Vorsicht, Kurven!

# Architekturen für die Cloud

Eberhard Wolff


*Cloud-Technologien sind gerade dabei, die Informationstechnik grundlegen zu ändern. Für Entwickler und Architekten stellt sich die Frage, was bei der Architektur und dem Design von Anwendungen für die Cloud zu beachten ist. Genau dem wird sich dieser Artikel widmen.*

Aus der Perspektive eines Entwicklers ist vor allem IaaS (Infrastructure as a Service) interessant. Dabei handelt es sich um Cloud-Angebote, die virtuelle Computer-Infrastruktur zur Miete anbieten. Beispiele sind Amazons Elastic Compute Cloud EC2 [EC2] für Computer oder Simple Storage Service [S3] für Speicher.

Und natürlich ist auch PaaS (Platform as a Service) relevant. Während IaaS einen nackten Computer liefert, bietet PaaS eine Plattform, auf der Entwickler nur noch Anwendungen installieren müssen und sie dann laufen lassen können. Beispiele sind Amazons Elastic Beanstalk [EBeanStalk] oder VMwares Cloud Foundry [CloudFoundry].

Der grundlegende Unterschied zwischen klassischen IT-Umgebungen und Cloud lässt sich an einigen Punkten festmachen:

- Klassische IT-Systeme nutzen einige wenige, sehr leistungsfähige Maschinen, um hohe Last zu bewältigen. Cloud-Lösungen bevorzugen viele kleine Rechner. Allerdings bietet die EC2-Cloud durchaus auch sehr schnelle Rechner mit bis zu 68 GB RAM oder auch Grafikprozessoren (GPUs) an.
- Auch der Ansatz für Hochverfügbarkeit ist anders: Klassisch ist die Hardware hoch verfügbar und so wird auch die Verfügbarkeit der Software sichergestellt. Cloud-Systeme hingegen bieten keine hohe Zuverlässigkeit einzelner Rechner. Laut Amazon Service Level Agreement (SLA) ist der Ausfall eines ganzen Rechenzentrums kein Ausfall des Service im Sinne dieser SLAs. Das bedeutet, dass die Architektur der Anwendung so angelegt sein muss, dass der Ausfall eines Rechners oder Rechenzentrums nicht den Ausfall der Anwendung zur Folge hat.

Wenn man nur diese beiden Punkte betrachtet, scheint die Cloud im Vergleich zu klassischen Umgebungen schlechtere Eigenschaften zu haben. In Wirklichkeit bietet die Cloud aber auch wesentliche Vorteile:

- Die Cloud kann kurzfristig wesentlich mehr Rechenleistung für eine Anwendung bereitstellen. Die Kapazitäten können auch wieder freigegeben werden, wenn sie nicht mehr benötigt werden. So ist der Umgang mit Lastspitzen wesentlich einfacher.
- Außerdem bieten Clouds meistens mehrere, weltweit verteilte Rechenzentren an – eine solche Infrastruktur ist ohne Cloud nur für sehr große Firmen bezahlbar.

## Ein Beispiel

Nehmen wir als Beispiel an, dass wir eine neue E-Commerce-Web-Site aufbauen wollen. Dafür benötigen wir einen Katalog mit den Waren, eine Bestandsführung für Informationen über Lieferbarkeiten und eine Möglichkeit, Bestellungen entgegenzunehmen. Für die tatsächliche Ausführung der Bestellungen gibt es ein Backend-System außerhalb der Cloud.

Dieses System soll nun in einer Public Cloud laufen. Das hat in diesem Szenario mehrere Vorteile:

- Es ist keine Investition notwendig, sondern das System verursacht nur Kosten, wenn es genutzt wird.
- Es kann schnell auch mit einer höheren Last zurechtkommen – das ist wichtig, wenn das Angebot plötzlich ein Erfolg wird.
- Die Installation neuer Versionen der Software ist sehr einfach und kann per Knopfdruck erfolgen. Die alte Version kann auch noch einige Zeit parallel betrieben werden, um so bei Fehlern in der neuen Version auf die alte zurückgreifen zu können. Einige PaaS bieten dazu recht ausgefeilte Ansätze, bei denen zum Beispiel die IP-Adressen des Angebots auf unterschiedliche Versionen des Systems umgeleitet werden können.
- Ebenfalls ist es einfach, Kopien der Produktionsumgebung zu erstellen und dabei beispielsweise mit einer Kopie der Datenbank Tests durchzuführen.

