NewSQL

NewSQL meint eine neue Generation von Datenbanken, welche die Vorteile der relationalen Datenbanken mit denen der NoSQL-Vertreter vereinigen wollen. Man spricht auch von **skalierbaren relationalen Systemen**.

Hintergrund

Der Begriff „NewSQL“ wurde durch das Marktforschungs- und Analyse-Unternehmen The 451 Group geprägt und beschreibt eine neue Generation von relationalen Datenbanken, welche die Flexibilität, Performance und Skalierbarkeit von NoSQL-Systemen unter Beibehaltung der ACID-Konsistenzeigenschaften und der SQL-Abfragesprache bieten möchte oder die Performance der relationalen Datenbanken derart steigern wollen, dass horizontale Skalierung nicht mehr nötig ist (Aslett, 2011). Dabei nutzen sie Techniken und Konzepte der NoSQL-Systeme wie Spaltenorientierung, In-Memory, Sharding und eine Verteilung der Aufgabenlast.

Als mit dem Web 2.0 immer mehr Daten in immer unstrukturierter Form in immer kürzerer Zeit anfielen und die Geschwindigkeit der Verarbeitung immer wichtiger wurde, stießen die traditionellen relationalen Datenbanken langsam an ihre Grenzen und neue, nichtrelationale Architekturen wurden unter der Bezeichnung NoSQL entwickelt, um mit den Massen der neuartigen Daten besser umgehen zu können. Um eine möglichst rasche Verarbeitung ermöglichen zu können, wurden die Einschränkungen der ACID-Eigenschaften flexibler durch das BASE-Prinzip umgesetzt, das Abstriche bei der Konsistenz macht.

Bei Online-Angeboten wie Webshops ist es nicht unbedingt wichtig, ob der virtuelle Einkaufskorb sofort aktualisiert wird. Es geht vielmehr darum, dass sofort auf Ereignisse reagiert wird, um dem Kunden ein schnelles und reibungsloses Einkaufserlebnis zu ermöglichen. Denn gerade im schnelllebigen Web, ist Geschwindigkeit unerlässlich, um mit der Konkurrenz mithalten zu können. Eine Untersuchung hat gezeigt, dass schon Verzögerungen von drei Sekunden dazu führen können, dass bis zu 40% der Besucher die Seite verlassen (Borland 2013). Hier spielt die Konsistenz der Verfügbarkeit gegenüber eine untergeordnete Rolle.

Die NewSQL-Bewegung versucht hier eine Brücke zu schlagen zwischen NoSQL und den traditionellen „OldSQL“ RDBMS.

Obwohl sich New-SQL-Systeme in ihrem Aufbau stark unterscheiden, haben sie im Grunde dieselben Eigenschaften:

- **SQL-Interface**
- **Relationales Datenmodell**
- **ACID-Transaktionen**
- **Horizontal Skalierbar**

Kategorien

Diese neuen SQL-Systeme lassen sich grob in drei Kategorien unterteilen (vgl. Venkatesh 2012):

- **Neue Datenbanken**: Das sind moderne Datenbanken, deren Architektur von Grund auf neu entwickelt wurden und dabei auch entsprechend die neuen Anforderungen an Performance und Skalierbarkeit, sowie den aktuellen Stand der Technik berücksichtigen. Beispiele: VoltDB, Clustrix, NuoDB.
- **Neue Speicher-Engines**: Sie bieten neue und optimierte Engines für bestehende SQL-Datenbanken an, die auch für horizontale Skalierung ausgelegt sind. Das hat den Vorteil, dass das bestehende Interface (etwa MySQL) weiterhin verwendet werden kann, aber eine Datenmigration von alten (My)SQL-Systemen ist nicht möglich. Beispiele: MySQL Cluster, TokuDB.
- **Transparentes Clustering**: Hierbei bleibt die eigentliche SQL-Datenbank unberührt und zum Einsatz kommt eine Middleware, die das Sharding der Datenbank übernimmt und es vor der Anwendung verbirgt, sodass sich der Anwender auf das Entwickeln seiner Applikation konzentrieren kann (vgl. Monash 2011). Beispiele: ScaleBase, dbShards.

