Einsatz von alternativen Datenbankstrukturen und -systemen im E-Commerce Bereich

Kathrin Kaufleitner

DIPLOMARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juli 2011

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 25. Juni 2011

Kathrin Kaufleitner

Inhaltsverzeichnis

Vorwort

Relationale Datenbanken begleiteten mich schon mein gesamtes Studium, war es bei deren praktischer Anwendung bei Projekten oder innerhalb einer Lehrveranstaltung. Oft war ich so in die Thematik vertieft, dass ich mir die gesamte reale Welt relational oder in einer Programmiersprache vorstellte. Dieses Phänomen entdeckten vermutlich auch viele andere meiner KommilitonInnen, wenn sie versuchten mittels IF/ELSE das Wetter und seine Folgen für den Ausgang der anstehenden Klausur zu beschreiben, oder auch mit „STRG + Z“ Geschehenes rückgängig zu machen.

Als ich vor zwei Jahren das erste Mal mit dem Begriff „NoSQL“ in einem Magazin konfrontiert war, stieg meine Neugier, wie eine so durchdachte und über Jahrzehnte lange bewährte Lösung von Relationen innerhalb einer Datenbank einfach über Bord geworfen werden konnte. Diese neuen alternativen NoSQL-Datenbanken sollten in der Theorie die Lösung aller Probleme der relationalen Datenbanken sein. Automatisierte Replikationsmechanismen, horizontale Skalierung, Ausfalltoleranz sowie flexible Datenstrukturen sind, so heißt es, ihre Stärken. Jedoch gibt es selten Stärken ohne Schwächen.

Zwar zeigen mittlerweile schon viele Unternehmen wie Facebook, Amazon und Twitter, dass der Einsatz von NoSQL-Datenbanken in der Praxis viele Vorteile mit sich bringt, jedoch will man sich immer selbst ein Bild davon schaffen. Mein Bild dieser NoSQL-Datenbanken spiegelt sich in folgender Masterarbeit wider. Um einen konkreten, praktischen Anwendungsfall zu schaffen, trieb es mich in den Bereich des E-Commerce. Einzig die Open-Source Lösung Magento spezialisierte sich hierbei auf das EAV-Modell, um die Datenstruktur in seiner relationalen MySQL Datenbank so flexibel als möglich zu gestalten. Flexibilität garantieren jedoch auch NoSQL-Datenbanken. Welche Datenbankstruktur bzw. welches Datenbanksystem ist also mit dem heutigen Wissensstand am Sinnvollsten für die Verwaltung von Daten einer E-Commerce Lösung? Dieser Fragestellung widmete ich mich mit Begeisterung in dieser Masterarbeit.

Abschließend möchte ich mich bei jenen Personen bedanken, welche mich im Laufe meines gesamten Studiums und vor allem während der Entstehung meiner Masterarbeit tatkräftig unterstützt und begleitet haben. Dieser Dank gilt in erster Linie allen Professoren der Fachhochschule Hagenberg, welche mich durch die Vermittlung ihres Fachwissens meiner Berufung ein großes

Stück näher gebracht haben. Speziell zu erwähnen ist hierbei mein Betreuer DI Martin Harrer für seine engagierte und ausgezeichnete Unterstützung während der Entstehung meiner Masterarbeit. Weiters möchte ich meinem Partner Robert danken. Durch sein Fachwissen und seiner Geduldigkeit unterstützte und inspirierte er mich im Laufe meines gesamten Studiums. Meiner Mutter Maria, sowie auch meinen Freunden sei ebenfalls ein großer Dank ausgesprochen. Sie glaubten an mich und halfen mir mit einer abwechslungsreichen Freizeitgestaltung, die Gedanken in meinem Kopf neu zu sortieren.

Kurzfassung

Alternative Speichersysteme sind im Zeitalter des Web 2.0 und des Cloud Computing gefragt wie nie zuvor. Webapplikationen mit einer großen Datenmenge im Hintergrund wenden sich immer mehr von relationalen Datenbankmanagementsystemen wie MySQL ab, da diese Probleme mit der Verarbeitung, Replikation und vor allem horizontalen Skalierung einer solch immensen Datenansammlung mit sich bringen. NoSQL, als Konzept verschiedener alternativer Datenbanken, setzt indes unter anderem auf eine hohe horizontale Skalierbarkeit, Ausfalltoleranz und weitgehend automatisierte Replikationsmechanismen.

In dieser Masterarbeit wird der Trend zu NoSQL näher erläutert, verschiedene NoSQL-Systeme und deren übergeordneten Datenbanktypen aufgelistet sowie ein Vergleich zu relationalen Datenbanken aufgestellt. Da diese alternativen Speichersysteme und auch -strukturen im E-Commerce Bereich noch sehr wenig bis gar keine Verwendung finden, werden ihre Stärken sowie auch Schwächen in Hinblick auf den dortigen Einsatz analysiert. Im Konkreten wird dabei die Verwendung des EAV-Modells im Rahmen eines relationalen Datenbankmanagementsystems sowie die dokumentorientierte Datenbank MongoDB in Verbindung mit der Open-Source E-Commerce Lösung osCommerce evaluiert und diskutiert.

Abstract

In accordance with the developments from Web 2.0 and the latest trend Cloud Computing, alternative data storage systems have experienced a surge in popularity. Web applications that handle large amounts of data are starting to turn away from relational database management systems such as MySQL, as they have problems dealing with such enormous data collections in terms of replication, processing and especially horizontal scaling. NoSQL, as a concept of different alternative databases, however, supports a high degree of horizontal scalability, fault tolerance and, in most cases, automated replication mechanisms.

In this master's thesis, the hype surrounding NoSQL is explained in detail. Several NoSQL systems and their parent database types are listed and a comparison to relational databases is provided. As these alternative storage systems and structures are currently not used in the field of e-commerce, their strengths and weaknesses in combination with an e-commerce system are analyzed. In concrete terms, the EAV model in the context of a relational database management system and the document-oriented database MongoDB are evaluated and discussed in connection with the open-source e-commerce solution osCommerce.

Kapitel 1

Einleitung

1.1 Motivation, Problemstellung und Zielsetzung

Die Größe der Datenmengen ist im Zeitalter des Web 2.0 besonders durch User-generated Content rasant gestiegen. In Hinblick auf die Verarbeitung und vor allem Skalierung und Replikation auf verschiedene Server von solch einer großen Menge an Daten, stoßen relationale Datenbanken schnell an ihre Grenzen. Somit waren bzw. sind nun alternative Datenbank-Speichersysteme gefragt wie nie zuvor. In den letzten Jahren wurden immer mehr dieser Alternativen mit einem nicht-relationalen Ansatz entwickelt und veröffentlicht. Um diese „neuen“ Systeme auch namentlich ansprechen zu können, wurde der Überbegriff „NoSQL“ (Not only SQL) Anfang 2009 von Johan Oskarsson neu eingeführt, welcher damals ein Treffen über verteilte strukturierte Datenspeicher einberief. In dieser Masterarbeit wird die Verwendung dieser sowie auch deren Stärken bzw. Schwächen analysiert.

1.1.1 NoSQL Datenbanken im E-Commerce-Bereich

In E-Commerce Systemen finden solche alternativen Datenbanken noch sehr wenig bis gar keine Verwendung. Dabei wäre es in diesem Bereich, vor allem bei der Verwaltung von Produkten, von Vorteil, würde das Schemata der Tabellen etwas offener gestaltet sein.

Relationale SQL-Datenbanken geben bei der Erstellung einer Tabelle eine Struktur für die darin zu speichernden Objekte vor. Würde einem Objekt im Nachhinein eine neue Eigenschaft zugewiesen werden, müsste mittels ALTER TABLE die gesamte Tabelle überarbeitet werden. Diese Datenbankoperation beansprucht unnötig viel Zeit, zusätzliche Ressourcen und Administrationsaufwand.

Am Beispiel eines Online-Shops, welcher wie auch Amazon[1] eine große Bandbreite von verschiedenen Produkttypen vertreibt, ist klar zu erkennen,

[1] http://www.amazon.de

dass zum Beispiel eine CD andere Produkteigenschaften besitzt als ein Kochtopf. Können diese Eigenschaften nicht flexibel in der Datenbank gespeichert werden, ist eine performante Suche nach CDs eines bestimmten Interpreten nahezu unmöglich. Der Online-Shop kann den Anforderungen des Kunden nicht zu hundert Prozent gerecht werden.

In der ersten Stufe dieser Arbeit hin zu einer skalierbaren, flexiblen und performanten Datenbank eines E-Commerce Systems wird das EAV-Modell in einem relationalen Datenbankmanagementsystem näher betrachtet, welches schon in der Open-Source E-Commerce-Plattform Magento[2] Verwendung findet. Da aber dieses Modell durch den Einsatz sehr komplexer SQL-Queries starke Nachteile hinsichtlich der Performance birgt, wird im zweiten Schritt der Einsatz einer dokumentorientierten Datenbank in einem E-Commerce System untersucht. Dabei wird die, seit 2009 bestehende, Open-Source Datenbank MongoDB[3] verwendet. Das zugrunde liegende E-Commerce System ist die Open-Source Lösung osCommerce[4], da es im Vergleich zu Magento einfacher und somit transparenter aufgebaut und die Datenbankstruktur somit schneller abänderbar ist.

Zudem wird noch eine theoretische Übersicht über weitere NoSQL-Alternativen wie Key/Value Datenbanken, spaltenorientierte Datenbanken und Graphdatenbanken sowie deren Vor- bzw. Nachteile in Bezug auf E-Commerce Systeme gegeben.

1.1.2 Abgrenzung

Die Umstellung der Datenbank erfolgt für die Tabellen in denen die größten Datenmengen verwaltet werden müssen, im Speziellen die Produkt- sowie die Kundentabellen. Alle weiteren Tabellen werden im Prototypen über die relationale Datenbank MySQL verwaltet. Horizontale Skalierung sowie Replikation der Datenbank wird bei der Auswertung vernachlässigt und dabei rein auf das theoretische Wissen vertraut.

1.2 Gliederung der Arbeit

Folgende Beschreibung des Aufbaus dieser Arbeit soll einen Überblick verschaffen sowie auch dem besseren Verständnis dieser dienen.

Kapitel 1 – Einleitung

Dieses Kapitel gibt einen Überblick über die Arbeit sowie der generellen Thematik, Problematik und Zielsetzung.

[2] www.magentocommerce.com
[3] http://www.mongodb.org/
[4] http://www.oscommerce.com

Kapitel 2 – Theoretische Datenbank-Grundlagen

Das zweite Kapitel umfasst die benötigten theoretischen Grundlagen in Hinblick auf Datenbankarchitekturen und -systeme, auf welche diese Arbeit aufbaut. Es dient der Gewinnung des Grundlagenverständnisses für die Thematiken der darauf folgenden Kapitel. Unterstützend dabei ist der Einsatz von Beispielen und Grafiken. Voraussetzung ist jedoch ein Basiswissen im Bereich Datenbanken.

Kapitel 3 – EAV-Modell

In diesem Kapitel wird der Einsatz des EAV-Modells in relationalen Datenbankmanagementsystemen theoretisch betrachtet und anhand von Beispielen erklärt. Zudem werden die Vor- sowie auch Nachteile bei der Verwendung dieses Modells erläutert. Dieses Kapitel dient dem Verständnis und der Vorbereitung auf das Kapitel 5.

Kapitel 4 – NoSQL-Datenbanken

Kapitel 4 befasst sich mit der erst seit 2009 bestehenden NoSQL-Bewegung im Allgemeinen und ihrer vier Hauptdatenbanktypen. Zudem wird zu jedem Typ ein populärer Vertreter hinsichtlich seines Datenmodells, seiner Skalierung, Replikation, Performance, Konsistenzerhaltung der Daten sowie seines Einsatzgebietes analysiert. Dieses Kapitel bereitet wie Kapitel 3 ebenfalls auf Kapitel 5 vor.

Kapitel 5 – Implementierung

In Kapitel 5 wird die Umsetzung des EAV-Modells sowie der letztendlich gewählten dokumentorientierten Datenbank MongoDB im Open-Source E-Commerce System osCommerce geschildert. Zudem wird die Wahl des zugrunde liegenden E-Commerce Systems sowie auch der NoSQL-Datenbank begründet.

Kapitel 6 – Evaluierung und Diskussion

In diesem Kapitel wird als Schritt nach der Implementierung der Einsatz des EAV-Modells sowie auch der dokumentorientierten Datenbank MongoDB bei einem E-Commerce System hinsichtlich ihrer Performance evaluiert und diskutiert. Ebenso werden spaltenorientierte Datenbanken, Key/Value Datenbanken sowie Graphdatenbanken theoretisch auf ihren Einsatz in einem E-Commerce System untersucht.

Kapitel 7 – Schlussbemerkung

Das letzte Kapitel resümiert die Arbeit sowie ihre Ergebnisse.

Kapitel 2

Theoretische Datenbank-Grundlagen

Um einen Überblick über die Probleme der verschiedenen Datenbanksysteme und deren Lösungen zu erhalten, wird in diesem Kapitel ein theoretischer Überblick über die wichtigsten Begriffe in Zusammenhang mit dem Aufbau eines Datenbanksystems bzw. einer Datenbankstruktur gegeben. Diese beinhalten:

- ACID
- CAP-Theorem
- BASE und Eventually Consistent
- MapReduce Framework
- Replikation
- Sharding
- MVCC
- Vektor Clocks
- Consistent-Hashing

2.1 Konsistenz, Verfügbarkeit und Partitionierung

2.1.1 ACID

ACID, das Akronym für *atomicity*, *consistency*, *isolation* und *durability*, steht, wie unter anderem in [45, S. 23] beschrieben für die Kriterien von gut funktionierenden Transaktionen innerhalb eines Datenbankmanagementsystems. Durch diese Eigenschaften ist die Verlässlichkeit eines solchen Systems vorausgesetzt.

Atomicity – Atomarität

Transaktionen müssen atomar sein, sie können also nicht partiell abgeschlossen werden. Scheitert eine einzelne Datenbankanweisung inner-

halb dieser Transaktion, wird die gesamte Transaktion nicht durchgeführt und alle vorangegangenen Datenbankanweisungen rückgängig gemacht. Man spricht vom „Alles oder nichts-Prinzip“.

Consistency – Konsistenzerhaltung

Konsistenz ist vor und nach einer Transaktion gewährleistet. Eine Transaktion führt eine Datenbank von einem konsistenten Zustand in einen anderen über. Dazwischen kann die Datenbank bei einzelnen Anweisungen oder bei Absturz während einer Transaktion auch inkonsistent sein.

Bei einem Absturz des Datenbanksystems wird die Konsistenz während der Restorephase wiederhergestellt. Die Daten der letzten aktiven Transaktion können verloren gehen, wenn sie nicht mehr mit COMMIT abgeschlossen wurden.

Isolation – Isolation

Jede Transaktion ist von anderen gleichzeitig laufenden Transaktionen isoliert und unsichtbar bis sie abgeschlossen ist. Dadurch beeinflussen sich mehrere Transaktionen bei ihrer Ausführung nicht gegenseitig.

Durability – Dauerhaftigkeit

Ergebnisse einer abgeschlossenen Transaktion werden dauerhaft gespeichert. Somit wird sichergestellt, dass die Daten bei einem Datenbanksystemabsturz nicht verloren gehen [45, S. 23].

Die Vorteile von ACID-Transaktionen liegen in der Gewährleistung der Verlässlichkeit des Datenbanksystems. Mit transaktionsunterstützenden Storage Engines, wie InnoDB für das Datenbankmanagementsystem MySQL, wird dem Anwendungsentwickler der komplexe Aufbau einer solchen Transaktionslogik abgenommen. Durch diese zusätzliche Sicherheit muss der Datenbank-Server jedoch mehr Arbeit erledigen und benötigt dafür mehr Ressourcen. Somit muss mit starken Performance-Einbußen gerechnet werden, wenn die Hardware nicht exakt an die Datenbank-Anforderungen angepasst wird [45, S. 24].

2.1.2 CAP-Theorem

Das CAP-Theorem wurde im Jahr 2000 in einem Vortrag im Rahmen eines ACM-PODC von Eric Brewer [5, S. 24] vorgestellt. Kernpunkt dieses Vortrages war, dass ein Web-Service niemals folgende drei Eigenschaften komplett erfüllen kann: Konsistenz, Verfügbarkeit und Ausfalltoleranz. Nur zwei dieser drei Eigenschaften können gleichzeitig erreicht werden. Nancy Lynch und Seth Gilbert belegten diese Theorie im Jahr 2002 ebenfalls mit Beispielen innerhalb von ansynchronen sowie synchronen Netzwerken [16, S. 24].

In Abhängigkeit mit den Anforderungen an ein System muss abgewägt werden, welche zwei der drei Eigenschaften wichtiger sind bzw. welche gelockert werden könnten. Im Zuge der NoSQL-Bewegung entstanden ebenfalls

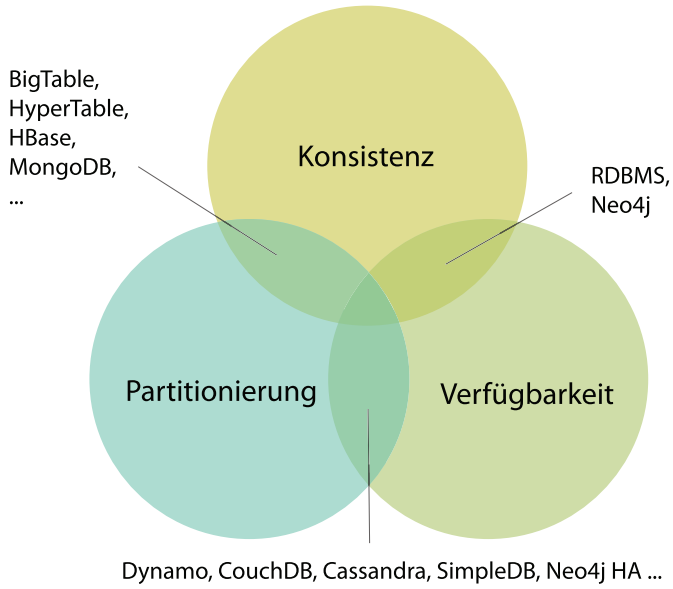


Abbildung 2.1: Einteilung der Datenbankmanagementsysteme nach dem CAP-Theorem.

viele alternative Datenbanksysteme, welche diese Eigenschaften verschieden abdecken, wie Abbildung 2.1 zeigt.

Ein vereinfachtes Beispiel aus [11, S. 32] zeigt mit zwei Knoten, wie Konsistenz und Verfügbarkeit sich nicht gleichzeitig mit Ausfalltoleranz vereinigen lassen. Dafür wird folgende Terminologie verwendet:

K ... Knoten mit Replikationen der Daten

D ... Daten

S ... Synchronisationsaufruf ausgehend von einem Knoten zum anderen

Im ersten Schritt enthalten K1 sowie auch K2 mittels Replikation denselben Datensatz D0. K1 ist hierbei für alle Schreiboperationen zuständig, K2 für alle Leseoperationen, wie in Abbildung 2.2 zu sehen ist.

Durch eine Schreiboperation auf K1 wechselt der Datensatz D0 in den Zustand D1. Mittels des Synchronisationsaufrufes S wird K2 aktualisiert und enthält somit ebenso den Zustand D1 (s. Abbildung 2.3).

Wäre aber die Kommunikation zwischen den Knoten K1 und K2 durch einen eventuellen Ausfall der Netzwerkverbindung nicht mehr gegeben, wird der alte Datensatz D0 gelesen, wie Abbildung 2.4 illustriert. Würde dieses Datenbanksystem ebenfalls ein Protokoll führen, welches erst bei vollständiger Synchronisation beider Knoten eine Transaktion abschließt, wären Teile

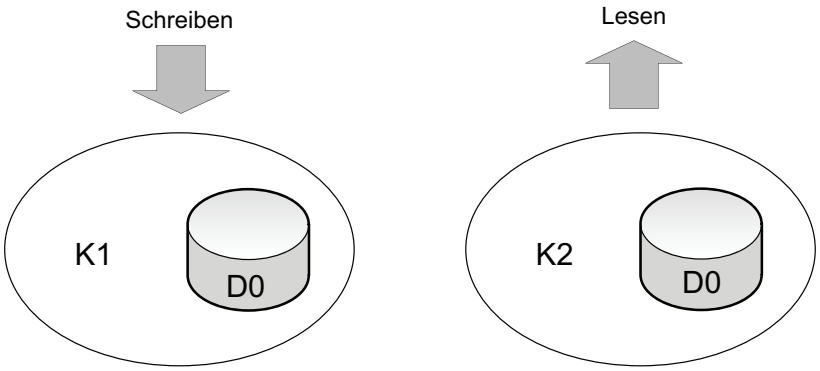


Abbildung 2.2: CAP-Theorem Beispiel: Beide Knoten enthalten denselben Datensatz.

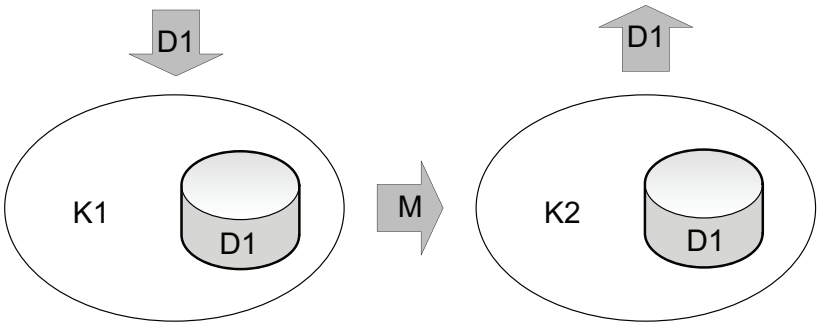


Abbildung 2.3: CAP-Theorem Beispiel: Datensatz wird von K1 auf K2 synchronisiert.

im Knoten K1 blockiert, was im nächsten Schritt bei weiteren Schreiboperationen schnell die Verfügbarkeit des Systems einbrechen ließe. Soll aber nun die Verfügbarkeit des Systems trotzdem gewährleistet sein, muss die Konsistenz der Daten auf K1 und K2 gelockert werden [11, S. 32].

2.1.3 BASE und Eventually Consistent

BASE (*Basically Available, Soft State, Eventually Consistent*) stellte die Alternative des ACID-Ansatzes dar. In diesem Konsistenzmodell steht die Verfügbarkeit des Systems über der Konsistenz. Konsistenz selbst ist hier nur eine Übergangsform und im Vergleich zu den ACID-Eigenschaften kein Zustand vor und nach einer Transaktion [11, S. 34].

Generell wird zwischen zwei Grundformen der Konsistenz unterschieden:

Strong consistency – bei dieser Form muss die Datenbank, wie in Abschnitt 2.1.1 beschrieben, vor sowie auch nach einer Anweisung darauf konsistent sein.

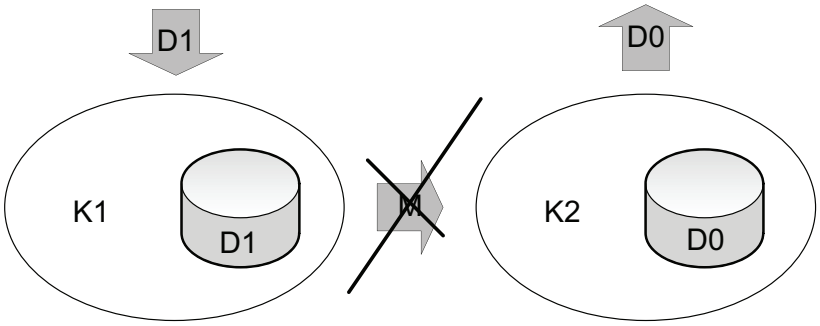


Abbildung 2.4: CAP-Theorem Beispiel: Problem bei Ausfall der Netzwerkverbindung.

Weak consistency – Die Datenbank muss nicht zu jedem Zeitpunkt konsistent sein, erst nach einer Reihe von Bedingungen muss sie wieder der konsistenten Form entsprechen.

Eventual consistency stellt eine spezielle Form der *weak consistency* dar. Das Datenbanksystem garantiert hier, dass eventuell alle Zugriffe den aktuellsten Wert zurückliefern, wenn dieser Wert inzwischen nicht neu aktualisiert wurde. Das sogenannte *inconsistency-window*, also der Zeitraum in dem der Wert nicht konsistent ist, wird durch Faktoren wie Kommunikations-Verzüge zur Datenbank, aktuelle Auslastung sowie der Anzahl der Replikationen beeinflusst [41].

Im Vergleich zu relationalen Datenbankmodellen findet diese Form der Konsistenzerhaltung vor allem in NoSQL-Datenbanken Verwendung. Das Modell der *eventual consistency* umfasst, wie in [41] beschrieben, verschiedene Variationen, welche auch miteinander kombiniert werden können:

Causal consistency – In dieser Variante sind nur kausal voneinander abhängige Schreibzugriffe zeitlich geordnet. Sie werden von anderen Prozessen ebenfalls in dieser Reihenfolge gesehen. Für Operationen, die nicht in dieser Beziehung stehen, ist die Reihenfolge gleichgültig (s. Abbildung 2.5).

Read–your–writes consistency – Der Prozess sieht in dieser Variante immer sein eigens aktualisiertes Datenobjekt. Andere Prozesse jedoch können im inconsistency-window immer noch auf einen veralteten Wert des selben Datenobjektes zugreifen (s. Abbildung 2.6).

Session consistency – Diese Variante ist eine spezielle Form des read-your-writes consistency, wobei der Prozess hier im Rahmen einer laufenden Session auf das Speichersystem zugreift. Solange die Session aufrechterhalten wird, wird dem Prozess vom System eine read-your-write consistency gewährt. Bricht die Session jedoch ab und wird eine neue gestartet, gilt diese Gewährleistung nur noch für Operationen innerhalb der neuen Session (s. Abbildung 2.7).

Monotonic read consistency – Wenn ein Prozess bei einer Leseope-

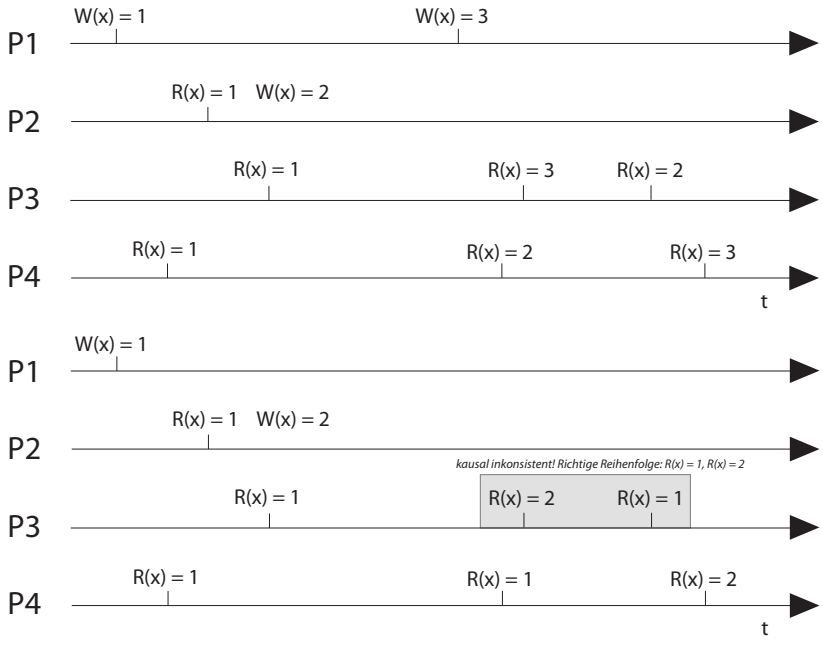


Abbildung 2.5: Causal consistency.