Die E-Commerce-Anwendung wird in der ersten Version Standard-Technologien wie eine relationale Datenbank und eine Java-Anwendung nutzen. Die Anwendung kann als WAR-Datei ausgeliefert werden. Im Moment gibt es noch wenige Angebote, die einen vollständigen Java EE-Server mit EJBs und EAR-Dateien in der Cloud anbieten.

Die Anwendung unterscheidet sich also kaum von anderen Java-Webanwendungen. Wieso kann diese Anwendung dennoch mit dem Ausfall einzelner Knoten zurechtkommen?

- Für die Webanwendung selbst können mehrere Knoten gestartet werden. Diese Knoten teilen sich die Last. So kann die Anwendung auch den Ausfall eines Knotens verkraften. Der dafür notwendige Loadbalancer ist bei den Cloud-Anbietern meistens im Angebot der Plattformen enthalten.
- Die Anwendung darf natürlich keine Daten auf den Knoten speichern, da diese beim Ausfall eines Knotens auch verloren gehen würden. Daher dürfen zum Beispiel keine Informationen in der HTTP-Session hinterlegt werden. Dieser Ansatz ist aber auch außerhalb der Cloud sinnvoll, um bessere Skalierbarkeit sicherzustellen.
- Die Datenbank muss natürlich auch mit dem Ausfall einzelner Knoten umgehen können. Dazu kann der Cloud-Nutzer entweder selbst einen Cluster in einem IaaS aufbauen oder Angebote wie Amazons RDS (Relational Database Service) nutzen, der schlüsselfertige MySQL- oder Oracle-Datenbank anbietet. Diese können auch als Cluster transparent über mehrere Rechenzentren verteilt werden. Außerdem werden Updates der Software von Amazon installiert.

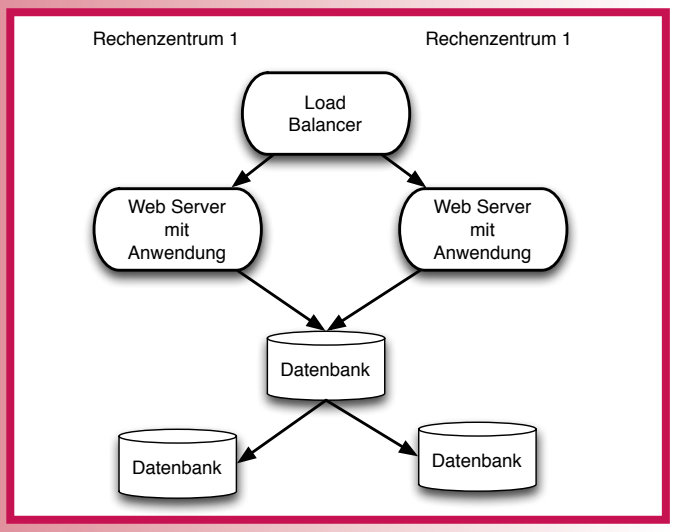


Abb. 1: Die Anwendung wird in mehreren Rechenzentren installiert. Ebenso ist die Datenbank über mehrere Rechenzentren verteilt, dadurch steht die Anwendung selbst bei Ausfall eines Rechenzentrums noch zur Verfügung

Allerdings haben viele Cloud-Anbieter Rechenzentren nur in den USA. Neben möglichen juristischen Problemen ergeben sich dadurch auch höhere Latenzzeiten, die bei E-Commerce-Angeboten auf den Umsatz durchschlagen können.

Daher lohnt sich der Einsatz eines Content Delivery Networks (CDN). Damit werden Bilder oder andere statische Inhalte auf eine Vielzahl von Servern an Netzknotenpunkten verteilt. Für jeden Kunden wird ein Server ausgewählt, der eine möglichst niedrige Latenzzeit hat, sodass der Zugriff auf die Ressourcen optimiert wird. Amazon und Microsoft bieten beide als Teil ihrer Cloud-Angebote ein solches CDN an.

