kommen, so können wir einfach ein neues lokales Cluster aufsetzen.

Hub-and-Spoke-Topologien werden überall da benutzt, wo wir von vornherein wissen, dass die Verbindung zum Hauptsystem unzuverlässig oder fast unmöglich ist, wie zum Beispiel bei Supermarktketten, Flugzeugen und Ähnlichem.

Die Firma Confluent bietet mit dem *Confluent Replicator* eine proprietäre Alternative zum MirrorMaker an.

4.1.6 Zusammenfassung

In diesem Kapitel haben wir uns sehr ausführlich mit dem Kafka-Ökosystem beschäftigt. Mit Kafka Connect haben wir zunächst ein sehr wichtiges Tool kennengelernt, um Kafka in andere Unternehmensprozesse beziehungsweise in andere Technologien einzubinden. Hierbei haben wir uns angesehen, wie genau Kafka Connect mit anderen Systemen kommuniziert, und das Ganze auch anhand eines Praxisbeispiels veranschaulicht. Außerdem sind wir kurz auf die Skalierbarkeit und Ausfallsicherheit von Kafka Connect eingegangen. Im Anschluss haben wir mit Kafka Streams ein weiteres wichtiges Tool der Kafka-Familie kennengelernt. In diesem Zusammenhang haben wir als Erstes Stream Processing an sich erläutert, bevor wir auf die genauen Funktionen von Kafka Streams eingegangen sind. Außerdem haben wir erfahren, dass wir verschiedene Operationen in Kafka Streams haben, welche es uns ermöglichen, verschiedene Streams miteinander zu kombinieren und auch zu filtern. Hierbei sind wir insbesondere auf das Konzept der KTables eingegangen, welches uns zum Beispiel ermöglicht, nur den aktuellen Wert zu einem Key zu speichern. Im Anschluss haben wir die Serialisierung und Deserialisierung unserer Nachrichten genauer beleuchtet. Wir haben verschiedene Schemas kennengelernt und sind dabei auch auf Herausforderungen bei nachträglichen Anpassungen von Schemas eingegangen, insbesondere auf die damit verbundenen potenziellen Kompatibilitätsprobleme. Anhand von Avro haben wir uns auch beispielhaft ein Schema angesehen. Wir haben außerdem erfahren, dass wir Schemas in sogenannten Schema Registrys speichern können. Zum Abschluss dieses Kapitels haben wir uns mit dem Disaster-Management in Kafka beschäftigt. Hierfür haben wir uns zunächst angeschaut, was überhaupt alles schiefgehen kann. Im Anschluss haben wir uns verschiedene Möglichkeiten zur Erhöhung der Ausfallsicherheit und zur Verhinderung eines Totalausfalls angesehen. Hierbei haben wir kurz Backups besprochen, welche allerdings eher ungeeignet sind. Im Anschluss haben wir ausführlich über sogenannte Stretched-Cluster gesprochen und wie diese helfen die Ausfallwahrscheinlichkeit zu reduzieren. Zu guter Letzt haben wir mit MirrorMaker ein Tool kennengelernt, welches es uns relativ einfach ermöglicht, ganze Cluster zu spiegeln.

■ 4.2 Vergleich mit anderen Technologien

Wir haben in diesem Buch Kafka ausführlich beschrieben und auch viele Themen angegangen, die für den Einsatz in Unternehmen relevant sind. Was wir bisher aber ausgelassen haben, ist die Diskussion, wann der Einsatz von Kafka überhaupt sinnvoll ist und wie wir Kafka mit anderen Technologien vergleichen können. Wir möchten an dieser Stelle darauf hinweisen, dass es die Aufgabe von IT-Architekten ist, die beste Technologie beziehungsweise Architektur für den gegebenen Anwendungsfall und insbesondere auch für die gegebenen Rahmenbedingungen zu finden. Was in einem Unternehmen ein sinnvoller

Technologieeinsatz ist, kann in einem anderen Unternehmen mit dem gleichen Anwendungsfall zu katastrophalen Folgen führen. So spannende neue Technologien wie Kafka können uns auf der einen Seite helfen, architekturelle und technische Herausforderungen zu lösen und Geschäftsmodelle zu ermöglichen, die so nicht ohne Weiteres umsetzbar wären. Aber am Ende geht es nie um die Technologie an sich, sondern wie wir diese im Unternehmen einsetzen und wie wir die Menschen, die von dieser Technologie beeinflusst werden, mitnehmen.

Es gibt definitiv keine Allheilösung für alle Probleme, und auch die eine alternativlose Lösung für ein gegebenes Problem ist sehr selten zu finden. Insbesondere fangen wir in den meisten Unternehmen nicht auf der grünen Wiese an zu planen, sondern müssen uns in die bestehende IT-Landschaft einfügen.

Mit welchen Technologien können wir dann überhaupt Kafka vergleichen? Dafür müssen wir noch mal genauer darauf eingehen, wofür wir Kafka überhaupt einsetzen wollen. Kafka ist ein System, um Daten zwischen Services auszutauschen. Wie im viel zitierten Paper *Data on the Outside versus Data on the Inside* von Pat Helland beschrieben, unterscheiden wir üblicherweise zwischen Daten in einem Service und Daten außerhalb von Services. Daten innerhalb eines Service sind an diesen Service gekapselt, und wir speichern sie in einem Format ab, um unsere Service-Ziele bestmöglich zu erreichen. Das kann in einem Such-Service ein ganz anderes Datenformat sein als in einem Analyseservice, obwohl die gleichen Daten gespeichert werden. Daten außerhalb von Services sind darauf ausgelegt, geteilt zu werden, und werden nach einem expliziten oder impliziten Schema erstellt oder konsumiert.

Oft nutzen wir für Daten innerhalb von Services relationale Datenbanken oder gar sehr spezielle Technologien, die am besten für die anfallenden Aufgaben geeignet sind. Für Daten außerhalb von Services nutzen wir auf der einen Seite Messaging-Systeme, oder wir haben gar keinen expliziten Speicher für solche Daten, sondern wir übermitteln sie über synchrone Protokolle wie HTTP-REST. In vielen Unternehmen haben wir dennoch meistens Datenbanksysteme, auf die von vielen unterschiedlichen Services zugegriffen wird; mit allen Vor- und Nachteilen, die sich dadurch ergeben.

4.2.1 Klassische Messaging-Systeme

Der Vergleich zu klassischen Messaging-Systemen drängt sich auch aus dem Grund auf, dass Kafka oft als solches eingesetzt wird. Das Ziel eines Messaging-Systems ist es, Nachrichten von Produzenten zu Konsumenten zu bewegen. Optimal erfolgt das performant und zuverlässig. Oft bieten Hersteller solcher Systeme zahlreiche zusätzliche Funktionen bis hin zu Enterprise-Service-Bussen (ESB), die darauf ausgelegt sind, alle Systeme eines Unternehmens miteinander zu integrieren.

Zu diesen Funktionen gehört zum Beispiel das Routing von Nachrichten. Der Producer produziert eine Nachricht, und das Messaging-System stellt sicher, dass der Consumer diese Nachricht bekommt, unabhängig davon, wo sich die Systeme befinden. Vielleicht befinden sich die Systeme auf unterschiedlichen Kontinenten in unterschiedlichen Jurisdiktionen und bei unterschiedlichen Konzernteilen. ESB kümmern sich um die Zustellung, ohne dass der Producer viel beachten muss. Vielleicht unterstützen die Producer und Consumer nur

zueinander inkompatible Nachrichtenformate. Solche Messaging-Systeme kümmern sich um die Konvertierung, ohne dass die Producer oder Consumer verändert werden müssen. Auch können die Messaging-Systeme darauf achten, dass Consumer nicht überlastet werden, und dem Producer melden, falls dieser langsamer produzieren soll.

Spätestens an dieser Stelle merken wir, dass Kafka eine andere Philosophie verfolgt. Es unterstützt diese ganzen Features nicht. Wir müssen diese selbst implementieren, wenn sie uns wichtig sind. Die Philosophie solcher klassischen Messaging-Systeme und vor allem die von ESBs ist es, dass die Services sich nicht um die Kommunikation kümmern müssen, sondern nur Daten an die Transportschicht schicken und diese sich um alles kümmert. Wir können auch sagen, dass bezogen auf die Kommunikation die Services selbst dumm sind und die Transportschicht, die Pipe, smart.