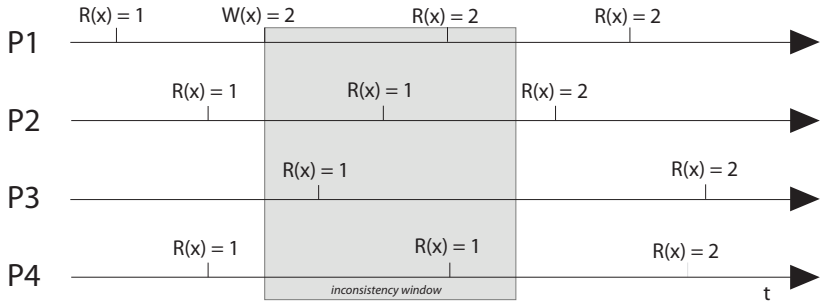


Abbildung 2.6: Read-your-writes consistency.

ration einen aktuelleren Wert gelesen hat, wird er in weiterer Folge keinen älteren Wert als diesen mehr lesen (s. Abbildung 2.8).

Monotonic write consistency – Eine zweite Schreiboperation eines Prozesses erfolgt erst dann, wenn der vorangegangene Schreibprozess vollständig abgeschlossen wurde – die Schreiboperationen werden also serialisiert.

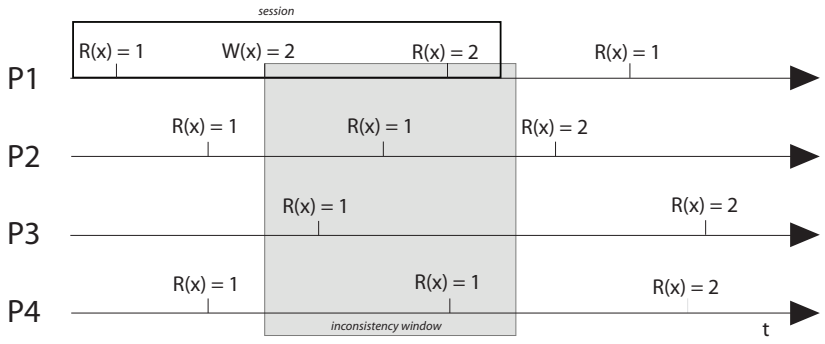


Abbildung 2.7: Session consistency.

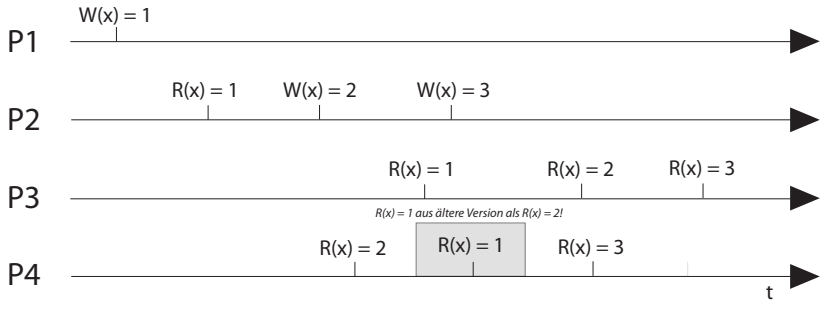


Abbildung 2.8: Monotonic read consistency.

2.2 MapReduce Framework

Das MapReduce Framework, entwickelt von den Google-Mitarbeitern Jeffrey Dean und Sanjay Ghemawat, wird für die nebenläufige Verarbeitung großer Datenmengen, über mehrere Computercluster hinweg, verwendet. Inspiriert wurden die Entwickler dieses Frameworks vom Map und Reduce Algorithmus der funktionellen Programmiersprache LISP[1].

2.2.1 Funktionsweise

Der Anwender selbst erstellt eine Map-Funktion, welcher ein Schlüssel/Wert-Paar übergeben wird. Diese Funktion bildet daraus eine Liste von Zwischenwerten in Form von neuen Schlüssel/Wert-Paaren. Die MapReduce Bibliothek gruppiert alle Werte, welche denselben Schlüssel besitzen, und gibt diese inklusive Schlüssel an die Reduce Funktion weiter. Der Reduce Funktion, welche ebenfalls vom Anwender geschrieben wird, fügt alle Werte dieses

[1]List Processing

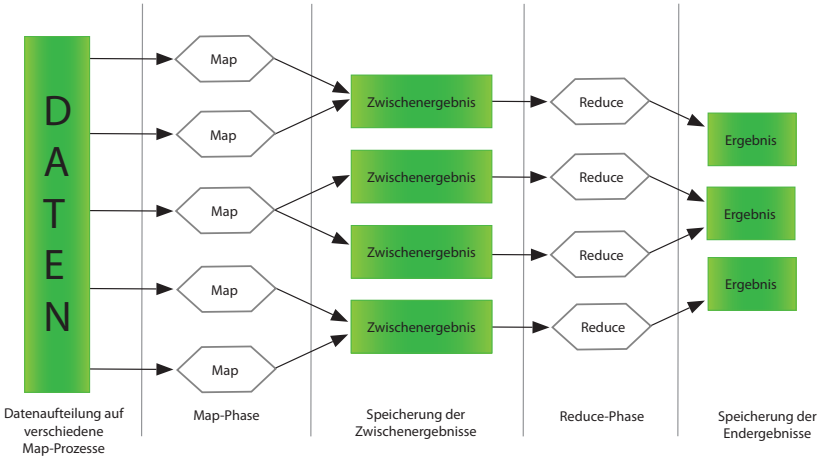


Abbildung 2.9: MapReduce Datenfluss.

Schlüssels zusammen, um eine möglichst kleine Liste von Werten oder auch nur einen endgültigen Wert zu erhalten [8]. Abbildung 2.9 aus [11, S. 18] gibt eine Illustration dieses Datenflusses wieder.

Das im Paper von Jeffrey Dean und Sanjay Ghemawat [8] genannte Beispiel ist eine Suche und Analyse der Worthäufigkeit in einer großen Sammlung von Dokumenten:

```
1  map(String key, String value):
2    // key: document name
3    // value: document contents
4    for each word w in value:
5    EmitIntermediate(w, "1");
6
7  reduce(String key, Iterator values):
8    // key: a word
9    // values: a list of counts
10   int result = 0;
11   for each v in values:
12   result += ParseInt(v);
13   Emit(AsString(result));
```

Der Map-Funktion wird hierbei als Schlüssel der Dokumentname und als Wert der Inhalt des Dokumentes übergeben. Nun wird für jedes Wort w ein Zwischenergebnis gespeichert. Weiters werden, nach der Wertegruppierung mit Hilfe der MapReduce Bibliothek, in der Reduce-Funktion alle Vorkommen eines einzelnen Wortes summiert.

2.2.2 Implementierung

Das MapReduce Framework kann verschieden implementiert werden, wobei die richtige Entscheidung die Anwendungsumgebung bestimmt. Eine Imple-

mentierung ist für ein kleines shared-memory System optimiert, eine andere für einen großen NUMA[2] Multiprozessor oder für große verteilte Netzwerke. Das Google MapReduce Framework wurde für ein großes Ethernet-Netzwerk mit handelsüblichen, auf Linux basierenden, PC's implementiert [8].

Die Datenspeicherung erfolgt auf verteilter Standardhardware mittels Googles eigenem Dateisystem GFS[3], welches für die Verarbeitung großer Datenmengen optimiert ist [11, S. 20].

2.2.3 Vorteile

Durch die Einfachheit dieses Frameworks kann es ebenso von Entwicklern verwendet werden, welche wenig Hintergrundwissen in Bezug auf Parallelsierung und verteilte Systeme haben. Parallelisierung, Ausfalltoleranz, Datentransfer und Load-Balancing laufen im Framework ab, ohne dass der Entwickler eingreifen muss [8].

Das MapReduce Framwork ist ein elementarer Bestandteil vieler NoSQL Datenbanken. Es ermöglicht eine effiziente Suche in großen verteilten Datenmengen bei paralleler Ausführung [11, S. 28].

2.3 Replikation

Verfügt eine Datenbank über eine große Menge an Daten, auf welche standortunabhängig ohne eine größere Verzögerung zugegriffen werden soll und von welcher ebenfalls eine aktuelle Sicherung bestehen muss, empfiehlt sich eine Replikation der Datenbank.

Eine Replikation funktioniert nach dem „Master-Slave"-Prinzip, wobei der Hauptdatenbankserver den „Master" bildet und alle übrigen Datenbankserver mit Kopien dieser Daten als „Slaves" fungieren. Im Master werden alle Abfragen in einem Binärlog gespeichert, welcher von den Slaves ausgelesen und auf ihre eigenen Kopien ausgeführt wird.

Eine durchaus gängige Anwendungsmöglichkeit von Replikation ist die Aufteilung der Slaves über eine große geographische Fläche, um die Zugriffszeit der User zu senken. Diese greifen nun automatisch auf den nächsten Slave-Datenbankserver in ihrer Umgebung zu und nicht auf den Hauptdatenbankserver, welcher möglicherweise auf der anderen Erdhalbkugel steht. Zudem gewährleisten verschieden Standorte der Datenbankserver eine hohe Sicherheit der Daten. Falls ein Server aufgrund höherer Gewalt zerstört wird, sind auf den verbleibenden Servern immer noch Kopien dieser Daten verfügbar.

Weiters wird durch eine Replikation ebenfalls das Load-Balancing optimiert, welches dafür sorgt, dass die Zugriffslast auf den einzelnen Servern

[2] Non-Uniform Memory Access
[3] Google File System

ausgeglichen bleibt. So kann es vorkommen, dass ein User aus Europa auf einen Datenbankserver mit Standort in Nordafrika zugreift, da der geographisch nähere Server in Europa höher ausgelastet ist.

Andererseits wird für eine Replikation weitaus mehr Speicherplatz sowie auch dementsprechend mehr Hardware benötigt, was sich wiederum auf die finanziellen Kosten auswirkt. Im Vergleich zu Leseoperationen bedeutet der schreibende Zugriff ebenfalls weitaus mehr Aufwand. Dabei wird zwischen einer *synchronen Replikation* und einer *asynchronen Replikation* unterschieden. Bei einer synchronen Replikation ist ein Update der Daten nur dann erfolgreich abgeschlossen, wenn dieses auch bei den Replikationen durchgeführt wurde. Im Gegensatz dazu liegt bei der asynchronen Replikation zwischen dem Update auf dem Master und dem auf seinen Slaves eine Latenzzeit. Diese ist meist so lange, bis der Master alle Queries für die Schreiboperationen in die von den Slaves auszulesenden Binärlogs geschrieben hat.

Bei der asynchronen Replikation sind also die Daten immer nur zum Zeitpunkt der Replikation ident, jedoch im weiteren Verlauf nie konsistent. Bei einer synchronen Replikation sind die Daten zwar immer konsistent, jedoch bis zum Abschluss der Replikation nicht verfügbar.

2.4 Sharding

Als Sharding wird die horizontale Skalierung einer Datenbank bezeichnet. Die erste Normalform wird also aufgehoben – einzelne Datenbanktabellen werden denormalisiert.

Wird die Größe der Datenmenge innerhalb einer Datenbank im Laufe der Zeit zu groß für die Speicherung auf einem einzelnen Server, oder soll die Performance gesteigert werden, können diese verteilt über ein beliebig großes Server-Cluster gespeichert werden. Zudem wird die Gesamtanzahl an Spalten in jeder einzelnen Tabelle reduziert und somit auch die Größe des Indizes vermindert, was folglich die Suchperformance steigert.

Eine weiterer Vorteil für Sharding findet sich bei großen Web-Services, welche weltweit fungieren, wieder. Dabei greifen User z.B. mit dem Sitz in Asien auf einen replizierten Datenbankserver mit Sharding in ihrer Nähe zu und nicht auf einen Server aus dem Land des Web-Service-Betreibers. Infolge dessen können die Daten schon im Vorhinein gezielter auf die User in den bestimmten Regionen abgestimmt und in einem solchen Shard gespeichert werden.

Eine *Partitionierung* stellt hier einen Spezialfall des Shardings dar. Dabei wird die Datenbank zwar ebenfalls in einzelne Partitionen aufgeteilt, liegt aber im Gesamten auf einem Server. Durch eine richtige Partitionierung der Datenbank werden Lese- sowie auch Schreiboperationen beschleunigt. Im Falle einer fehlerhaften Partition, kann diese, ohne den Zugriff von Prozessen auf andere Partitionen einzuschränken, aus dem Backup wieder hergestellt

werden.

2.5 MVCC

Multiversion Concurrency Control ist eine, besonders bei NoSQL-Datenbanken und manchen relationalen Datenbanken, immer weiter verbreitete Art mehrere parallele Zugriffe auf einen Datensatz zu organisieren, ohne dass dieser blockiert werden muss. Dieses Verfahren wurde schon 1978 in einer Dissertation von David Reed [37] vorgestellt.

Die klassische Version, das sogenannte *Locking*, sperrt den jeweiligen Datensatz auf den geschrieben werden soll und gibt diesen erst nach Abschluss dieses Vorgangs wieder für Lese- und Schreiboperationen anderer Anwender frei. Diese Sperre wird *„Write-Lock“* genannt. Wird vom Datensatz gelesen, kommt der *„Read-Lock“* zum Einsatz. Der Datensatz wird somit für Schreiboperationen gesperrt, ist jedoch für Leseoperationen aller Anwender offen, wie Abbildung 2.10 visualisiert [11, S. 41].

Wird eine Sperre jedoch nicht aufgehoben, kann der Fall eintreten, dass andere Prozesse unendlich lange auf die Freigabe des Datensatzes warten und daher „verhungern“ (engl. „starving“). Weiters kann eine solche Sperre ebenfalls Deadlocks verursachen. Hierbei warten zwei Prozesse auf die Freigabe des gesperrten Datensatzes des jeweiligen anderen. Diese Probleme können jedoch unter anderem durch einen Abbruch des Prozesses mittels eines gesetzten Timers beseitigt werden. Nach Abbruch startet der Prozess erneut, mit der Hoffnung bei diesem Anlauf nicht zu „verhungern“ oder in einen Deadlock zu geraten [11, S. 41].

Ein hierbei entstehendes gravierendes Problem ist jedoch, dass teilweise längere Lesesperren auftreten können. Ebenso der zusätzliche Traffic für die Kommunikation unter den Prozessen kann besonders in verteilten Datenbank-Clustern sehr teuer und ineffizient werden [3].

MVCC löst diese Problematiken, indem es Datensätze nicht mehr sperrt, sondern mit Versionierung arbeitet. Mit einer Änderung eines Datensatzes wird eine neue Version sowie eine zugehörige eindeutige Identifikationsnummer und ein Zeitstempel erzeugt. Somit werden Leseoperationen auf den Datensatz nicht mehr blockiert, sondern bekommen die Letztversion des Datensatzes vor einer eventuell gerade stattfindenden Schreiboperation zurückgeliefert (s. Abbildung 2.11).

Würde der Fall eintreten, dass ein Prozess auf einen Datensatz schreibt, welcher im selben Moment auch von einem anderen Prozess beschrieben wird, wird dieser Konflikt bei der Versionskontrolle am Ende des Schreibvorganges erkannt und der gescheiterte Schreibvorgang rückgängig gemacht [11, S. 41](s. Abbildung 2.12).

Dieses System hat jedoch nicht nur Vorteile. Durch das Zustandekommen vieler verschiedener Versionen, welche für eine längere Zeit in der Daten-

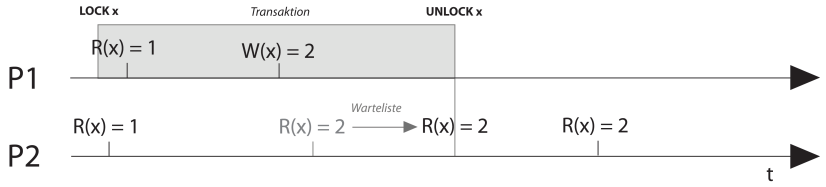


Abbildung 2.10: Klassischer Locking-Mechanismus.

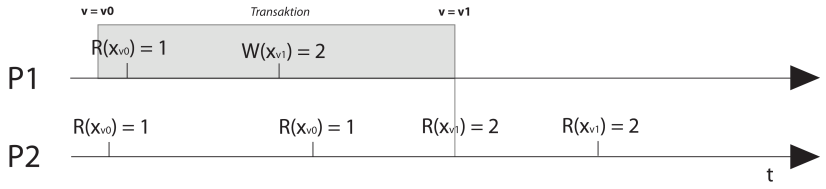


Abbildung 2.11: Versionierung beim MVCC Verfahren.

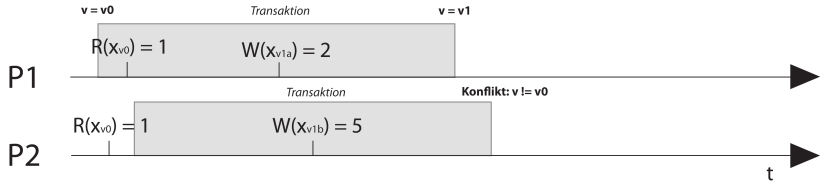


Abbildung 2.12: Konflikterkennung beim MVCC Verfahren.

bank bestehen bleiben, steigen die Speicher- und somit auch die finanziellen Kosten. Im Gegenzug dazu werden Leseoperationen nie blockiert. Dies ist besonders bei einer ausgelasteten Datenbank sehr wichtig[4].

2.6 Vektor Clocks

Da in verteilten Datenbankmanagementsystemen mehrere Instanzen Schreiboperationen durchführen können, benötigt es eine nachträgliche Synchronisation der Daten sowie eine Ordnung der nacheinander folgenden Operationen [11, S. 43]. Diesem Versionierungs-Problem können, neben dem MVCC-Verfahren, auch Vektor Clocks entgegenwirken. Sie zählen ebenfalls zu den Grundbausteinen vieler NoSQL-Datenbanksystemen wie Amazon's Dynamo [9] oder Riak[5].

Jede Datenbankinstanz erhält hierbei einen Zähler (*Clock*), welcher beim Sender, oder auch beim Empfänger erhöht wird. Diese Instanz speichert zu-

[4] http://de.wikipedia.org/wiki/Multiversion_Concurrency_Control
[5] http://wiki.basho.com

dem die Quellinstanz des soeben gesendeten Objektes sowie auch dessen jeweiligen aktuellen Zähler. Mittels dieser Werte erstellt jede einzelne Instanz einen *Vector* mit dem Typ *Sender x Zähler* [11, S. 43].

Vector Clocks verfolgen das Ziel, dem Client die Erkennung verschiedener Versionen zu ermöglichen, der diese in weiterer Folge ordnet und zwischen den Versionen entscheiden kann. Dabei ist der Zähler von hoher Bedeutung, da aus diesem gelesen werden kann, welches Objekt von welchem anderen kausal abhängig ist [11, S. 44]. Wie in [44] definiert kann nun folgendes behauptet werden:

$VC(x)$ definiert eine Vector Clock eines Events x. $VC(x)_z$ steht für die Komponente dieser Clock für Prozess z. $VC(x)$ ist die Ursache von $VC(y)$, wenn und nur wirklich wenn $VC(x)_z$ kleiner oder gleich $VC(y)_z$ für alle Prozesse z gilt und zumindest eine dieser Beziehungen strikt gleich ist. Diese strikte Gleichheit ergibt sich aus $VC(x) \neq VC(y)$. Daraus ergibt sich, wie in [44] definiert, folgende Formel:

$$VC(x) < VC(y) \iff \forall z \left[VC(x)_z \leq VC(y)_z\right] \wedge \exists z' \left[VC(x)_{z'} < VC(y)_{z'}\right]$$

Diese Ursachen-Relation kann ebenfalls folgendermaßen geschrieben werden, wobei hier x die Ursache von y ist und somit $VC(x) < VC(y)$:

$$x \Rightarrow y$$

Jedoch kann auch der Fall eintreten, dass keine Ursachen-Relation vorhanden ist. Somit sind beide Prozesse nebenläufig und werden folgendermaßen geschrieben[6]: $x \parallel y$.

An folgendem vereinfachten Beispiel aus Riak [14] in Kombination mit Abbildung 2.13 wird dieses Senden und Empfangen von Daten anhand von Personen als Instanzen demonstriert:

Anna, Ben, Karin und David planen ein gemeinsames Mittagessen. Anna schlägt Mittwoch vor und teilt dies allen mit:

```
1 date = Mittwoch
2 vclock = Anna:1
```

David wiederum entscheidet sich mit Ben gemeinsam für Dienstag. David bringt diesen Vorschlag ein:

```
1 date = Dienstag
2 vclock = Anna:1, David:1
```

Da Ben Dienstag zustimmt, stellt er ebenfalls seinen Zähler auf 1, da diese Version die erste ist, welche er liest.

```
1 date = Dienstag
2 vclock = Anna:1, David:1, Ben:1
```

[6] http://de.wikipedia.org/wiki/Vektoruhr

Karin wird von dem Vorschlag von David für Dienstag nicht verständigt. Sie weiß nur von Annas Vorschlag für Mittwoch, stimmt aber selbst für Donnerstag und sendet diesen Vorschlag an Ben zurück.

```
1 date = Donnerstag
2 vclock = Anna:1, Karin:1
```

Ben steht nun im Konflikt mit zwei verschiedenen Versionen an Wochentagen. Dies schließt er daraus, dass weder alle Werte aus $vclockA$ höher als die Werte aus $vclockB$ sind und umgekehrt, also $vclockA(Karin) < vclockB(Karin)$ sowie $vclockB(David, Ben) < vclockA(David, Ben)$. Er kann nicht unterscheiden, welche Version aktueller ist. Anzumerken ist hierbei, dass alle in den vclocks nicht erscheinenden Personen den Zähler 0 erhalten, da sie diese Version nicht kennen.

```
1 date = Dienstag
2 vclockA = Anna:1, David:1, Ben:1, Karin:0
3
4 date = Donnerstag
5 vclockB = Anna:1, Karin:1, David:0, Ben:0
```

Ben trifft nun seine eigene Entscheidung für Donnerstag. Da er den Zähler-Status von David ebenfalls weiß, fügt er diesen zu seiner Wahl hinzu und sendet seine Entscheidung zurück an Karin.

```
1 date = Donnerstag
2 vclock = Anna:1, Karin:1, David:1, Ben:2
```

Sobald Anna von David und Karin eine endgültige Entscheidung verlangt, bekommt sie von David folgende Daten.

```
1 date = Dienstag
2 vclock = Anna:1, David:1, Ben:1
```

Karin sendet ihr eine andere Version zurück.

```
1 date = Donnerstag
2 vclock = Anna:1, Karin:1, David:1, Ben:2
```

Daraus stellt Anna fest, dass Ben seine vorher getroffene Entscheidung mit David in Korrespondenz mit Karin geändert hat. Anna zeigt im weiteren Schritt David Karins Version der Vector Clock. David weiß nun, dass er überstimmt wurde und seine Version veraltet ist.

2.7 Consistent-Hashing

Dieses Hashing-Verfahren wird bei vielen NoSQL-Datenbanken, wie bei Amazon's Dynamo [9], für die Replikation und Partitionierung von Datenbankservern eingesetzt.

Zwischen einem Client und einem Server werden oftmals Caching-Server verwendet, um die Last vom Hauptserver zu nehmen. Mittels eines Hashing-Verfahrens werden dabei Objekte einem bestimmten Speicherort eines Caches zugewiesen. Der Client erkennt diesen zugewiesenen Speicherort mittels

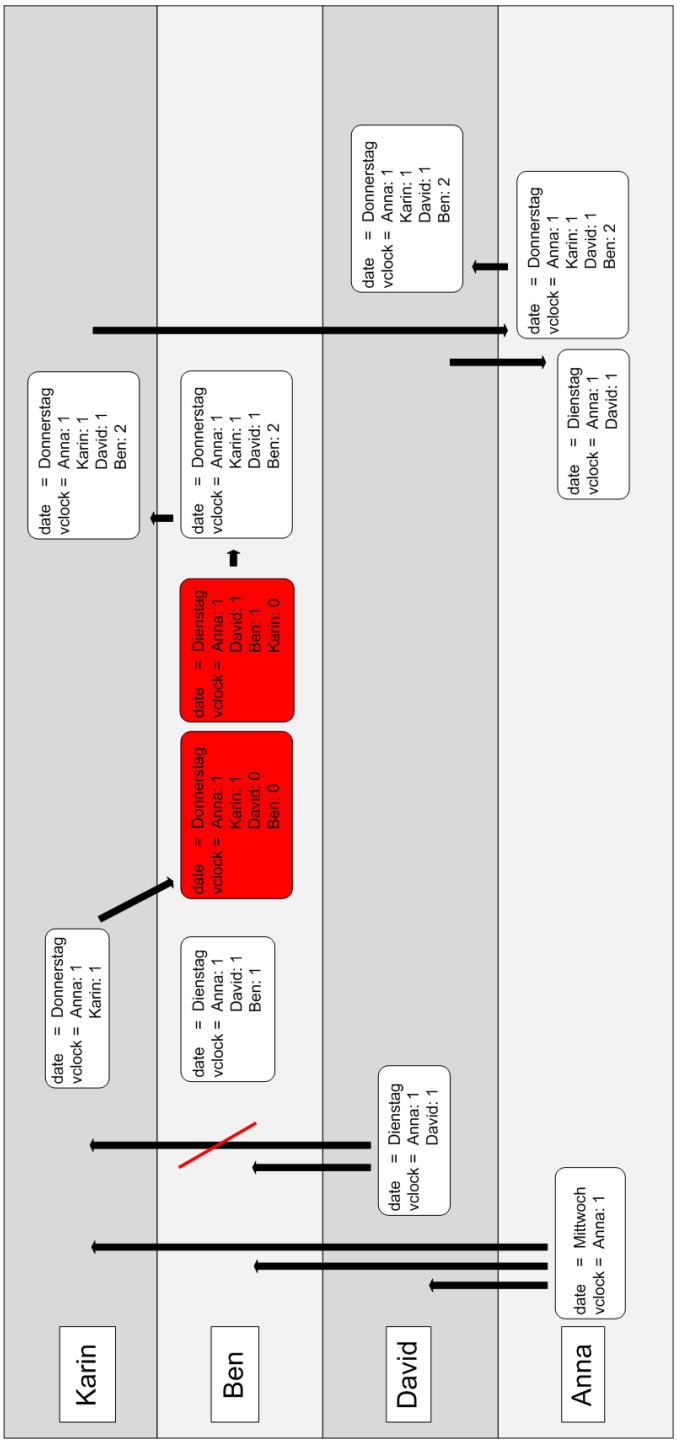


Abbildung 2.13: Einsatz von Vector Clocks über mehrere Instanzen.