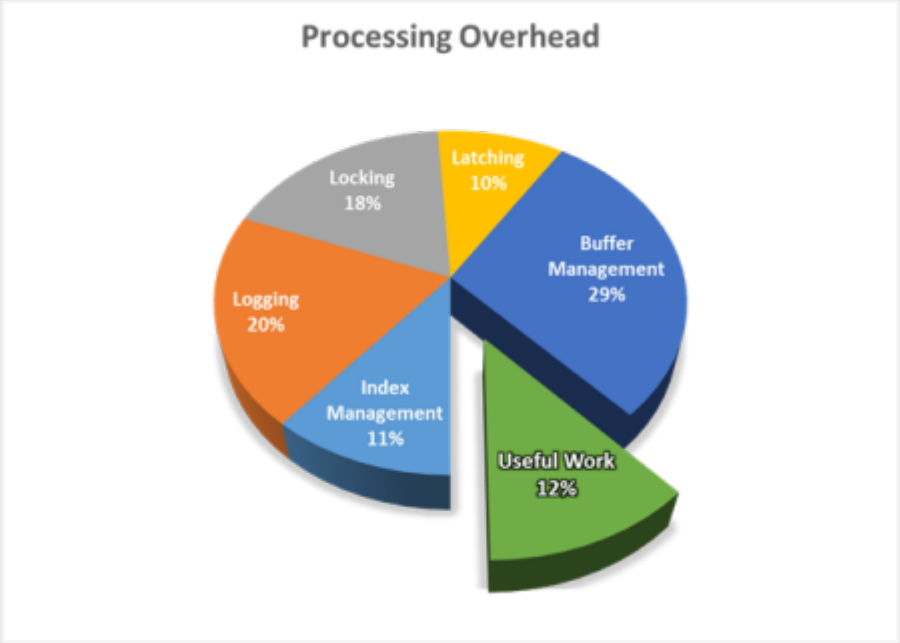
Architektur-Prinzipien

Grundgedanke hinter den NewSQL-Systemen ist, dass die traditionellen RDBMS den neuen Herausforderungen des Big-Data-Zeitalters vor allem deshalb nicht mehr gewachsen sind, weil sie immer noch auf (Hardware-)Grundlagen bauen, die in den 70er und 80er Jahren Stand der Dinge waren. Seit dieser Zeit hat es zwar einige Anpassungen, jedoch keine komplette Neuentwicklung mehr gegeben. So implementieren sie immer noch Architektureigenschaften von IBMs „System R“ (das erste RDBMS, mit SQL-Implementierung Mitte der 70er) (Stonebraker et al. 2007: S. 1):

- Festplatten-orientierte Speicher- und Indexstrukturen
- Multithreading, um Latenzzeiten zu vertuschen
- Lock-basierte Nebenläufigkeit
- Log-basiertes Wiederherstellungsverfahren

Diese Verfahren, die für die Datenintegrität gedacht waren, sorgen für einen Transaktions-Overhead, der letztlich einer Skalierung des Systems entgegensteht (vgl. VoltDB o. J.: S. 2).

In einer Untersuchung einer quelloffenen SQL-Datenbank „Shore“ hat sich gezeigt, dass nur etwa 12% der Zeit für tatsächlich „sinnvolle“ Aufgaben genutzt werden, der Rest der Zeit wird für **Locking**, **Latching**, **Puffer-** und **Index-Management** und das **Log-basiertes Recoveryprotokoll** benötigt. Diese fünf Schritte bei Verarbeitungen bilden den Flaschenhals in traditionellen RDBMS.