Kafka verfolgt die gegenteilige Philosophie. Kafka selbst ist dumm (bezogen auf die Kommunikation). Es ist lediglich ein System, wo wir Daten in Logs ablegen können, diese dann aber persistiert und von Consumern zu einem beliebigen späteren Zeitpunkt (beschränkt durch die Retention-Zeit) abgeholt werden. Die Services selbst müssen sich darum kümmern, wie sie auf Kafka zugreifen, welche Nachrichten gelesen und wie sie geschrieben werden. Das gibt uns zwei Ebenen von Entkopplung, einmal die physische Entkopplung, das heißt, die Producer und Consumer müssen nicht räumlich beieinanderliegen. Diese Entkopplungsebene bekommen wir auch bei den anderen Messaging-Systemen. Der Producer muss darüber hinaus nicht wissen, wer der Consumer ist und ob dieser überhaupt existiert. Die spannendere Ebene ist aber die zeitliche Entkopplung. Vielleicht ist der Consumer noch gar nicht existent, ist in Wartung oder hat gerade mit erhöhter Last zu kämpfen. Vielleicht soll der Consumer nur einmal die Woche laufen. Das alles interessiert den Producer in Kafka nicht. Dieser kann einfach Daten in Topics schreiben, ganz unabhängig davon, in welchem Zustand die Consumer sind.

Ein weiterer Punkt, der insbesondere in größeren Unternehmen relevant ist, ist die Frage, wer für den korrekten Betrieb und das Management der Messaging-Technologie zuständig ist. Meistens gibt es ein zentrales Team, welches sich darum kümmert. Bei ESB-Systemen besagt unsere Erfahrung, dass sich dieses Team um jegliche Konfigurationen kümmert und meistens komplexe Prozesse durchlaufen werden müssen, bevor Änderungen vorgenommen werden dürfen. Dies sorgt auf der einen Seite dafür, dass Wildwuchs im Zaum gehalten wird, aber auf der anderen Seite verlieren wir dadurch viel Agilität, weil es mitunter Wochen oder Monate dauert, bis gewünschte Änderungen umgesetzt werden. Wir plädieren auch beim Einsatz von Kafka für Prozesse zum Konfigurieren von Topics, technischen Benutzern und Zugriffsrechten. Teilautomatisierte Prozesse helfen hier Teams, einen guten Kompromiss zwischen Verwaltbarkeit und Flexibilität zu finden. Wir glauben auch, dass durch Kafkas philosophischen Ansatz, nur unveränderbare Ströme von Fakten zu zentralisieren und wenig Logik in die Transportschicht zu legen, der Konfigurationsaufwand massiv reduziert und so den Teams die Freiheit gegeben wird, sich unabhängiger voneinander weiterzuentwickeln und Änderung schneller umzusetzen.

An dieser Stelle möchten wir nochmals betonen, dass Messaging-Systeme auch abseits von Kafka viele gute Anwendungsfälle haben und wir Kafka oft im Einsatz zusammen mit anderen Messaging-Systemen finden. Insbesondere an Schnittstellen zwischen agilen, sich schnell entwickelnden Services, Teams und Systemen, die Kern-Businessprozesse verwalten, wie Host-Systeme oder SAP-Systeme.

4.2.2 REST

Für Microservice-basierte Architekturen wurde und werden oft HTTP-REST oder andere synchrone Kommunikationssysteme als Mittel der Wahl eingesetzt und vorgeschlagen. REST-ähnliche Protokolle finden sich überall in der IT und sind eine einfache und effiziente Art, Services miteinander zu verbinden. Wenn es aber darum geht, Services voneinander zu entkoppeln und damit unabhängiger voneinander zu machen, kommen solche Technologien an ihre Grenzen. Auf den ersten Blick scheint es wenige Fallstricke zu geben, wenn wir andere Services einfach per HTTP-Call aufrufen. Aber was passiert, wenn der aufgerufene Service mal nicht erreichbar oder wenn dieser langsam ist? Was passiert, wenn der aufgerufene Service einen anderen REST-Service aufrufen muss, um unsere Anfrage zu bearbeiten? Diese synchronen Aufruf-Kaskaden können schnell zu sehr unübersichtlichen und schwer zu verstehenden Systemarchitekturen führen.

Asynchronität ist nicht die Lösung für alle Probleme, kann aber in vielen Fällen die Systeme voneinander entkoppeln und fehltoleranter machen. Wenn Services synchrone Kommunikationskanäle nutzen, bewegen sich Daten meistens erst, wenn sie benötigt werden, und sind somit meistens woanders, wenn wir sie lokal benötigen. In Kafkas event-basierter Philosophie sollten Daten stets fließen, und jeder Service holt sich kontinuierlich genau die Daten, die dieser benötigt. Wir können hier eine viel bessere Unterscheidung zwischen Daten innerhalb und außerhalb von Services treffen.

4.2.3 Relationale Datenbanken

Es ist heutzutage üblich, es als Anti-Pattern zu betrachten, wenn es mehr als einen oder einige wenige Services gibt, die auf eine gemeinsame Datenbank zugreifen. Relationale Datenbanken sind ideal für Daten innerhalb eines Service, aber sollten im Allgemeinen nicht als Kommunikationsplattform zwischen Services dienen. Aber wir möchten Kafka gar nicht auf dieser Ebene mit Datenbanken vergleichen.

Wenn wir uns Datenbanken genauer anschauen, stellen wir fest, dass so gut wie jede (relationale) Datenbank intern auf Logs basiert. Der Commit Log hält alle Änderungen an der Datenbank fest, bevor die eigentlichen Tabellen aktualisiert werden. Diese Logs werden einerseits benutzt, um Daten zwischen unterschiedlichen Instanzen einer Datenbank zu replizieren, aber auch für Fehlerfälle. Wenn unser Datenbankserver abstürzt und wieder hochgefahren wird, nutzen Datenbanken oft den Commit Log, um den korrekten Zustand wiederherzustellen. Basierend auf den Daten im Commit-Log können wir nicht nur die Tabellen erstellen, sondern auch Indizes, *Materialized Views* und anderes aktuell halten.

Wir können nun die Topics in Kafka als genau solche Logs betrachten. Aber statt nun in Kafka selbst die Funktionen einer Datenbank nachzubilden, lagern wir diese Funktionalitäten in unsere Services aus. Genau so, wie Pat Helland es mit den Daten innerhalb und außerhalb von Services beschrieben hat. Das Log als perfekte Datenstruktur, um Daten zwischen Services auszutauschen und Services zu entkoppeln. Und im Service selbst bauen wir *Materialized Views* der Daten im Log auf. Wenn wir über die Daten im Log suchen möchten, warum nicht einfach *Elasticsearch*¹⁶ benutzen? Wenn wir die Daten für den Abruf auf

¹⁶⁾ <https://www.elastic.co/de/elasticsearch/>

unserer viel besuchten Website warmhalten möchten und dafür einen Cache benutzen möchten, warum die Daten dann nicht in einem In-Memory Cache wie *Redis*¹⁷ vorhalten? Vielleicht wollen wir auch komplexe Ad-hoc-Abfragen über die Daten machen? Dann nutzen wir im Service eine relationale Datenbank wie PostgreSQL.

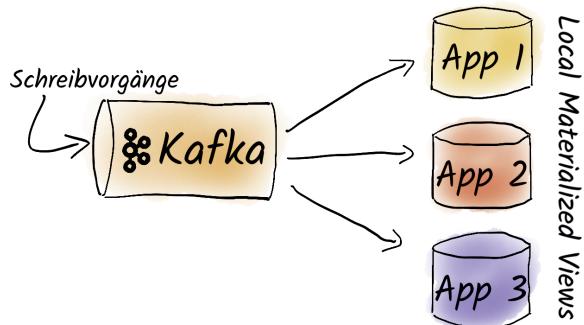


Bild 4.12 Local Materialized Views mit Kafka

Aber wir müssen nicht mal so weit gehen und andere Produkte einsetzen. In vielen Fällen kann uns Kafka Streams im Zusammenspiel mit den zustandsbehafteten Operationen den Einsatz von zusätzlichen Datenbanksystemen ersetzen. Sollten wir im Unternehmen Confluent's Kafka-Distribution einsetzen wollen, können wir sogar weitergehen und teilweise statt Kafka Streams *ksqlDB*¹⁸ einsetzen. *ksqlDB* ist eine auf Kafka Streams basierende Software, um mit einem SQL-ähnlichen Dialekt Abfragen direkt über Kafka durchzuführen. In vielen Fällen ist es deutlich einfacher, direkt mit *ksqlDB* kontinuierliche Abfragen auszuführen, als selbstständig Kafka Streams-Applikationen zu schreiben. Die Resultate der Abfragen können wir entweder zurück in ein Kafka-Topic schreiben oder per REST-API abholen.