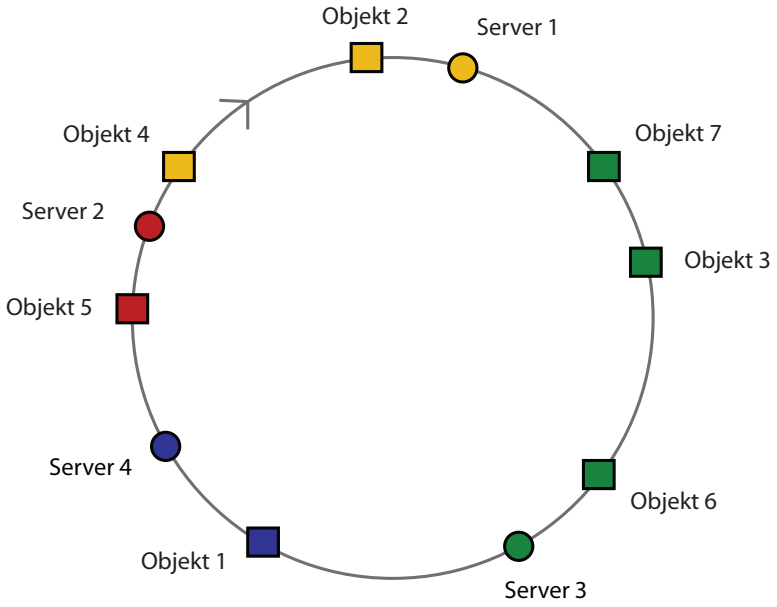


Abbildung 2.14: Aufteilung der Objekte und Server auf einen Adress-Ring.

dem selben Hashing-Verfahren [20]. Der Adressraum einer Hash-Funktion kann dabei als ein Ring mit den Werten $0 - x$ gesehen werden, welcher im Uhrzeigersinn gelesen wird (s. Abbildung 2.14). Die Caching-Server werden ebenfalls wie die Objekte anhand ihres Hashwertes an eine Adresse gesetzt [21].

Würde sich die Anzahl der Caching-Server im Laufe der Zeit ändern, ändern sich somit auch die Bereiche einer Hash-Funktion und nahezu jedem Objekt wird ein neuer Speicherort auf einem Caching-Server zugewiesen – der Cache wird für den Client nutzlos [20]. In einem verteilten Datenbanksystem mit einer großen Datenmenge ist diese dynamische Anpassung nahezu nicht vertretbar, da die Transfergeschwindigkeit der Daten vergleichsweise langsam ist [11, S. 37].

Consistent-Hashing steuert diesem Problem entgegen, indem es bei der Entfernung eines Caching-Servers nur die darauf gespeicherten Objekte an den im Uhrzeigersinn darauf folgenden Caching-Server weitergibt, wie Abbildung 2.15 illustriert. Wird ein neuer Caching-Server dem System hinzugefügt, übernimmt er alle Objekte, welche zwischen ihm und seinem gegen den Uhrzeigersinn liegenden Vorgänger gespeichert sind [11, S. 38] (s. Abbildung 2.16).

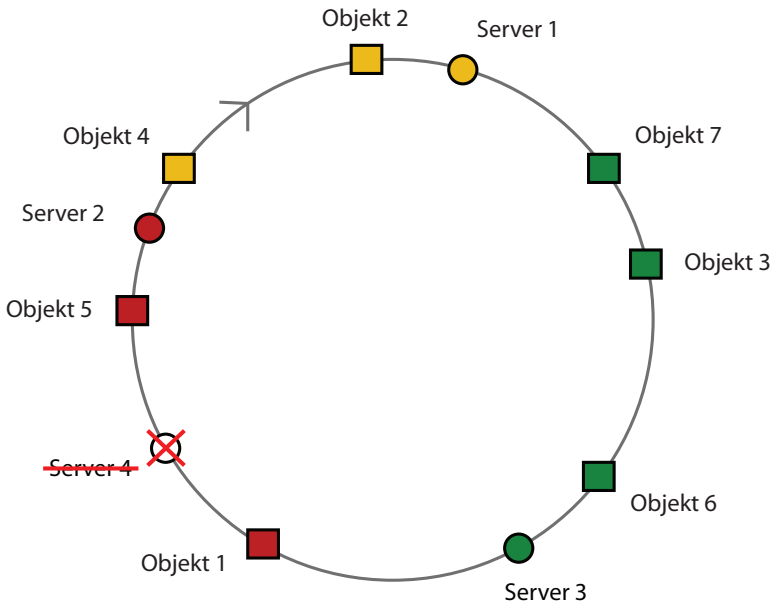


Abbildung 2.15: Entfernung eines Caching-Servers.

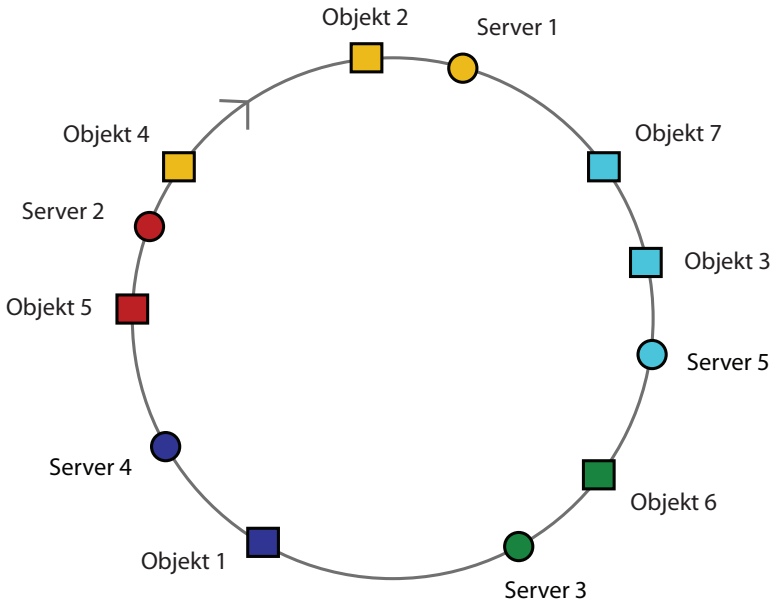


Abbildung 2.16: Hinzufügung eines Caching-Servers.

Kapitel 3

EAV-Modell

Das EAV-Modell bietet eine Alternative zu einem relationalen Speichermodell, wobei es aber ebenfalls innerhalb eines relationalen Datenbankmanagementsystem zum Einsatz kommt. Der vermutlich bekannteste Vertreter mit dem Einsatz dieses Modells ist die Open-Source-E-Commerce-Plattform Magento[1]. Prinzipiell wird es eingesetzt, wenn die Anzahl der Attribute zur Beschreibung des Objektes im Vorfeld nicht bestimmt werden kann [15].

EAV[2] steht für **E**ntity - **A**ttribute - **V**alue:

Entity – ein Objekt, welchem Attribute zugeordnet werden

Attribute – der Name eines Attributes

Value – der Wert des jeweiligen Attributes

Abbildung 3.1 veranschaulicht in einfacher Form die Umsetzung des EAV-Modells im Vergleich zu einem relationalen Modell. Bei diesem sehr einfachen Modell tritt jedoch das Problem auf, dass der Value-Spalte keine expliziter Datentyp zugeordnet werden kann. Zudem ist die Integrität nicht gewährleistet. Um diese Probleme zu beseitigen, wird für jeden einzelnen Datentyp eine eigene Value-Tabelle erstellt. Die Entities sowie die Attributes werden

[1] http://www.magentocommerce.com/de/

[2] http://en.wikipedia.org/wiki/Entity-attribute-value_model

Relationales Modell

person_id	firstname	lastname	age
123	Max	Muster	29
156	Moritz	Muster	15

EAV Modell

person_id	attr_name	attr_value
123	firstname	Max
123	lastname	Muster
123	age	29
156	firstname	Moritz
156	lastname	Muster
156	age	15

Abbildung 3.1: Aufbau des EAV-Modells im Vergleich zu RDBMS.

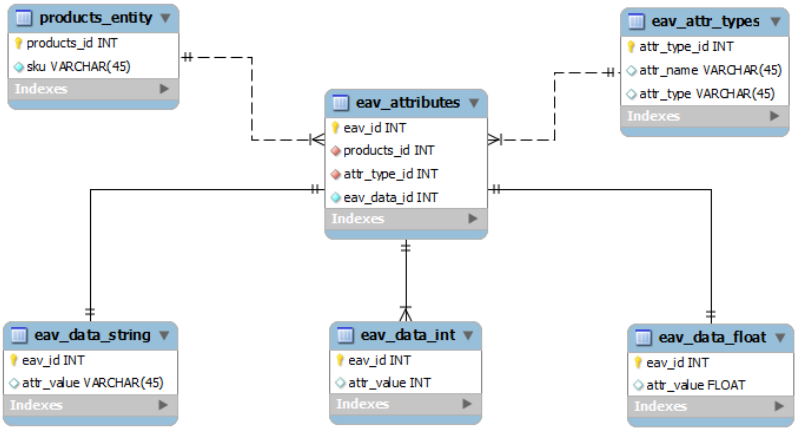


Abbildung 3.2: EAV-Modell.

ebenfalls in eigene Tabellen gespeichert. Zudem können eigene Attribute-Sets bestimmt werden, [40] wie Abbildung 3.2 veranschaulicht.

3.1 Vorteile

Der stärkste Vorteil dieser Modellierung liegt in der hohen Flexibilität bei der Gestaltung von Objektstrukturen. Wird in einem relationalen Modell einem Objekt bei einem schon bestehenden Schemata eine neue Eigenschaft zugewiesen, so muss mittels ALTER TABLE die gesamte Tabelle überarbeitet werden. Mit dem EAV-Modell genügen einfache INSERT-Statements. Da der User selbst alle Attribute im Backend verwalten kann, wird zudem keine zusätzliche Programmierung benötigt [35].

3.2 Nachteile

Die größte Schwäche dieses Modells ist die Performance. Durch komplexere SQL-Queries, welche gezwungenermaßen mehrere JOIN-Anweisungen beinhalten, sinkt die Performance beim Laden mehrere Attributwerte beträchtlich. Diesem Problem kann teilweise durch die Verwendung von Query Caching und Indizes entgegengewirkt werden, obwohl beim Indizieren Vorsicht geboten ist, da die Daten somit doppelt in der Datenbank gespeichert werden (in der Tabelle und im Index). Dies ist vergleichbar mit der Indizierung aller Spalten in einer Tabelle [40]. Zudem erfordern komplexere SQL-Queries wiederum eine komplexere Programmierung dieser [35].

Kapitel 4

NoSQL-Datenbanken

In den letzten Jahren sind eine Vielzahl verschiedenster NoSQL-Datenbanken entwickelt worden, wobei jede einzelne Variante für bestimmte Anwendungsbereiche optimiert wurde. Generell wird zwischen vier großen Hauptgruppen unterschieden:

- Dokumentorientierte Datenbanken
- Key/Value Datenbanken
- Spaltenorientierte Datenbanken
- Graphdatenbanken

Dieses Kapitel gibt einen Überblick über die Vor- sowie auch Nachteile von NoSQL-Datenbanken gegenüber relationalen Datenbanken und warum sie sich in den letzten Jahren immer mehr durchsetzen konnten. Weiters wird genauer auf die Unterschiede in den soeben genannten Hauptgruppen eingegangen sowie auch einige der bekanntesten Vertreter vorgestellt.

4.1 Allgemeines

Da die Größe der Datenmenge, besonders durch Web 2.0 Dienste und nicht zuletzt dem Cloud Computing, in den letzten Jahren rasant gestiegen ist, stießen relationale Datenbankmanagementsysteme schnell an ihre Grenzen hinsichtlich der Verarbeitung, Flexibilität, Skalierung und Replikation dieser Daten.

Dadurch wurden alternative Speichersysteme entwickelt, welche an genau diesen Problemen ansetzen sollten. Der Italiener Carlo Strozzi prägte den Begriff „NoSQL“ und entwickelte dazu 1998 die erste Open-Source-Datenbank ohne einer SQL API [39].

Der Begriff „NoSQL“ entstand jedoch im Jahr 2009 bei einem Treffen mit dem Thema „distributed structured data storage“ neu und steht von da an als Überbegriff für alle nicht relationale, verteilte Datenbankmanagementsysteme [13]. Im Vergleich zu Strozzi's Ansicht, gänzlich ohne SQL zu arbeiten,

steht dieser neu geformte Begriff NoSQL für „Not only SQL“[1], SQL fließt in diese Datenspeichersysteme also zum Teil sehr wohl auch mit ein.

Streng genommen gibt es NoSQL-Datenbanken jedoch schon weitaus länger. Lotus Notes[2] kann schon seit den 1980er Jahren in seiner dokumentorientierten Form als eine der ersten NoSQL-Systeme gezählt werden. BerkeleyDB[3] verfolgte ebenso schon früh den Ansatz von nicht relationalen Systemen. Doch erst mit dem rasanten Wachstum der Datenmenge, in Folge der Web 2.0-Bewegung, setzten sich diese Systeme immer mehr durch [11, S. 1].

4.2 Probleme mit RDBMS und deren Lösungen im NoSQL Bereich

4.2.1 Flexibilität der Datenstruktur

Relationale Datenbankmanagementsysteme kämpfen immer mehr mit fehlender Flexibilität aufgrund strikter Schemavorgaben in Tabellen. Vor einigen Jahren mussten Anwendungen nicht unbedingt flexibel hinsichtlich ihrer Datenstruktur sein, Web 2.0-Portale jedoch sehr. Zwar kann die Struktur relationaler Tabellen mittels ALTER TABLE auch im Nachhinein geändert werden, aber die betroffene Tabelle wird somit für die komplette Dauer dieses Vorgangs gesperrt. Bei einer großen Datenmenge von einigen Gigabyte bis Tera-, oder sogar Petabyte, kann dies mehrere Stunden in Kauf nehmen. Infolge dessen wäre das Portal in dieser Zeit nicht erreichbar [11, S. 4].

Viele NoSQL-Systeme setzen dabei auf die Versionierung von Daten, wie in Abschnitt 2.5 beschrieben. Dabei entsteht nur ein minimales „inconsistency-window“, in welchem die Daten nicht konsistent sind und eventuell ältere Daten ohne die neu hinzugefügten, oder mit bereits gelöschten oder abgeänderten Felder zurückgeliefert werden [11, S. 4].

4.2.2 Replikation

Replikationen lassen sich in RDBMS ebenfalls meist nur sehr umständlich umsetzen. Die meisten NoSQL-Systeme sind in Hinsicht auf dieses Problem so einfach konzipiert, dass sie eine Repliaktion oft nur mit einem einzigen Kommando durchführen können (s. CouchDB oder Redis) [11, S. 4].

4.2.3 Skalierung

Als die Menge der Daten im Vergleich zu heute noch nicht so rasant stieg, wurde auf die *vertikale Skalierung* gesetzt, falls der Datenbankserver diese

[1] http://de.wikipedia.org/wiki/NoSQL_(Konzept)
[2] http://www-01.ibm.com/software/at/lotus/
[3] http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html

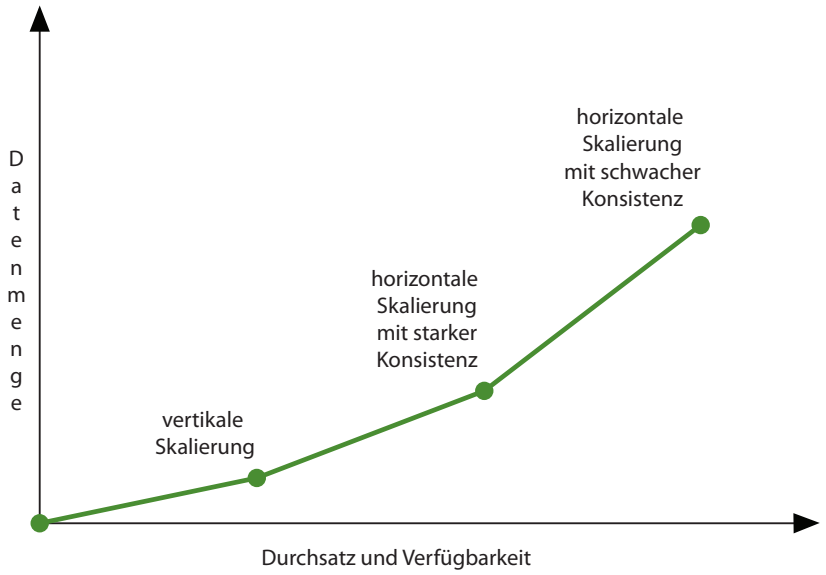


Abbildung 4.1: Auswirkung auf Verfügbarkeit und Durchsatz durch Lockerung der Konsistenz.

Daten nicht mehr performant verarbeiten konnte. Neue teure Hardware wurde angeschafft, um die Leistungsfähigkeit zu steigern. Dies wirkte sich jedoch stark auf die finanziellen Kosten aus.

Die in der heutigen Zeit jedoch häufiger genutzte *horizontale Skalierung* setzt indes auf günstige Standardhardware und verteilt die Datenbank auf einzelne Rechnercluster. Diese Systemarchitektur ist vor allem günstiger und kann zudem in einer dementsprechenden Größe des Clusters auch unendlich viele Daten gleichzeitig verarbeiten.

NoSQL Datenbanken sind meist von Anfang an auf eine verteilte horizontale Skalierbarkeit ausgerichtet. Da laut dem CAP-Theorem Konsistenz und Verfügbarkeit nicht gleichzeitig mit Partitionstoleranz erreicht werden können, verzichten sie auf die starke Konsistenz der Daten und lockern diese im Vergleich zum relationalen Ansatz. Somit erreichen NoSQL-Datenbanken im Schnitt einen höheren Durchsatz der Daten, gekoppelt mit einer ebenfalls höheren Verfügbarkeit bei einer steigenden Datenmenge, wie Abbildung 4.1 als Statistik demonstriert.

Relationale Datenbanken riskieren jedoch durch den Erhalt der starken Konsistenz der Daten die Verfügbarkeit des Systems.

4.2.4 Nachteile im Vergleich zu RDBMS

Mit den eigenen APIs kann der Anwender, im Gegensatz zur SQL-API, zwar meist noch einfacher auf eine NoSQL-Datenbank zugreifen, aber sie bringen durchschnittlich weniger Funktionalitäten mit, als das schon sehr ausgereifte

SQL. Besonders bei komplexeren Abfragen muss der Anwender einen anderen Weg finden und diese zum Beispiel als Map/Reduce-Abfrage formulieren. Dieser Bereich befindet sich jedoch noch in der Entwicklung, um letztendlich doch auf eine Stufe mit der SQL-API zu gelangen [11, S. 4].

4.2.5 Einsatzgebiete

Je nach Anwendung muss verglichen werden, welches Datenbankmanagementsystem die angeforderten Eigenschaften besitzt. Für transaktionale Finanzsoftware werden immer noch relationale Datenbanksysteme mit ACID-Eigenschaften herangezogen, da dortige Daten zu kritisch sind, um die Konsistenz zu lockern. Bei Banküberweisungen, zum Beispiel, könnte dies schwere Folgen nach sich ziehen. Social Webs hingegen können auf die BASE-Eigenschaften der NoSQL-Datenbanken vertrauen, da es dort wichtiger ist, das System nahezu rund um die Uhr verfügbar zu halten, als dass eine Verlinkung zu einem Bild des Users immer konsistent ist.

Im E-Commerce-Bereich erschließt sich hier eine Mischform. Zwar kann bei der Verwaltung von Produkten und Kundendaten auf eine NoSQL-Datenbank gesetzt werden, ein Bestellvorgang sollte jedoch bevorzugt mittels Transaktionen über eine ACID-fähige Datenbank abgewickelt werden, da möglicherweise Bestellungen und damit eventuell auch Kunden verloren gehen könnten.

4.3 Dokumentorientierte Datenbanken

4.3.1 Allgemein

Mit der Entwicklung von Lotus Notes[4] entstand in den 1980er Jahren auch die erste bekannte dokumentorientierte Datenbank. Wie sich aus dem Namen herleiten lässt, werden in dieser Datenbank unter anderem Dokumente in Form von strukturierten Dateien wie Textverarbeitungsprogrammdateien gespeichert.

Die in der NoSQL-Bewegung entwickelten dokumentorientierten Datenbanken, wie CouchDB[5] von Apache oder MongoDB[6], speichern diese eindeutig identifizierbaren Dokumente in Form des JSON- (CouchDB) bzw. BSON-Formates (binary JSON in MongoDB) ab. In diesen strukturierten Dateien ist es möglich, wiederum schemalose Daten abzuspeichern, welche jeweils aus einem Schlüssel-Wert-Paar bestehen[7]. Der Wert kann hierbei ebenfalls als Schlüssel für einen weiteren Wert dienen. Somit können Daten beliebig verschachtelt werden, wie das angeführte Beispiel eines BSON-Objektes in

[4] http://www-01.ibm.com/software/at/lotus/
[5] http://couchdb.apache.org/
[6] http://www.mongodb.org/
[7] http://de.wikipedia.org/wiki/Dokumentenorientierte_Datenbank

MongoDB mit dem Schlüssel „tutorials“ zeigt.

```
1 {
2    "\_id": ObjectId("4d73971f86b3c91812000000"),
3    "url": "http: \/\/www.dummy.com",
4    "software": "typo3",
5    "tutorials": {
6      "0": "php",
7      "1": "javascript",
8      "2": "java"
9     }
10 }
```

4.3.2 MongoDB

MongoDB zählt derzeit zu einer der bekanntesten und verbreitetsten dokumentorientierten Open-Source Datenbankmanagementsystemen. Das Unternehmen 10gen[8] startete die Entwicklung dieser Datenbank 2007, mit Ende 2009 erschien die erste öffentliche Version [28] unter der GNU AGPL v3.0[9] [4, S. 13]. Auf Wunsch steht ebenfalls eine kommerzielle Version mit Support von 10gen selbst zur Verfügung.

Datenmodell

Eine MongoDB-Datenbank besteht aus mehreren *Collections*, vergleichbar mit Tabellen in relationalen RDBMS. Diese sind schemafrei und beinhalten eine beliebige Anzahl von Dokumenten [7, S. 5].

Dokumente wiederum können mit Zeilen einer Tabelle eines RDBMS verglichen werden. In ihnen werden Key/Value-Paare gespeichert, welche zudem auch ineinander verschachtelt werden können, wie das Beispiel in 4.3 zeigt. Diese Daten liegen im BSON[10]-Format vor. Zu einem Dokument wird ebenfalls eine *Object_ id* gespeichert, um dieses eindeutig identifizieren zu können. Diese Object_id ist ähnlich einem Primary Key in einem RDBMS.

Bei den Key/Value-Paaren ist zu beachten, dass MongoDB Type- sowie auch Case-Sensitive arbeitet [7, S. 6] und es somit einen großen Unterschied macht, ob der Wert als Integer (z.B. „foo“ : 1) oder als String (z.B. „foo“: „1“) gespeichert wird.

Einer der größten Vorteile bei MongoDB liegt in der integrierten Query-Funktion, mit welcher auf einfachste Weise Daten abgefragt, geändert oder gelöscht werden können. Im Vergleich zu anderen bekannten dokumentorientierten Datenbanken wie CouchDB[11] muss bei MongoDB somit nicht immer direkt mit MapReduce gearbeitet werden [4, S. 12].

[8] http://10gen.com
[9] http://www.gnu.org/licenses/agpl-3.0.html
[10] Binary JSON
[11] http://couchdb.apache.org/

Skalierung

MongoDB wurde für eine, für den Anwender einfache, horizontale Skalierung mit integriertem *Balancer* optimiert. Dabei teilt es ein Shard, also die Daten auf einem Knoten im Rechnercluster in einzelne *Chunks* auf, welche verschiedene Bereiche der Daten beinhalten. Einem Chunk wird dabei ein Feld eines Dokumentes, welches in allen aufzuteilenden Dokumenten vorhanden ist, ein sogenannter *Shard Key* zugewiesen. Mittels dieses Keys können die Daten in einzelne Chunk-Bereiche aufgeteilt werden. Diese Aufteilung wird in folgendem, stark vereinfachten Beispiel mit der Bestimmung des Shard Keys auf das Feld „Alter“ und des Wertebereiches eines Chunks von 10–40 demonstriert [6]:

Dokumente in Shard:

```
1 {"`Name"' : "`Max"', "`Alter"' : 30}
2 {"`Name"' : "`Moritz"', "`Alter"' : 14}
3 {"`Name"' : "`Martin"', "`Alter"' : 69}
4 {"`Name"' : "`Maria"', "`Alter"' : 23}
5 {"`Name"' : "`Miriam"', "`Alter"' : 51}
6 {"`Name"' : "`Marta"', "`Alter"' : 78}
```

Dokumente im Chunk 10–40 eines Shards:

```
1 {"`Name"' : "`Max"', "`Alter"' : 30}
2 {"`Name"' : "`Moritz"', "`Alter"' : 14}
3 {"`Name"' : "`Maria"', "`Alter"' : 23}
```

Üblicherweise enthält ein Shard einen gewissen Bereich von Daten. Wird dieses Shard zu groß, lagert es alle überschüssigen Daten auf das nächste Shard aus, dies auf sein folgendes usw.. Ein Kaskadeneffekt entsteht, wie Abbildung 4.2 illustriert. Tritt eine solche Situation öfters ein, entsteht eine enorme Performance-Last im Cluster [6].

Die Aufteilung eines Shards erlaubt der Datenbank, einzelne Chunks bei dessen Überlast einfach und schnell auf ein weniger ausgelastetes Shard zu verschieben (s. Abbildung 4.3). Diese Aufgabe des *Balancing* übernimmt automatisch der Balancer, welcher ebenso auch auf Wunsch deaktiviert werden kann [6].

Wird ein neuer Knoten dem Cluster hinzugefügt, werden einzelne Chunks automatisch auf diesen verlagert, ohne dass der Anwender eingreifen muss. So bleibt das Gleichgewicht der Daten auf den einzelnen Shards bestehen [6].