- **Logging**: Aus Sicherheitsgründen werden sämtliche Transaktionen nicht einfach nur ausgeführt, sondern zusätzlich auch noch in einem Log protokolliert, das auf Festplatte geschrieben werden muss, um die Datenbank bei einem Fehler wiederherstellen zu können. Dieser Vorgang kostet also viel Zeit.
- **Locking**: In parallelen Systemen schützt ein Thread durch Locks/Sperren Ressourcen vor Zugriffen anderer Threads. Es kommt bei konkurrierendem Zugriff also zu Wartezeiten, bis eine Ressource wieder freigegeben wurde.
- **Latching**: Für Updates auf Datenstrukturen in (verteilten) Multi-Threading-Datenbanken, werden Ressourcen für den Zugriff kurzzeitig gesperrt (anders als beim Lock wird nicht auf eine Warteschlage gesetzt, sondern regelmäßig angefragt).
- **Puffer-Management**: Daten, die auf RDBMS auf Festplatte gespeichert sind, werden in Seiten fester Länge gespeichert. Ein Pufferpool entscheidet, welche dieser Seiten im Hauptspeicher zwischengelagert werden. Zum Bearbeiten eines Records muss dieser geladen und wieder gespeichert werden.
- **Index-Management**: Die Verwaltung von Indexstrukturen (B-Bäume, Hash, etc.) ist CPU-intensiv und benötigt viele Disk-Zugriffe.

(Vgl. Harizopoulos et al. 2008; VoltDB o. J.)

(Grafik nach: VoltDB o. J.)

Neuer Ansatz

Auf Grundlage von Shore wurde dann mit „H-Store“, aus dem später „VoltDB“ hervorging, eine moderne, relationale Datenbank geschaffen, bei der versucht wurde, diesen Overhead zu eliminieren oder zu minimieren, um ein möglichst effizientes System zu schaffen, das auch für modernes OLTP (Online Transaction Processing/Online Transaktionsverarbeitung) geeignet ist.

Für die Umsetzung nutzten sie Techniken wie die **Shared-Nothing**-Architektur, bei der jeder Cluster-Knoten unabhängig ist, da er keine Ressourcen (Speicher) teilt und selbst auch nur auf seinem Teil der Daten arbeitet. Somit stellt ein einzelner Knoten auch keinen Single Point of Failure für das ganze System mehr dar, da die anderen im Falle des Versagens unabhängig weiterarbeiten können. Dafür kommt es bei Anfragen zu Performance-Einbußen, wenn auf Daten über mehrere Knoten/Shards

hinweg zugegriffen werden muss (Pavlo 2012).

Vor allem wurde auf In-Memory-Technik gesetzt, womit auch sämtliche Puffer-Verwaltung und Plattenzugriffe vermieden werden, da die Daten allzeit im RAM sofort verfügbar gehalten werden. Transaktionen dauern so auch nur wenige Milli- bis Mikrosekunden. Datenhaltung auf Festplatten ist nicht vorgesehen, zur Ausfallsicherheit gehört es daher, dass Daten (im RAM) auf verschiedenen Rechnern repliziert werden.

Um Latching- und Locking-Overhead zu vermeiden, wurde auf klassisches Multithreading mit geteilten Ressourcen verzichtet, stattdessen wird pro Prozessorkern ein **Single-Thread** betrieben, der einen eigenes Stück RAM für sich bekommt und keinerlei Ressourcen oder Datenstrukturen teilt. In einem System mit n CPUs wird der Hauptspeicher also in n Teile geteilt. Somit ist auch keine aufwendige Ressourcenverwaltung wie in aktuellen (multithreaded) RDBMS nötig. (Pavlo 2012)

Für die Sicherheit des Datenbestandes setzen H-Store und VoltDB auf **Command-Logging**, bei dem ein Logfile auf ein Minimum reduziert wird, indem nur die „Befehle“ (in diesen Systemen sind es „Stored Procedures“, die wiederum mehrere SQL-Befehle enthalten können) geloggt werden, die die Daten verändern. Zusammen mit Snapshots lässt sich bei einem Ausfall der letzte Datenstand wiederherstellen. Um Plattenzugriffe zu minimieren, werden die Logs zunächst im RAM gehalten, und erst nach festgelegtem Zeitintervall auf Platte persistiert. (VoltDB 2015) Um zusätzlich für Ausfallsicherheit und einen konsistenten Datenbestand zu sorgen, werden Knoten mit Partitions-Kopien geführt. Vor einem Transaktions-Commit werden die Daten synchron an alle replizierten Partitionen gesendet (VoltDB o.J.).