Somit können wir mit Kafka eine Architektur kreieren, in der wir unveränderliche Daten zentral für alle Interessenten vorhalten und sich alle Services genau die Daten, die sie benötigen, selbstständig abholen und diese in genau dem Datenformat mit der Technologie in sogenannten Lokalen Materialized Views abspeichern und weiterbenutzen können. Martin Kleppmann hat das in seinen Konferenzbeiträgen und Blogartikeln¹⁹ als *Turning the Database inside out* bezeichnet. Das bedeutet, dass wir mit Kafka die Datenbank von innen nach außen umkrepeln und das zentrale, aber versteckte Element von Datenbanken, das Log, öffentlich verfügbar machen. Darauf basierend bauen wir die Features von Datenbanken wie Tabellen, Views, Materialized Views, Indizes und so weiter auf verteilte Weise auf. Dieses Architekturmuster bezeichnen wir auch gerne als zentrales Nervensystem für Daten oder auch Streaming-Plattformen.

¹⁷⁾ <https://redis.io/>

¹⁸⁾ <https://ksqldb.io/>

¹⁹⁾ <https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/>

4.2.4 Kafka als Streaming-Plattform

Wir haben nun Kafka mit einigen unterschiedlichen Technologien verglichen und einen Einblick in unsere Erfahrungen gegeben, wann Kafka seine Stärken ausspielt. Aber wir sind nicht darauf eingegangen, was Kafka aus architektureller Sicht eigentlich ist. Manche bezeichnen Kafka als Messaging-System, aber aus unserer Sicht ist das nur eine mögliche Anwendung und nicht weit genug gedacht. Für manche reicht es aus, dass es ein verteiltes Log ist. Wir glauben aber, dass sich Apache Kafka am besten als Kern einer Streaming-Plattform beschreiben lässt.

Für uns ist die Kernidee einer Streaming-Plattform, dass Daten immer in Bewegung sind. Sobald etwas in unserem Unternehmen passiert, und es passiert quasi immer irgendetwas, löst dieses Ereignis ein Event in einem produzierenden System aus, welches dann in das zentrale Log geschrieben wird. Andere Systeme können dann in Nahe-Echtzeit oder zu einem späteren Zeitpunkt dieses Event aufgreifen und darauf reagieren; statt also wie in batch-basierten Systemen bis zu einem Tag oder gar länger zu warten, bis wir gewisse Ereignisse mitbekommen und darauf basierend Entscheidungen treffen können.

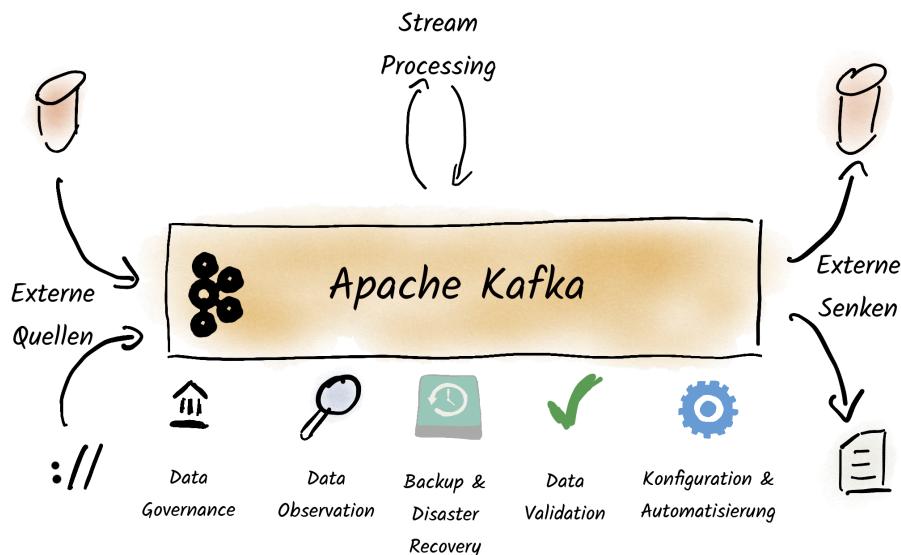


Bild 4.13 Kafka als zentrales Nervensystem für Daten im Unternehmen

Wenn wir aber sagen, dass Kafka Kern einer Streaming-Plattform ist, was benötigen wir zusätzlich? Bei allen unseren Kunden wird Kafka nicht auf der sogenannten *Grünen Wiese* aufgesetzt, sondern es existieren bereits zahlreiche andere Systeme. Mit Kafka Connect haben wir ein Werkzeug kennengelernt, um Daten aus Drittsystemen nach Kafka zu schreiben oder auch von Kafka an Drittsysteme wie Datenbanken, Dateien oder gar andere Messaging-Systeme zu schicken. Wir glauben, dass Kafka Connect integraler Bestandteil von Architekturen ist, die Kafka nicht nur zum Bewegen von Massendaten zwischen zwei Punkten benutzen.

Sobald die Daten in Kafka sind, möchten wir diese meist nicht nur an andere Ziele weiterleiten, sondern oft weitere Verarbeitungsschritte vornehmen. Dafür eignet sich ein Stream Processing-Werkzeug wie Kafka Streams oder ksqlDB hervorragend. Damit können wir unsere Daten nahezu in Echtzeit verarbeiten und auf Veränderungen reagieren.

Diese aufbereiteten Daten können dann von unseren Services benutzt werden oder gar von Services anderer Teams im gesamten Unternehmen.

Spätestens wenn mehr als ein Team mit Kafka arbeiten soll, gibt es einen ganzen Themenkomplex, der oft so lange ignoriert wird, bis es fast zu spät ist, nämlich das Thema Kafka-Management und Automatisierung. Auf der einen Seite geht es dabei natürlich darum, wie wir Kafka möglichst einfach und automatisiert aufsetzen, wenn wir es selbst betreiben möchten. Aber für den alltäglichen Gebrauch finden wir es nicht minder wichtig, das Management der Kafka-Ressourcen und der Prozesse möglichst weit zu automatisieren: Wie werden Topics angelegt, welche Namensschemata werden benutzt? Wie werden Nutzer angelegt, welche Rechte bekommen diese Nutzer? Wie werden die Zugangsdaten an unsere Services verteilt? Wie automatisieren wir das Konfigurieren von Schemas, Connectoren und ksqlDB SQL-Skripten? Das sind alles Fragen, die sich von Unternehmen zu Unternehmen stark unterscheiden und es nicht immer klare Lösungen gibt. Wir favorisieren da einen Git-basierten Ansatz, wo Teams Pull- (oder Merge-)Requests stellen können und diese auf der einen Seite automatisiert überprüft und zusätzlich, zumindest für Produktionsumgebungen, noch von Menschen durchgesehen werden. Damit haben wir gute Erfahrungen gemacht, und Unternehmen finden so eine gute Balance zwischen Wartbarkeit und Agilität.

Wenn wir das alles umgesetzt haben, befinden wir uns auf einem guten Weg, Kafka als zentrales Nervensystem für unsere Daten zu benutzen. Wobei auch hier immer gilt: Die besten *Best Practices* bringen nichts, wenn sie nicht zum Unternehmen passen und der Rest der Architektur nicht stimmig ist. Es gehört mehr als nur Kafka dazu, unsere Kommunikationssysteme weg von *Daten in Ruhe* hin zu Daten, die durch eine Organisation fließen, umzustellen.

4.2.5 Zusammenfassung

In diesem Kapitel haben wir uns angeschaut, wie Kafka im Vergleich zu ähnlichen Technologien einzuordnen ist. Hierbei haben wir uns verschiedenste Technologien von klassischen Messaging-Systemen über REST bis hin zu relationalen Datenbanken angesehen. Wir haben dabei sowohl Gemeinsamkeiten als auch Unterschiede erörtert und sind in diesem Kontext auch auf Vor- und Nachteile der unterschiedlichen Systeme eingegangen. Zum Abschluss dieses Kapitels haben wir noch Kafka als Streaming-Plattform betrachtet und die damit einhergehenden Herausforderungen ausführlich erörtert.

■ 4.3 Kafka-Referenzarchitektur

Nachdem wir nun einen guten Überblick darüber bekommen haben, was Kafka ist, wie wir es einsetzen und wie es sich architekturell in unsere bestehenden Unternehmens-IT-Landschaften einbindet, stellt sich nun die große Frage: Was brauchen wir, um Kafka erfolgreich zu betreiben?

Wir haben bereits in den letzten Abschnitten besprochen, dass Kafka allein in den meisten Fällen nicht ausreicht, um unsere Unternehmensziele zu erreichen. Ein typisches Kafka-Setup besteht meist aus deutlich mehr als nur den eigentlichen Brokern und dem Koordinationscluster, wie wir im Bild erkennen können.