Replikation

Für Replikationen setzt MongoDB das bekannte Master-Slave-Prinzip ein, wobei alle Slaves eine Kopie des Masters besitzen. Zudem können noch *Replica Sets* (s. Abbildung 4.4) bestimmt werden, welche grundsätzlich auch auf das Master-Slave-Prinzip aufbauen, jedoch den Master selbst ernennen. Fällt der Master aus, wird ein zufällig gewählter Slave automatisch der neue Master. Wird der alte Master wieder im Cluster registriert, übernimmt er

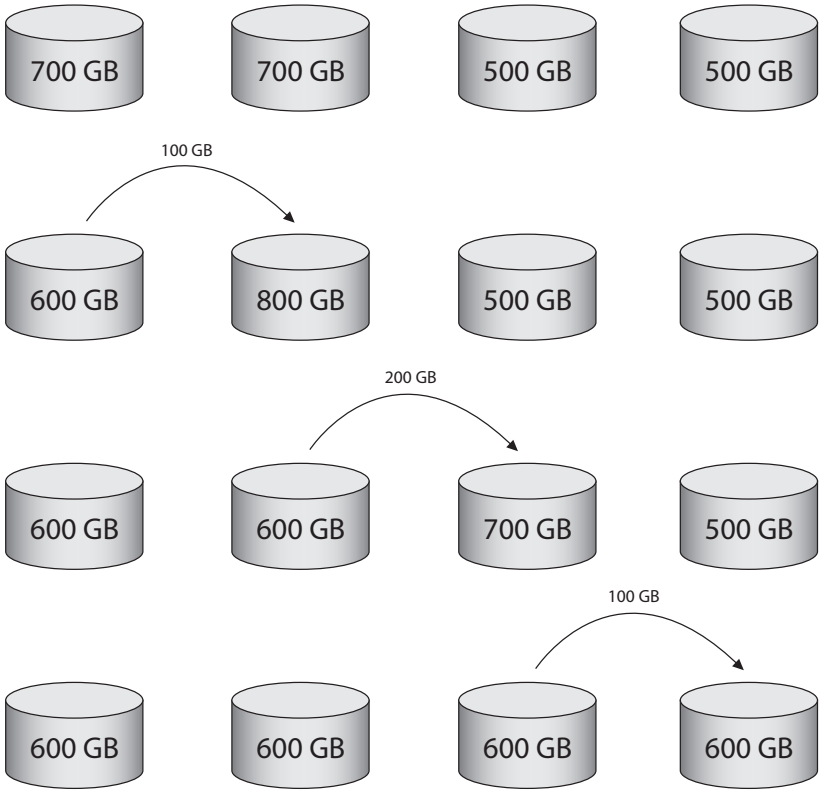


Abbildung 4.2: Einfaches Balancing der Shards mit daraus erfolgendem Kaskadeneffekt.

die Rolle eines Slaves. Dieses voll automatisierte Vorgehen garantiert somit eine hohe Ausfallsicherheit eines Clusters [7, S. 127–130].

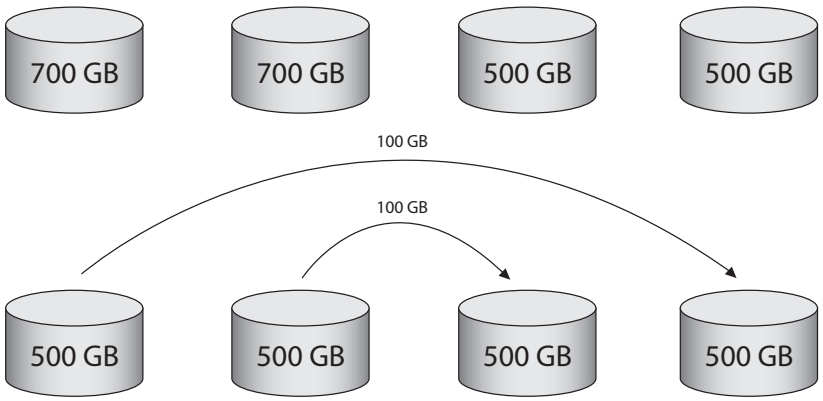


Abbildung 4.3: Balancing in MongoDB mit Einsatz von Chunks.

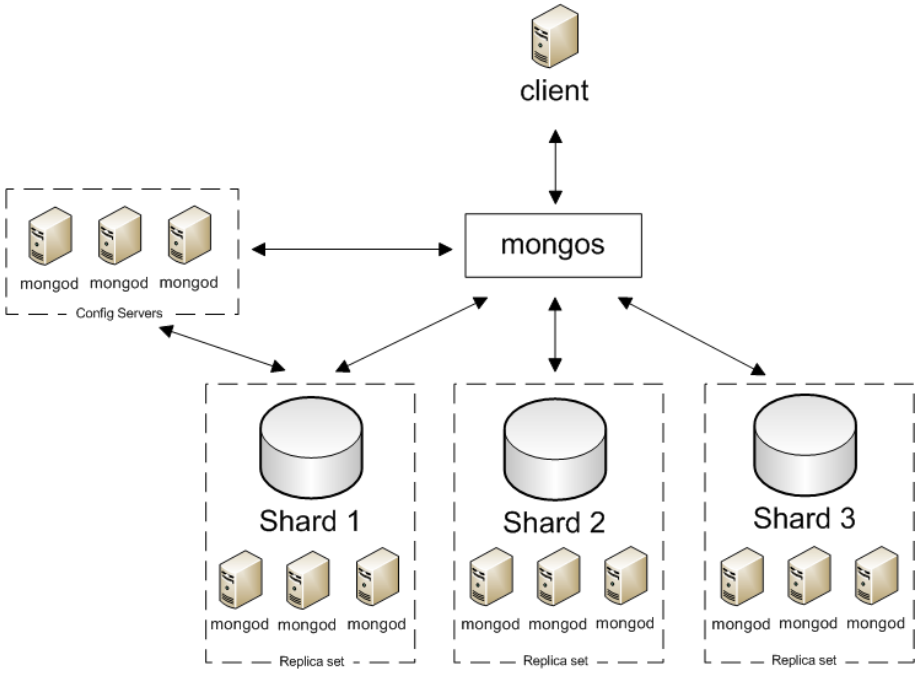


Abbildung 4.4: MongoDB Rechnerarchitektur bei Sharding.

Clusterarchitektur

Abbildung 4.4 visualisiert ein Beispiel eines MongoDB-Clusters. Der Client, welcher Daten abfragen will, wird im ersten Schritt zu *mongos*, einem von MongoDB zur Verfügung gestellten Routing- und Koordinations-Prozess, geleitet. Dadurch erscheinen dem Client alle einzelnen Cluster-Komponenten wie ein gesamtes System. Der Prozess mongos wiederum leitet die eingehenden Requests an die einzelnen Shards weiter und liefert deren Resultate an den Client zurück [27].

Jedes einzelne Shard sowie dessen Replikationen für die Absicherung bei einem Systemausfall führen einen Prozess namens *mongod* aus, welcher als Haupt-Datenbankprozess für MongoDB fungiert [27].

Der *Config Server* beinhaltet Metadaten des Clusters in Bezug auf Basisinformationen der Shards und ihren Chunks. Dieser wird ebenfalls über mehrere Server hinweg repliziert. Diese Replikation scheint jedoch nicht als *replica set* wie in einem Shard auf, sondern bildet ein eigenes Replikationsmodell. Fällt einer der Config-Server aus, kann von den übrigen Config-Server-Replikationen nur mehr gelesen werden. Auf das Cluster selbst können jedoch immer noch Schreib- sowie auch Leseoperationen erfolgen [27].

Performance

Eines der größten Ziele von MongoDB ist eine hohe Performance. Dies erreicht diese Datenbank unter anderem mit der Server-Interaktion über ein *Binary Wire Protocol*[12], mit welchem eigene Client Driver für verschiedene Programmiersprachen geschrieben werden können und einem dynamischen Query-Optimizer, welcher sich die schnellste Art eine Query abzusetzen „merkt“ [7, S. 3].

Zudem schaffen noch folgende Eigenschaften zusätzliche Performance in der Datenbank [26]:

- Memory-Mapped-Files für die Datenspeicherung – dabei wird die Verantwortung des Memory-Managements an das Betriebssystem abgegeben [7, S. 3]
- Collection-orientierter Speicher – allen Objekten einer Collection werden aneinander angrenzenden Speicherplätze zugeteilt, um sie schnell wieder von der Festplatte lesen zu können
- Daten werden durch Überschreibung dieser aktualisiert – Vermeidung von MVCC
- Entwicklung in C++

Konsistenzerhaltung der Daten

MongoDB kann in Bezug auf das CAP-Theorem, wie auch sein „Inspirator“ Google BigTable[13], dem Bereich CP, also der starken Konsistenz sowie Partitionstoleranz, zugeordnet werden.

Mit der Eigenschaft des automatisierten Shardings sowie Replikation besteht immer ein Master für ein Shard, wobei nur auf diesen geschrieben werden kann [23]. MongoDB unterstützt zudem atomare Schreib- bzw. Leseoperationen auf einzelne Dokumente, jedoch nicht auf mehrere Dokumente gleichzeitig. Aus dieser Standard-Einstellung folgt, dass die Daten stark konsistent sind [25].

Eine Partitionstoleranz ist durch die Verwendung von *Replica Sets* ebenfalls gegeben. Dadurch ist diese Datenbank jedoch nicht durchgehend verfügbar.

MongoDB ist in der Form der Konsistenzerhaltung jedoch sehr variabel. Falls gewünscht, können die Konsistenz-Einstellungen manuell abgeändert werden, um eine höhere Verfügbarkeit des Systems zu erreichen [24].

[12] http://www.mongodb.org/display/DOCS/Mongo+Wire+Protocol
[13] http://labs.google.com/papers/bigtable.html

Einsatzgebiete

Bekannte Unternehmen, welche auf MongoDB setzen, sind unter anderem Foursquare[14] mit ihrer Geolocation und Capped Collection, New York Times[15] für einen Formulargenerator für Bilderuploads, ShopWiki[16] für die Speicherung der Produktdaten und Diaspora[17], das dezentrale Pendant zu Facebook[18] [4, S. 14].

MongoDB lässt sich vor allem für mobile und Web-Anwendungen einsetzen, da die unterstützten Libraries für den Zugriff auf die Datenbank vor allem im Web-Bereich liegen. Die Arbeit mit der Datenbank mittels BSON ist ebenfalls bestens für Webanwendungen geeignet [4, S. 17]. Muss die Datenstruktur des Öfteren geändert werden, kann dies in den schemalosen Collections, im Vergleich zu einem RDBMS, schnell und einfach vollzogen werden. Soll die Anwendung ebenfalls einfach skalierbar und performant sein und eine starke Konsistenz der Daten garantieren, ist gegen MongoDB nicht mehr viel einzuwenden.

Diese dokumentorientierte Datenbank bietet weiters die Möglichkeit, Geospatial Indexing zu verwenden, um aus gewonnen Daten Geo-basierte Abfragen, wie etwa eine Umkreissuche, zu implementieren. Dieser Punkt ist vor allem für Anwendungen mit integrierten Geolocation Services von Vorteil [4, S. 17].

MongoDB vs. MySQL

Marc Boeker vergleicht in seinem Buch [4, S. 23] in der übersichtlichen Tabelle 4.1, welche Vorteile oder auch Nachteile MongoDB zu MySQL hat und somit auch, wie ausgereift diese dokumentorientierte Datenbank aktuell ist.

4.4 Key/Value Datenbanken

4.4.1 Allgemein

Key/Value Datenbanken zeichnen sich durch die Abspeicherung von Daten mittels Schlüssel-Wert-Paaren aus. Values können hierbei beliebige Datentypen wie zum Beispiel Arrays, Objekte oder einfache Zeichenketten sein [11, S. 7].

Dadurch sind sie dokumentorientierten Datenbanken sehr ähnlich, welche im Grunde auch auf einfache Schlüssel-Wert-Paare hinunter gebrochen werden können. In der Key/Value Datenbank Riak[21] lässt sich zum Beispiel

[14] https://foursquare.com/
[15] www.nytimes.com/
[16] www.shopwiki.com/
[17] https://joindiaspora.com/
[18] www.facebook.com/
[21] http://wiki.basho.com/

Tabelle 4.1: Aufstellung der Vor- sowie auch Nachteile von MongoDB zu MySQL.

Funktion	MySQL	MongoDB
Open Source Typ	GNU GPL v2.0 + LE	GNU AGPL v3.0
Protokoll der Kommunikationsschicht	TCP/IP	TCP/IP
Datenstruktur	Spaltenorientiert	Dokumentorientiert
Schlüssel	Primary Key & Foreign Keys	ObjectID
Basic-, Unique- und Compound-Indexes	Ja	Ja
Unterstützung von Transaktionen	Ja	Nein, durch atomare Updates gelöst
Stored Procedures	Ja	Ja, durch Ausführung von JavaScript auf Server
Trigger	Ja	Nein
Cursor	Ja	Ja
Views	Ja	Evt. über MapReduce lösbar
Volltextsuche	Ja	Nein
Sharding	Ja, mit Unterstützung externer Hilfen	Ja, Out of the box
Replikation	mittels Master/Slave	mittels Replica Sets (ein Master, n Slaves)
GUI	PHPMyAdmin	RockMongo[19], Fang-of-Mongo[20], ...
Binärdatenspeicherung	BLOB	GridFS oder ByteString
Funktionen in Query Language (STRLEN(),...)	Ja	Nein, mit JS-Workarounds zu lösen
Authentifikation	Ja	Ja
MapReduce Framework	Nein	Ja

zwar ebenso das JSON-Format als Values speichern, ihre API besitzt jedoch im Vergleich zu dokumentorientierten Datenbanken wie MongoDB nur weniger komplexere Abfragemechanismen [10], wodurch diese Datenbank für manche Anforderungen etwas schwieriger zu handhaben ist.

Der Vorteil dieser Datenbanken hingegen liegt unter anderem ebenfalls in der Speicherung von schemalosen Daten sowie in der Skalierbarkeit, Synchronisation und Fehlertoleranz [2].

Key/Value Datenbanken lassen sich weiters in zwei Gruppen unterteilen, die In-Memory- sowie die On-Disk-Varianten. In-Memory-Datenbanken werden oftmals als verteilte Cache-Speichersysteme verwendet, da sie die Daten im Speicher behalten und eine hohe Performance erreichen. On-Disk-Varianten hingegen speichern die Daten auf der Festplatte und werden primär als Datenspeicher genutzt [42].

4.4.2 Amazon Dynamo

Die hoch verfügbare und dezentralisierte Key/Value Datenbank Dynamo von Amazon wurde speziell für Anwendungen des Amazon Web Services[22] entwickelt und ist nicht öffentlich verfügbar. Sie gilt in ihrer verteilten Architektur jedoch als Vorbild vieler nachfolgender Open-Source NoSQL-Systeme [11, S. 259] und ist dahin gehend hoch interessant, da sie in Amazon's E-Commerce Umgebung eine enorme Last an Daten verlässlich verwaltet.

Datenmodell und System Interface

Objekte werden, mit einem eindeutigen Schlüssel verknüpft, über ein einfaches Interface gespeichert. Relationen sind in diesem Sinne nicht vorhanden. Die API stellt für den Zugriff auf die Datenbank folgende zwei Funktionen bereit [9]:

get(key) – liefert das mit dem Schlüssel verknüpfte Objekt, oder eine Liste von Objekten zurück, falls Konflikte mit Objektversionen auftreten.

put(key, context, object) – speichert ein Objekt gemäß seines Schlüssel in der Datenbank. Der Parameter *context* enthält Informationen, wie auch die Version des zu speichernden Objektes.

Skalierung

Eine Partitionierung der Datenbank erfolgt mittels Dynamos eigener Variante des Consistent-Hashings (Abschnitt 2.7). Da auf manche Daten häufiger ein Zugriff stattfindet als auf andere und zudem auch jeder einzelne Rechnerknoten andere Hardwareeigenschaften und somit andere Performance mit sich bringt, benutzt Dynamo *virtuelle Knoten*. Ein virtueller Knoten verhält

[22] http://aws.amazon.com/

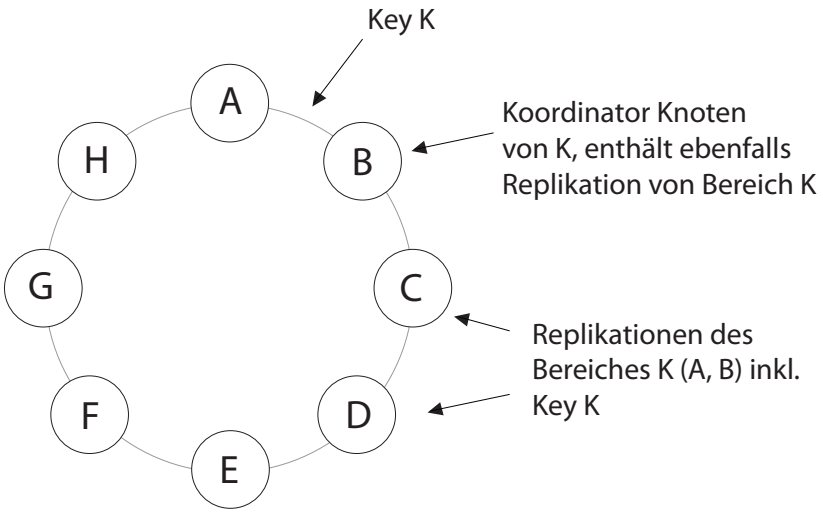


Abbildung 4.5: Partitionierung und Replikation in Dynamo.

sich zwar nach außen wie ein einzelner Rechnerknoten, jedoch kann ein realer Rechnerknoten mehrere virtuelle Knoten verwalten.

Der größte Vorteil virtueller Knoten liegt darin, dass deren Anzahl auf einem realen Knoten abhängig von dessen Hardwareeigenschaften ist. Je leistungsstärker ein solch realer Knoten ist, desto mehr virtuelle Knoten kann er verwalten. Dadurch besitzen alle virtuellen Knoten eine ähnliche Rechenleistung [9].

Replikation

Um eine hohe Verfügbarkeit sowie eine Beständigkeit des Datenbanksystems zu gewährleisten, repliziert Dynamo seine Daten auf mehrere Rechnerknoten.

Dabei werden einem Schlüssel K N Replikas zugeteilt. Der Schlüssel K ist weiters einem Koordinator-Knoten zugewiesen. Dieser Koordinator-Knoten ist verantwortlich für die Replizierung der Daten auf alle N Knoten. Zudem repliziert er alle Schlüssel zusätzlich zu der lokalen Speicherung in ihre vordefinierten Regionen auch in den $N - 1$ nachfolgenden Knoten. Daraus ergibt sich ein Cluster in welchem jeder Knoten für eine Region zwischen ihm selbst und seinem Nten Nachfolger verantwortlich ist [9]. Abbildung 4.5 zeigt ein vereinfachtes Beispiel dieses Vorganges.

Konsistenzerhaltung der Daten

Alle Daten in Dynamo sind *eventually consistent*, um eine hohe Verfügbarkeit und Partitionstoleranz zu erreichen. Durch die Lockerung der Konsistenz und der Verwendung von Versionierung können Aktualisierungen der Daten

Tabelle 4.2: Daten in zweidimensionaler Tabellenform.

PersonID	Vorname	Nachname	Alter
1	Max	Mustermann	41
2	Moritz	Musterperson	25
3	Maria	Mustergartner	67

zwischen den Replikationen asynchron erfolgen. Besteht ein Konflikt zwischen mehreren Versionen eines Objektes verwendet Dynamo *Vector Clocks* (Abschnitt 2.6), um diese zu beheben.

Einsatzgebiete

Dynamo wird ausschließlich bei Amazons E-Commerce Plattform eingesetzt. Dabei verwaltet sie Daten der Warenkörbe, Bestsellerlisten und vieles mehr [9].

Sie inspirierte durch ihre Systemarchitektur viele nachfolgende Open-Source NoSQL-Datenbanken wie Apaches Cassandra (Abschnitt 4.5.2), CouchDB[23], Voldemort[24] oder Riak[25] [19, S. 21].

4.5 Spaltenorientierte Datenbanken

4.5.1 Allgemein

Column Datenbanken, also spaltenorientierte Datenbanken, speichern Daten im Vergleich zu einem RDBMS in Spalten über mehrere Tabellen hintereinander und nicht in Zeilen in einer einzigen Tabelle ab. Somit ist die Tabelle einer spaltenorientierten Datenbank im physischen Speicher des Computers nach Spalten gruppiert. Im relationalen Modell würden alle Datenwerte einer Zeile hintereinander gereiht werden[26].

Diese Speicherform wird an einem einfachen Beispiel mit Personendaten in Tabelle 4.2 näher erklärt.

Speicherung der Daten in einem zeilenorientierten Datenbankmodell:

```
1 1, Max, Mustermann, 41; 2, Moritz, Musterperson, 25;
2 3, Maria, Mustergartner, 67;
```

Speicherung der Daten in einer spaltenorientierten Datenbank, in welcher die Spalten gruppiert werden:

[23] http://couchdb.apache.org/
[24] http://project-voldemort.com/
[25] http://wiki.basho.com/
[26] http://de.wikipedia.org/wiki/Spaltenorientierte_Datenbank

```
1 1, 2, 3; Max, Moritz, Maria; Mustermann, Musterperson,
2 Mustergartner; 41, 25, 67;
```

Eine solche spaltenorientierte Datenbank ist vor allem für Data-Warehouse- sowie auch OLAP-Anwendungen[27] geeignet, da sie Vorteile in Bezug auf die Performance von Leseoperationen bietet und nur Attribute ausliest, welche auch wirklich aktuell benötigt werden. Ebenso wird die Datenkompression durch diese physische Speicherung der Daten positiv beeinflusst [1].

Nachteile ergeben sich jedoch bei der Suche, wenn gleichzeitig viele Spalten einer Zeile ausgelesen werden sowie auch beim Einfügen neuer Zeilen, wenn alle Daten dieser Zeile auf einmal vorliegen. Somit sind zum Beispiel für OLTP-Aufgaben[28] zeilenorientierte Datenbanken zu bevorzugen[29].

4.5.2 Apache Cassandra

Die spaltenorientierte Open-Source Datenbank Cassandra[30] wurde 2008 von den Facebook-Entwicklern Avinash Lakshman und Prashant Malik veröffentlicht und Anfang 2009 in die Projekte von Apache aufgenommen. Seither erreichte sie zwar noch nicht die stabile Version 1.0, ist aber bereits jetzt schon sehr verbreitet und wird unter anderem bei Facebook, Twitter und Digg für Teile der Datenverwaltung verwendet [19, S. xvii].

Cassandra besticht durch eine verteilte Architektur, elastische Skalierbarkeit, hohe Vefügbarkeit sowie Fehlertoleranz und stellt hinsichtlich ihre verteilten Architektur und dem Datenmodell eine Mischform aus Amazons Dynamo (s. Abschnitt 4.4.2) und Googles BigTable[31] dar.

Datenmodell

In diesem Datenmodell werden zwar die Nomen Spalten und Zeilen verwendet, diese haben jedoch wenig mit den schon bekannten Spalten und Zeilen in einem RDBMS gemeinsam.

Eine Tabelle bei Cassandra entspricht dabei einer verteilten multidimensionalen Map, welche durch einen eindeutigen Schlüssel (Row Key) indiziert wird [22]. Eine solche Map kann entsprechend ihrer Dimension als *Super Column Family*, *Column Family* oder als *Column* bezeichnet werden. Dabei bauen die Dimensionen, wie in [19, S. 43] erklärt, folgendermaßen aufeinander auf (s. Abbildung 4.6):

Column – entspricht einem Key/Value-Paar, darin sind die Daten als Values enthalten.

[27] OLAP: Online Analytical Processing
[28] OLTP: Online Transaction Processing
[29] http://de.wikipedia.org/wiki/Spaltenorientierte_Datenbank
[30] http://cassandra.apache.org/
[31] http://labs.google.com/papers/bigtable-osdi06.pdf

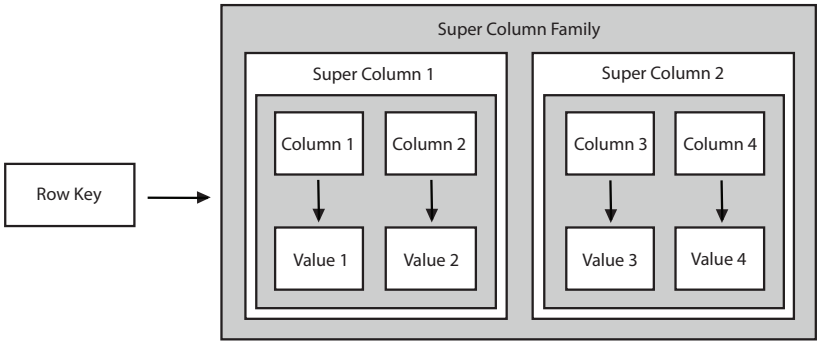


Abbildung 4.6: Cassandra Datenmodell mit flexibler Datenstruktur.

Column Family – eine Gruppierung mehrerer Columns. Diese Dimension kann in etwa mit einer zweidimensionalen Tabelle in einem RDBMS verglichen werden.

Super Column Family – diese Dimension vereint mehrere zusammenhängende Column Families miteinander.

Cassandra erlaubt für alle Werte beliebige Datentypen. Somit können Daten nicht nur als Value in einer Column, sondern auch im Row Key gespeichert werden [19, S. 43].

Die Datenstruktur innerhalb der Maps lässt sich ebenfalls flexibel anpassen. Gibt es in einer Column Family zu einem bestimmten Key aktuell keinen Wert, so wird das gesamte Key/Value-Paar nicht gespeichert.

Skalierung

Cassandra bietet eine *elastische Skalierbarkeit*. Diese Spezialform der horizontalen Skalierbarkeit bedeutet, dass das Rechnercluster übergangslos hoch sowie auch wieder hinunter skalieren kann. Dies wird erst durch die Eigenschaft des dezentralisierten Clusters (kein Master/Slave-Prinzip) möglich. Dabei können neue Knoten dem Cluster ohne dessen Neukonfiguration hinzugefügt werden. Cassandra übernimmt hierbei die gesamten Aufgaben, um einen neuen Knoten im Cluster zu registrieren und zu partitionieren. Ähnliches gilt für die Entfernung eines Knotens aus dem Cluster. Hierbei balanciert Cassandra die Daten des zu entfernenden Knotens automatisch auf die übrigen Knoten aus [19, S. 16].

Replikation

Durch die Eigenschaft einer verteilten Datenbankarchitektur wird diese spaltenorientierte Datenbank erst wirklich performant. Das Master/Slave-Prinzip aus den RDBMS wird hierbei komplett verworfen und stattdessen ein dezentralisiertes Cluster aufgebaut. Dabei sind alle Knoten dieses Clusters ident.

Mittels eines *P2P-Protokolls* und dem sogenannten *Gossiper*[32] stehen alle Knoten miteinander in Verbindung. Dabei gibt jeder Knoten in einem gewissen Zeitintervall seine Status-Informationen an einen anderen beliebigen Knoten weiter [19, S. 15]. Somit ist sichergestellt, dass auch bei einem Ausfall eines Knotens alle anderen Knoten des Clusters diese Information erhalten und das Datenbanksystem stabil weiterläuft.