## Die Grenze klassischer Architekturen

Bis jetzt unterscheidet sich der Entwurf für das System nicht signifikant von einem klassischen Enterprise-Java-System. Bisher war die Anwendung auch nur auf Deutschland begrenzt. Wenn wir nun aber die Anwendung weltweit verfügbar machen wollen, ist der Zugriff auf eine zentrale Datenbank nicht mehr sinnvoll:

- Die Latenzzeiten beim Zugriff sind zu groß.
- Angebote wie Amazons RDS bieten keine Datenbank, die weltweit verteilt ist.

Letztendlich ist es also so, dass bei dem weltweiten System die Grenze bei der Datenhaltung erreicht worden ist. Abhängig von dem zu entwickelnden System kann diese Grenze auch schon bei einem lokalen System erreicht worden sein. Wenn etwa sehr große Datenmengen verwaltet werden müssen oder die Daten nur auf eine bestimmte Art und Weise genutzt werden, kann die Grenze einer klassischen Architektur noch früher erreicht werden. In dem betrachteten Beispiel ist die Grenze aber absichtlich recht hoch gelegt.

Bei der Diskussion rund um Big Data und Cloud wird oft übersehen, dass nicht alle Entwickler ein System mit Skalierungsansprüchen und Datenmengen wie Twitter oder Facebook entwickeln werden. Auch in der Cloud sind relationale Datenbanken eine Alternative für die Entwicklung von Anwendungen, denn sie sind generell gut verstanden und die meisten Entwickler haben sie auch schon in zahlreichen Projekten verwendet.

## Das CAP-Theorem

Wenn das System zu einem globalen System umgebaut wird, ist es notwendig, das CAP-Theorem über verteilte Systeme zu kennen [Brew00]. CAP ist eine Abkürzung:

- Mit C (Consistency) ist die Konsistenz der Daten auf allen Knoten gemeint. C ist also erfüllt, wenn alle Knoten dieselben Daten haben.
- Availability (A) bedeutet, dass der Ausfall einzelner Knoten nicht die anderen Knoten davon abhält, weiterzuarbeiten. Insbesondere werden keine Knoten abgeschaltet.
- Partition Tolerance (P) bedeutet, dass das System mit Kommunikationsstörungen zwischen den Knoten umgehen kann. Man spricht von einer Partitionierung eines Netzwerks, wenn nicht mehr alle Knoten mit allen anderen Knoten kommunizieren können.

Das CAP-Theorem besagt nun, dass von den drei Eigenschaften maximal zwei in einem System gleichzeitig erreicht werden können. Natürlich kann ein System auch so schlecht entworfen sein, dass keine dieser Eigenschaften erfüllt ist, aber auf keinen Fall wird ein System alle drei gleichzeitig erfüllen. Einige Beispiele sollen dies illustrieren:

- Systeme mit C und A sind beispielsweise solche mit einem Zwei-Phasen-Commit-Protokoll. Dabei wird in einer ersten Phase jeder beteiligte Knoten gefragt, ob er die in einer Transaktion geänderten Daten schreiben kann. Falls alle Knoten das können, werden die Daten in einem zweiten Schritt tatsächlich geschrieben. Dadurch sind die Daten immer verfügbar (C) und es wird nie ein Knoten abgeschaltet, obwohl er eigentlich arbeiten könnte (A). Aber wenn die Knoten nicht mehr miteinander kommunizieren können, fällt das Gesamtsystem aus. P ist also nicht erfüllt. Das ist in klassischen Systemen akzeptabel, da dort mit Hochverfügbarkeitslösungen eine Partitionierung ausreichend unwahrscheinlich gemacht werden kann.
- Systeme, die mit Replikation arbeiten, haben die Eigenschaften P und A. Ein Beispiel ist DNS (Distributed Naming System), das im Internet zur Auflösung von Hostnamen zu Internet-Adressen genutzt wird. Dieses System erlaubt die Replikation von Daten auf mehrere Server. Dadurch sind Daten verfügbar, auch wenn gerade nicht jeder Server erreichbar ist (P), und es wird auch nie ein Server deaktiviert (A). Aber die Daten können inkonsistent sein, weil nicht alle Knoten alle Änderungen bekommen haben. Hier ist also Konsistenz (C) nicht erfüllt.
- Schließlich gibt es Systeme, die bei einer Änderung die verfügbaren Knoten „durchzählen“ (Quorum) und die Änderung nur annehmen, wenn genügend Knoten erreichbar sind. Dadurch ist Konsistenz gewährleistet und auch mit einer Partitionierung kann akzeptabel umgegangen werden, aber einige Knoten werden abgeschaltet, weil sie nicht mit genügend anderen Knoten kommunizieren können.