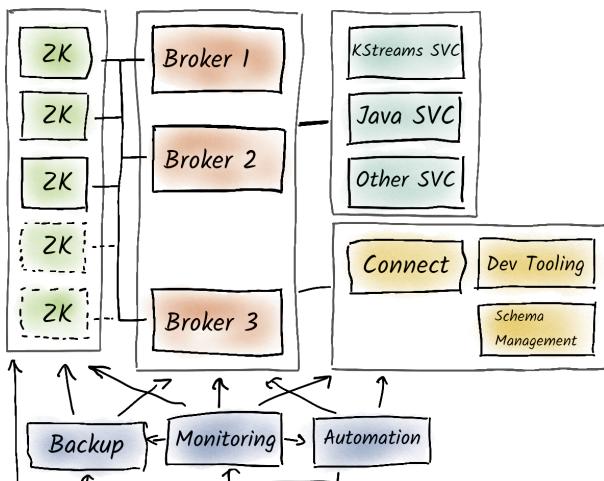


Bild 4.14 Ein typischer Aufbau einer Kafka-Installation mit Zusatzwerkzeugen

4.3.1 Deployment-Modelle und Hardware-Anforderungen

Kafka lässt sich auf sehr vielfältige Arten und Weisen betreiben. Wir haben in unserer Praxis vieles gesehen: von selbst betriebenen Kafka-Clustern über *Kafka in Kubernetes* bis hin zu vollständig gemanagten *Kafka as a Service*-Lösungen, wo wir nicht einmal wissen, wie viele Broker uns zur Verfügung stehen.

Unsere Meinung ist, dass es im Grunde egal ist, auf welcher Infrastruktur Kafka betrieben wird. Hauptsache die Infrastruktur ist gut, und es gibt ein Team, welches sich gut mit Kafka auskennt und es betreiben kann. Wenn entweder keine zuverlässige Infrastruktur zur Verfügung steht oder kein Team Kapazitäten hat, sich mit Kafka intensiv auseinanderzusetzen, zeigt unsere Erfahrung, dass es in diesen Fällen besser ist, einen gemanagten Kafka-Service einzukaufen.

4.3.1.1 Kafka auf eigener Hardware

Die wohl klassischste Möglichkeit, Software zu betreiben, ist im eigenen Rechenzentrum auf eigener Hardware. Wir empfehlen, alle Prozesse rund um Kafka, auch im eigenen Rechenzentrum, weitestgehend zu automatisieren. Das stellt uns in diesem Anwendungsfall vor Herausforderungen, ist aber machbar. Wir werden in den nächsten Abschnitten auf konkrete Hardware-Empfehlungen eingehen, aber insbesondere im eigenen Rechenzentrum benötigen wir eine gute Planung des Kafka-Einsatzes. Wie organisieren wir Ersatz bei Hardwareausfällen? Was tun wir bei zusätzlichem Ressourcenbedarf? Dabei dürfen wir nicht vergessen, dass wir außer den Produktionsclustern auch Kafka in Entwicklungs- und Testumgebungen aufsetzen sollten. Wir sollten darauf achten, dass die Broker sich nicht im gleichen Rack befinden oder am besten sogar unterschiedliche Brandabschnitte benutzen, um zusammen mit der Rack-Awareness für bessere Zuverlässigkeit zu sorgen.

4.3.1.2 Kafka in virtualisierten Umgebungen

In virtualisierten Umgebungen in eigenen Rechenzentren sind zum Großteil dieselben Hinweise zu beachten, als würden wir Kafka auf der Hardware selbst betreiben. Wichtig ist hier, zusätzlich darauf zu achten, dass die Kafka-Broker möglichst gleichmäßig über VM-Hosts verteilt werden. Außerdem sind SAN-Systeme für den Einsatz von Kafka in seltenen Fällen geeignet. Wenn unser SAN-System zuverlässig und sehr performant ist, kann dies gut funktionieren, aber wenn Sie hier Zweifel haben, sollte immer lokale SSDs präferiert werden.

4.3.1.3 Kafka in der Public Cloud selbst betreiben

Immer mehr Unternehmen betreiben Kafka in Public Cloud-Angeboten wie Amazon Web Services (AWS), Google Cloud oder Microsoft Azure. Wir haben mit allen diesen Cloud-Angeboten gute Erfahrungen gemacht, und bei sorgfältiger Planung funktionieren diese Deployments sehr gut. Obwohl wir gesagt haben, dass wir auf SAN-Systeme verzichten sollen, haben alle diese Anbieter sehr gut funktionierende Storage-Systeme. Bei AWS sollten wir nur darauf achten, sogenannte EBS-optimized Instanzen zu benutzen. Wir empfehlen auch, sich vorher gut Gedanken über die Netzwerkarchitektur zu machen. Falls Zookeeper noch eingesetzt wird, sollte Zookeeper von allen anderen Systemen bis auf Kafka abgeschottet werden.

4.3.1.4 Kafka in Kubernetes

Mit dem Aufstieg von Docker und später Kubernetes wandern aus unserer Erfahrung immer mehr Anwendungen in Kubernetes-Cluster. Auch wenn zu Beginn unserer Arbeit mit Kafka und Kubernetes dies auf sehr viel berechtigte Skepsis gestoßen ist, lässt sich Kafka inzwischen ganz gut betreiben. Wir sollten dafür aber stets aktuelle Kubernetes-Versionen benutzen und vor allem auf die Performance unserer Persistenz-Schicht achten. Nach unseren Erfahrungen mit GlusterFS oder NFS raten wir davon sehr stark ab. Da sich Kubernetes-Deployments sehr gut automatisieren lassen, empfehlen wir unseren Kunden, Kafka auf Kubernetes von Anfang an sehr stark zu automatisieren. Wir empfehlen, sogenannte Kafka Operators wie *Strimzi*²⁰ zu benutzen, um den Betrieb von Kafka so einfach wie möglich zu gestalten.

²⁰⁾ <https://strimzi.io/>

Bei Kafka auf Kubernetes müssen wir unbedingt mit sogenannten Anti-Affinity-Regeln darauf achten, dass unterschiedliche Kafka-Broker nicht auf dem gleichen Kubernetes Node liegen, sonst kann dies zu Datenverlust führen.

4.3.1.5 Kafka as a Service

Wir sehen immer mehr Unternehmen, die nicht nur Cloud-Angebote einsetzen, sondern zusätzlich dazu so viele Dienste wie möglich nicht mehr selbst betreiben, sondern den Betrieb an Drittanbieter auslagern. Sowohl AWS als auch Azure haben unterschiedliche Kafka-Dienste im Angebot. Diese sind meistens insbesondere bei der maximalen Retention-Zeit stark limitiert. Wenn diese aber zu den eigenen Anforderungen passen, kann es sich lohnen, diese Services zu evaluieren.

Um das meiste aus Kafka herauszubekommen, empfehlen wir, stattdessen auf Kafka spezialisierte Dienstleistungen wie die *Confluent Cloud*²¹ oder *Aiven*²² einzusetzen. Wir haben mit beiden Dienstleistern gute Erfahrungen gemacht und können beide je nach Anforderungen und Vertragsgestaltung empfehlen. Beide, und auch andere Anbieter, lassen Kafka in den drei Cloud-Umgebungen bei AWS, Azure und Google Cloud laufen und unterstützen auch das sogenannte VNet Peering. Es ist bei Aiven auch auf Anfrage möglich, Kafka im eigenen Cloud-Account verwalten zu lassen. Confluents Alleinstellungsmerkmal ist, dass wir uns keine Gedanken über Cluster-Größen machen müssen und wir uns stattdessen transparent gewünschte Kapazitäten buchen können.

4.3.2 Broker

Die Broker sind Kafkas Herzstück. Wir empfehlen, mit mindestens drei Brokern zu starten, um für erhöhte Zuverlässigkeit zu sorgen. Je nachdem, wie groß die Broker sind und wie viel Last erzeugt wird, können wir dann Stück für Stück weitere Broker hinzufügen. Da zum Zeitpunkt des Schreibens Kafka Version 3.0 erst in der Veröffentlichung ist, können wir keine Erfahrungen zu den Ressourcenanforderungen der Koordinationsknoten nennen. Wir gehen aber davon aus, dass sie ähnlich zu Zookeeper sein werden.

Die benötigten Ressourcen sind sehr von den Anforderungen abhängig. Wir empfehlen hier, sich nicht von den offiziellen Empfehlungen von mindestens 32 GB RAM abschrecken zu lassen. Wir haben auch deutlich kleinere Cluster im erfolgreichen Einsatz gesehen.

Der Fokus sollte bei den Brokern auf den eingesetzten Festplatten und dem Hauptspeicher liegen. Der verwendete Massenspeicher sollte zuverlässig und schnell sein. Je nach Anforderung ist es empfehlenswert, eher mehr kleine SSDs zu verwenden als wenige sehr große. Zum Beispiel ist die Empfehlung für große Kafka-Cluster, mit zwölf 1-TB-SSDs zu arbeiten. RAID ist abseits von RAID 0 nicht sinnvoll, da die Replikation in Kafka für die Zuverlässigkeit zuständig ist. Insbesondere raten wir vom Einsatz von RAID 5 und 6 ab, da diese RAID-Systeme meistens zu langsam sind.