Performance

Cassandra ist durch ihre verteilte und dezentralisierte Architektur schon von Beginn an auf hohe Performance über ein großes Rechnercluster hinweg ausgelegt. Ein Benchmark von Avinash Lakshman und Prashant Malik[33] verdeutlicht den Unterschied von Cassandra zu MySQL hinsichtlich der Datenbankperformance:

MySQL > 50 GB Daten
Schreibzugriffe Durchschnitt: ~300ms
Lesezugriffe Durchschnitt: ~350ms

Cassandra > 50 GB Daten
Schreibzugriffe Durchschnitt: 0.12ms
Lesezugriffe Durchschnitt: 15ms

Konsistenzerhaltung der Daten

Diese spaltenorientierte Datenbank zählt zu den NoSQL-Datenbanken mit *eventual consistency*, da sie eine hohe Verfügbarkeit der Rechnerknoten sowie Partitionstoleranz gewährleistet. Bei Cassandra kann die Konsistenz der Daten manuell gesteuert und nach den eigenen Bedürfnissen angepasst werden, zum Leidwesen der Verfügbarkeit [19, S. 17].

Einsatzgebiete

Cassandra eignet sich vor allem für große, verteilte Datenbank-Anwendungen mit besonders vielen Schreiboperationen. Als System auf einem einzigen Datenbankserver kann sie zwar auch verwendet werden, dort jedoch nicht komplett ihre Mächtigkeit entfalten. Die bekanntesten Unternehmen, welche aktuell auf Cassandra setzen, sind Twitter bei der Datenanalyse, Facebook bei der Inbox-Suche und Digg [19, S. 25–26].

[32] http://wiki.apache.org/cassandra/ArchitectureGossip
[33] http://www.slideshare.net/Eweaver/cassandra-presentation-at-nosql

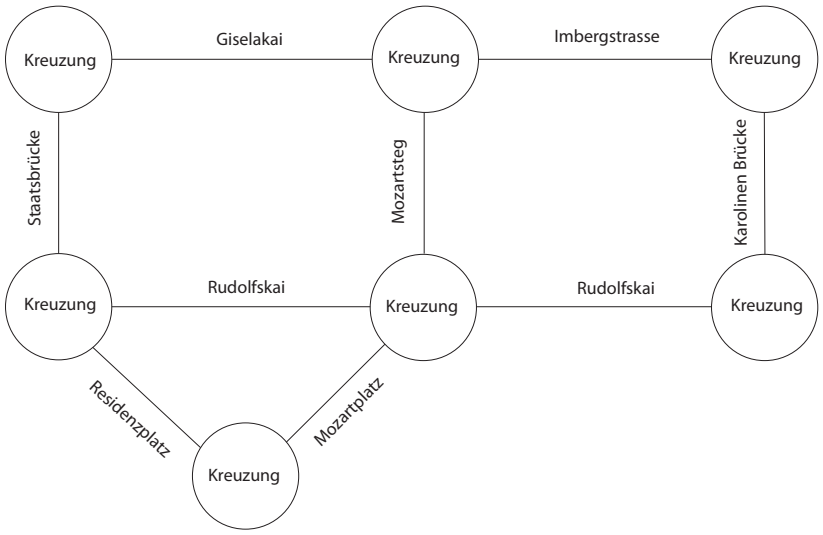


Abbildung 4.7: Beispiel eines Graphen anhand eines Straßennetzes.

4.6 Graphdatenbanken

4.6.1 Allgemein

Mit der steigenden Popularität von sozialen Netzwerken, Empfehlungssystemen in Webshops, Routenplanungs-Systemen und semantischen Verknüpfungen von Daten, änderten sich die Anforderungen an Datenbanksysteme drastisch. Da relationale Datenbankmodelle diese stark vernetzten Informationen nur mit sehr großem Aufwand abbilden können, erfreuen sich Graphdatenbanken an dieser Stelle immer mehr an Beliebtheit [11, S. 169]. Graphdatenbanken bestehen, wie sich aus dem Namen schon herleiten lässt, aus Graphen, welche in der Mathematik ihren Ursprung fanden. Ein Graph lässt sich in Knoten sowie Kanten unterteilen. Knoten beinhalten hierbei Informationen, wobei Kanten diese Knoten untereinander vernetzen, sie stellen also Beziehungen zwischen den Knoten dar [43].

Graphen lassen sich am einfachsten anhand eines Straßen-Beispiels von Ralf Hartmut Güting [18] in Kombination mit Abbildung 4.7 erklären. Knoten stellen hierbei Straßenkreuzungen dar, wobei Kanten Straßenabschnitte repräsentieren. Straßen an sich sind dabei Pfade über Teile des Graphen. Mit diesem Aufbau sollte es möglich sein, die kürzeste Route zwischen Standort A und Standort B zu finden, oder auch einen Subgraphen innerhalb eines bestimmten Radius von Knoten (Standort) C ausfindig zu machen [18].

Kanten können wiederum gerichtet oder ungerichtet sein. *Gerichtete Kanten* erlauben nur eine Richtung der Beziehung zwischen den Knoten (s. Abbildung 4.8), wobei *ungerichtete Kanten* eine beidseitige Beziehung darstellen (Abbildung 4.9) [43].

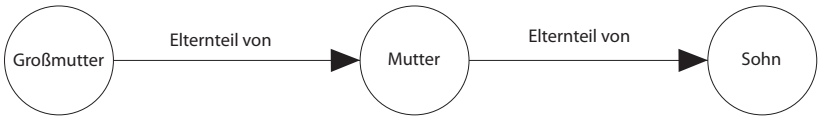


Abbildung 4.8: Gerichteter Graph.

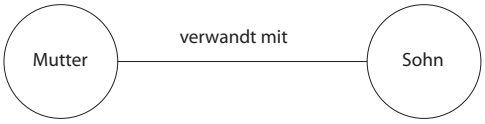


Abbildung 4.9: Ungerichteter Graph.

Zudem wird noch zwischen *Multigraphen*, bei welchen zwei Knoten durch mehrere Kanten in Beziehung gesetzt werden können, und *Hypergraphen* unterschieden. Hypergraphen, auch Hyperkanten genannt, verbinden mehrere Knoten mit einer Kante gleichzeitig [43].

Da das einfache Graphdatenmodell jedoch mit den soeben vorgestellten Eigenschaften nicht alle Problemstellungen darstellen kann, hat sich in den letzten Jahren ein neues, erweitertes Modell durchgesetzt, das *„Property-Graph-Modell“*, welches bislang bei fast allen aktuellen Graphdatenbanken Verwendung findet [11, S. 173].

Ein Property Graph ist dabei laut [17] ein Key/Value-basierter, gerichteter und multi-relationaler Graph:

Key-Value basiert – Knoten sowie auch Kanten können eine beliebige Anzahl an Eigenschaften besitzen, welche untereinander verknüpft sind.

Gerichtet – es wird nur eine Richtung der Beziehung zwischen den Knoten erlaubt

Multi-relational – von einem Knoten können mehrere Kanten hin zu einem zweiten Knoten führen.

Abbildung 4.10 veranschaulicht einen solchen Property Graph anhand eines kleinen sozialen Netzwerkes.

4.6.2 Neo4j

Neo4j[34] zählt zu einer der am weitest verbreiteten Graphdatenbanken und ist seit 2003 durchgehend im Produktionseinsatz [34]. Dieses Open-Source Projekt, veröffentlicht unter der GPLv3[35], ist ebenfalls unter einer kommerziellen Lizenz mit professioneller Unterstützung der Firma Neo Technology[36] erhältlich [31].

[34] http://neo4j.org/
[35] GNU General Public License Version 3
[36] http://neotechnology.com

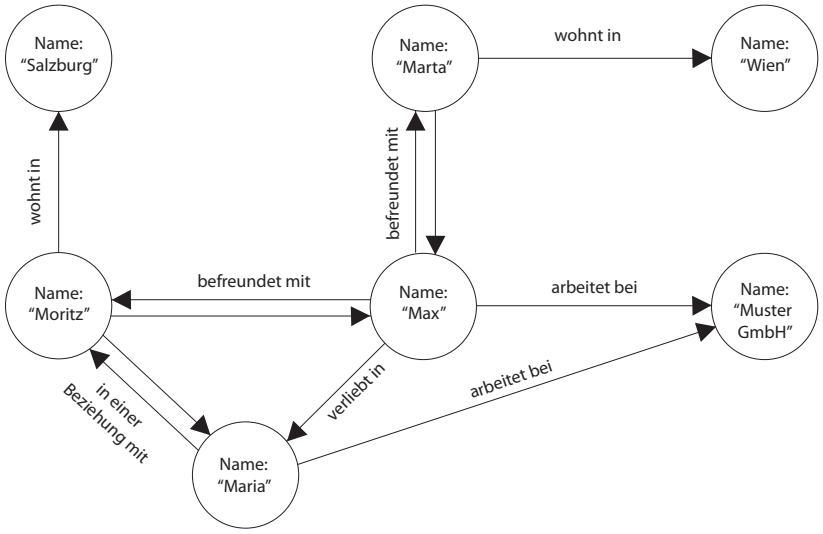


Abbildung 4.10: Property Graph in einem sozialen Netzwerk.

Die in Java entwickelte Graphdatenbank Neo4j ist vollkommen ACID-transaktional, hoch verfügbar [32] und kann sowohl standalone als auch als eingebetteter Server betrieben werden [33]. Zudem unterstützt die API unterschiedliche REST-Wrapper [34] und zahlreiche Programmiersprachen wie:

- Java
- PHP
- Ruby
- Erlang
- uvm.

Datenmodell

Das Datenmodell in Neo4j entspricht dem eines Graphen und besteht somit aus Knoten, Relationen (Kanten) und Eigenschaften. Diese Knoten können durch mehrere Kanten beidseitig sowie auch in nur eine Richtung miteinander verknüpft sein. Knoten und Kanten können beliebige Eigenschaften besitzen, welche ebenfalls aussagekräftig für die Beziehung zwischen den Knoten sind. Diese Eigenschaften können beliebige primitive Datentypen wie Booleans, Integer-Variablen, Float-Variablen sowie Strings oder auch homogene Arrays annehmen [30]. Abbildung 4.11 visualisiert dieses Datenmodell an einem einfachen Beispiel.

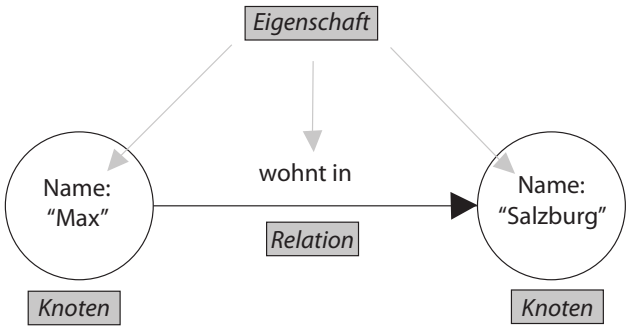


Abbildung 4.11: Neo4j Datenmodell.

Skalierung

Mit einer Neo4j Datenbank alleine ist es möglich, abhängig von der Leistung des Datenbankrechners, mehrere Milliarden Knoten und Kanten zu verwalten [32]. Reicht eine Datenbank alleine dennoch nicht aus, um alle Daten eines Graphen zu speichern, müsste sie mittels Sharding oder Partitionierung horizontal skaliert werden.

Aktuell steht für Neo4j kein offizieller Sharding-Mechanismus zur Verfügung, jedoch wird daran gearbeitet. Somit sollte Sharding mit der Version 1.3 ebenfalls möglich sein. Die Logik dafür kann jedoch auch in der aktuellen Version Client-seitig vom Entwickler selbst implementiert werden [29].

Replikation

Hochverfügbarkeits-Features, wie auch die Replikation, sind nur mit der kommerziellen Neo4j Enterprise Edition (Neo4j HA[37]) möglich. Dabei können mehrere Neo4j Slave-Datenbanken konfiguriert werden, welche Replikationen einer Master-Datenbank abbilden. Auf diese Slaves können bei Neo4j HA ebenfalls Schreiboperationen ausgeführt werden, ohne diese im ersten Schritt an den Master weiterzuleiten [32, S. 39]. Fällt ein Master aus, wird automatisch ein Slave als neuer Master ernannt [32, S. 42]. Dies gewährleistet ebenfalls eine Ausfalltoleranz. Abbildung 4.12 zeigt wie eine solche Master/Slave-Datenbankarchitektur unter Neo4j aufgebaut sein kann. Neo4j HA verwendet für die Überwachung der Datenbankrechner, Auswahl des Masters und die Ausbreitung des generellen Clusters Apache ZooKeeper[38] [32, S. 39].

In Neo4j HA ist die Konsistenz der Daten, im Vergleich zu einer Neo4j Datenbank, auf nur einem Server gelockert, da nach Abschluss einer Schreiboperation auf einem Slave nicht sofort alle anderen Slaves das Update der

[37] Neo4j High Availability
[38] http://hadoop.apache.org/zookeeper/

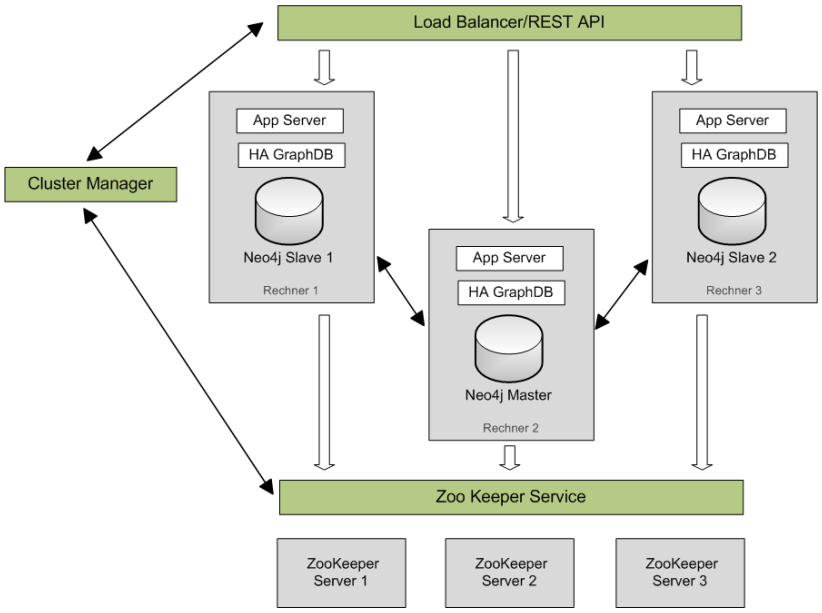


Abbildung 4.12: Neo4j Master/Slave Datenbankarchitektur.

Daten durch den Master erhalten. Alle restlichen ACID-Eigenschaften bleiben jedoch bestehen [32, S. 39]. Diese Lockerung der Konsistenz ist wiederum ein Beweis für das CAP-Theorem – soll das System gleichzeitig hoch verfügbar und ausfalltolerant sein, so kann nicht ebenfalls auch eine starke Konsistenz der Daten gegeben sein.

Performance

Zur Steigerung der Performance bietet Neo4j zum einen die Optimierung des Neo4j Caches, zum anderen die optimale Konfiguration der JVM[39], in welcher Neo4j läuft [32, S. 8].

Cache optimieren Neo4j verfügt über zwei verschiedene Caches [32, S. 8–11]:

File Buffer Cache – dieser Cache, auch *„Low Level Cache“* genannt, speichert alle Daten im selben Format wie die physische Festplatte. Dadurch kann die Performance der Lese- sowie auch der Schreiboperationen optimiert werden.

Object Cache – im Object Cache *(„High Level Cache“)* befinden sich alle Knoten, Kanten und Eigenschaften in einem Format, welches für die schnelle Traversierung[40] und Transaktions-Mutationen optimiert ist.

[39] Java Virtual Machine
[40] Untersuchung der Knoten im Graphen in einer systematisch bestimmten Reihenfolge

Zudem ist noch auf die Wahl der Strings in den Knoten- sowie auch Kanteneigenschaften zu achten. Können diese als *Short-Strings* eingestuft werden, werden sie direkt, ohne Umweg über den Property-Speicher, gespeichert und benötigen somit keinen Eintrag für die Speicherung. Somit können diese Eigenschafts-Strings mit nur einem einzelnen Zugriff gelesen sowie auch geschrieben werden. Dies wiederum steigert ebenfalls die Performance [32, S. 13].

Mit Neo4j ist es möglich, durch den Einsatz der Caches über 2000 Kantenschritte pro Millisekunde zu erreichen. Bei einem sozialen Netzwerk mit 1000 Personen ist es bei der Suche von Freunden-meiner-Freunde um das 1000fache schneller als MySQL, wobei der Unterschied hier mit der Größe des Netzwerkes exponentiell steigt [34].

Konsistenzerhaltung der Daten

Dadurch dass Neo4j vollkommen ACID-transaktional arbeitet, ist die starke Konsistenz der Daten ebenso wie in ACID-transaktionalen RDBMS gegeben.

Dadurch müssen alle Datenaufrufe in Neo4j in einer Transaktion vollzogen werden. Die Standard-Isolationsebene ist hierbei READ_COMMITTED, welche auf die jeweiligen Bedürfnisse der Anwendung angepasst werden kann. Durch die gesetzten Sperrmechanismen können Deadlocks auftreten, welche von Neo4j als solche erkannt und dementsprechend behandelt werden [32, S. 14]. Im Unterschied dazu lockert jedoch die kommerzielle Version Neo4j HA die Konsistenz, um eine hohe Verfügbarkeit über mehrere Replikationen hinweg zu erhalten.

Einsatzgebiete

Neo4j findet mit seinen Charakteristiken besonders Verwendung in Bereichen wie LinkedData, sozialen Netzen, Semantic Web und RDF sowie auch tiefen Empfehlungsalgorithmen [34].

Kapitel 5

Implementierung

Ziel dieser Arbeit war es, alternative Datenbanksysteme bzw. -strukturen für ein E-Commerce System zu testen. Diese sollten vor allem besser skalierbar, flexibler in ihrer Datenstruktur und mindestens gleichbleibend performant sein.

Der erste Schritt zu einer besser skalierbaren sowie auch flexibleren Datenbank führt zum Ansatz des EAV-Modells, welches schon in der Open-Source E-Commerce-Plattform Magento Verwendung findet. Da dieses Modell aber durch komplexer gestaltete SQL-Queries Nachteile in der Performance birgt, was auch bei Magento ein großes Problem ist, wird im zweiten Schritt eine den Anforderungen entsprechende NoSQL-Datenbank auf ihre Skalierbarkeit und Performance geprüft.

Für den Prototypen erfolgte die Umstellung der Datenbankstruktur in den produkt- sowie kundenbezogenen Tabellen. Das zugrunde liegende E-Commerce System ist die Open-Source Lösung osCommerce, da es im Vergleich zu Magento und anderen Open-Source E-Commerce Systemen einfacher aufgebaut und die Datenstruktur infolge dessen schneller abänderbar ist.

5.1 osCommerce

osCommerce[1] ist eine Open-Source E-Commerce Lösung basierend auf PHP und MySQL. Sie wird durch die Open-Source Community ständig weiterentwickelt. Durch die Out-of-the-box Installation entsteht mit minimalem Aufwand ein einfacher Online-Shop, welcher im Design und der flexiblen Modulzusammensetzungen variabel gestaltet werden kann.

Die Entscheidung für diese Lösung fiel vor allem aufgrund der Einfachheit der dahinter liegenden Datenbank (50 Tabellen insgesamt) und der Eigenschaft, dass der Quelltext von osCommerce offen liegt und dadurch unter einer Open-Source Lizenz steht. Somit kann schnell ein Überblick über die

[1] http://www.oscommerce.de/

Relationen innerhalb dieser Datenbank geschaffen und die Datenstruktur des Prototypes abgeändert werden.

5.2 Wahl der NoSQL-Datenbank

In Bezug auf die vier NoSQL-Hauptkategorien, wie in Abschnitt 4.1 erklärt, sind für die umzustrukturierenden Tabellen (Produkte und Kunden) des E-Commerce Systems Key/Value-Datenbanken oder dokumentorientierte Datenbanken von Relevanz.

Oren Eini [12] schrieb in seinem Blog über den großen Unterschied zwischen Key/Value- und dokumentorientierte Datenbanken:

> „The major benefit of using a document database comes from the fact that while it has all the benefits of a key/value store, you aren't limited to just querying by key."

Da aber auch die APIs der dokumentorientierten Datenbanken zum Großteil schon komplexere Queries als Key-Value Datenbanken absetzen können, ohne dass der Anwender selbst eine eigene MapReduce Funktion für diesen Zweck schreiben muss, fiel die Wahl schnell auf die Gruppe der dokumentorientierten Datenbanken.

In dieser Sparte wurde weiters zwischen zwei der größten und am weitesten entwickelten Open-Source Datenbanksystemen CouchDB[2] und MongoDB[3] entschieden.

In Hinblick auf das Pre-Filtering für die MapReduce Funktion bringt MongoDB den Vorteil mit sich, dass dieses schon in der Funktion *„query"* eingebaut ist. Bei CouchDB müsste diese Logik vom Datenbankentwickler selbst implementiert werden [36].

CouchDB setzt automatisch einen Index auf jede MapReduce Query. Dieser Index ist zu vergleichen mit einer temporären Tabelle, welche gespeicherte Ergebnisse dieser Query beinhaltet. Dadurch benötigt sie aber bei der Erstausführung dieser Query eine weitaus längere Abfragezeit, um diesen Index zu setzen. Jede nachfolgende Query ist jedoch hoch performant. MongoDB setzt diesen Index nicht und hat somit bei ein und derselben Query gleich lange Abfragezeiten [36].

Daraus lässt sich der Schluss ziehen, dass CouchDB für Datenbanken, auf welche immer dieselben Abfragen getätigt werden, bevorzugt wird, wobei MongoDB für Datenbanken mit stetig variierenden Abfragen eingesetzt werden kann [36].

Ein weiterer großer Unterschied liegt darin, dass CouchDB im Vergleich zu MongoDB MVCC (s. Abschnitt 2.5) verwendet [26]. Dieser aufwändig zu

[2] http://couchdb.apache.org/
[3] http://www.mongodb.org/

implementierende und performanceschwächende Kontrollmechanismus kann bei E-Commerce-Systemen wie osCommerce vernachlässigt werden, da dabei statt komplexen Master-Master Replikationen, bei welchen MVCC starke Vorteile mit sich bringt, Master-Slave Architekturen zum Einsatz kommen [26].

MongoDB bedient sich, ähnlich wie Google BigTable, des Shardings (s. Abschnitt 2.4) um die Datenbank horizontal skalierbar zu halten, wobei CouchDB Datenbanken nur repliziert [26].

Durch die ebenfalls einfache Handhabung der MongoDB-API in PHP und das kürzlich entwickelte sehr übersichtliche GUI „RockMongo“[4], wurde zugunsten der dokumentorientierten Datenbank MongoDB entschieden.

5.3 Implementierung des EAV-Modells

5.3.1 EAV-Modell in Magento

Als Vorbild dieser Umsetzung diente die Datenbankstruktur der Open-Source E-Commerce Lösung Magento[5]. Magento setzt hier in fast allen Tabellen überwiegend auf das EAV-Modell, wie Anhang A zeigt. Dieses UML-Diagramm soll veranschaulichen, um welche komplexen Dimensionen der Datenbank es sich dabei handelt.

5.3.2 Umsetzung und Eingrenzungen

Um das Projekt etwas einzugrenzen, wurden nur die wichtigsten produktbezogenen sowie kundenbezogenen Tabellen in das EAV-Modell umgewandelt. Zudem ist im neuen Modell nur eine Sprachwahl möglich, da es für dieses Projekt wichtiger war, das EAV-Modell zu integrieren, anstatt auf erweiterte Funktionalitäten des Systems zu achten.

Wie die Umstellung der Datenbankstruktur genau erfolgte, kann aus den UML-Diagrammen in Anhang B sowie Anhang C entnommen werden.

Anpassung der SQL-Query-Abfragen

Der gesamte Online-Shop Ordner wurde kopiert, um dieselben Produkte ebenfalls im neuen „EAV-Shop“ verwalten zu können. Nachdem alle für das EAV-Modell benötigten Tabellen hinzugefügt wurden, konnten alle PHP-Dateien im System auf ihre SQL-Queries und die Zusammenarbeit untereinander analysiert werden. Alle betroffenen Queries wurden so umgestellt, dass sie die neuen EAV-Tabellen ansprechen und aus diesen ihre Daten beziehen.

Schnell trat das Problem auf, dass SQL-Query-Abfragen mit mehreren Attributen sofort an die Performance-Grenzen stießen. Nach dem Versuch

[4] http://code.google.com/p/rock-php/wiki/rock_mongo
[5] http://www.magentocommerce.com/de/

der Indizierung aller EAV-Tabellen konnte diesem Problem im ersten Schritt schon etwas entgegengewirkt werden. Ab 6–7 Attributen stieg die Abfragezeit jedoch trotz Indizierung von 1–2 Sekunden auf beträchtliche 50–60 Sekunden an. Diese Zeiten wären nicht mehr vertretbar gewesen.

Somit wurden eigene Produkt- bzw. Kundenklassen erstellt, da über viele Query-Abfragen fast alle Attribute der Produkte bzw. Kunden aus der Datenbank geladen werden. Diese laden mit Übergabe der dementsprechenden Objekt-IDs alle objektbezogenen Attribute, durch die Verwendung von performanteren LEFT JOIN Anweisungen, aus der Datenbank und speichern die resultierenden Attributwerte in Arrays.

Bei den ursprünglich implementierten INNER JOIN Abfragen wurde pro Attribut über die Entity Tabelle, die EAV-Attribute Tabelle, die Attribute Tabelle (um den Attribut-Namen zu ermitteln) und weiters über die Value Tabelle gejoint. Im Vergleich dazu werden in diesen Klassen nur 2 INNER JOIN Anweisungen über die Entity-Tabelle mit der EAV-Attribute Tabelle sowie über die EAV-Attribute-Tabelle mit der Attribute-Tabelle benötigt. Die Values werden mittels LEFT JOIN Anweisungen der EAV-Attribute Tabelle mit allen Value-Tabellen ermittelt. Somit werden alle Values nach Typ sortiert und mittels der Spalten „attribute_ name“ und „attribute_ type“ schnell und einfach in ein Array gespeichert.