Ein Cloud-System benötigt ab einer bestimmten Menge von Knoten Partitionstoleranz, da nicht garantiert werden kann, dass alle Knoten immer mit allen anderen kommunizieren können. Wie dargestellt, ist in der Cloud der Ausfall einzelner Systeme sehr wahrscheinlich. Ebenfalls wird niemand Availability opfern – gerade bei E-Commerce-Systemen ist die Verfügbarkeit sehr wichtig, da sonst kein Umsatz erzielt wird. A und P müssen also gegeben sein – damit kann wegen des CAP-Theorems das System die Konsistenz der Daten nicht sicherstellen. Dieser Ansatz widerspricht den Architekturansätzen, die man sonst bei Enterprise-Systemen findet und die Konsistenz als höchstes Ziel haben.

Dieser Ansatz ist unter dem Akronym *BASE* (Basically Available, Soft State, Eventually Consistent) [Prit08] bekannt.

Dabei steht das System zwar immer zur Verfügung, aber es hat keinen einheitlichen Zustand über alle Knoten und ist nur schlussendlich konsistent („eventually consistent"). Also werden alle Knoten konsistent, wenn sie lange genug miteinander kommunizieren können und keine weiteren Updates mehr in das System kommen. Natürlich ist BASE (engl: Base bzw. Lauge) eine Anspielung auf *ACID* (engl: Säure) [HäReu83], was die Eigenschaften Atomizität, Konsistenz (Consistency), Isolation und Dauerhaftigkeit (Durability) einer Transaktion bezeichnet.

## CAP im Beispiel

Im Beispiel kann die Anwendung nun so weiterentwickelt werden, dass für die verschiedenen Regionen jeweils eine eigene Datenbank genutzt wird. Diese Datenbanken bekommen die Änderungen am Datenbestand als Nachrichten zugestellt. Fällt nun die Verbindung zu einer der Datenbanken aus, so werden die Änderungen erst später übertragen. So kommt es zwar kurzfristig zu Inkonsistenzen, die aber langfristig behoben werden. Vor allem sind alle Datenbank unabhängig voneinander verfügbar.

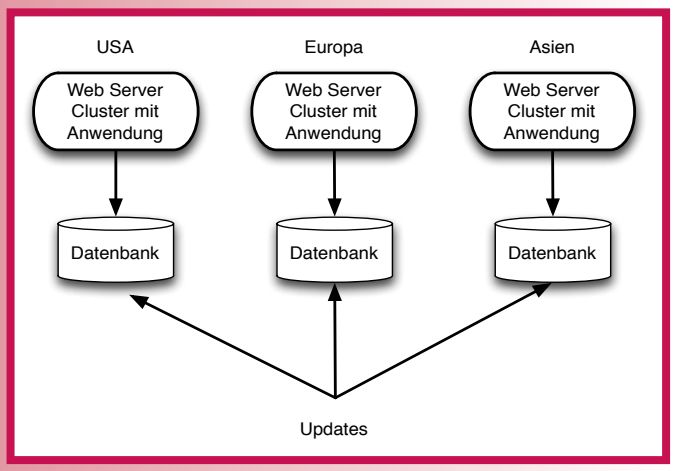


Abb. 2: Die Anwendung läuft nun in verschiedenen Regionen über die Welt verteilt. Für jede Region muss es eine eigene Datenbank geben. Änderungen werden an alle Datenbanken weitergeleitet. Das Verwalten der Updates geschieht durch eine MOM oder in der Persistenz mit einer NoSQL-Datenbank