Für kleine Anwendungsfälle ist es möglich, mit einigen JVM-Optimierungen den RAM-Verbrauch sehr stark zu reduzieren und mit 1 GB RAM pro Broker gut klarzukommen. Im Ex-

²¹⁾ <https://www.confluent.de/confluent-cloud/>

²²⁾ <https://aiven.io/kafka>

tremfall lässt sich selbst das reduzieren. Aber auch in der maximalen Ausbaustufe benötigt der Kafka-Heap nicht mehr als 6 GB RAM und wächst auch über längere Zeit nicht stark an. Alles, was nicht vom Heap benötigt wird, kann das Betriebssystem als Page Cache benutzen. Üblich sind Broker mit 32 oder 64 GB RAM.

Ein weiterer wichtiger Punkt ist ein schnelles und zuverlässiges Netzwerk. Wir empfehlen den Einsatz von mindestens 1 GBit/s Ethernet. Nach oben gibt es aber keine Grenzen. Wenn Netzwerkspeicher benutzt werden soll, dann sollte ein eigenständiges Storage-Netzwerk benutzt werden.

Die CPU ist weniger wichtig, sondern hängt bei den meisten Anbietern vom eingesetzten RAM ab. Es ist im Allgemeinen empfohlen, eher mehr kleine CPU-Kerne als wenige große einzusetzen, da sich in Kafka viele Operationen parallelisieren lassen.

Wie mehrmals schon beschrieben, ist die physische Trennung der Kafka-Broker essenziell. Kafka-Broker sollten nicht auf dem gleichen VM-Host oder im gleichen Rack stehen, um im Falle eines Stromausfalls des Hosts oder Racks nicht Daten bei mehreren Brokern zu verlieren. Mit Kafkas Rack-Awareness-Funktionalität ist es empfehlenswert, die Broker, wenn möglich, auf unterschiedliche Availability-Zonen zu verteilen.

Es ist nicht sinnvoll, einzelne Kafka-Cluster über mehrere Cloud-Regionen zu verteilen. Die Latenz zwischen den Brokern sollte stets unter 30 ms betragen.

4.3.3 Koordinationscluster

Wie bereits beschrieben, können wir für Kafka 3.0 noch keine Empfehlungen für die Koordinationscluster geben. Wir gehen aber davon aus, dass die Ressourcenanforderungen ähnlich oder sogar geringer als für Zookeeper sind.

Zookeeper selbst ist sehr latenzempfindlich, und wir sollten hier das beste Netzwerk benutzen, das wir haben. Die Rechner selbst müssen nicht groß sein, und sogar für große Kafka-Cluster braucht Zookeeper nicht mehr als vier bis acht GB RAM pro Knoten.

Üblicherweise haben wir drei Zookeeper-Knoten für kleinere Kafka-Cluster im Einsatz und für große fünf.

Mit Kafka 3.0 gehen wir davon aus, dass Kafka-Cluster, bestehend aus drei bis fünf Brokern, keinen eigenständigen Koordinationscluster haben werden und erst bei größeren Clustern der Koordinationscluster ausgelagert wird. Auch hier gilt, drei oder fünf Controller-Knoten werden benötigt.

4.3.4 Monitoring und Logging

Für den erfolgreichen Betrieb von IT-Systemen sind Monitoring und Logging essenziell. Kafka setzt Javas JMX (Java Management Extensions) ein, um unterschiedlichste Parameter an Monitoring-Werkzeuge weiterzuleiten. Wir möchten an dieser Stelle gar nicht zu tief in dieses Thema eintauchen, dafür gibt es online oder in Schulungen bessere und aktuellere Ressourcen. Die meisten uns bekannten Werkzeuge unterstützen JMX mehr oder weniger

von Haus aus. Für das Monitoring mit Prometheus benutzen wir zum Beispiel den Prometheus JMX Exporter²³ mit der empfohlenen Konfiguration für Kafka.

Eine der allerwichtigsten Metriken ist die sogenannte *underreplicated partitions*-Metrik. Diese besagt, wie viele Partitionen es gibt, die nicht vollständig repliziert werden. Dieser Wert sollte immer 0 sein. Ist er konstant größer, deutet es darauf hin, dass ein Broker offline ist. Fluktuiert dieser Wert ständig, dann deutet dies auf Ressourcenprobleme im Cluster hin. Oft ist nur die Last ungleich verteilt, und wir sollten dies schnellstmöglich beheben. Weiterhin sollten wir folgende JMX-Metriken monitoren: Wie viele Partitionen gibt es pro Broker, wie viele Leader etc. Wichtig ist hier, dass diese Zahlen über alle Broker hinweg halbwegs gleichverteilt sind. Das Gleiche gilt auch für die Netzwerk-Auslastung. Unser Ziel sollte immer eine Gleichverteilung der Last sein.

Dafür ist es auch wichtig, die Betriebssystem-Metriken zu beobachten. Die Festplattenauslastung sollte nicht zu groß sein. Empfohlen werden sogar Alerts bei einer Auslastung von über 60%, damit genug Zeit bleibt, neue Festplatten zu bestellen. Bei einem Rebalancing von Partitionen benötigen wir unter Umständen zusätzlichen Speicherplatz, und deshalb ist es wichtig, genug Puffer zu haben. Die Netzwerk-Auslastung muss auch unter 60% liegen, da wir sonst beim Hinzufügen neuer Ressourcen Probleme haben werden, die Partitionen auf die neuen Broker zu bewegen. Außerdem würden Ausfälle einzelner Broker die Netzwerkkapazitäten an die Grenze bringen. Den Hauptspeicher sollten wir auch im Auge behalten und schauen, dass wir noch Kapazität für unerwartete Last haben. Kafka benutzt sehr aktiv den Page Cache und kommt auch gut mit weniger RAM zurecht, aber die Performance leidet dann darunter.

Wir empfehlen unseren Kunden immer, zusätzliches Monitoring zu installieren. Eine der wichtigsten Metriken für Benutzer ist der sogenannte Lag. Der Lag zeigt an, wie viel ein Consumer hinter dem aktuellen Stand ist. Den Lag können wir einerseits in Anzahl von Nachrichten angeben oder messen, wie viel Zeit zwischen der zuletzt vom Consumer gelesenen Nachricht und der neusten Nachricht in der Partition liegt. Kafka selbst bietet diese Metrik nicht an, aber es gibt zahlreiche Werkzeuge, die sich hier gut integrieren lassen. Wir benutzen gerne *Kafka Lag Exporter*²⁴, um diese Metrik von Prometheus beobachten zu lassen. Zusätzlich empfehlen wir externe Health-Checks. Diese schreiben Nachrichten auf ein Kafka-Topic, lesen wieder von dem Topic und messen, wie lange dies dauert. Damit können wir die Ende-zu-Ende-Latenz im Auge behalten. Leider haben wir hier keine gute Empfehlung, da der *LinkedIn Kafka Monitor*²⁵ sehr komplex zu konfigurieren ist und Alternativen zum Zeitpunkt des Schreibens rar oder uns nicht bekannt sind.

4.3.5 Zusätzliche Werkzeuge

Wir sind der Meinung, dass Automatisierung von Prozessen Kernbestandteil moderner IT-Architekturen ist. Deswegen empfehlen wir unseren Kunden stets, so viel wie sinnvoll möglich ist zu automatisieren. In Kubernetes-Umgebungen ist dies selbst für die Cluster-Ver-

²³⁾ https://github.com/prometheus/jmx_exporter

²⁴⁾ <https://github.com/lightbend/kafka-lag-exporter/>

²⁵⁾ <https://github.com/linkedin/kafka-monitor>

waltung mit wenig Aufwand möglich. Aber auch Cloud-Umgebungen lassen sich meistens sehr gut automatisieren.

Wichtiger als die Cluster-Verwaltung zu automatisieren ist es, das gesamte Management rund um Topics, Nutzer und ACLs vollständig zu automatisieren. Unsere Kunden nutzen dafür meist selbst geschriebene Lösungen oder, wenn möglich, vom Kafka-Anbieter offerte Lösungen, meistens basierend auf Terraform.

Wir sind große Verfechter des sogenannten Hands-off-Betriebs. Wir sollten nur im absoluten Notfall manuell an Konfigurationen und an Live-Systemen arbeiten. Grafische Oberflächen wie zum Beispiel die kommerzielle Lösung *Conduktor*²⁶ *KaDeck*²⁷ oder das *Confluent Control Center*²⁸ sind gute Anwendungen, um Einsicht in den laufenden Betrieb zu bekommen, aber wir raten sehr stark davon ab, mit solchen Werkzeugen Änderungen an Produktionssystemen vorzunehmen. Sie sind aber vor allem in Entwicklungsumgebungen Werkzeuge, die nach einiger Einarbeitung niemand missen möchte. Alternativ gibt es Werkzeuge wie *Kafka Topics UI*²⁹ von Lenses.io, um sich den Inhalt von Topics anzeigen zu lassen.