Logging kann alternativ auf ein kurzlebigen, im Hauptspeicher gehaltenen Undo-Log (für den Fall, dass eine Transaktion fehlschlägt) reduziert werden, der nach einer erfolgreichen Transaktion wieder gelöscht wird (Stonebraker et al. 2007: S. 3).

Für Abfragen kommen vom Nutzer vordefinierte **Stored Procedures** zum Einsatz, die in Java geschrieben werden und eingebetteten SQL-92-Code zur Datenbankabfrage nutzen.

Eine weitere interessante NewSQL-Implementierung ist „**Splice Machine**“. Dabei handelt es sich um eine relationale Datenbank, die auf Hadoop aufsetzt. Genauer ersetzt sie die Storage-Engine der leichtgewichtigen relationalen Datenbank Apache „Derby“ mit dem HDFS und HBase. Damit ist es möglich, auch große Datenvolumen zu verarbeiten, die die Kapazitäten eines einzelnen RDBMS-Servers oder eines NewSQL-In-Memory-Clusters übersteigen würden. Sie eignet sich sowohl für ACID-Transaktionen, als auch analytische Aufgaben und bietet volle ANSI-SQL-Kompatibilität. (Vgl. Splice Machine 2015) Es vereint dabei SQL mit der Skalierbarkeit von Hadoop. Aufgaben verteilt, parallel bearbeitet und nachher wieder zusammengeklebt (engl. „splice“). Splice Machine sieht sich als Ersatz, wenn Systeme wie Microsoft SQL Server, IBM DB2 oder MySQL an ihre Performance- oder Kostengrenzen stoßen (nach Henschen 2014).

Vergleich OldSQL, NoSQL und NewSQL

Im Gegensatz zu den traditionellen RDBMS (OldSQL) sind NewSQL-Systeme, ebenso wie die NoSQL-Systeme, skalierbar und hochverfügbar, sind aber dennoch relationale Systeme, die vollwertige ACID-Transaktionen und SQL unterstützen.

Ein Vorteil gegenüber NoSQL ist, dass sie mit den standardisierten Zugriffs-APIs und Treiber der traditionellen Systeme kompatibel sind, das heißt, dass bestehende Applikationen auch nicht angepasst werden müssen. Dies ist auch ein Vorteil gegenüber den NoSQL-Systemen, die noch über keinen einheitlichen Zugriffs-Standard verfügen. (Welkenbach/Schmutz 2012) Da sie jedoch auf das relationale Modell und SQL als Abfragesprache bauen, liegt ihnen, anders als den NoSQL-Datenbanken, ein Schema zugrunde, was sie in Hinsicht auf die Flexibilität im Umgang mit Big Data doch einschränkt.

Allgemein bestimmt die Art der Daten auch die Wahl der Datenbank. Wird Datenintegrität und Konsistenz verlangt, steht die Wahl zwischen RDBMS und NewSQL. NewSQL-Systeme wurden dabei mit besonderem Fokus auf Performance und Skalierbarkeit entworfen. OldSQL-Systeme wurden für die Hardware-Architekturen der 70er und 80er entwickelt und werden so teilweise durch unnötig großen Overhead gebremst, wohingegen die neue Generation versucht, diesen unter Einbeziehung neuer Hardwareentwicklungen weitestgehend zu minimieren, um so um Größenordnungen schneller als traditionelle RDBMS zu werden (vgl. Stonebraker et al 2007).

Ist es nötig, dass das System zu skalieren, bieten sich NoSQL- und NewSQL-Systeme an, da sie, im Gegensatz du RDBMS, nicht auf ein kostspieliges vertikales Skalieren beschränkt sind, sondern frei horizontal skalierbar sind, sich also ohne große Komplikationen durch neue Maschinen (mit kostengünstiger Standardhardware) ergänzen lassen. Dabei gelten jedoch insofern Einschränkungen, als dass die meisten NewSQL-Systeme hauptsächlich für schnelle Transaktionen und Operationen ausgelegt wurden, diese also nur in kleinem Umfang machen und JOINs über zu viele Knoten vermeiden, da sich dies negativ auf die Performance auswirkt (vgl. Cattel 2010: S. 9; Pavlo 2012).

(Grafik-Quelle: Choudhary 2014)