Dieses grundlegende Konzept kann mithilfe einer Message Oriented Middleware (MOM) implementiert werden. Dazu zählen Systeme, die auf JMS (Java Messaging Service, [JSR914]) oder AMQP (Advanced Message Queuing Protocol, [AMQP]) setzen. Letztendlich implementiert das System dabei das Entwurfsmuster Event Sourcing [Fowl05]: Die Änderungen an den Daten werden als eine Sequenz von Events modelliert. Diese Idee stammt aus dem Domain-Driven Design [Evans03]. Das Muster ermöglicht nicht nur Replikation, sondern auch, dass sich, beispielsweise durch Kompensation, einzelne Änderungen an den Daten wieder rückgängig machen lassen. Die Änderungen können auch wieder „abgespielt" werden, um sie auf einem anderen Knoten zu replizieren. Außerdem kann mithilfe von Complex Event Processing auf den Ereignisstrom kontinuierlich und zeitnah reagiert werden.

Bei diesem Ansatz wird der BASE-Ansatz „von Hand" mit der MOM implementiert. Das System muss darauf ausgelegt sein, mit Messages umgehen zu können, und der Versand der Nachrichten als Ergebnis bestimmter fachlicher Szenarien muss implementiert werden. Eine Alternative ist es, die Implementierung dieser Ansätze der Infrastruktur zu überlassen. Dazu haben sich in letzter Zeit einige NoSQL-Datenbanken etabliert, die ebenfalls BASE-Ansätze verfolgen.

Dazu zählt beispielsweise CouchDB [CouchDB]. Diese Datenbank ist dokumentenorientiert. Das bedeutete, dass sie Dokumente direkt abspeichert, konkret in diesem Fall als JSON-Objekte (JavaScript Object Notation). Spannend ist dabei vor allem, dass CouchDB sehr einfach eine Replikation zwischen mehreren Knoten erlaubt. Dabei sind auch Master-Master-Replikationen möglich, bei denen also alle Server zum Schreiben neuer Daten genutzt werden können. Änderungen propagieren sich schrittweise durch die einzelnen Server. Dadurch ergibt sich eine Umsetzung von BASE. Allerdings verwalten alle Server dieselben Daten, was für große Datenmenge unrealistisch ist. CouchDB kann um Sharding erweitert werden: Dann werden die Datensätze unter den Servern aufgeteilt. Natürlich können die Daten immer noch repliziert werden, um so Ausfallsicherheit herzustellen.

Einen anderen Ansatz nutzt das Apache-Projekt Cassandra [Cassandra]. Jeder Knoten ist bei Cassandra der Master für bestimmte Datensätze. Die Daten werden auf N Knoten repliziert. Beim Schreiben von Daten wird gewartet, bis W Knoten das Schreiben bestätigt haben. Und Daten werden von R Knoten gelesen. Die Anzahlen N, W und R sind konfigurierbar. Dadurch kann der Nutzer das System den eigenen Anforderungen anpassen: Bei einem hohen R wird das System eher konsistente Daten erzeugen. Bei hohem N bietet das System bessere Verfügbarkeit. Mithilfe von NoSQL kann also ebenfalls ein BASE-System implementiert werden.

## Anwendung aufteilen

Eine weitere Herausforderung ist, dass die verschiedenen Teile des Systems eigentlich unterschiedlich Anforderungen an Konsistenz, Verfügbarkeit und Skalierung stellen.