Mit diesen und weiteren Werkzeugen sind wir gut aufgestellt, unsere Arbeit, und auch die unserer Teams, so einfach wie möglich zu gestalten und damit Kafka zur Erfolgsgeschichte zu machen.

4.3.6 Zusammenfassung

In diesem Kapitel drehte sich alles um den Betrieb eines Kafka-Clusters. Wir haben uns verschiedene Möglichkeiten angesehen, ein Kafka-Cluster zu deployen, und haben hierbei verschiedene Möglichkeiten vom Hosten auf eigener Hardware bis hin zu komplett eingekauften Lösungen besprochen. In diesem Zusammenhang sind wir noch mal gesondert auf die Hardwareanforderungen von Brokern, aber auch vom Koordinationscluster eingegangen. Außerdem haben wir uns mit Monitoring und Logging in Kafka beschäftigt und sind zum Abschluss noch auf einige Tools eingegangen, welche für die Automatisierung eines Kafka-Clusters hilfreich sind.

²⁶⁾ <https://www.conduktor.io/>

²⁷⁾ <https://www.xeotek.com/>

²⁸⁾ <https://docs.confluent.io/platform/current/control-center/index.html>

²⁹⁾ <https://github.com/lensesio/kafka-topics-ui>

5

Anhang: Kafka-Testumgebung aufsetzen



Bis zum Redaktionsschluss wurde Kafka 3.0 noch nicht veröffentlicht.
Sobald dies geschehen ist, veröffentlichen wir auf der Website <https://streamcommit.de/buch/updates/> eine Anleitung, wie Sie die Testumgebung ohne Zookeeper mit Kafka 3.0 aufsetzen können.

Für unsere praktischen Beispiele setzen wir eine beispielhafte Testumgebung auf. Wir erstellen ein Zookeeper-Ensemble mit drei Servern und starten drei Kafka-Broker. Normalerweise würden wir dies auf getrennten Servern aufsetzen, aber Einfachheitshalber starten wir alles auf einem Computer und nutzen dafür verschiedene Ports:

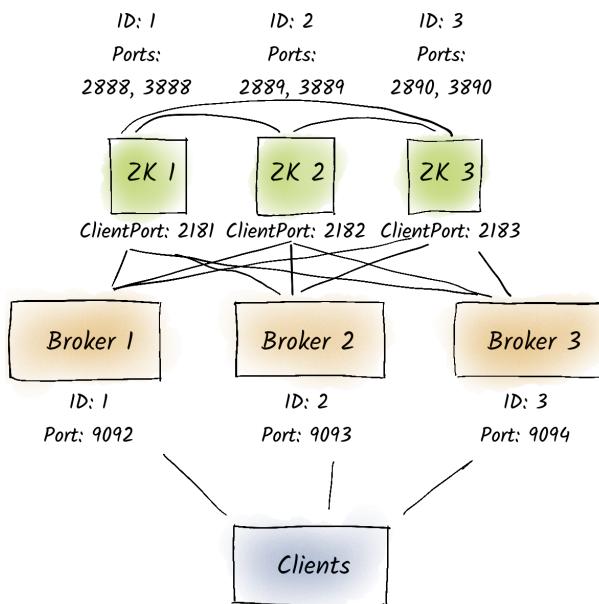


Bild 5.1 Unser Testaufbau

■ 5.1 Betriebssysteme

Apache Kafka läuft in der Java Virtual Machine und ist somit auf allen populären Betriebssystemen lauffähig. Wir haben unser Setup unter folgenden Betriebssystemen getestet: *macOS Big Sur 11.4 mit Intel Prozessor, Ubuntu 20.04 LTS*. Windows-Nutzern empfehlen wir das *Windows-Subsystem für Linux*.

■ 5.2 Kafka herunterladen

Die aktuelle Kafka-Version finden Sie unter <https://kafka.apache.org/downloads>. Zum Zeitpunkt der Manuskripterstellung war die Version 2.8.0 aktuell.

Wir empfehlen den binären Download in der aktuellen Scala-Version, welche zum Zeitpunkt der Manuskripterstellung `kafka_2.13-2.8.0.tgz` ist. Wir laden die Datei herunter und entpacken sie an einem von uns gewählten Ort (z.B. `~/kafka`):

```
$ wget "https://downloads.apache.org/kafka/2.8.0/kafka_2.13-2.8.0.tgz"
$ tar xfz kafka_2.13-2.8.0.tgz
$ rm kafka_2.13-2.8.0.tgz
$ cp -a kafka_2.13-2.8.0/. ~/kafka
$ rm -r kafka_2.13-2.8.0
```

Im Ordner `~/kafka` finden wir folgende Ordner und Dateien:

- `LICENSE, NOTICE`: Dateien mit Lizenzinformationen
- `bin`: Kommandozeilenwerkzeuge zur Interaktion mit Kafka und Zookeeper
- `config`: Konfigurationsdateien
- `libs`: Java-Bibliotheksdateien (`*.jar`)
- `site-docs`: Dokumentation

Damit wir beim Ausführen der Kommandozeilenwerkzeuge nicht immer den gesamten Pfad angeben müssen, fügen wir den `bin`-Ordner zur PATH-Variablen hinzu:

```
$ export PATH=~/kafka/bin:"$PATH"
```

Natürlich können wir diese Zeile auch in unsere `~/.bashrc`, `~/.zshrc` o.Ä. hinzufügen, je nachdem, welche Shell wir benutzen.

■ 5.3 Zookeeper starten

Bevor wir Kafka starten können, müssen wir unser Zookeeper-Ensemble, bestehend aus drei Zookeeper-Servern, konfigurieren und starten.

Wir beginnen damit, für jeden Zookeeper-Knoten einen Ordner zu erstellen, um darin die Zookeeper-Daten zu speichern:

```
$ mkdir -p ~/kafka/data/zk1
$ mkdir -p ~/kafka/data/zk2
$ mkdir -p ~/kafka/data/zk3
```

In jedem dieser Ordner erstellen wir eine Datei `myid`, in die wir die ID des Zookeeper-Knotens schreiben, dem dieser Ordner gehört. Wir gehen hier sehr einfach vor: zk1 bekommt die ID 1 und so weiter:

```
$ echo 1 > ~/kafka/data/zk1/myid
$ echo 2 > ~/kafka/data/zk2/myid
$ echo 3 > ~/kafka/data/zk3/myid
```

Nun erstellen wir den Ordner `~/kafka/config` und darin drei Konfigurationsdateien; `zk1.properties` bis `zk3.properties`. Alle drei Dateien sind bis auf den `clientPort` und das `dataDir` identisch. Um im Buch ein paar Seiten zu sparen, zeigen wir hier beispielhaft die Datei `zk1.properties`. Die Dateien für die anderen beiden Zookeeper müssen wir entsprechend anpassen:

```
# Bei zk1.properties
clientPort=2181
dataDir=[PFAD-ZUM-NUTZER-VERZEICHNIS]/kafka/data/zk1
# Bei zk2.properties
#clientPort=2182
#dataDir=[PFAD-ZU-kafka]/data/zk2
# Bei zk3.properties
#clientPort=2183
#dataDir=[PFAD-ZU-kafka]/data/zk3
# Rest bleibt bei allen gleich.
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890

# Nicht in Produktion benutzen!
client.secure=false
tickTime=2000
initLimit=10
syncLimit=5
```

Nun können wir Zookeeper starten. Standardmäßig startet Zookeeper im Vordergrund. Das heißt, wir benötigen pro Zookeeper ein eigenes Terminal (Wir sollten nicht vergessen immer die PATH-Variable anzupassen). Bis alle Zookeeper-Server gestartet sind, werden wir viele Fehlermeldungen sehen. Danach sollten keine Fehler mehr auftreten.

```
$ zookeeper-server-start.sh ~/kafka/config/zk1.properties
$ zookeeper-server-start.sh ~/kafka/config/zk2.properties
$ zookeeper-server-start.sh ~/kafka/config/zk3.properties
```

Um zu testen, ob Zookeeper erfolgreich gestartet ist, können wir die ZookeeperShell benutzen:

```
$ zookeeper-shell localhost:2181 ls /
Connecting to localhost:2181

WATCHER:::

WatchedEvent state:SyncConnected type:None path:null
[zookeeper]
```