Abfrage mittels Inner Joins bei 5 Attributen (Abfragezeit 1.2810 Sekunden):

```
 1 select
 2 p.products_id,
 3 pimage.value products_image,
 4 pname.value products_name,
 5 pmanuID.value manufacturers_id,
 6 pprice.value products_price,
 7 ptci.value products_tax_class_id
 8 from products_entity p
 9 join eav_attribute as eavattr_tci on (p.entity_id = eavattr_tci.
       entity_id)
10 join attribute as attr_tci on (eavattr_tci.attribute_id = attr_tci.
       attribute_id)
11 join product_value_int as ptci on (eavattr_tci.eav_attribute_id = ptci.
       eav_attribute_
12 id) and eavattr_tci.attribute_id = attr_tci.attribute_id and attr_tci.
       attribute_
13 name = 'tax_class_id'
14 join eav_attribute as eavattr_price on (p.entity_id = eavattr_price.
       entity_id)
15 join attribute as attr_price on (eavattr_price.attribute_id = attr_price
       .attribute_
16 id)
17 join product_value_decimal as pprice on (eavattr_price.eav_attribute_id
       =
18 pprice.eav_attribute_id) and eavattr_price.attribute_id = attr_price.
       attribute_id and
```

```
19 attr_price.attribute_name = 'price'
20 join eav_attribute as eavattr_manuID on (p.entity_id = eavattr_manuID.
       entity_id)
21 join attribute as attr_manuID on (eavattr_manuID.attribute_id =
       attr_manuID.attribute_
22 id)
23 join product_value_int as pmanuID on (eavattr_manuID.eav_attribute_id =
       pmanuID.
24 eav_attribute_id) and eavattr_manuID.attribute_id = attr_manuID.
       attribute_id and attr_
25 manuID.attribute_name = 'manufacturers_id'
26 join eav_attribute as eavattr_status on (p.entity_id = eavattr_status.
       entity_id)
27 join attribute as attr_status on (eavattr_status.attribute_id =
       attr_status.attribute_
28 id)
29 join product_value_int as pstatus on (eavattr_status.eav_attribute_id =
       pstatus.
30 eav_attribute_id) and eavattr_status.attribute_id = attr_status.
       attribute_id and attr_
31 status.attribute_name = 'status'
32 join eav_attribute as eavattr_name on (p.entity_id = eavattr_name.
       entity_id) join
33 attribute as attr_name on (eavattr_name.attribute_id = attr_name.
       attribute_id)
34 join product_value_varchar as pname on (eavattr_name.eav_attribute_id =
       pname.
35 eav_attribute_id) and eavattr_name.attribute_id = attr_name.attribute_id
        and attr_name.
36 attribute_name = 'name'
37 join eav_attribute as eavattr_image on (p.entity_id = eavattr_image.
       entity_id)
38 join attribute as attr_image on (eavattr_image.attribute_id = attr_image
       .attribute_
39 id)
40 join product_value_varchar as pimage on (eavattr_image.eav_attribute_id
       = pimage.
41 eav_attribute_id) and eavattr_image.attribute_id = attr_image.
       attribute_id and
42 attr_image.attribute_name = 'image'
43 where p.products_id = '65'
```

Abfrage über alle verfügbare Attribute in den Objekt-Klassen (Abfragezeit 0.0127 Sekunden):

```
1 select
2 p.products_id,
3 a.attribute_id,
4 b.attribute_name,
5 b.attribute_type,
6 c.value product_varchar,
7 d.value product_decimal,
8 e.value product_text,
9 f.value product_datetime,
```

```
10 g.value product_int
11 FROM products_entity p JOIN eav_attribute a USING (entity_id)
12 join attribute b on a.attribute_id = b.attribute_id
13 left join product_value_varchar c on (a.eav_attribute_id = c.
       eav_attribute_
14 id)
15 left join product_value_decimal d on (a.eav_attribute_id = d.
       eav_attribute_
16 id)
17 left join product_value_text e on (a.eav_attribute_id = e.eav_attribute_
18 id)
19 left join product_value_datetime f on (a.eav_attribute_id = f.
       eav_attribute_
20 id)
21 left join product_value_int g on (a.eav_attribute_id = g.eav_attribute_
22 id)
23 where p.products_id = '65'
```

5.3.3 SQL Query Cache

Um die Performance der relationalen Datenbank mit dem EAV-Modell noch zu optimieren, kam der MySQL Query Cache[6] zum Einsatz, welcher die über die SQL-Queries geladenen Daten in einem Cache zwischenspeichert, um zu einem späteren Zeitpunkt schneller wieder darauf zugreifen zu können. Um den Cache zu aktivieren, werden zwei Zeilen der Datei „my.ini“ hinzugefügt und der MySQL-Server neu gestartet.

```
1 query-cache-type = 1
2 query-cache-size = 64M
3 query-cache-limit = 512M
```

Das Limit der Cache-Größe wurde auf 512 MB gesetzt, um zu gewährleisten, dass so viele Daten wie möglich im Cache gespeichert bleiben. Um die Ausführungszeiten aus dem Cache für die spätere Auswertung erfassen zu können, mussten nach dem Neustart des MySQL-Servers alle Aktionen, bei denen Daten aus der Datenbank geladen werden, doppelt ausgeführt werden. Bei erster Ausführung werden die Daten noch direkt aus der Datenbank geladen und gleichzeitig auch im Cache gespeichert, beim zweiten Durchlauf sollten diese Daten direkt aus dem Cache geladen werden. Bei manchen Aktionen wie INSERT, UPDATE oder DELETE Statements war der SQL Query Cache jedoch nicht verwendbar, da hier nie dieselben Daten übergeben werden.

5.3.4 eAccelerator

eAccelerator[7] ist eine freie Software, welche PHP-Seiten auf Webservern optimiert und cached und kam im Zuge der EAV-Modell Performance-Evaluierung

[6] http://dev.mysql.com/tech-resources/articles/mysql-query-cache.html
[7] http://eaccelerator.net/

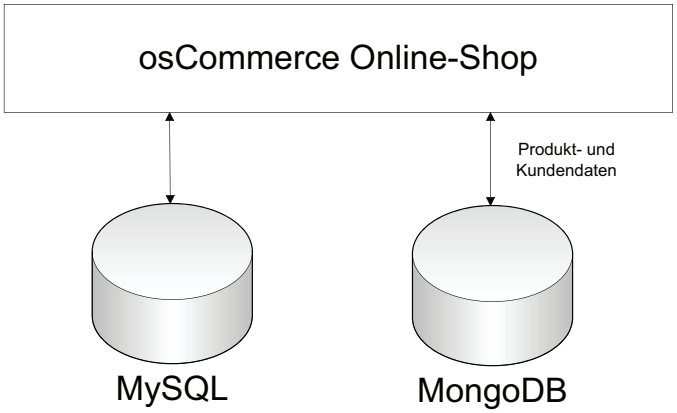


Abbildung 5.1: Anbindung von MongoDB an osCommerce.

ebenfalls zum Einsatz. Diese Software erzielte im Vergleich zum MySQL Query Cache jedoch weniger zufriedenstellende Ergebnisse, da sie hauptsächlich PHP-Dateien und nicht MySQL Queries gecached und optimiert werden. Um eAccelerator unter Windows zu installieren, wird die vorkompilierte eAccelerator DLL-Datei passend zur eigenen PHP-Version benötigt und den PHP-Extensions inklusive der Zuweisung in der PHP-Konfiugrationsdatei „php.ini“ hinzugefügt.

5.4 Implementierung der MongoDB Datenbank

Bei der Anbindung der MongoDB Datenbank war, wie auch beim EAV-Modell, der erste Schritt, alle wichtigen kunden- sowie auch produktbezogenen Tabellen auszulesen und die dementsprechenden Daten in die MongoDB-Datenbank zu importieren. Alle übrigen Tabellen verblieben in der relationalen Datenbank. Die Zusammenarbeit dieser beiden Datenbanken mit osCommerce visualisiert Abbildung 5.1.

Da zwar in MongoDB mittels der schemalosen Struktur innerhalb der Dokumente alle objektspezifischen Attribute frei wählbar sind und verschiedene Produkttypen ebenfalls verschiedene Attribute besitzen können, blieb es vorübergehend bei der Struktur, wie sie auch im originalen relationalen Modell verwendet wird. Durch diese Entscheidung blieb ein großer Programmieraufwand erspart, welcher für die Analyse dieser Datenbank in einem E-Commerce System nicht von Relevanz gewesen wäre.

Ein einheitliches Dokument für ein Produkt sieht nun im BSON-Format in MongoDB folgendermaßen aus:

```
1 {
2    '_id': ObjectId('4d94937c86b3c9880b00620a'),
3    'id': 25101,
4    'model': 'ABC24994',
```

```
 5    'quantity': 15,
 6    'image': 'microsoft/msimpro.gif',
 7    'price': 80.54,
 8    'date_added': 'Thu, 31 Mar 2011 16: 40: 12 +0200',
 9    'last_modified': 'Thu, 01 Jan 1970 01: 00: 00 +0100',
10    'date_available': 'Thu, 31 Mar 2011 16: 40: 12 +0200',
11    'weight': 5.24,
12    'status': 1,
13    'tax_class_id': 1,
14    'manufacturers_id': 2,
15    'products_ordered': 0,
16    'details': {
17      'name': 'Dummy Produkt',
18      'description': 'Dummy Produkt Beschreibung',
19      'url': 'http://www.microsoft.com',
20      'viewed': 0,
21      'language_id': 1
22   }
23 }
```

Da immer noch Relationen zwischen diesen MongoDB-Dokumenten und MySQL-Tabellen bestehen, war es nötig, neben der *_id* für die MongoDB Datenbank selbst ebenfalls noch eine *id* für die Erhaltung der Relationen zu den MySQL-Tabellen mitzuführen. Selbes gilt für die Dokumente in der Collection der Kunden.

Die Verwaltung zweier verschiedener Datenbankmanagementsysteme für eine E-Commerce Anwendung bedeutet zwar insgesamt einen komplexeren Programmier- sowie auch mehr Speicheraufwand, wurde aber für eine schnelle Implementierung des Prototypen akzeptiert, um den Umfang des Programmieraufwandes so gering als möglich zu halten. Der Prototyp sollte grundsätzlich der Evaluierung der dokumentorientierten Datenbank dienen und nicht ein komplett neues E-Commerce System mit einer dahinter liegenden dokumentorientierten Datenbank schaffen. Die Umstellung der Produkt- sowie Kundentabellen auf MongoDB reichte für diese Zwecke.

Für die Beibehaltung der Übersicht aller Collections und darin gespeicherter Dokumente war das MongoDB-Administrations-GUI *RockMongo*[8] von großer Bedeutung. Mit Hilfe dieses Werkzeuges, vergleichbar mit phpMyAdmin in MySQL, ist es unter anderem möglich, Serverinformationen, Serverstatus, Replikationsvorgänge, Datenbanken und vieles mehr zu überwachen und zudem noch deren Collections und Dokumente zu verwalten.

5.4.1 Verwendung der MongoDB Query API für PHP

Mit der Verwendung der MongoDB Query API für PHP wurden alle Queries aus dem osCommerce Quellcode, welche in MongoDB ausgelagerte Daten ansprachen, umgestellt.

[8] http://code.google.com/p/rock-php/wiki/rock_mongo

Zu beachten war, dass jedes Produkt sowie auch jeder Kunde noch eine eigene Tabelle mit einem Primary-Key sowie auch Foreign-Keys in der MySQL Datenbank führte, um die Relationen zu den übrigen Tabellen zu erhalten.

Durch die schrittweise Umstellung der Queries und der regelmäßigen Überprüfung der betroffenen PHP-Dateien konnten Fehler schnell identifiziert und behoben werden.

MongoDB PHP-Wrapper Klasse

Um einen schnellen Zugriff auf die MongoDB Funktionen in PHP zu erhalten, wurde eine eigene sogenannte MongoWrapper-Klasse, zu finden in Anhang D, zwischen die osCommerce-PHP-Dateien und der MongoDB Query API geschaltet. Darin wurden die benötigten Funktionen wie *get*, *getAll*, *update*, *insert* und *delete* mit der Unterstützung eines Tutorials[9] implementiert.

Die Erzeugung einer Datenbank sowie auch der darin enthaltenen Collections, erfolgt „on the fly“. Es ist somit nicht notwendig, die Datenbank im Vorhinein über die MongoDB-Javascript-Shell oder das Administrations-GUI zu erstellen. MongoDB erkennt eine noch nicht vorhandene Datenbank oder Collection als neu und erstellt diese.

Im nächsten Schritt wurde eine neue Instanz der Mongo-Datenbank angelegt.

```
1 $mongoP = new mongoWrapper();
2 $mongoP->setDatabase('oscommerce');
3 $mongoP->setCollection('products');
```

Das Einfügen neuer Daten in eine MongoDB-Datenbank-Collection geschieht am Beispiel eines neuen Produktes folgendermaßen:

```
1  $mongoP->insert(array(
2        'id' => 15,
3        'model' => 'EBF67',
4        'quantity' => 24,
5        'image' => 'images/dummy.jpg',
6        'price' => 12.50,
7        'date_added' => new MongoDate(time()),
8        'last_modified' => new MongoDate(time()),
9        'date_available' => new MongoDate(time()),
10       'weight' => 0.48,
11       'status' => 1,
12       'tax_class_id' => 1,
13       'manufacturers_id' => 5,
14       'products_ordered' => 0,
15       'details' => array(
16         'name' => 'Dummy Product',
17         'description' => 'Dummy Product Description',
18         'url' => 'www.dummy.com',
```

[9] http://query7.com/mongodb-php-tutorial

```
19          'viewed' => 0,
20          'language_id' => 1
21        )
22        ));
```

Um ein Produkt aus der Datenbank auslesen zu können, wird auf die Funktion *get* zugegriffen, welcher ein Array mit den Suchparametern übergeben wird. Bei einer Übereinstimmung wird der auf das gefundene Objekt zeigende MongoDB-Cursor wie ein Iterator durchlaufen, die Daten in ein Array gespeichert und dieses in weiterer Folge zurückgeliefert.

```
1 $product = $mongoP->get(array('id' => 15, 'status' => 1));
```

Das Array *$product* enthält nun folgende Daten:

```
1  Array
2  (
3    [0] => Array
4      (
5        [_id] => MongoId Object
6          (
7            [$id] => 4d94935286b3c9880b000001
8
9          )
10       [id] => 15
11       [model] => 'EBF67'
12       [quantity] => 24
13       [image] => 'images/dummy.jpg'
14       [price] => 12.50
15       [date_added] =>  => MongoDate Object
16                 (
17                     [sec] => 1290669462
18                     [usec] => 0
19                 )
20       [last_modified] =>  => MongoDate Object
21                 (
22                     [sec] => 1290669462
23                     [usec] => 0
24                 )
25       [date_available] => MongoDate Object
26                 (
27                     [sec] => 1290669462
28                     [usec] => 0
29                 )
30       [weight] => 0.48
31       [status] => 1
32       [tax_class_id] => 1
33       [manufacturers_id] => 5
34       [products_ordered] => 0
35       [details] => Array
36       (
37         [name] => 'Dummy Product'
38         [description] => 'Dummy Product Description'
39         [url] => 'www.dummy.com'
40         [viewed] => 0
```

```
41          [language_id] => 1
42        )
43
44      )
45 )
```

Ist die Abfrage-Query etwas komplexer gestaltet, wird über die Mongo-Wrapper-Klasse hinweg direkt auf dessen MongoDB-Instanz zugegriffen und damit weitergearbeitet:

```
1 $products_cursor = $mongoP->collection->find(array('status' => 1))->sort
      (array('date_added' => -1))->limit(100);
```

Updates sind ebenfalls einfach zu implementieren. Wichtig ist hier der Parameter *$set*. Ohne den würden die restlichen Daten im Dokument gelöscht werden und nur der aktualisierte Wert würde darin enthalten sein.

```
1 $mongoP->update(array('id' => 15, array('$set'=>array('details.viewed'
      =>((1))), array('upsert' => true));
```

Um ein Produkt aus der Collection zu löschen, wird ein *delete* benötigt. Durch die Übergabe eines Arrays werden alle mit dessen Inhalt übereinstimmenden Objekte aus der Datenbank gelöscht.

```
1 $mongoP->delete(array('id' => 15));
```

5.4.2 Probleme im Laufe der Implementierung

Das Hauptproblem war die Arbeit mit zwei verschiedenen Datenbankmanagementsystemen. Besonders war dabei zu beachten, dass jedes neu angelegte Objekt in der MongoDB-Datenbank ebenfalls auch in der MySQL-Datenbank mit einer fortlaufenden ID und den richtigen Foreign-Keys gespeichert werden muss. Selbes gilt bei der Löschung eines Objektes. Dies muss neben der MongoDB-Datenbank ebenfalls auch in der MySQL-Datenbank vollzogen werden.

In Hinblick auf die Evaluierung der MongoDB-Datenbank war es wichtig, alle Daten in beiden Datenbanken im selben Muster abzufragen. Ein Beispiel hierbei ist die detaillierte Suche nach Produkten. Im osCommerce-Shop mit der MySQL und MongoDB-Datenbank wird dabei folgendermaßen vorgegangen:

1. MongoDB-Abfrage über die Produkte betreffend Preis, Veröffentlichungsdatum, Name und Beschreibung
2. MySQL-Abfrage innerhalb der von MongoDB zurückgelieferten Produkt-IDs betreffend Kategorie und Hersteller
3. MongoDB-Abfrage um Detailinformationen der weiters aus der MySQL-Abfrage gefilterten Produkt-IDs zu erhalten.

Dieser Ablauf musste im originalen osCommerce-Shop mit MySQL ebenfalls gleich erfolgen, um die Abfragezeiten gezielt vergleichen zu können. Um

den sehr komplexen Ablauf zu vereinfachen, wurde die Schlüsselwort-Suche auf Produktnamen sowie Produktbeschreibungen begrenzt und somit die Suche nach Herstellern vernachlässigt. Auf die Berücksichtigung eines reduzierten Preises wurde bei der neu implementierten Suche ebenfalls verzichtet, um die Implementierung der Query-Abfragen nicht noch mehr zu erschweren.

Alle osCommerce-Tabellen in MongoDB-Collections zu konvertieren wäre in Hinblick auf die Implementierung zwar sinnvoller, der mit sich bringende Aufwand im Zuge des Masterprojektes jedoch nicht gerechtfertigt gewesen. Letztendlich ist aus den in die MongoDB ausgelagerten Produkt- sowie auch Kundendaten schon klar zu erkennen wie sich diese Umstellung auf die Performance eines E-Commerce Shops auswirkt.

Kapitel 6

Evaluierung und Diskussion

6.1 Auswertungsumfeld

6.1.1 Hardwareanforderungen

Die Auswertung der Daten fand über eine lokale Apache-Installation statt. Der PC lief mit einem 1,30 Ghz Intel Core 2 U7300 Prozessor und 4 GB RAM. Das darauf verwendete 64Bit-Betriebssystem war Windows 7.

Da neben den Auswertungsläufen auch andere Prozesse abgearbeitet wurden, welche ebenfalls Ressourcen der CPU und des Hauptspeichers benötigten, können die ermittelten Zeitwerte nicht exakt nachvollzogen werden, sie geben jedoch für dieses System Richtwerte für den Vergleich untereinander vor. Ebenso können diese Werte nicht mit einem auf Linux basierenden Datenbankserver verglichen werden, welcher rein für diese Zwecke optimiert ist.

6.1.2 Datenbankdimension

Für die Auswertung des EAV-Modells reichten schon mehrere hundert Produkte aus, um Unterschiede in der Performance im Vergleich zum relationalen Ansatz zu erkennen. Für die Performance-Auswertung der MongoDB-Datenbank wurden über 25.000 Produkte in beide Datenbanken eingespeist.

6.1.3 Abgrenzung

Ziel war es, rein die Performance der unterschiedlichen Datenbankstrukturen sowie -systeme zu testen. Das Verhalten bei einer Partitionierung, horizontalen Skalierung sowie auch bei auftretenden Fehlern wurde hierbei nicht überprüft. Ein Lasttest mit mehreren gleichzeitigen Zugriffen wurde ebenfalls nicht durchgeführt, alle Zeitwerte gelten somit für einen einzelnen User, welcher sich in einem Online-Shop bewegt. Grund dafür war der schon bei einem einzelnen User ersichtliche Performanceunterschied der zu vergleichenden Datenbanken.

6.2 Praktisch angewandte Datenbankstrukturen und -systeme

6.2.1 Vorgehensweise

Um die Ergebnisse bezüglich Ausführungszeit der verschiedenen Ansätze in der Datenbank messen zu können, wurden alle vom jeweiligen Datenbanksystem abgesetzten Queries mitgeloggt.

Für das Logging der MySQL-Datenbank musste der Parameter, ab welcher Ausführungszeit ein Query als Slow-query gilt, angepasst werden, da nur der MySQL Slow-query-Log diese sowie auch die jeweiligen Locking-Zeiten mitspeichert. Dies ist in der MySQL-Konfigurationsdatei „my.ini“ vorzunehmen.

```
1 long_query_time = 0
2 log_error = "C:/xampp/mysql/data/mysql.err"
3 pid_file = "mysql.pid"
4 general_log = 1
5 general_log_file = "C:/xampp/mysql/data/mysql.log"
6 slow_query_log = 1
7 slow_query_log_file = "C:/xampp/mysql/data/mysql-slow.log"
```

Der Parameter *„long_query_log“* gibt an, ab welcher Ausführungszeit in Sekunden die Query als *„slow-Query“* gilt. *„log_error“* bezeichnet die Datei, in welche alle auftretenden Fehler gespeichert werden. *„general_log“* bzw. *„slow_query_log“* entsprechen einer boolschen Variable. Mit dem Wert 1 ist das Logging aktiviert, mit 0 deaktiviert. Weiters kann bei den Parametern *„general_log_file“* bzw. *„slow_query_log_file“* angegeben werden, in welche Datei die dementsprechenden Logs geschrieben werden.

In MongoDB können ebenfalls alle Zugriffe auf die Datenbank mittels eines *Database Profiler*[1] mitgeloggt werden. Dieser wird über die mitgelieferte MongoDB Javascript-Shell folgendermaßen für alle Queries aktiviert:

```
1 db.setProfilingLevel(2);
```

Da diese Daten nicht in einer extra Datei, wie bei MySQL, gespeichert werden, müssen sie über die JavaScript-Shell manuell in eine Datei kopiert werden. Mit folgendem Aufruf erscheinen die aktuellsten Logs der Datenbank:

```
1 db.system.profile.find().sort({\$natural:-1})
```

Alle Log-Files der Testergebnisse wurden nach jedem Testlauf einzeln gespeichert und im weiteren Schritt ausgewertet.

6.2.2 EAV-Modell

Tabelle 6.1 zeigt einen Auszug aus der Auswertungszeit der Daten in Sekunden. Hierbei ist anzumerken, dass die Performance des EAV-Modells noch

[1] http://www.mongodb.org/display/DOCS/Database+Profiler

Tabelle 6.1: Auswertungszeiten der Daten im EAV-Modell im Vergleich zum originalen relationalen Modell unter anderem mit dem Einsatz des SQL-Caches.

	MySQL	EAV-Modell	EAV-Modell & SQL-Cache
Gesamte Ausführungszeit aller Datenbankabfragen	0,1269	35,6830	0,108006
Gesamte Ausführungszeit aller Datenbankabfragen*	0,1189	3,9362	0,106006
Durchschnittliche Ausführungszeit aller Datenbankabfragen	0,00264	0,74340	0,00284
Durchschnittliche Ausführungszeit aller Datenbankabfragen*	0,00258	0,085570	0,00283
Schnellste Datenbankabfrage	0,001	0,002	0,001
Langsamste Datenbankabfrage*	0,0100	0,4370	0,0440
*ausgenommen erweiterte Suche			

stark optimiert werden kann. Das Open-Source E-Commerce System Magento[2] demonstriert diese Möglichkeit, jedoch hat es aufgrund des Einsatzes des EAV-Modells ebenfalls noch Probleme mit der Datenbankperformance.

Auswertung EAV-Modell vs. relationales Modell

Durch die Auswertungstabelle 6.1 ist schnell ersichtlich, dass die Ausführungszeiten beim EAV-Modell teilweise beträchtlich höher waren als die des relationalen Modells. Jedoch blieben die Zeiten, bis auf ein paar wenige Ausnahmen, unter dem Zehntel-Sekunden-Bereich.

Eine sehr hohe Ausführungszeit erscheint zur Zeit noch im Fall der erweiterten Suche. Wenn Produkte nach Schlagwort, Datum, Preis, Hersteller und Kategorie gesucht werden, benötigt die Datenbank zur Ausführung der Query noch 27,921597 Sekunden. Dieser nicht tolerierbare Wert müsste noch stark reduziert werden.

[2] http://www.magentocommerce.com/

Auswertung EAV-Modell mit MySQL-Cache vs. relationales Modell

Wird der MySQL-Cache aktiviert, reduzieren sich alle Ausführungszeiten ab dem zweiten Ladevorgang auf ein Minimum von 0.001 Sekunden, da die Daten direkt aus dem Cache geladen werden. Einige Queries waren nicht auf ihre optimierte Ausführungszeit auswertbar, da sie auch nach mehrmaligem Ausführen nicht im Slow-Query-Log-File, jedoch aber im Log-File ohne Ausführungszeitangabe aufschienen. Dieser Fall trat nur beim aktivierten MySQL-Cache auf, wobei das Problem sehr wahrscheinlich durch dessen Aktivierung hervorgerufen wurde. Jedoch war schon bei der Ausführung der dementsprechenden Queries bemerkbar, dass diese eine signifikant kürzere Ladezeit benötigten, als beim ersten Durchgang. So ist anzunehmen, dass der Cache auch bei diesen Abfragen wirksam wurde. Durch diese Umstände können die Werte der Ausführungszeiten mit aktivierten MySQL-Cache in der Tabelle 6.1 nicht exakt mit den Werten der anderen beiden Spalten verglichen werden.

Der MySQL-Cache optimiert zwar Queries aus dem EAV-Modell, birgt jedoch Nachteile. So müssen alle Daten mindestens einmal nach Neustart des MySQL-Servers mit der ursprünglichen Ausführungszeit aus der Datenbank geladen werden, bevor sie im Cache gespeichert werden können. Zudem ist dieser auch nicht endlos groß und so können in einer großen Datenbank Daten im Cache überschrieben werden, welche in weiterer Folge erneut aus der Datenbank geladen werden müssen.

Grundsätzlich ist der MySQL-Cache beim EAV-Modell jedoch ein starker Faktor in Bezug auf Performanceoptimierung und sollte hierfür in der Regel aktiviert sein.

Auswertung EAV-Modell mit eAccelerator vs. relationales Modell

Da die Software eAccelerator[3] dafür bestimmt ist, PHP-Seiten zu optimieren und zu beschleunigen, war er für die Optimierung der SQL-Queries im EAV-Modell nicht hilfreich. Zwar ist der Großteil der Ausführungszeiten bei aktiviertem eAccelerator etwas niedriger als bei den original Zeiten des EAV-Modells, dies kann aber darauf zurückzuführen sein, dass dem MySQL-Server in diesem Moment mehr Rechenleistung zugewiesen wurde.

6.2.3 MongoDB

Da der Einsatz des EAV-Modells in einem relationalen Datenbankmanagementsystem hinsichtlich der schwachen Performance nicht zufriedenstellend war, wurde ebenfalls der Einsatz von MongoDB für die Verwaltung von Produkt- sowie auch Kundendaten getestet und analysiert. Tabelle 6.2 listet

[3] http://eaccelerator.net/

Tabelle 6.2: Auswertungszeiten der Daten in der MongoDB-Datenbank im Vergleich zur MySQL-Datenbank.