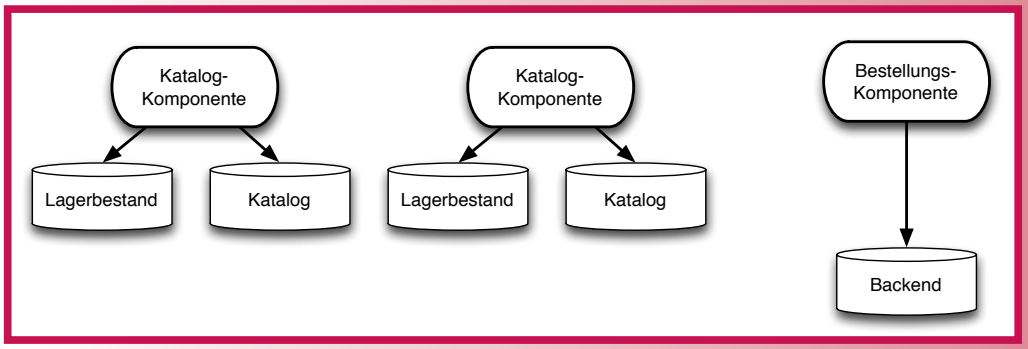


Abb. 3: Die Anwendung wird in Komponenten aufgeteilt. Je nach Last für die einzelnen Komponenten werden sie auf unterschiedlich vielen Servern installiert. Außerdem werden die Daten auf verschiedenen Servern gehalten, die je nach Konsistenzbedingungen konfiguriert sein können. Die Bestellungskomponente hat keine eigene Datenhaltung, sondern schickt die Daten direkt in das Backend

So muss der Katalog der Teile für unsere E-Commerce-Web-Site immer verfügbar sein. Kleine Inkonsistenzen sind akzeptabel: Wenn eine neue Ware nicht sofort überall zur Verfügung steht, sondern dieser Vorgang einige Minuten dauert, kann das akzeptabel sein. Für die Informationen über die Lagerbestände kann das anders sein: Wegen einer Inkonsistenz kann einem Kunden eine Ware zugesagt werden, obwohl sie nicht mehr verfügbar ist. Das kann zu enttäuschten Kunden und damit zu Problemen führen. Denkbar wäre also, für Lagerbestände strengere Anforderungen an die Konsistenz zu stellen. Bei Bestellungen kann auf eine Speicherung der Daten verzichtet werden, sie werden einfach in das Backend geschickt.

Für die Skalierung gilt, dass der Katalog wesentlich mehr Last haben wird als der Teil für die Bestellungen und die Lagerinformationen. Dementsprechend müssen für den Katalog mehr Server zur Verfügung stehen.

Diesen Herausforderungen kann begegnet werden, indem die Anwendung in mehrere Komponenten aufgeteilt wird. Diese können auf getrennten Servern installiert werden. Jede Komponente kann jeweils so viele Server nutzen, wie für sie notwendig sind. Außerdem kann die Datenhaltung so gewählt werden, dass die Konsistenz- und Skalierungsanforderungen erfüllt werden.

## Map/Reduce

Bleibt noch eine Anforderung: Für das Marketing sind Informationen darüber interessant, welche Waren wie oft abgerufen oder bestellt werden. Klassisch könnte dies mit einem Data-Warehouse gelöst werden. Der Ansatz hat aber den Nachteil, dass die Daten in das Data-Warehouse importiert werden müssen. Die Daten über Zugriffe und Verkäufe sind auf den verschiedenen Servern weltweit verteilt, was diese Option problematisch erscheinen lässt. Außerdem sind für Data-Warehouses recht leistungsfähige und damit teure Server notwendig.

Eine andere Möglichkeit wäre es, die Daten in einem Batch zu prozessieren und dadurch die Auswertungen laufen zu lassen. Das ist jedoch schwierig, da die Daten ebenfalls alle auf dem Knoten zur Verfügung stehen müssen, auf dem der Batch läuft.

In dieser Situation hat sich Google ein anderes Verfahren einfallen lassen: Map/Reduce ([DeGh04], Abb. 4). Zunächst wird auf die Daten der einzelnen Knoten eine Funktion angewendet. Das ist der Map-Schritt. Im konkreten Fall könnte für jeden Zugriff auf eine Ware ein Datensatz mit dem Namen der Ware erzeugt werden und analog für jeden Verkauf ebenfalls ein Datensatz mit der erworbenen Ware. Diese Datensätze werden in einem zentralen Knoten gesammelt und sortiert.