Wenn dies funktioniert, können wir alle Zookeeper stoppen. Um Terminals zu sparen, starten wir nun Zookeeper mit dem zusätzlichen Flag `-daemon` im Hintergrund. Danach sollten wir noch testen, ob der `zookeeper-shell.sh`-Befehl funktioniert:

```
$ zookeeper-server-start.sh -daemon ~/kafka/config/zk1.properties
$ zookeeper-server-start.sh -daemon ~/kafka/config/zk2.properties
$ zookeeper-server-start.sh -daemon ~/kafka/config/zk3.properties
```

■ 5.4 Kafka starten

Nachdem Zookeeper erfolgreich läuft, können wir nun endlich Kafka konfigurieren und es danach starten. Ähnlich wie bei Zookeeper legen wir für jeden Kafka-Broker einen eigenen Ordner an:

```
$ mkdir -p ~/kafka/data/kafka1
$ mkdir -p ~/kafka/data/kafka2
$ mkdir -p ~/kafka/data/kafka3
```

Nun erstellen wir für jeden Broker eine Konfigurationsdatei im Ordner `~/kafka/config`: `kafka1.properties` für Broker 1 und `kafka2.properties` und `kafka3.properties` für die beiden anderen Broker. Wieder zeigen wir hier nur die Konfigurationsdatei für den Broker 1. Für die beiden anderen Broker muss die Datei entsprechend angepasst werden:

```
# Bei kafka1.properties
broker.id=1
log.dirs=[PFAD-ZUM-NUTZER-VERZEICHNIS]/kafka/data/kafka1
listeners=PLAINTEXT://:9092

# Bei kafka2.properties
#broker.id=2
#log.dirs=[PFAD-ZUM-NUTZER-VERZEICHNIS]/kafka/data/kafka2
#listeners=PLAINTEXT://:9093

# Bei kafka3.properties
#broker.id=3
#log.dirs=[PFAD-ZUM-NUTZER-VERZEICHNIS]/kafka/data/kafka3
#listeners=PLAINTEXT://:9094

# Rest bleibt bei allen gleich.
zookeeper.connect=localhost:2181
```

Wir starten testweise einen Broker, um zu überprüfen, ob die Konfigurationsdatei korrekt ist und das Zusammenspiel mit Zookeeper einwandfrei funktioniert:

```
$ kafka-server-start.sh ~/kafka/config/kafka1.properties
```

Nach sehr vielen Log-Meldungen sollte nach einigen Sekunden Ruhe einkehren und wir sollten folgende Zeile sehen:

```
[...] INFO [broker-1-to-controller-send-thread]: Recorded new controller,
from now on will use broker localhost:9092 (id: 1 rack: null)
(kafka.server.BrokerToControllerRequestThread)
```

Wenn wir diese Zeile nicht sehen, ist entweder der Broker fehlerhaft konfiguriert, oder Zookeeper läuft nicht. Um zu überprüfen, dass alles wirklich läuft, können wir in einem weiteren Fenster folgenden Befehl ausführen:

```
$ kafka-broker-api-versions.sh \
--bootstrap-server localhost:9092
localhost:9092 (id: 1 rack: null) -> (
# Viele Text ...
)
```

Wir stoppen die Broker wieder mit **STRG+C**. Dies dauert einige Sekunden, damit der Broker Zeit hat, sich sauber abzumelden. Nun starten wir Kafka, ähnlich wie Zookeeper, im Hintergrund mithilfe des Daemon-Modes:

```
$ kafka-server-start.sh -daemon \
~/kafka/config/kafka1.properties
$ kafka-server-start.sh -daemon \
~/kafka/config/kafka2.properties
$ kafka-server-start.sh -daemon \
~/kafka/config/kafka3.properties
```

Obwohl die Befehle sich sofort beenden, benötigt Kafka dennoch einige Sekunden, um zu starten. Wenn wir möchten, können wir das Kommandozeilenwerkzeug `kafka-broker-api-versions.sh` nutzen, um sicherzugehen, dass alle Broker wirklich funktionieren:

```
$ kafka-broker-api-versions.sh \
--bootstrap-server localhost:9092
localhost:9092 (id: 1 rack: null) -> (
# Viele Informationen zu kafka1
)
localhost:9093 (id: 2 rack: null) -> (
# Viele Informationen zu kafka2
)
localhost:9094 (id: 3 rack: null) -> (
# Viele Informationen zu kafka3
)
```

Nun läuft unsere Kafka-Testumgebung. Nach einem Neustart des Rechners starten weder Zookeeper noch Broker von selbst. Wir nutzen die Befehle `zookeeper-server-start.sh` und `kafka-server-start.sh` um die Server zu starten. Wichtig ist dabei, erst alle Zookeeper zu starten und erst dann die Kafka-Broker!

Unser hier beschriebenes Setup ist sehr nützlich, wenn wir Dinge auf unserem eigenen Rechner testen möchten. Wir sollten keinesfalls diese Konfigurationsdateien für mehr als lokale Tests benutzen; schon gar nicht in Produktionsumgebungen! Dafür ist dieses Setup zu unsicher, und es läuft auf einem einzigen Rechner, was jegliche Annahmen, die Kafka trifft, ad absurdum führt. Für das Testen und Ausprobieren ist es aber vollkommen ausreichend.

■ 5.5 Einzelne Broker stoppen

Das mit Kafka ausgelieferte Skript `kafka-server-stop.sh` stoppt leider alle Broker auf unserem Rechner statt nur einen gewünschten Broker. Um nur einen Broker herunterzufahren, benutzen wir folgendes Skript. Am besten speichern wir es in der Datei `~/kafka/bin/kafka-broker-stop.sh`:

```
#!/bin/bash
BROKER_ID="$1"

if [ -z "$BROKER_ID" ]; then
    echo "usage ./kafka-broker-stop.sh [BROKER-ID]"
    exit 1
fi

PIDS=$(ps ax | grep -i 'kafka\|.Kafka' | grep java | grep "kafka${BROKER_ID}.properties" | grep -v grep | awk '{print $1}')

if [ -z "$PIDS" ]; then
    echo "No kafka server to stop"
    exit 1
else
    kill -s TERM $PIDS
fi
```

Wir müssen das Skript nur noch ausführbar machen und können es danach benutzen:

```
$ chmod +x ~/kafka/bin/kafka-broker-stop.sh
```

■ 5.6 Umgebung aufräumen

Nach unseren Tests ist es sinnvoll, die Testumgebung wieder herunterzufahren. Auch hier ist es wichtig, auf die Reihenfolge zu achten: Wir stoppen erst alle Kafka-Broker und Zookeeper:

```
$ kafka-broker-stop.sh 1
$ kafka-broker-stop.sh 2
$ kafka-broker-stop.sh 3
$ zookeeper-server-stop.sh ~/kafka/config/zk1.properties
$ zookeeper-server-stop.sh ~/kafka/config/zk2.properties
$ zookeeper-server-stop.sh ~/kafka/config/zk3.properties
```

Optional können wir danach den `~/kafka`-Ordner löschen, um alle Spuren unserer Installation zu beseitigen.

Register

Symbole

2-Phase-Commit-Protokoll 124

Buffer 79

Byte-Array 19, 77

A

Acknowledgements (ACKs) 42, 63, 78
- Strategien 43
ACLs 145
Aiven 161
Annahmen 26
Anwendungsfälle 127
Architektur 12, 25
At least once 48, 120
At most once 48, 120
Atomares Schreiben 122
Ausfallsicherheit 12, 134
Authentifizierung 144
Autorisierung 144
Avro 19

C

Chaos Monkey 27
Checkpoints 82, 108
cleanup.policy 106
Clients 38, 75
Cloud, Public 160
Cluster 6
Cluster-Management 115
Commit-Log 22
Compaction 21
Compaction Lag 112
Conduktor 164
Confluent Cloud 161
Confluent Control Center 164
Confluent Replicator 152
Consumer 6, 13, 30, 38, 93
- Daten parallel lesen 8
- Konfiguration 68
- Performance messen 69
Consumer Groups 31, 59, 99
- Group-ID 99
- Heartbeat-Signale 101
- Static Memberships 103
Controller 52, 116, 118
CooperativeStickyAssignor 104

B

Backups 33, 49, 147
Batching 63, 78
Batch-Systeme 2
Betriebssysteme 166
Bootstrap Server 74
Broker 13, 38, 79, 161
- Ausfall 51
- Konfiguration 67
- Optimierung 81
- Performance 66
- starten 168
- stoppen 170