	MySQL	MongoDB
Gesamte Ausführungszeit aller Datenbankabfragen	4,2432	2,6511
Durchschnittliche Ausführungszeit aller Datenbankabfragen	0,1697	0,1060
Schnellste Datenbankabfrage	0,001	0
Langsamste Datenbankabfrage	0,6310	0,6400
Größte Differenz MySQL < MongoDB	0,0520	0,221
Größte Differenz MongoDB < MySQL	0,9071	0,001
25.000 Produkte einfügen (ohne Indizierung)	116	19
25.000 Produkte einfügen (mit Indizierung)	124	21
25.000 Produkte auslesen	0,1806	0,0200

eine Zusammenfassung der Auswertungsergebnisse von MySQL zu MongoDB auf. Die nachfolgenden Zeiten sind hierbei in Sekunden angegeben.

Die angegeben Werte in der Tabelle 6.2 gelten rein für Produktabfragen im Rahmen des E-Commerce Systems osCommerce. Durch die Auswertung der Performance bei einer Datenbankdimension von über 25.000 Produkten ist klar ersichtlich, dass MongoDB, obwohl es bei vielen tabellen- bzw. collectionübergreifenden Abfragen in Zusammenarbeit mit der MySQL-Datenbank steht, eindeutig schneller arbeitet. Durch die richtige Setzung von Indizes betragen die Ausführungszeiten in der MongoDB-Datenbank nicht mehr als einige Millisekunden. Einzig zwei MongoDB-Queries erhöhen diesen Schnitt, da bei diesen bewusst kein Index auf die spezielle Sortierung gesetzt wurde. Diese beiden Queries sollen demonstrieren, inwiefern sich die korrekte Setzung eines Index auf die Performance der Datenbank auswirkt.

Die höhere durchschnittliche Ausführungszeit von 0,1060 Sekunden in der Tabelle sowie die langsamste Ausführungszeit im System, wird bei MongoDB rein von langsameren, datenbanksystem-übergreifenden Queries verursacht.

Einzig die nachträglich implementierte Volltext-Suche in MySQL übertrifft MongoDB hinsichtlich ihrer Performance (s. Tabelle 6.2 Spalte „Größte

Differenz MySQL < MongoDB “).

Würde die gesamte Datenbankarchitektur des osCommerce Online-Shops auf MongoDB umgestellt werden, bedeute dies, dass die Datenbank noch schneller arbeiten könnte und abgesehen von den weiteren Vorteilen von MongoDB im Vergleich zu MySQL (s. Abschnitt 4.3.2) eine Schemafreiheit der Daten gegeben wäre. Letzterer Punkt ist besonders für Produkte von Vorteil, da diese in ihren Eigenschaften stark variieren.

Durch die im Standard-Modus eingestellte starke Konsistenzerhaltung bei MongoDB [25], können ebenfalls kritischere Abläufe, wie der Bestellvorgang in einem E-Commerce System, mit dieser Datenbank umgesetzt werden. Zu beachten ist hier jedoch, dass die Unterstützung von Transaktionen über mehrere Dokumente hinweg seitens MongoDB nicht gegeben ist. Durch die starke Konsistenz der Daten sowie der Partitionstoleranz ergibt sich eine geringere Verfügbarkeit des gesamten Systems, im Vergleich zu NoSQL-Datenbanken mit eventual consistency. Durch eine manuelle Lockerung der Konsistenz auf einzelne Collections, steigt wiederum deren Verfügbarkeit.

6.3 Theoretische Betrachtung anderer alternativer Datenbanken in Hinblick auf E-Commerce Systeme

Abgesehen von einer dokumentorientierten Datenbank sind auch andere Datenbankgruppen für den teilweisen Einsatz in einem E-Commerce System geeignet. Da diese im Rahmen des Masterprojektes jedoch nicht auf ihre praktischen Vor- bzw. auch Nachteile getestet wurden, muss auf das theoretische Wissen vertraut werden.

Generell gilt, dass besonders Bestellvorgänge in einem Online-Shop auf eine starke Konsistenz der Daten bauen sollten, da mit der Lockerung dieser womöglich die Kundenzufriedenheit geschmälert wird. Somit sollten im Hinblick auf die Konsistenz der Daten nur ACID-transaktionale Datenbanken kritische Daten verwalten.

6.3.1 Key-Value Datenbanken

Ein einfacher Key-Value Store könnte die relationale Datenbank eines E-Commerce Systems zwar ablösen, da die Vorteile der Schemafreiheit und der hohen Performance gegeben sind, jedoch bringen sie durch ihre komplette Strukturlosigkeit und den damit verbundenen komplizierteren Abfragen, im Vergleich zu dokumentorientierten Datenbanken, mehr Nachteile als Vorteile mit sich.

Ein mögliches Einsatzgebiet in einem Online-Shop könnte die Speicherung und Verwaltung eines beständigen Caches sein, da hier nur weitgehend einfache Abfragen getätigt werden müssen.

6.3.2 Spaltenorientierte Datenbanken

Generell gesehen kann eine spaltenorientierte Datenbank ebenfalls Daten eines E-Commerce Systems verwalten. Durch die Eigenschaft nur Attribute auszulesen, welche aktuell benötigt werden, ist eine spaltenorientierte Datenbank im Vergleich zu einer relationalen, welche immer die gesamte Tabelle durchsucht, performanter beim Lesen von Daten. Der Nachteil liegt jedoch in der schwachen Schreibperformance. Da in einem gut laufenden großen Online-Shop täglich ebenfalls auch viele Schreiboperationen, besonders auf die Produkt-, Bestell- sowie auch Kundentabellen erfolgen, bleibt einer spaltenorientierten Datenbank im Vergleich zu einer zeilenorientierten, relationalen Datenbank nur noch der Vorteil der Schemafreiheit.

6.3.3 Graphdatenbanken

Graphdatenbanken sind durch ihren Aufbau von Knoten und Kanten nicht für ein gesamtes E-Commerce System einsetzbar. Vorstellbar wäre die Analyse und Speicherung von kundenspezifischen Produktempfehlungen sowie Zahlungsarten [38].

Produktempfehlungen basieren dabei auf der Bestellhistorie des jeweiligen Kunden in Verbindung mit Produkten, welche andere Kunden ebenfalls mit den Produkten in der jeweiligen Bestellhistorie gekauft haben [38]. Weitere andere Einflussfaktoren spielen dabei ebenfalls eine Rolle. Diese Vernetzungen können als Graph mit Knoten für die Produkte und Kanten für die Verknüpfungen untereinander kompakt und performant in einer Graphdatenbank gespeichert werden. Somit kann der Betreiber eines Online-Shops mit dem Einsatz von auf den Kunden zugeschnittenen Produktempfehlungen seinen Umsatz steigern und durch die geringere Speicherkapazität dieser Daten und höherer Datenbank-Performance Kosten für neue Hardware sparen.

Kapitel 7

Schlussbemerkung

In dieser Masterarbeit wurden verschiedene Ansätze von alternativen Datenbankstrukturen sowie auch -systemen im Einsatz bei E-Commerce Systemen analysiert und diskutiert. Zudem wird auf die theoretischen Grundlagen dieser Datenbanksysteme eingegangen und ein Überblick über die populärsten im Einsatz befindlichen Systeme, welche sich der Gruppe der NoSQL-Datenbanken zuschreiben lassen, geschaffen.

Besonders bei der Datenmenge von großen Online-Shops wie Amazon[1] stoßen relationale Datenbankmanagementsysteme in Bezug auf die horizontale Skalierung, Replikation und performante Verarbeitung schnell an ihre Grenzen. Um zudem eine, für ein E-Commerce System vorteilhafte, flexible Datenstruktur innerhalb der jeweiligen Datenbank aufbauen zu können, muss der traditionell relationale Ansatz großteils verworfen und eine Alternative gefunden werden.

Der Einsatz des EAV-Modells innerhalb eines relationalen Datenbankmodells bietet hierbei eine hohe Flexibilität der Datenstruktur, jedoch zum Leidwesen der Performance, welche auf komplexere Queries in Verbindung mit einer hohen Anzahl von JOIN-Anweisungen zurückzuführen ist. Im Vergleich dazu behebt schon eine teilweise Auslagerung der Daten in die dokumentorientierte Datenbank MongoDB nahezu alle Probleme der relationalen Datenbank in Bezug auf die Flexibilität der Datenstruktur und Performance. Im Hinblick auf die Konsistenzerhaltung der Daten ermöglicht MonoDB, im Vergleich zu vielen anderen NoSQL-Vertretern, eine starke Konsistenz. Somit können auch kritische Datenbankanweisungen, wie Bestellvorgänge, mittels MongoDB vollzogen werden.

Andere Untergruppen der NoSQL-Datenbanken lassen sich ebenfalls für den teilweisen Einsatz in einem E-Commerce System verwenden, können auf Grund ihrer Eigenschaften ein relationales Datenbankmodell im E-Commerce Bereich jedoch nicht komplett ablösen. Das EAV-Modell ermöglicht zwar flexiblere Datenstrukturen in einer relationalen Datenbank, jedoch scheidet es

[1] http://www.amazon.com

aufgrund seiner schwachen Performance im praktischen Teil dieser Arbeit als Alternative zu einem strikt relationalen Datenbankmodell aus. Um diese Performance zu optimieren, müsste die Implementierung des EAV-Modells noch mehr an die Methoden von Magento angepasst werden. Letztendlich wird es aber die Performance einer normal relationalen Datenbank niemals erreichen.

Ein sinnvollerer Ansatz ist die Übernahme der gesamten Datenverwaltung eines E-Commerce Systems durch MongoDB. Damit kann dessen Performance noch mehr optimiert und zudem die Last über ein Rechnercluster mittels Replikationen und Sharding verteilt werden. Dass MongoDB sehr wohl eine starke Alternative zu relationalen Datenbankmanagementsystemen im E-Commerce Bereich ist, demonstriert das Start-Up Unternehmen *Open Sky*[2].

Abzuwarten bleibt, wie lange aktuell bestehende E-Commerce Lösungen noch auf ein relationales Datenbankmanagementsystem bauen, wenn unter den NoSQL-Vertretern jetzt schon hoch entwickelte Systeme den Problemen des relationalen Modells Abhilfe schaffen können.

[2] https://opensky.com/

Anhang A

Datenbankmodell in Magento

MAGENTO - Database Diagram [v1.3.2.4] - revised 10/02/2009

LEGEND

- Area: System
- Area: Product
- Area: Customer
- Area: EAV
- Area: Sales
- Reference Tables [Static]
- Reference Tables [Dynamic]
- Entity Tables
- Entity Detail Tables [by DataType]
- "Important Looking" Tables
- Cross-Reference Tables
- Other Tables
- CORE_WEBSITE
- CORE_STORE
- EAV_ENTITY_TYPE
- CUSTOMER_GROUP
- CATALOG_PRODUCT_ENTITY
- CATALOG_CATEGORY_ENTITY
- CUSTOMER_ENTITY
- EAV_ATTRIBUTE
- EAV_ATTRIBUTE_SET
- COUNTRY
- Recently Added Tables

SYSTEM

ADMIN

admin_role
admin_user
adminnotification_inbox
admin_rule
admin_assert

CORE CONFIGURATION

core_email_template
core_resource
core_flag
core_config_data
cron_schedule

SYSTEM API

api_role
api_user
api_assert
api_rule
api_session

EVENT REPORTER

report_event_types
report_event

LOGS

log_summary_type
log_summary
log_url_info
log_url
log_visitor_info
log_visitor
log_visitor_online
log_customer
log_quote
sendfriend_log

TAG / GOOGLE

tag
tag_summary
tag_relation
googlebase_items
googleoptimizer_code
googlebase_types
googlebase_attributes

WEBSITE / STORE

core_website
core_session
core_store_group
core_store
core_translate

SYSTEM LAYOUT

core_url_rewrite
cms_page
core_layout_update
cms_block
core_layout_link
design_change
cms_page_store
cms_block_store

DIRECTORY

directory_country
directory_country_region
directory_country_region_name
directory_currency_rate
directory_country_format

DATA FLOW

dataflow_profile
dataflow_profile_history
dataflow_batch
dataflow_batch_import
dataflow_batch_export
dataflow_session
dataflow_import_data

CATALOG RULE

catalogrule
catalogrule_product_price
catalogrule_product

CATALOG INDEX

catalogindex_price
catalogindex_eav
catalogindex_minimal_price
catalogindex_aggregation_tag
catalogindex_aggregation
catalogindex_aggregation_to_tag

CATALOG QUERY

catalogsearch_query
catalogsearch_fulltext
catalogsearch_result
catalog_compare_item

CUSTOMER GROUP

customer_group

CUSTOMER NEWSLETTER

newsletter_template
newsletter_queue
newsletter_queue_link
newsletter_problem
newsletter_subscriber
newsletter_queue_store_link

EAV ENTITY STORE

eav_entity_store

EAV ENTITY TYPE

eav_entity_type

SALES RULE

salesrule
salesrule_customer

SALES ORDER

sales_order
sales_flat_order_item

SALES QUOTE

sales_flat_quote
sales_flat_quote_address

CATEGORY

catalog_category_product
catalog_category_product_index
catalog_category_flat

PRODUCT

catalog_product_entity_tier_price
catalog_product_enabled_index
catalog_product_website

CATEGORY ENTITY

catalog_category_entity
catalog_category_entity_datetime
catalog_category_entity_decimal
catalog_category_entity_int
catalog_category_entity_text
catalog_category_entity_varchar

PRODUCT ENTITY

catalog_product_entity
catalog_product_entity_datetime
catalog_product_entity_decimal
catalog_product_entity_int
catalog_product_entity_text
catalog_product_entity_varchar

PRODUCT LINK

catalog_product_super_link
catalog_product_link_type
catalog_product_link
catalog_product_link_attribute
catalog_product_link_attribute_decimal
catalog_product_link_attribute_int
catalog_product_link_attribute_varchar

CUSTOMER ENTITY

customer_entity
customer_entity_datetime
customer_entity_decimal
customer_entity_int
customer_entity_text
customer_entity_varchar

CUSTOMER ADDRESS ENTITY

customer_address_entity
customer_address_entity_datetime
customer_address_entity_decimal
customer_address_entity_int
customer_address_entity_text
customer_address_entity_varchar

EAV ENTITY

eav_entity
eav_entity_datetime
eav_entity_decimal
eav_entity_int
eav_entity_text
eav_entity_varchar

SALES ORDER ENTITY

sales_order_entity
sales_order_entity_datetime
sales_order_entity_decimal
sales_order_entity_int
sales_order_entity_text
sales_order_entity_varchar

[DOWNLOADABLE]

downloadable_link_purchased
downloadable_link_purchased_item

SALES TRANSACTION SETUP

GIFT / GIFT CERTIFICATE

gift_message

SHIPPING

shipping_tablerate

WEEE TAX (electronic waste tax)

weee_tax
weee_discount

TAX

tax_calculation_rule
tax_class
tax_calculation_rate
tax_calculation_rate_title
tax_calculation

CHECKOUT AGREEMENT

checkout_agreement_store
checkout_agreement

PAYMENT OPTIONS

googlecheckout_api_debug
paypal_api_debug
paypaluk_api_debug
paygate_authorizenet_debug
amazonpayments_api_debug

PRODUCT

PRODUCT OPTION

catalog_product_option
catalog_product_option_title
catalog_product_option_price
catalog_product_option_type_value
catalog_product_option_type_title
catalog_product_option_type_price

PRODUCT BUNDLE

catalog_product_bundle_option
catalog_product_bundle_option_value
catalog_product_bundle_selection
catalog_product_bundle_price_index

PRODUCT SUPER ATTRIBUTE

catalog_product_super_attribute
catalog_product_super_attribute_label
catalog_product_super_attribute_pricing

PRODUCT INVENTORY

cataloginventory_stock
cataloginventory_stock_item
cataloginventory_stock_status

DOWNLOADABLE

downloadable_link_price
downloadable_link_title
downloadable_link
downloadable_sample_title
downloadable_sample

PRODUCT GALLERY

catalog_product_entity_gallery
catalog_product_entity_media_gallery
catalog_product_entity_media_gallery_value

PRODUCT ALERT

product_alert_price
product_alert_stock

CUSTOMER

REVIEW

review_entity
review_status
review_store
review
review_entity_summary
review_detail

RATING

rating_entity
rating
rating_option
rating_option_vote
rating_store
rating_title
rating_option_vote_aggregated

POLL

poll_store
poll
poll_answer
poll_vote

WISH LIST

wishlist
wishlist_item

EAV

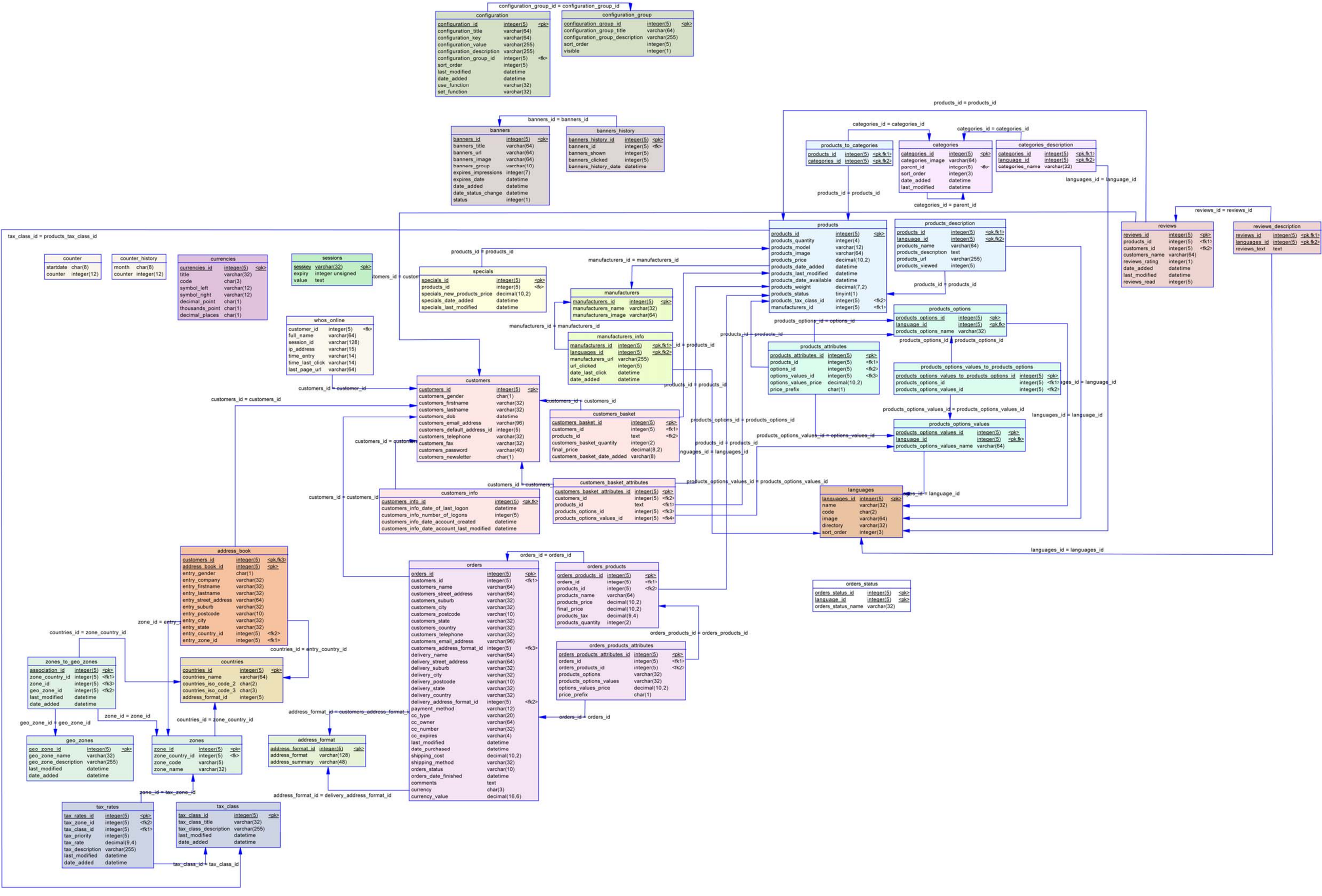
EAV ATTRIBUTE

eav_attribute_set
eav_attribute_group
eav_attribute
eav_attribute
eav_attribute_option
eav_attribute_option_value