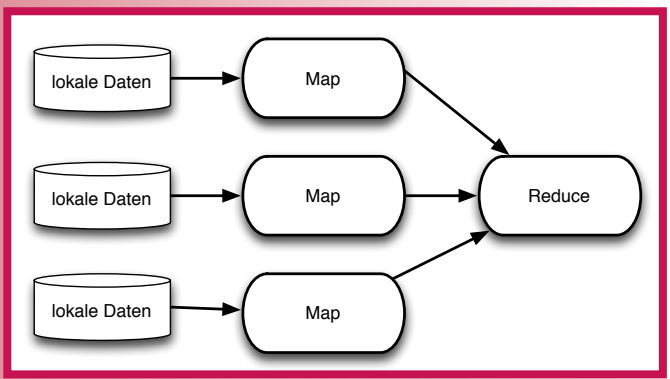


Abb. 4: Map/Reduce: In einem Map-Schritt wird eine Funktion auf lokale Daten auf zahlreichen Knoten angewendet. Im nächsten Schritt werden die Ergebnisse zu einem einzigen Ergebnis verdichtet (Reduce)

Anschließend können die Datensätze für eine Ware zu einem einzigen Datensatz mit dem Namen der Ware und der Anzahl Zugriff bzw. Verkäufe zusammengefasst werden. Das ist der Reduce-Schritt. Dieses Verfahren hat einige offensichtliche Ineffizienzen, aber auch Vorteile. So kann der Algorithmus durchgeführt werden, auch wenn einige der Knoten ausgefallen sind. Das macht ihn in verteilten Systemen einfacher einsetzbar.

Dieser Ansatz kann am besten mit Harvest/Yield [FoBr99] verdeutlicht werden. Yield ist die Wahrscheinlichkeit, eine Auswertung erfolgreich zu beenden. Harvest ist der Anteil Daten, der im Ergebnis berücksichtig worden ist. Diese beiden Zahlen beeinflussen sich gegenseitig. Ein hohes Yield führt zu einem niedrigen Harvest und umgekehrt. Natürlich sind diese Begriffe verwandt mit dem CAP-Theorem und sind auch ein guter Ansatz, um Cassandra-Konfigurationen zu bewerten.

## Wie nutzt man Cloud?

Die Nutzung von Cloud-Technologien ermöglicht auch ohne Nutzung von Public-Cloud-Ansätzen ganz andere Herangehensweisen. Mithilfe von Cloud-Architekturen können hochverfügbare Systeme mit billiger Standard-Hardware implementiert werden. Gleichzeitig bietet dieser Ansatz auch eine gute Skalierung, weil die Anzahl der Knoten erhöht werden kann, um mit größeren Lasten umzugehen. Dieser Ansatz ist generell attraktiv, um Enterprise-Systeme kosteneffizient zu implementieren. Dabei kann das System dank Private-Cloud-Technologien nach wie vor im eigenen Rechenzentrum laufen – eine Nutzung einer Public Cloud ist nicht notwendig. Bei vielen Firmen müssen hier noch Bedenken und teilweise auch juristische Hürden genommen werden, bevor neben einer Private Cloud auch eine Public Cloud genutzt werden kann.

Ein anderer wesentlicher Fortschritt der Cloud-Technologie ist, dass das Bereitstellen von Infrastruktur und die Installation von Software durch einen einfachen Aufruf der passenden APIs implementiert werden kann. Mit diesem Ansatz ist der Aufbau von Test- und Produktions-Umgebungen automatisierbar. Das hat mehrere Vorteile: Die Installation von Software wird zuverlässiger und schneller. Dadurch kann das Risiko bei der Einführung einer neuen Version minimiert werden. Außerdem können neue Features schneller in Produktion gebracht werden. Diese Ansätze sind die Grundlage für Continuous Delivery [HuFa10] und außerdem ein Zeichen des Zusammenwachsens von Betrieb und Entwicklung zu DevOps [DevOps]. Auch hier gilt, dass als Infrastruktur nicht unbedingt eine Public Cloud genutzt werden muss, sondern ebenfalls eine Private Cloud möglich ist.

## Fazit

Moderne Cloud-Angebote bieten die Möglichkeit, Anwendungen mit bekannten Ansätzen für Enterprise-Anwendungen wie beispielsweise relationalen Datenbanken zu betreiben. Allerdings kann es abhängig von den Skalierungsanforderungen notwendig sein, früher oder später eine andere Architektur zu nutzen. Dabei spielt das CAP-Theorem eine Rolle: In einer Cloud-Umgebung sind bezüglich der Konsistenz Kompromisse notwendig, um mit dem Ausfall einzelner Rechner in der Cloud umzugehen.

Dadurch ergeben sich BASE-Systeme: Sie haben eine bessere Verfügbarkeit, aber der Zustand auf den einzelnen Knoten kann sich unterscheiden. Solche Ansätze können in der Anwendung selbst realisiert werden, zum Beispiel durch Event Sourcing.

Die Alternative ist eine NoSQL-Lösung, die das Problem auf der Datenbank löst. Ebenfalls kann die Anwendung für eine bessere Skalierung in einzelne Teile aufgeteilt werden, die unabhängig voneinander skalieren und unterschiedliche Kompromisse bezüglich der CAP-Eigenschaften eingehen können. Schließlich können mit Map/Reduce große Datenmengen in einer Cloud analysiert werden. Dabei wird dank Harvest/Yield auch ein Ergebnis erzeugt, wenn einige der Knoten mit Daten nicht zur Verfügung stehen.

Die Herausforderungen in der Cloud sind also sehr spannend, aber mit modernen Ansätze auch lösbar.

## Links


**[AMQP]** Advanced Message Queuing Protocol, `http://amqp.org/`

**[Brew00]** R. A. Brewer, Folien zu: Towards Robust Distributed Systems, PODC Keynote, 19.7.2000, s. a. `http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf`

**[Cassandra]** `http://cassandra.apache.org/`

**[CloudFoundry]** `http://cloudfoundry.org/`

**[CouchDB]** `http://couchdb.apache.org/`

**[DeGh04]** J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, Dezember, 2004, `http://labs.google.com/papers/mapreduce.html`

**[DevOps]** D. Edwards, What is DevOps?, 22.2.2010, `http://dev2ops.org/blog/2010/2/22/what-is-devops.html`

**[EBeanStalk]** `http://aws.amazon.com/elasticbeanstalk/`

**[EC2]** Amazon Elastic Compute Cloud (EC2), `http://aws.amazon.com/ec2/`

**[Evans03]** E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003

**[FoBr99]** A. Fox, E. A. Brewer, Harvest, Yield, and Scalable Tolerant Systems, in: Proc. of the Seventh Workshop on Hot Topics in Operating Systems (1999), IEEE Comput. Soc, s. a. `http://radlab.cs.berkeley.edu/people/fox/static/pubs/pdf/c18.pdf`

**[Fowl05]** M. Fowler, Development of Further Patterns of Enterprise Application Architecture: Event Sourcing, 2005, `http://martinfowler.com/eaaDev/EventSourcing.html`

**[HäReu83]** Th. Härder, A. Reuter, Principles of Transaction-Oriented Database Recovery, in: ACM Comput. Surv. 15(4): 287-317 (1983), s. a. `http://www.minet.uni-jena.de/dbis/lehre/ss2004/dbs2/HaerderReuter83.pdf`

**[HuFa10]** J. Humble, D. Farley, Continous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, Addison-Wesley, 2010

**[JSR914]** Java Specification Request 914: Java Message Service (JMS) API, `http://jcp.org/aboutJava/communityprocess/final/jsr914/index.html`

**[Prit08]** D. Pritchett, BASE: An Acid Alternative, ACM Queue, 1.5.2008, `http://queue.acm.org/detail.cfm?id=1394128`

**[S3]** Amazon Simple Storage Service (Amazon S3), `http://aws.amazon.com/de/s3/`



**Eberhard Wolff** (Twitter: @ewolff) arbeitet als Architecture & Technology Manager für die adesso AG. Seine Schwerpunkte liegen auf Cloud-Technologien, Enterprise Java und anderem mit Spring und Softwarearchitekturen.
E-Mail: eberhard.wolff@gmail.com