D

Dateistrukturen 82
Dateisystem 80

- Daten
 - historische 8
 - Datenbank 155
 - Datenformat 19
 - Datenhaltbarkeit 41
 - Datenmengen, große 27
 - Datenstrukturen 23
 - Deltas 18
 - Desaster-Management 145
 - MirrorMaker 149
 - Stretched Cluster 147
 - Deserializer 78
 - Design-Ziele 27
 - Doppelte Nachrichten 121
 - Download, Kafka 166
- E**
- Echtzeit 3
 - Einsatzgebiete 2
 - Ende-zu-Ende-Verschlüsselung 144
 - Enterprise-Service-Bus 127, 153
 - Epochen 85
 - Ereignisse 18
 - Erreichbarkeit 41
 - Events 18
 - Exactly once 48, 97, 120
- F**
- Fehler, Umgang mit 27
 - Fetch-Request 90, 94f.
 - Follower 35, 51
 - from-beginning 14
 - fsync 81
- G**
- Grafische Oberflächen (GUI) 164
 - Group Coordinator 100, 115
 - Group Leader 101
- H**
- High Watermark (HWM) 90
- I**
- Idempotenz 48, 66, 79, 120
 - Index-Dateien 86
 - In-Sync Replicas (ISR) 13, 35, 44, 53, 89
 - min.insync.replicas 47, 92
 - IO-Threads 80
- J**
- JSON 19
- K**
- KaDeck 164
 - Kafka 3.0 13, 36, 44, 60, 119, 120, 145, 162, 165
 - Kafka as a Service 161
 - Aiven 161
 - Confluent Cloud 161
 - Kafka Backup 130
 - Kafka-Cluster 36
 - Kafka Connect 38f., 129f., 157
 - Connect Hub 129
 - Distributed Mode 132
 - REST API 133
 - Stand-alone Mode 132
 - Kafka Lag Exporter 163
 - Kafka Monitor 163
 - Kafka Streams 38, 40, 129, 135f.
 - KStream 138
 - KTables 138
 - Operationen 136, 138
 - Topology Visualizer 139
 - Kafka Topics UI 164
 - Keys 19f., 29, 58
 - Komponenten 13, 36, 39
 - Komprimierung 63
 - Konsistenz 42
 - Koordinationscluster 13, 36f., 162
 - KRaft 118
 - ksqlDB 129, 156
 - Kubernetes 160
- L**
- Last Committed Offset (LCO) 91
 - Lastenverteilung 13, 16, 31, 59

Leader 13, 35
 - preferred 54
 - rebalancing 53
 Leader-Election 54, 84
 Leader-Follower-Prinzip 34, 50
 lenses.io 129
 Leseoperationen 24
 librdkafka 76
 linger.ms 66
 Log 4, 22
 - verteilt 26
 Log Cleaner 105, 111, 113
 Log Compaction 105, 109
 - Tombstone 114
 Log-Dateien 86
 Log-End-Offset 98
 Log Retention 105f.
 Logging 162

M

Materialized Views 155
 Messaging-System 127, 153
 Messwerte 17
 Metadata-Request 74, 93
 Metriken 163
 Microservices 3, 19, 128
 MirrorMaker 40, 130, 149
 - Aktiv-Aktiv-Paarung 150
 - Aktiv-Passiv-Paarung 149
 - Hub and Spoke 151
 Monitoring 162
 Monolithen 3

N

Nachrichten 17
 - Größe 17
 - mit Keys 19
 - Position 24
 - Reihenfolge 20
 - Typen 17
 Nachvollziehbarkeit 23
 Network Threads 80
 Nutzlast 12

O

Offsets 14, 24, 96
 - retention 109
 - Verwaltung 96
 Ökosystem 39
 Orchestrierung 37

P

Parallelisierung 28, 59
 Partitioned Stream 131
 Partitionen 6, 12, 58, 60, 83
 - Anzahl berechnen 60
 - Anzahl verändern 16, 61
 - Assignments 101
 - Dateistrukturen 83
 - Verteilung 15
 Partitioner 76
 Partitionierung 27
 Paxos-Protokoll 116
 Performance 56
 - Konfiguration 57
 Persistenz 2
 Produce-Request 80
 Producer 6, 13, 38
 - Beispielcode 76
 - mehrere 9
 - Performance messen 64
 - Performance optimieren 62
 Producer-ID 121
 Protobuf 19
 Purgatory 81, 95

R

Rack-Awareness 162
 Rack-ID 95
 Rakete 4
 RangeAssignor 102
 Rebalance-Protokoll 100
 Referenzarchitektur 159
 Reihenfolge 20, 23
 Remote Partitions 150
 Remote Topics 150
 Replicas 12, 34
 Replication Factor 12, 33, 47, 50
 Replikation 33, 49, 89

Request-Queue 80

REST 155

REST-Proxy 129

Richtung, Schreiben und Lesen 23

RoundRobinAssignor 102

S

SAN-Systeme 160

SASL 144

Schema 19, 139

- Avro 141

- Formate 141

- Kompatibilität 140

- Serialisierung 141

Schema-Management 139

Schema Registry 40, 129, 139, 142

Schreiboperationen 24

Segmente 87

Sequence-ID 121

Serializer 78

Sharding 28

Sicherheit 143

- Zookeeper 145

Single Source of Truth 127

Skalierbarkeit 134

Skalierung 26

StickyAssignor 103

Streaming-Plattform 127, 157

Stretched Cluster 147

T

Timeouts 78

TLS 143

Tombstone 114

Topic 5, 12, 58, 82

- ändern 16

- auflisten 13

- beschreiben 12

- __consumer_offsets 14, 97

- erstellen 5

- Konfiguration 12

- leeren 109

- löschen 14, 88

- mehrere Partitionen 28

Transaktionen 121

Transportverschlüsselung 143

U

Unveränderbarkeit 23

V

Value 19

Verarbeitungsgarantien 119

Verschlüsselung 143

Verteiltes System 26

VNet Peering 161

W

Windows 166

X

XML 19

Z

Zeitreisen 24

Zentrales Nervensystem für Daten 2

Zookeeper 13, 36, 116

- Sicherheit 145

- starten 166

Zustände 17

Zustellungsgarantien 47

Zuverlässigkeit 33, 41

APACHE KAFKA //

- Aktuelles Kafka-Wissen für Software-Architekten, Entwickler und Administratoren
- Grundlegende Konzepte und fortgeschrittene Techniken für den Produktiveinsatz
- Integrierbare Best-Practice-Lösungen für Ihr Unternehmen
- Unterhaltsame, klare und informative Darstellung
- **Online:** Updates zum Buch und Code-Beispiele

Die Zukunft Ihres Unternehmens ist nur dann sicher, wenn Sie richtig mit Daten umgehen. Neben dem Sammeln, Speichern und Auswerten ist der Austausch von Daten zwischen unterschiedlichen Systemen hierbei von immenser Bedeutung. Denn er soll zuverlässig und möglichst in Echtzeit erfolgen.

Diese Anforderungen erfüllt Apache Kafka und hat sich deshalb als Standard etabliert. Immer mehr Unternehmen setzen Apache Kafka heute als Streaming-Plattform und Messaging-System ein, um die Komplexität moderner IT-Architekturen beherrschbar zu machen.

Für den Erfolg dieser Mission ist Wissen entscheidend. Und weil geteiltes Wissen höheren Mehrwert hat, nimmt dieses Buch Software-Architekten, Entwickler und Administratoren gleichermaßen auf die spannende Apache-Kafka-Mission mit – eine Mission, die von den Grundlagen bis zum Produktiveinsatz Kafka-basierter Daten-Pipelines reicht. Erleben Sie, wie Apache Kafka Zuverlässigkeit und Performance erreicht und wie Sie Komplikationen frühzeitig meistern. Anhand zahlreicher Praxisbeispiele lernen Sie überdies, wie Sie Best-Practice-Lösungen in Ihrem eigenen Unternehmen umsetzen.



Anatoly **ZELENIN** und Alexander **KROPP** begeistern sich seit ihrer Kindheit für IT. Heute sind beide Apache Kafka-Experten und haben ein Faible für moderne IT-Architekturen. Anatoly Zelenin ist darüber hinaus ein versierter Mentor in IT-Schulungen und Trainer für lebendige Apache Kafka-Sessions. Alexander Kropp forscht als Informatiker, unterstützt Unternehmen bei der Digitalisierung und ist Experte für Cloud-Services.

AUS DEM INHALT //

- Warum Kafka? – Motivation und Nutzungsarten
- Was ist Kafka? – Kafka als verteilter Log
- Wie erreicht Kafka seine Performance? – Partitionen und Consumer Groups
- Wie erreicht Kafka seine Zuverlässigkeit? – Replikation, ACKs und Exactly Once
- Nachrichten produzieren und konsumieren: Was passiert in den Clients und auf den Brokern?
- Nachrichten aufräumen: Wie können nicht mehr benötigte Daten gelöscht und Speicherplatz freigeräumt werden?
- Einblicke in das Kafka Ökosystem: Kafka über Kafka Connect mit Datenbanken und externen Systemen verbinden

HANSER