SALES

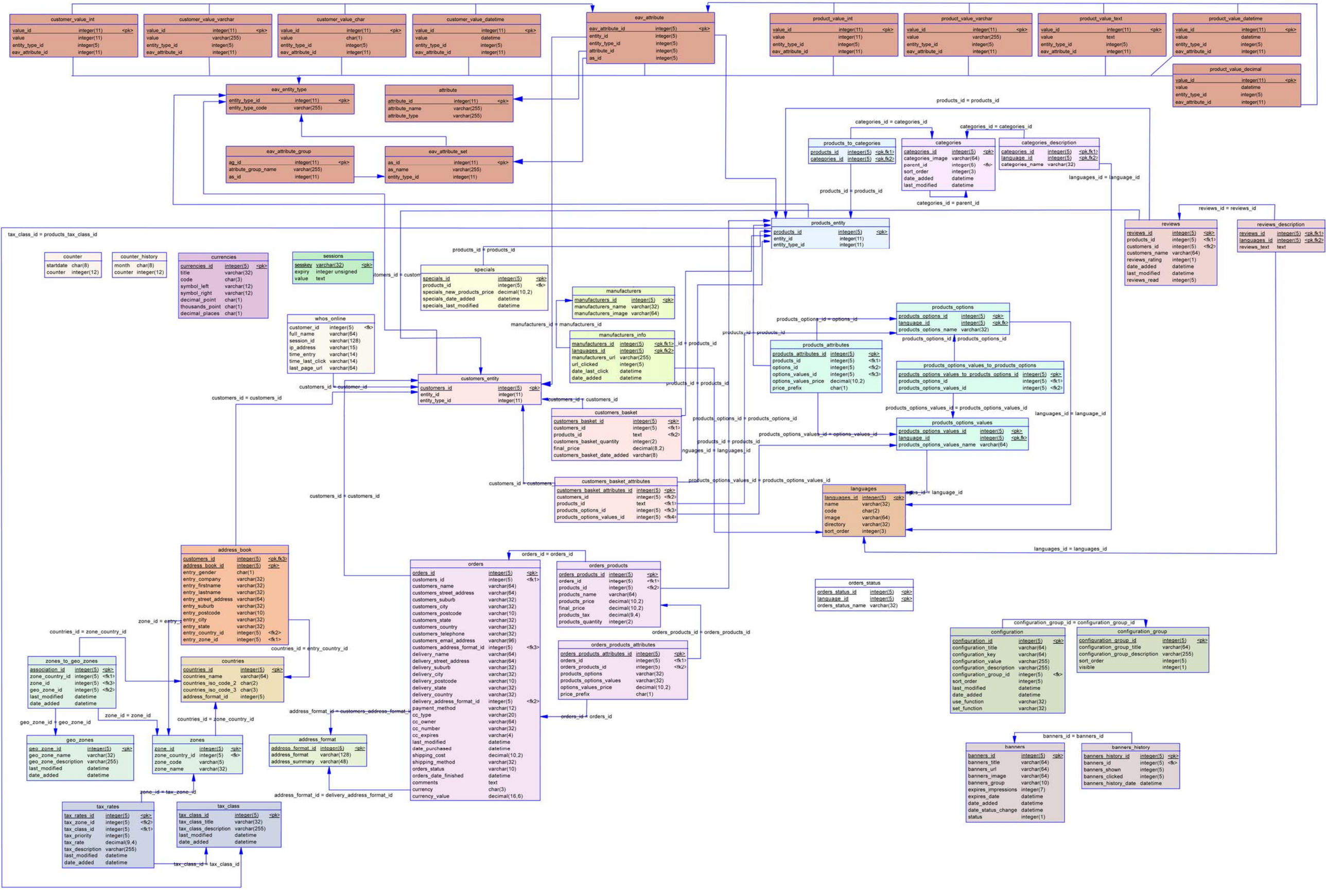
Anhang B

Datenbankmodell osCommerce Original


configuration_group_id = configuration_group_id
configuration
configuration_id integer(5) <pk>
configuration_title varchar(64)
configuration_key varchar(64)
configuration_value varchar(255)
configuration_description varchar(255)
configuration_group_id integer(5) <fk>
sort_order integer(5)
last_modified datetime
date_added datetime
use_function varchar(32)
set_function varchar(32)
configuration_group
configuration_group_id integer(5) <pk>
configuration_group_title varchar(64)
configuration_group_description varchar(255)
sort_order integer(5)
visible integer(1)
banners_id = banners_id
banners
banners_id integer(5) <pk>
banners_title varchar(64)
banners_url varchar(64)
banners_image varchar(64)
banners_group varchar(10)
expires_impressions integer(7)
expires_date datetime
date_added datetime
date_status_change datetime
status integer(1)
banners_history
banners_history_id integer(5) <pk>
banners_id integer(5) <fk>
banners_shown integer(5)
banners_clicked integer(5)
banners_history_date datetime
products_id = products_id
categories_id = categories_id
categories_id = categories_id
products_to_categories
products_id integer(5) <pk,fk1>
categories_id integer(5) <pk,fk2>
categories
categories_id integer(5) <pk>
categories_image varchar(64)
parent_id integer(5) <fk>
sort_order integer(3)
date_added datetime
last_modified datetime
categories_description
categories_id integer(5) <pk,fk1>
language_id integer(5) <pk,fk2>
categories_name varchar(32)
languages_id = language_id
products_id = products_id
categories_id = parent_id
reviews_id = reviews_id
reviews
reviews_id integer(5) <pk>
products_id integer(5) <fk1>
customers_id integer(5) <fk2>
customers_name varchar(64)
reviews_rating integer(1)
date_added datetime
last_modified datetime
reviews_read integer(5)
reviews_description
reviews_id integer(5) <pk,fk1>
languages_id integer(5) <pk,fk2>
reviews_text text
tax_class_id = products_tax_class_id
products
products_id integer(5) <pk>
products_quantity integer(4)
products_model varchar(12)
products_image varchar(64)
products_price decimal(10,2)
products_date_added datetime
products_last_modified datetime
products_date_available datetime
products_weight decimal(7,2)
products_status tinyint(1)
products_tax_class_id integer(5) <fk2>
manufacturers_id integer(5) <fk1>
products_description
products_id integer(5) <pk,fk1>
language_id integer(5) <pk,fk2>
products_name varchar(64)
products_description text
products_url varchar(255)
products_viewed integer(5)
products_id = products_id
counter
startdate char(8)
counter integer(12)
counter_history
month char(8)
counter integer(12)
currencies
currencies_id integer(5) <pk>
title varchar(32)
code char(3)
symbol_left varchar(12)
symbol_right varchar(12)
decimal_point char(1)
thousands_point char(1)
decimal_places char(1)
sessions
sesskey varchar(32) <pk>
expiry integer unsigned
value text
products_id = products_id
specials
specials_id integer(5) <pk>
products_id integer(5) <fk>
specials_new_products_price decimal(10,2)
specials_date_added datetime
specials_last_modified datetime
manufacturers_id = manufacturers_id
manufacturers
manufacturers_id integer(5) <pk>
manufacturers_name varchar(32)
manufacturers_image varchar(64)
manufacturers_id = manufacturers_id
manufacturers_info
manufacturers_id integer(5) <pk,fk1>
languages_id integer(5) <pk,fk2>
manufacturers_url varchar(255)
url_clicked integer(5)
date_last_click datetime
date_added datetime
products_options_id = options_id
products_options
products_options_id integer(5) <pk>
language_id integer(5) <pk,fk>
products_options_name varchar(32)
products_options_id = products_options_id
products_attributes
products_attributes_id integer(5) <pk>
products_id integer(5) <fk1>
options_id integer(5) <fk2>
options_values_id integer(5) <fk3>
options_values_price decimal(10,2)
price_prefix char(1)
products_options_values_to_products_options
products_options_values_to_products_options_id integer(5) <pk>
products_options_id integer(5) <fk1>
products_options_values_id integer(5) <fk2>
products_options_values_id = products_options_values_id
products_options_values
products_options_values_id integer(5) <pk>
language_id integer(5) <pk,fk>
products_options_values_name varchar(64)
whos_online
customer_id integer(5) <fk>
full_name varchar(64)
session_id varchar(128)
ip_address varchar(15)
time_entry varchar(14)
time_last_click varchar(14)
last_page_url varchar(64)
customers_id = customer_id
customers
customers_id integer(5) <pk>
customers_gender char(1)
customers_firstname varchar(32)
customers_lastname varchar(32)
customers_dob datetime
customers_email_address varchar(96)
customers_default_address_id integer(5)
customers_telephone varchar(32)
customers_fax varchar(32)
customers_password varchar(40)
customers_newsletter char(1)
customers_id = customers_id
customers_basket
customers_basket_id integer(5) <pk>
customers_id integer(5) <fk1>
products_id text <fk2>
customers_basket_quantity integer(2)
final_price decimal(8,2)
customers_basket_date_added varchar(8)
languages_id = languages_id
customers_basket_attributes
customers_basket_attributes_id integer(5) <pk>
customers_id integer(5) <fk2>
products_id text <fk1>
products_options_id integer(5) <fk3>
products_options_values_id integer(5) <fk4>
customers_info
customers_info_id integer(5) <pk,fk>
customers_info_date_of_last_logon datetime
customers_info_number_of_logons integer(5)
customers_info_date_account_created datetime
customers_info_date_account_last_modified datetime
languages
languages_id integer(5) <pk>
name varchar(32)
code char(2)
image varchar(64)
directory varchar(32)
sort_order integer(3)
languages_id = language_id
languages_id = languages_id
address_book
customers_id integer(5) <pk,fk3>
address_book_id integer(5) <pk>
entry_gender char(1)
entry_company varchar(32)
entry_firstname varchar(32)
entry_lastname varchar(32)
entry_street_address varchar(64)
entry_suburb varchar(32)
entry_postcode varchar(10)
entry_city varchar(32)
entry_state varchar(32)
entry_country_id integer(5) <fk2>
entry_zone_id integer(5) <fk1>
orders_id = orders_id
orders
orders_id integer(5) <pk>
customers_id integer(5) <fk1>
customers_name varchar(64)
customers_street_address varchar(64)
customers_suburb varchar(32)
customers_city varchar(32)
customers_postcode varchar(10)
customers_state varchar(32)
customers_country varchar(32)
customers_telephone varchar(32)
customers_email_address varchar(96)
customers_address_format_id integer(5) <fk3>
delivery_name varchar(64)
delivery_street_address varchar(64)
delivery_suburb varchar(32)
delivery_city varchar(32)
delivery_postcode varchar(10)
delivery_state varchar(32)
delivery_country varchar(32)
delivery_address_format_id integer(5) <fk2>
payment_method varchar(12)
cc_type varchar(20)
cc_owner varchar(64)
cc_number varchar(32)
cc_expires varchar(4)
last_modified datetime
date_purchased datetime
shipping_cost decimal(10,2)
shipping_method varchar(32)
orders_status varchar(10)
orders_date_finished datetime
comments text
currency char(3)
currency_value decimal(16,6)
orders_products
orders_products_id integer(5) <pk>
orders_id integer(5) <fk1>
products_id integer(5) <fk2>
products_name varchar(64)
products_price decimal(10,2)
final_price decimal(10,2)
products_tax decimal(9,4)
products_quantity integer(2)
orders_products_id = orders_products_id
orders_products_attributes
orders_products_attributes_id integer(5) <pk>
orders_id integer(5) <fk1>
orders_products_id integer(5) <fk2>
products_options varchar(32)
products_options_values varchar(32)
options_values_price decimal(10,2)
price_prefix char(1)
orders_id = orders_id
orders_status
orders_status_id integer(5) <pk>
language_id integer(5) <pk>
orders_status_name varchar(32)
countries_id = zone_country_id
zones_to_geo_zones
association_id integer(5) <pk>
zone_country_id integer(5) <fk1>
zone_id integer(5) <fk3>
geo_zone_id integer(5) <fk2>
last_modified datetime
date_added datetime
countries
countries_id integer(5) <pk>
countries_name varchar(64)
countries_iso_code_2 char(2)
countries_iso_code_3 char(3)
address_format_id integer(5)
zone_id = zone_id
geo_zone_id = geo_zone_id
countries_id = zone_country_id
address_format_id = customers_address_format_id
geo_zones
geo_zone_id integer(5) <pk>
geo_zone_name varchar(32)
geo_zone_description varchar(255)
last_modified datetime
date_added datetime
zones
zone_id integer(5) <pk>
zone_country_id integer(5) <fk>
zone_code varchar(5)
zone_name varchar(32)
address_format
address_format_id integer(5) <pk>
address_format varchar(128)
address_summary varchar(48)
zone_id = tax_zone_id
address_format_id = delivery_address_format_id
tax_rates
tax_rates_id integer(5) <pk>
tax_zone_id integer(5) <fk2>
tax_class_id integer(5) <fk1>
tax_priority integer(5)
tax_rate decimal(9,4)
tax_description varchar(255)
last_modified datetime
date_added datetime
tax_class
tax_class_id integer(5) <pk>
tax_class_title varchar(32)
tax_class_description varchar(255)
last_modified datetime
date_added datetime
tax_class_id = tax_class_id

Anhang C

Datenbankmodell osCommerce mit EAV-Modell


customer_value_int
value_id integer(11) <pk>
value integer(11)
entity_type_id integer(5)
eav_attribute_id integer(11)
customer_value_varchar
value_id integer(11) <pk>
value varchar(255)
entity_type_id integer(5)
eav_attribute_id integer(11)
customer_value_char
value_id integer(11) <pk>
value char(1)
entity_type_id integer(5)
eav_attribute_id integer(11)
customer_value_datetime
value_id integer(11) <pk>
value datetime
entity_type_id integer(5)
eav_attribute_id integer(11)
eav_attribute
eav_attribute_id integer(5) <pk>
entity_id integer(5)
entity_type_id integer(5)
attribute_id integer(5)
as_id integer(5)
product_value_int
value_id integer(11) <pk>
value integer(11)
entity_type_id integer(5)
eav_attribute_id integer(11)
product_value_varchar
value_id integer(11) <pk>
value varchar(255)
entity_type_id integer(5)
eav_attribute_id integer(11)
product_value_text
value_id integer(11) <pk>
value text
entity_type_id integer(5)
eav_attribute_id integer(11)
product_value_datetime
value_id integer(11) <pk>
value datetime
entity_type_id integer(5)
eav_attribute_id integer(11)
product_value_decimal
value_id integer(11) <pk>
value datetime
entity_type_id integer(5)
eav_attribute_id integer(11)
eav_entity_type
entity_type_id integer(11) <pk>
entity_type_code varchar(255)
attribute
attribute_id integer(11) <pk>
attribute_name varchar(255)
attribute_type varchar(255)
eav_attribute_group
ag_id integer(11) <pk>
atribute_group_name varchar(255)
as_id integer(11)
eav_attribute_set
as_id integer(11) <pk>
as_name varchar(255)
entity_type_id integer(11)
products_id = products_id
categories_id = categories_id
categories_id = categories_id
products_to_categories
products_id integer(5) <pk,fk1>
categories_id integer(5) <pk,fk2>
categories
categories_id integer(5) <pk>
categories_image varchar(64)
parent_id integer(5) <fk>
sort_order integer(3)
date_added datetime
last_modified datetime
categories_description
categories_id integer(5) <pk,fk1>
language_id integer(5) <pk,fk2>
categories_name varchar(32)
languages_id = language_id
products_id = products_id
categories_id = parent_id
reviews_id = reviews_id
products_entity
products_id integer(5) <pk>
entity_id integer(11)
entity_type_id integer(11)
reviews
reviews_id integer(5) <pk>
products_id integer(5) <fk1>
customers_id integer(5) <fk2>
customers_name varchar(64)
reviews_rating integer(1)
date_added datetime
last_modified datetime
reviews_read integer(5)
reviews_description
reviews_id integer(5) <pk,fk1>
languages_id integer(5) <pk,fk2>
reviews_text text
tax_class_id = products_tax_class_id
counter
startdate char(8)
counter integer(12)
counter_history
month char(8)
counter integer(12)
currencies
currencies_id integer(5) <pk>
title varchar(32)
code char(3)
symbol_left varchar(12)
symbol_right varchar(12)
decimal_point char(1)
thousands_point char(1)
decimal_places char(1)
sessions
sesskey varchar(32) <pk>
expiry integer unsigned
value text
products_id = products_id
specials
specials_id integer(5) <pk>
products_id integer(5) <fk>
specials_new_products_price decimal(10,2)
specials_date_added datetime
specials_last_modified datetime
manufacturers
manufacturers_id integer(5) <pk>
manufacturers_name varchar(32)
manufacturers_image varchar(64)
manufacturers_id = manufacturers_id
manufacturers_info
manufacturers_id integer(5) <pk,fk1>
languages_id integer(5) <pk,fk2>
manufacturers_url varchar(255)
url_clicked integer(5)
date_last_click datetime
date_added datetime
products_id = products_id
products_options
products_options_id integer(5) <pk>
language_id integer(5) <pk,fk>
products_options_name varchar(32)
products_options_id = options_id
products_id = products_id
products_options_id = products_options_id
products_attributes
products_attributes_id integer(5) <pk>
products_id integer(5) <fk1>
options_id integer(5) <fk2>
options_values_id integer(5) <fk3>
options_values_price decimal(10,2)
price_prefix char(1)
products_options_values_to_products_options
products_options_values_to_products_options_id integer(5) <pk>
products_options_id integer(5) <fk1>
products_options_values_id integer(5) <fk2>
products_options_values_id = products_options_values_id
languages_id = language_id
products_options_values
products_options_values_id integer(5) <pk>
language_id integer(5) <pk,fk>
products_options_values_name varchar(64)
products_options_values_id = options_values_id
languages_id = languages_id
whos_online
customer_id integer(5) <fk>
full_name varchar(64)
session_id varchar(128)
ip_address varchar(15)
time_entry varchar(14)
time_last_click varchar(14)
last_page_url varchar(64)
customers_id = customer_id
customers_entity
customers_id integer(5) <pk>
entity_id integer(11)
entity_type_id integer(11)
customers_id = customers_id
customers_basket
customers_basket_id integer(5) <pk>
customers_id integer(5) <fk1>
products_id text <fk2>
customers_basket_quantity integer(2)
final_price decimal(8,2)
customers_basket_date_added varchar(8)
customers_id = customers_id
customers_basket_attributes
customers_basket_attributes_id integer(5) <pk>
customers_id integer(5) <fk2>
products_id text <fk1>
products_options_id integer(5) <fk3>
products_options_values_id integer(5) <fk4>
products_options_values_id = products_options_values_id
languages_id = language_id
languages
languages_id integer(5) <pk>
name varchar(32)
code char(2)
image varchar(64)
directory varchar(32)
sort_order integer(3)
languages_id = languages_id
address_book
customers_id integer(5) <pk,fk3>
address_book_id integer(5) <pk>
entry_gender char(1)
entry_company varchar(32)
entry_firstname varchar(32)
entry_lastname varchar(32)
entry_street_address varchar(64)
entry_suburb varchar(32)
entry_postcode varchar(10)
entry_city varchar(32)
entry_state varchar(32)
entry_country_id integer(5) <fk2>
entry_zone_id integer(5) <fk1>
zone_id = entry_zone_id
countries_id = entry_country_id
orders
orders_id integer(5) <pk>
customers_id integer(5) <fk1>
customers_name varchar(64)
customers_street_address varchar(64)
customers_suburb varchar(32)
customers_city varchar(32)
customers_postcode varchar(10)
customers_state varchar(32)
customers_country varchar(32)
customers_telephone varchar(32)
customers_email_address varchar(96)
customers_address_format_id integer(5) <fk3>
delivery_name varchar(64)
delivery_street_address varchar(64)
delivery_suburb varchar(32)
delivery_city varchar(32)
delivery_postcode varchar(10)
delivery_state varchar(32)
delivery_country varchar(32)
delivery_address_format_id integer(5) <fk2>
payment_method varchar(12)
cc_type varchar(20)
cc_owner varchar(64)
cc_number varchar(32)
cc_expires varchar(4)
last_modified datetime
date_purchased datetime
shipping_cost decimal(10,2)
shipping_method varchar(32)
orders_status varchar(10)
orders_date_finished datetime
comments text
currency char(3)
currency_value decimal(16,6)
orders_id = orders_id
orders_products
orders_products_id integer(5) <pk>
orders_id integer(5) <fk1>
products_id integer(5) <fk2>
products_name varchar(64)
products_price decimal(10,2)
final_price decimal(10,2)
products_tax decimal(9,4)
products_quantity integer(2)
orders_products_id = orders_products_id
orders_products_attributes
orders_products_attributes_id integer(5) <pk>
orders_id integer(5) <fk1>
orders_products_id integer(5) <fk2>
products_options varchar(32)
products_options_values varchar(32)
options_values_price decimal(10,2)
price_prefix char(1)
orders_id = orders_id
orders_status
orders_status_id integer(5) <pk>
language_id integer(5) <pk>
orders_status_name varchar(32)
configuration_group_id = configuration_group_id
configuration
configuration_id integer(5) <pk>
configuration_title varchar(64)
configuration_key varchar(64)
configuration_value varchar(255)
configuration_description varchar(255)
configuration_group_id integer(5) <fk>
sort_order integer(5)
last_modified datetime
date_added datetime
use_function varchar(32)
set_function varchar(32)
configuration_group
configuration_group_id integer(5) <pk>
configuration_group_title varchar(64)
configuration_group_description varchar(255)
sort_order integer(5)
visible integer(1)
banners_id = banners_id
banners
banners_id integer(5) <pk>
banners_title varchar(64)
banners_url varchar(64)
banners_image varchar(64)
banners_group varchar(10)
expires_impressions integer(7)
expires_date datetime
date_added datetime
date_status_change datetime
status integer(1)
banners_history
banners_history_id integer(5) <pk>
banners_id integer(5) <fk>
banners_shown integer(5)
banners_clicked integer(5)
banners_history_date datetime
countries_id = zone_country_id
countries
countries_id integer(5) <pk>
countries_name varchar(64)
countries_iso_code_2 char(2)
countries_iso_code_3 char(3)
address_format_id integer(5)
zones_to_geo_zones
association_id integer(5) <pk>
zone_country_id integer(5) <fk1>
zone_id integer(5) <fk3>
geo_zone_id integer(5) <fk2>
last_modified datetime
date_added datetime
zone_id = zone_id
geo_zone_id = geo_zone_id
geo_zones
geo_zone_id integer(5) <pk>
geo_zone_name varchar(32)
geo_zone_description varchar(255)
last_modified datetime
date_added datetime
zones
zone_id integer(5) <pk>
zone_country_id integer(5) <fk>
zone_code varchar(5)
zone_name varchar(32)
zone_id = tax_zone_id
tax_rates
tax_rates_id integer(5) <pk>
tax_zone_id integer(5) <fk2>
tax_class_id integer(5) <fk1>
tax_priority integer(5)
tax_rate decimal(9,4)
tax_description varchar(255)
last_modified datetime
date_added datetime
tax_class_id = tax_class_id
tax_class
tax_class_id integer(5) <pk>
tax_class_title varchar(32)
tax_class_description varchar(255)
last_modified datetime
date_added datetime
address_format_id = customers_address_format_id
address_format
address_format_id integer(5) <pk>
address_format varchar(128)
address_summary varchar(48)
address_format_id = delivery_address_format_id

Anhang D

Quellkode

D.1 MongoWrapper Class

```
1 <?php
2
3 class mongoWrapper
4 {
5
6   public \$connection;
7   public \$collection;
8
9   public function \_\_construct(\$host = 'localhost:27017')
10  {
11    \$this->connection = new Mongo(\$host);
12  }
13
14  public function setDatabase(\$c)
15  {
16    \$this->db = \$this->connection->selectDB(\$c);
17  }
18
19  public function setCollection(\$c)
20  {
21    \$this->collection = \$this->db->selectCollection(\$c);
22  }
23
24  public function insert(\$f)
25  {
26    \$this->collection->insert(\$f);
27  }
28
29  public function get(\$f)
30  {
31    \$cursor = \$this->collection->find(\$f);
32
33    \$k = array();
34    \$i = 0;
35
```

```
36      while( \$cursor->hasNext())
37      {
38        \$k[\$i] = \$cursor->getNext();
39        \$i++;
40      }
41
42      return \$k;
43    }
44
45    public function update(\$f1, \$f2)
46    {
47      \$this->collection->update(\$f1, \$f2);
48    }
49
50    public function getAll()
51    {
52      \$cursor = \$this->collection->find();
53      foreach (\$cursor as \$id => \$value)
54      {
55        echo "`\$id: "`;
56        var\_dump( \$value );
57      }
58    }
59
60    public function delete(\$f, \$one = FALSE)
61    {
62      \$c = \$this->collection->remove(\$f, \$one);
63      return \$c;
64    }
65
66    public function ensureIndex(\$args)
67    {
68      return \$this->collection->ensureIndex(\$args);
69    }
70
71
72  }
73
74  ?>
```

Anhang E

Inhalt der CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

E.1 Masterarbeit

Pfad: /Masterarbeit

/Latex	LaTeX-Quelldateien dieser Diplomarbeit
DA.pdf	Diplomarbeit (Gesamtdokument)

E.2 Quellen

Pfad: /Quellen

/Online	Gespeicherte Web-Seiten der Online-Quellen

E.3 Quellcode

Pfad: /Quellcode

/osCommerce	Original OsCommerce-Dateien inkl. SQL-Datei der Datenbank
/osCommerce_eav . . .	Modifizierte OsCommerce-Dateien mit Implementierung des EAV-Modells inkl. SQL-Datei der Datenbank
/osCommerce_mongodb	Modifizierte OsCommerce-Dateien mit Anbindung der MongoDB-Datenbank inkl. SQL-Datei der Datenbank

E.4 Sonstiges

Pfad: /Abbildungen

*.jpg, *.png . . . . . Original Rasterbilder
*.eps . . . . . Bilder und Grafiken im EPS-Format

Literaturverzeichnis

[1] Abadi, D. J., S. R. Madden und N. Hachem: *Columnstores vs. rowstores: How different are they really?* Techn. Ber., Massachusetts Institute of Technology, Cambridge, MA, 2008. http://db.csail.mit.edu/projects/cstore/abadi-sigmod08.pdf.

[2] Bin, S. und J. Brekle: *Key Value Stores.* Techn. Ber., Universität Leibzig, Leibzig, 2009. http://dbs.uni-leipzig.de/file/seminar_0910_Bin_Brekle.pdf.

[3] Birman, K. P.: *Maintaining consistency in distributed systems.* Techn. Ber. TR 91-1240, Department of Computer Science, Cornell University, Ithaca, NY, 1991.

[4] Boeker, M.: *MongoDB Sag ja zu NoSQL.* Software & Support Media GmbH, entwickler.press, Frankfurt am Main, 2010.

[5] Brewer, E. A.: *Towards robust distributed systems.* In: *Principles of Distributed Computing*, Portland, Oregon, 2000. http://www.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[6] Chodorow, K.: *Scaling MongoDB.* O'Reilly Media, Inc., Sebastopol, California, 2011.

[7] Chodorow, K. und M. Dirolf: *MongoDB The Definitive Guide.* O'Reilly Media, Inc., Sebastopol, California, 2010.

[8] Dean, J. und S. Ghemawat: *Mapreduce: Simplied data processing on large clusters.* Techn. Ber., Google, Inc., 2004. http://static.googleusercontent.com/external_content/untrusted_dlcp/labs.google.com/de//papers/mapreduce-osdi04.pdf.

[9] DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall und W. Vogels: *Dynamo: Amazon's highly available key-value store.* Techn. Ber., Amazon, 2007. http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.

[10] Drost, I. und L. George: *Datenmassenspeicher.* C'T Magazin, 15:168–173, Juli 2010.

[11] Edlich, S., A. Friedland, J. Hampe und B. Brauer: *NoSQL Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Carl Hanser Verlag, München, 2010.

[12] Eini, O.: *That no sql thing – document databases*, 2011. http://ayende.com/Blog/archive/2010/04/11/that-no-sql-thing-ndash-document-databases.aspx.

[13] Evans, E.: *Nosql 2009*, 2009. http://blog.sym-link.com/2009/05/12/nosql_2009.html.

[14] Fink, B.: *Why vector clocks are easy*, 2010. http://blog.basho.com/2010/01/29/why-vector-clocks-are-easy/.

[15] GeekInterview: *Entity attribute value (eav)*, 2008. http://www.learn.geekinterview.com/it/data-modeling/entity-attribute-value-eav-.html.

[16] Gilbert, S. und N. Lynch: *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. ACM SIGACT News, 33/2, 2002.

[17] GitHub: *Defining a property graph*, 2010. https://github.com/tinkerpop/gremlin/wiki/Defining-a-Property-Graph.

[18] Gueting, R. H.: *Graphdb: Modeling and querying graphs in databases*. In: *Proceedings of the 20th International Conference on Very Large Data Bases*, S. 297–308. Morgan Kaufmann, Santiago de Chile, 1994.

[19] Hewitt, E.: *Cassandra The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, California, 2010.

[20] Karger, D., E. Lehman, T. Leighton, M. Levine, D. Lewin und R. Panigrahy: *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web*. In: *Proceedings of the 29th annual ACM symposium on Theory of computing*, New York, Mai 1997. ACM.

[21] Karger, D., A. Sherman, A. Berkheimer, B. Bogstad, K. Dhanidina, Rizwan an Iwamoto, B. Kim, L. Matkins und Y. Yerushalmi: *Web caching with consistent hashing*. In: *Proceedings of the 8th international conference on World Wide Web*, New York, Mai 1999. Elsevier North-Holland, Inc. http://www8.org/w8-papers/2a-webserver/caching/paper2.html.

[22] Lakshman, A. und P. Malik: *Cassandra - a decentralized structured storage system*. Techn. Ber., Facebook, 2009. http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf.

[23] Merriman, D.: *On distributed consistency part 1*, 2010. http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1.

[24] Merriman, D.: *On distributed consistency part 2*, 2010. http://blog.mongodb.org/post/498145601/on-distributed-consistency-part-2-some-eventual.

[25] Merriman, D.: *On distributed consistency part 6*, 2010. http://blog.mongodb.org/post/523516007/on-distributed-consistency-part-6-consistency-chart.

[26] Merriman, D.: *Comparing mongo db and couch db*, 2011. http://www.mongodb.org/display/DOCS/Comparing+Mongo+DB+and+Couch+DB.

[27] MongoDB: *Sharding introduction*, 2010. http://www.mongodb.org/display/DOCS/Sharding+Introduction.

[28] MongoDB: *State of mongodb, march, 2010*, 2010. http://blog.mongodb.org/post/434865639/state-of-mongodb-march-2010.

[29] Neo4j: *Faq*, 2011. http://wiki.neo4j.org/content/FAQ#How_about_sharding_the_data.3F.

[30] Neo4j: *Graph data model*, 2011. http://wiki.neo4j.org/content/Graph_Data_Model.

[31] Neo4j: *Neo4j - the graph database*, 2011. http://neo4j.org/.

[32] Neo4j: *Neo4j - the graph database*, 2011. http://dist.neo4j.org/neo4j-manual-stable.pdf.

[33] Neo4j: *Neo4j wiki - main page*, 2011. http://wiki.neo4j.org/content/Main_Page.

[34] Neubauer, P.: *Neo4j - die High-Performance-Graphdatenbank*, 2010. http://it-republik.de/jaxenter/artikel/Neo4j-die-High-Performance-Graphendatenbank-2919.html.

[35] Nitz, D.: *Was ist eigentlich EAV?*, 2010. http://www.slideshare.net/danielnitz/was-ist-eigentlich-eav.

[36] Osborne, R.: *Sql or nosql?*, 2010. http://rickosborne.org/blog/2010/02/sql-or-nosql/.

[37] Reed, D. P.: *Naming and Synchronization in a Decentralized Computer System*. Dissertation, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sep. 1978.

[38] Sones: *CeBIT 2010: sones GraphDB verbessert Kundenservice in Online-Shops*, 2010. http://www.sones.com/c/document_library/get_file?uuid=53d77d51-0135-4e3c-93d7-47e398ef6ae6&groupId=11476.

[39] Strozzi, C.: *Nosql - a relational database management system*, 1998. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home.

[40] Vogelbacher, D.: *Das EAV Modell (Entity-Attribute-Value)*. http://www.informave.org/de/database-handbook/article/book/design_eav.html.

[41] Vogels, W.: *Eventually consistent - revisited*, Dez. 2008. http://www.allthingsdistributed.com/2008/12/eventually_consistent.html.

[42] Walker-Morgan, D.: *NoSQL im Überblick*, 2010. http://www.heise.de/open/artikel/NoSQL-im-Ueberblick-1012483.html?artikelseite=2.

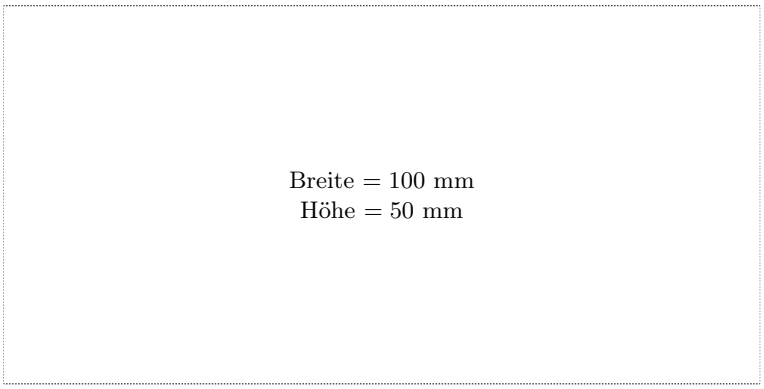
[43] Wikipedia: *Graph (Graphentheorie)*, 2011. http://de.wikipedia.org/wiki/Graph_(Graphentheorie), language=german.

[44] Wikipedia: *Vector clocks*, 2011. http://en.wikipedia.org/wiki/Vector_clock.

[45] Zawodny, J. D. und D. J. Balling: *High Performance MySQL Optimierung, Datensicherung, Replikation und Lastverteilung*. O'Reilly Verlag GmbH & Co. KG, Köln, 2005.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —