Anforderungen an Datenbanksysteme für Multi-Tenancy- und Software-as-a-Service-Applikationen

Stefan Aulbach[‡] Dean Jacobs[$] Jürgen Primsch[$] Alfons Kemper[‡]

[‡]Technische Universität München, Garching
{stefan.aulbach, alfons.kemper}@in.tum.de

[$]SAP AG, Walldorf
{dean.jacobs, j.primsch}@sap.com

Abstract: Für Multi-Tenancy-Applikationen und Software as a Service (SaaS) stellen sich konventionelle Datenbanksysteme oftmals als ungeeignet heraus. In einem solchen Umfeld müssen Anforderungen, wie Erweiterbarkeit und Verfügbarkeit, trotz geringer Kosten für den Service Provider erfüllt werden. Dieser Beitrag spezifiziert Anforderungen an ein Datenbanksystem, das eigens für den Einsatz in Multi-Tenancy-Applikationen gedacht ist, und evaluiert, in welchem Umfang eine Umsetzung mittels aktuell verfügbarer Datenbanktechnik möglich ist. Weiterhin wird ein Modell eines Datenbanksystems präsentiert, das speziell für Multi-Tenancy und SaaS optimiert ist und konsequent auf Mandantenvirtualisierung, Grid-Computing-Infrastrukturen und Hauptspeicherlokalität setzt.

1 Einführung

Das Vertriebsmodell *Software as a Service* (SaaS) wird immer beliebter. Anstatt Applikationen im eigenen Haus zu betreiben und hierfür Hard- und Software sowie Personal bereithalten zu müssen, lagern viele Unternehmen die Bereitstellung an Service Provider aus. Für die Benutzer reichen dann ein Internetzugang und ein Browser aus, allerdings erwartet der Kunde auch, dass sich die Applikation so verhält, als ob sie im eigenen Haus installiert wäre. Der Service Provider hingegen will die *Total Cost of Ownership* (TCO) reduzieren, indem er seine Infrastruktur vielen Mandanten zur Verfügung stellt.

Die folgende Arbeit diskutiert genau diese Ansprüche und Anforderungen, die Kunden, Service Provider und Anwendungsentwickler an das zu Grunde liegende Datenbanksystem (DBMS) stellen. Setzt man für *Multi-Tenancy*, also für die Fähigkeit, mehrere Mandanten auf einem gemeinsamen System zu konzentrieren, marktübliche DBMS ein, so stellt man schnell fest, dass die dazu beworbenen Fähigkeiten nicht direkt im DBMS, sondern nur als Modul implementiert sind. Zwar werden die Anforderungen aus Sicht der Kunden mehr oder weniger erfüllt, jedoch existieren weiterhin grundlegende Probleme wie Isolation und Ressourcenverwaltung, die sich erst beim Einsatz der Anwendung in einem Data Center bemerkbar machen. Anwendungsentwickler stehen bei konventionellen DBMS ebenfalls vor Problemen: Soll zum Beispiel die Applikation erweiterbar und anpassbar sein, so muss dies innerhalb der Applikation implementiert werden, falls sich Mandanten eine gemeinsame Datenbankinstanz teilen. Dabei wird eine zusätzlich zu wartende Zwischenschicht notwendig, in der die eigentliche Datenverwaltung implementiert wird, und das DBMS

wird zu einem reinen Speichermedium degradiert, das die Semantik und die Beziehungen der Daten nicht mehr zur Optimierung und zur Konsistenthaltung heranziehen kann. Andere Implementierungen, bei denen Mandanten ihr eigenes Schema oder gar ihre eigene Instanz erhalten, scheiden aufgrund ihres hohen Speicherverbrauchs aus [JA07, AGJ+08].

Als Antwort auf die mangelnde Unterstützung von Multi-Tenancy in aktuellen DBMS stellen wir in dieser Arbeit ein Modell eines DBMS vor, das speziell an die Anforderungen von Multi-Tenancy-Applikation und SaaS angepasst ist. Einsatzgebiete sind hierbei OLTP- und OLAP-Applikationen mittlerer Komplexität, wie sie in typischen Business-Anwendungen vorkommen. Ein solcher Workload ist damit eher I/O- als CPU-lastig [LMP+08]. Wir haben uns deshalb an einem Zitat von Jim Gray orientiert, das optimale Hauptspeicherlokalität postuliert: *„Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King“*

Der Beitrag ist wie folgt aufgebaut: Zunächst werden in Abschnitt 2 die Anforderungen an ein Multi-Tenancy-DBMS genauer erläutert, ehe in Abschnitt 3 beschrieben wird, in welchem Umfang sich aktuell am Markt erhältliche DBMS für die Umsetzung dieser Anforderungen eignen. Abschnitt 4 widmet sich der Behebung der übrigen Unzulänglichkeiten mittels einer neuen DBMS-Generation, bevor Abschnitt 5 schließlich die Ergebnisse zusammenfasst und einen Ausblick auf zukünftige Arbeiten gibt.

2 Anforderungen an Datenbanksysteme für Multi-Tenancy

Der Einsatz konventioneller DBMS ist auf den Betrieb innerhalb eines Unternehmens ausgerichtet, wo sich gewöhnlich nur die Daten und Anwendungen eines einzelnen Unternehmens oder einer Unternehmensgruppe finden. Bei Anwendungen, die als SaaS vermarktet werden, kommen zwangsläufig Technologien zum Einsatz, mit deren Hilfe auch die Datenbanken im Hintergrund für mehrere parallel agierende Mandanten verfügbar gemacht werden. Aus dem Vermarktungsmodell kombiniert mit Mehrmandantenfähigkeit ergeben sich spezielle Anforderungen, denen sich dieser Abschnitt widmet. Frühere Arbeiten [JA07, AGJ+08] zeigen, dass sich diejenigen Ansätze, bei denen sich mehrere Mandanten eine Instanz teilen, bezüglich des Ressourcenbedarfs besser positionieren als Ansätze, bei denen Mandanten eine eigene Datenbankinstanz erhalten.

Bei der Anforderungsdefinition lassen sich drei Rollen erkennen. Der *Service Provider*, der die Anwendung und die dafür notwendige Infrastruktur bereitstellt, legt besonderen Wert auf niedrige laufende Kosten bei hoher Verfügbarkeit. Der *Kunde* möchte im Wesentlichen eine hohe Verfügbarkeit bei hoher Datensicherheit, während sich der *Applikationsentwickler* einfache, aber mächtige Funktionsmodelle wünscht. Der Einfachheit halber verzichten wir bei der Aufstellung dieser Anforderungen auf die Trennung dieser drei Rollen.

Mandantenkonsolidierung ermöglicht dem Service Provider, mehrere Mandanten in einer Datenbankinstanz zu aggregieren. Bei der Konsolidierung darf weder die Anzahl, noch die Größe oder die Verteilung der Mandanten als limitierender Faktor auftreten. Das heißt insbesondere, dass es Mechanismen geben muss, einen Mandanten über mehrere Rechner zu verteilen, sollte dieser bis an die Grenzen eines einzelnen Rechners wachsen.

Erweiterbarkeit der Kernapplikation ermöglicht, die vom Service Provider angebote-

ne Applikation an unterschiedliche Geschäftsprozesse oder Spezialisierungen eines jeden Mandanten anzupassen. Der Mandant kann sowohl zu bestehenden Kern-Entitäten weitere Attribute, als auch gänzlich neue Entitäten definieren. Solche Erweiterungen müssen dann in der gesamten Anwendung berücksichtigt werden. Diese Anforderung ergibt sich aus der strengen Trennung zwischen Applikation und DBMS: die Applikation sorgt für die Datenpräsentation; das DBMS für die Datenhaltung und ist somit Eigentümer des Schemas.

Isolation der Mandanten ist erforderlich, um Zugriff nur auf die eigenen Daten des Mandanten zu gewähren. Aus diesem Grund muss das DBMS Mechanismen bereitstellen, um einen mandantenübergreifenden Zugriff von vornherein auszuschließen. Angelehnt an Zugriffsschutzmechanismen aus marktüblichen DBMS wird hier ein *Tenant Level Based Access Control* gefordert. Ein zentrales Identitätsmanagement verwaltet die Zugriffsrechte und verteilt diese in der Datenbank und den Applikationsservern.

Derzeit ist es üblich, dass Applikationsserver mit höchstmöglicher Autorisierung (also als Superuser) auf das DBMS zugreifen. Die eigentliche Autorisierung der Klienten erfolgt im Applikationsserver, so dass das Autorisierungssystem des DBMS gar nicht genutzt wird. Diese Vorgehensweise der Überprivilegierung des Applikationsservers birgt jedoch Gefahren, da ein erfolgreich kompromittierter Applikationsserver vollen Zugriff auf die sensiblen Daten aller Klienten in der Datenbank ermöglicht. Deshalb sollte man anwendungsspezifische und möglichst klientenspezifische so genannte *Autorisierungskorridore* [WEEK04] generieren, die den Anwendungen nur die benötigten Daten zugreifbar machen. Diese minimal autorisierten Datenbankkonten sollten automatisch aus den Anwendungen generiert werden.

Ressourcenverwaltung und -überwachung sind Voraussetzung für die bessere Auslastung von kritischen Ressourcen in einem DBMS, wie Hauptspeicher, CPU und verfügbare I/O-Bandbreite. In einem gewöhnlichen DBMS müssen diese Ressourcen zwischen den parallel abzuarbeitenden Anfragen verteilt werden; bei den Multi-Tenancy-DBMS findet zusätzlich noch eine Aufteilung auf Mandaten statt.

Ein Service Provider möchte seine physischen Ressourcen vollständig ausgenutzt wissen, weswegen das DBMS eine Methode zur Erfassung der aktuellen Lastsituation bereitstellen muss. Diese Erfassung muss so feingranular sein, dass sie sowohl die Lastsituation des Rechners, als auch den Ressourcenbedarf von mandantenspezifischen Tasks (z.B. Datenbankanfragen oder Operatoren im Auswertungsplan) erfassen kann und an eine zentrale Überwachungsinstanz oder gar einen Adaptiven Controller [GKS+08] melden kann.

Ein Adaptiver Controller steuert die Einhaltung von *Service Level Agreements (SLA)*, indem er Datenbankanfragen gemäß der aktuellen SLA-Einhaltung der einzelnen Mandanten priorisiert. Ein derartiges SLA-Management ermöglicht eine optimale Ressourcenausnutzung durch Überbuchung der Systeme bei gleichzeitiger Bereitstellung von *Quality of Service (QoS)*. Eine solche Überbuchung ist sinnvoll, da nicht immer alle Mandanten gleichzeitig online sind, beispielsweise wegen verschiedener Zeitzonen. Zudem müssen Amok laufende Anfragen vom DBMS erkannt und gestoppt werden können. Eine proaktive Ressourcenbeschränkung erweist sich möglicherweise als besseres Verfahren, da das Abbrechen von Anfragen sich auch negativ auf die Ressourcensituation auswirken kann.

Lastverteilung und Migration von Mandanten ist eng mit dem Ressourcenmanagement

verbunden. Die *Lastverteilung* dient der Abfederung von Lastspitzen und der Gewährleistung einer SLA-konformen Dienstverfügbarkeit. Solche Lastspitzen können sowohl periodisch (z.B. bei Buchungsabschlüssen), als auch sporadisch auftreten, etwa wegen einer Fehleinschätzung der Überbuchungssituation.

Beim *initialen Staging* eines Mandaten wird ein Rechner ausgewählt, wobei dieser entweder statisch beim Vertragsabschluss festgelegt, oder dynamisch bei der Aktivierung des Mandanten anhand der aktuellen Ressourcenauslastung ausgewählt wird. Eine solche *Aktivierung*, zusammen mit der *Passivierung*, ermöglicht es, dass mehrere Schichten auf dem System gefahren werden können. Nur solange ein Mandant aktiv ist, belegt er Ressourcen auf dem System, während in den Passivzeiten keinerlei Ressourcen belegt werden. Insbesondere sind die Metadaten der passiven Mandanten nicht geladen. Eine *statische Lastverteilung* allokiert die Mandanten aufgrund Lastvorhersagen, greift aber bei einer Änderung des Ressourcenbedarfs zur Laufzeit eines Mandanten nicht ein und passt nur die Lastvorhersagen entsprechend an. Erst bei der nächsten Aktivierung wird der Mandant auf einen besser geeigneten Rechner platziert. Im Gegensatz dazu kann eine *dynamische Lastverteilung* Mandanten in Überlast auch zur Laufzeit auf einen anderen Rechner migrieren.

Wächst der Bedarf innerhalb der Vertragslaufzeit, so reichen dem Mandanten oftmals die initial geplanten Kapazitäten nicht mehr aus. In einem solchen Fall muss dieser Mandant auf einen anderen Rechner *migriert* werden: entweder wieder auf ein von mehreren Mandanten gemeinsam genutztes System oder auf einen dedizierten Rechner.

Schemaänderungen zur Laufzeit treten häufig bei Updates einer Applikation auf. Da ein Multi-Tenancy-DBMS wegen der besseren Ausnutzung rund um die Uhr aktiv sein muss, müssen Schemaänderungen zur Laufzeit des Systems durchgeführt werden. Aber auch parallel agierende Mandanten machen eine Schemaänderung zur Laufzeit sehr wahrscheinlich, wenn ein Mandant die Anwendung nach seinen Bedürfnissen anpasst.

DDL-Anweisungen ziehen oft deutliche Leistungseinbußen mit sich, wenn sie auf einem aktiven System ausgeführt werden. Während das Anlegen von Tabellen nur eine Sperrung des Data Dictionaries erfordert, führen manche Änderungen an der Tabellenstruktur zu einer Änderung der Recordstruktur, also wie die Tupel auf dem Hintergrundspeicher liegen. Dementsprechend wird bei Strukturänderungen einer Tabelle oft eine temporäre Tabelle angelegt, die Tupel kopiert und anschließend die Tabelle umbenannt, womit ein solcher Eingriff die I/O-Leistung des DBMS beeinflusst. Daraus ergibt sich die Anforderung, jegliche Schemaänderungen ohne belastende Reorganisation durchführen zu können.

Geringer Administrationsaufwand sorgt bei einem Service Provider dafür, seine Leistung mit möglichst geringen laufenden Kosten, wie Hardwarekosten, Lizenzkosten, Infrastrukturkosten und Personalkosten, zu erbringen. Aus diesem Grund scheiden komplexe Cluster-Systeme als Infrastruktur aus, da nicht nur teure Spezialhardware notwendig wäre, sondern auch speziell geschulte Administratoren eingesetzt werden müssten, um den ungleich höheren Administrationsaufwand von Cluster-Systemen verglichen mit Shared-Nothing-Systemen gleicher Leistungsfähigkeit zu bewältigen.

Verfügbarkeit spielt eine so zentrale Rolle bei SaaS, dass SLAs oft integraler Bestandteil von Service-Verträgen sind. Damit ist der Service Provider fest an diese SLAs gebunden und muss bei einer Verletzung mit Vertragsstrafen rechnen. Ein Multi-Tenancy-DBMS

muss deshalb sicherstellen, dass die Applikation für den Mandanten gemäß der vereinbarten SLAs verfügbar ist, was bedeutet, dass die SLAs auch dann gewährleistet sein müssen, wenn Teile des Systems ausgefallen sind. Hierbei muss der Service Provider einen Kompromiss zwischen Kosten der Redundanz und den Vertragsstrafen eingehen, was sich im Grad der Verfügbarkeit widerspiegelt: Eine Non-Stop-Verfügbarkeit, bei der im Fehlerfall garantiert keine Transaktion zurückgerollt wird, erfordert viel zu hohe Investitionen, verglichen mit einer Hochverfügbarkeit, bei der Transaktionen zurückgerollt werden dürfen.

3 Verwendung aktueller Datenbanksysteme

Unsere früheren Arbeiten [AGJ+08] zeigen, dass sich die größten Probleme bei der Verwendung marktüblicher DBMS für Multi-Tenancy und SaaS aus der Verwaltung des Data Dictionaries und der Bufferpools ergeben. Bufferpools werden von den DBMS auf Tabellenbasis belegt, was bedeutet, dass bei einer sehr großen Anzahl von Tabellen die Bufferpool-Seiten nur eine sehr geringe Auslastung besitzen. Auf diese und weitere Probleme wird in diesem Abschnitt näher eingegangen.

3.1 Nutzung bereits vorhandener Techniken

Mit vorhandenen DBMS lassen sich auch ohne besondere Erweiterungen Multi-Tenancy-Applikationen implementieren, wenn auch nur mit Einschränkungen [JA07]: Eine Variante besteht darin, jedem Mandanten eine eigene Instanz auf einem Rechner zu geben, allerdings verbunden mit dem großen Nachteil, dass dabei mit jedem Mandanten ein sehr großer Overhead einhergeht, der aus dem initialen Ressourcenbedarf einer Datenbankinstanz resultiert. Jedem Mandanten ein eigenes Schema zu geben, führt innerhalb des DBMS zu einer sehr großen Anzahl von Tabellen, was sich in einem sehr hohen Hauptspeicherbedarf aufgrund wenig effizienter Metadatenverwaltung und in nur partiell gefüllten Bufferpool-Seiten niederschlägt. Eine dritte Variante besteht darin, alle Mandanten in einem gemeinsamen Schema zu konsolidieren und dann Sichten zu verwenden, um die Isolation der Daten zu ermöglichen. Bei dieser Variante müssen dann Maßnahmen ergriffen werden, um die Erweiterbarkeit zu gewährleisten.

Eine solche Maßnahme stellen *Chunk Tables* [AGJ+08] dar. Die Tabellen der Kernapplikation werden auf eigene physische Tabellen abgebildet, mandantenspezifischen Erweiterungen hingegen auf eine *generische Struktur*. Um eine Trennung der Mandanten zu erreichen, wird für jedes Tupel die Mandaten-ID gespeichert. Der große Vorteil dieses Verfahrens ist, dass sich die Anzahl der Tabellen reduziert und somit der Bufferpool besser ausgelastet ist. Nachteilig wirkt sich allerdings aus, dass ein einzelnes Tupel mittels Joins wieder zusammengesetzt werden muss. Auch andere Lösungen, wie pureXML für IBM DB2 [DB2], verfolgen diesen Ansatz.

Allen diesen Lösungen ist gemeinsam, dass die Mehrmandantenfähigkeit von außen aufgesetzt worden ist. Eine zusätzliche Zwischenschicht enthält nicht nur das Wissen über die verwendete Technologie, um die benutzerdefinierten Attribute auf die physischen Tabelle

abzubilden, sondern auch das Wissen über die Datenverteilung. Der Nachteil dieser Zwischenschicht ist ein höherer Wartungsaufwand und die Tatsache, dass die Datenbank zu einem großen Datentopf degradiert wird, der über die Verknüpfung der Daten nichts weiß, und diese somit nicht mehr zur Optimierung verwenden kann.

Neu aufkommende Datenbanktechnologien, wie Googles *BigTable* [CDG+08], Amazons *Dynamo* [DHJ+07] oder Yahoo!'s *PNUTS* [CRS+08], spielen hierbei auch eine wichtige Rolle, werden in dieser Arbeit jedoch in einem anderen Zusammenhang in einem folgenden Abschnitt näher diskutiert.

3.2 Geringfügige Modifikationen im Datenbankkernel

Während die geringe Bufferpool-Auslastung nur schwer in den Griff zu bekommen ist, kann man die Probleme mit der Metadatenverwaltung vergleichsweise einfach lösen. Jede einzelne Tabelle belegt einen gewissen Hauptspeicherbereich, in dem zum einen die Struktur der Tabelle vorgehalten wird, zum anderen die Abbildung von der logischen Sichtweise auf die physische Struktur (File Pointer, B-Tree, etc.) stattfindet. Gibt man jedem Mandanten ein eigenes Schema, so nimmt mit zunehmender Anzahl von Mandanten – und somit auch mit zunehmender Anzahl von Tabellen – der davon beanspruchte Arbeitsspeicherbereich zu. Eine Reduktion der Tabellenanzahl mittels eingangs erwähnter Verfahren führt allerdings zu keiner endgültigen Lösung, da mittels generischer Strukturen die Metadatenverwaltung vom DBMS nur in die Zwischenschicht ausgelagert wird.

Eine endgültige Lösung der Metadatenproblematik bietet letztlich nur eine geänderte Implementierung mit dem Ziel vererbbarer Metadaten, so dass möglichst viele Schemainformationen gemeinsam genutzt werden und somit nur einmal im System vorhanden sind. Das Schema der Kerntabellen kann als abstrakte Klasse aufgefasst werden; jeder Mandant bildet dann sein eigenes Schema durch Ableitung.

Um eine weitere Reduktion des Hauptspeicherbedarfs der Metadaten zu erreichen, sollte das Data Dictionary denselben Paging-Mechanismen unterliegen wie die gesamte Datenbank. Zudem sollte die Abbildung auf die File Pointer nur solange im Arbeitsspeicher gehalten werden, wie die Tabelle wirklich genutzt wird.

Weiterhin kann bei bestehenden DBMS auch die Informationsdarstellung auf dem Hintergrundspeicher so angepasst werden, dass Schemaänderungen keine Reorganisation erforderlich machen und somit keine I/O-Belastung auftritt. Dies geschieht vor allem durch Verfahren wie *Update in Place*, *Sort by Insertion* und *Delete in Place*. Eine mögliche Implementierung findet sich in [GT07].

Eine andere Modifikation besteht darin, die Daten und Indexe der Mandanten untereinander physisch zu isolieren, indem jeder Mandant seinen eigenen Container erhält. Dabei werden die Daten so verteilt, dass jeder Mandant in sich abgeschlossen ist, um sicherzustellen, dass eine Migration durch Verwendung von einfachen Dateioperationen anstelle von teurem Entladen und Laden möglich ist. Zudem kann der Container bei inaktiven Mandanten ohne Belastung anderer Mandanten gesichert werden. Für Indexe kann ein Verfahren eingesetzt werden, dass sich an partitionierte B-Bäume [Gra03] anlehnt.

3.3 Verbleibende Schwierigkeiten und inhärente Schwächen

Zu den verbleibenden Problemen zählen nahezu alle administrativen Aufgaben, die in konventionellen DBMS auf Instanzebene, in Multi-Tenancy-Systemen jedoch auf Mandantenebene implementiert sind. Solche administrativen Tätigkeiten sind beispielsweise Wiederherstellung der Anwendung nach Fehlbedienung (Point-in-Time-Recovery, PITR), aber auch Migration und (Hoch-) Verfügbarkeit, insbesondere auch Aktualisierungen der Applikation. Wird eine solche durchgeführt, so ändert sich oft auch das Schema der Tabellen. Dabei muss sichergestellt werden, dass Mandanten, die ihr privates Schema erweitert haben, auch mit der neuen Version der Kerntabellen kompatibel bleiben. In diesem Zusammenhang muss das Data Dictionary einer Versionierung unterworfen werden, so dass auch die Möglichkeit besteht, nicht alle Mandanten auf einmal auf die neue Version zu aktualisieren. Schemakompatibilität spielt zudem eine große Rolle in einem Multi-Tenancy Data Center mit unterschiedlichen Applikationsdomänen (z.B. SCM und CRM), die zwecks besserer Auslastung auf demselben operationalen System beheimatet sind.

Stonebraker et al. [SMA+07] fordern das Ende von den General Purpose-DBMS, indem sie spezialisierte DBMS für jeden Verwendungszweck postulieren. Da sich SaaS fundamental von klassischen OLTP-Anwendungen unterscheidet, ist eine solche Forderung auch in diesem Szenario durchaus gerechtfertigt.

4 Wege zum Glück

In diesem Abschnitt wird das Modell eines Multi-Tenancy-DBMS entworfen. Die Grundidee, auf denen alle hier vorgestellten Methoden aufbauen, ist der konsequente Einsatz von Virtualisierungstechniken und Grid-Technologien, sowie die Ausnutzung der von Jim Gray postulierten Hauptspeicherlokalität. In diesem Zusammenhang ist jeder Mandant eine „virtuelle Maschine“, die auf dem Hypervisor „Multi-Tenancy-fähiges DBMS“ zur Ausführung gebracht wird.

Speziell für den Einsatz von Grid-Technologien spricht die geringe Hardwarekomplexität der beteiligten Komponenten. Auch andere Projekte, wie Googles *BigTable* [CDG+08], verwenden aus gleicher Motivation diese Technologien. BigTable dient als hochgradig skalierbarer Datenspeicher, der auf einer Vielzahl einfacher Rechner verteilt wird, mit Verzicht auf ein relationales Datenmodell. BigTable speichert Informationen in einer Zelle, die über die Dimensionen Zeile, Spalte und Zeitstempel referenziert wird. Zellen werden so abgelegt, dass semantisch verwandte Dimensionen physikalisch möglichst benachbart platziert werden. Amazons *Dynamo* [DHJ+07] und *SimpleDB* [Sim] verfolgen einen ähnlichen Ansatz, gewähren allerdings andere Garantien hinsichtlich der Datenkonsistenz. Im Gegensatz zu diesen einfachen Key-Value-Stores stellt *PNUTS* [CRS+08] ein relationales Datenmodell zu Verfügung. Auch hier stehen Skalierbarkeit, Verteilung und Hochverfügbarkeit an erster Stelle, jedoch werden, um das zu erreichen, die Zugriffsmöglichkeiten und die Konsistenzgarantien verglichen mit traditionellen DBMS eingeschränkt.

Allen diesen Projekten ist gemeinsam, dass sie sehr stark von Verteilung und Replikation

Gebrauch machen und ihr Datenmodell danach ausrichten. Je nach den Anforderungen an die Replikation (beispielsweise, wie oft sie stattfindet) und damit verbunden auch an die Konsistenz der Replikate, wird das Datenmodell gewählt und die operationalen Garantien, wie man sie von konventionellen DBMS kennt, entweder gewährt oder eingeschränkt. Für Multi-Tenancy-DBMS können ähnliche Verfahren verwendet werden.

4.1 Flexible Datenbankschemas

Für die bessere Anpassbarkeit der SaaS-Applikation an die Bedürfnisse der einzelnen Mandanten, muss das DBMS Verfahren bereitstellen, die eine Änderung des Schemas zur Laufzeit erlauben. Flexible Schemas ermöglichen die *spontane Schemaanpassung* durch den Mandanten, ohne dass ein Datenbankadministrator des Service Providers notwendig ist. Hierfür werden zwei mögliche Ansätze vorgestellt. Zunächst sollen flexible Schemas mit Hilfe von bereits vorhandenen Technologien in modernen DBMS implementiert werden, ehe auf einen radikaleren Ansatz eingegangen wird, der umfangreiche Änderungen am Datenbankkernel erfordert.

4.1.1 Vorhandene Ansätze in aktuellen Datenbanksystemen

Einige moderne Datenbanksysteme besitzen die Fähigkeit, manche Schemaänderungen zur Laufzeit auch ohne Datenreorganisation durchzuführen, indem sie die notwendigen Änderungen zunächst nur auf den Metadaten ausführen. Demnach lassen sich Schemaänderungen in zwei unterschiedliche Typen aufteilen: auf der einen Seite befinden sich alle leichtgewichtigen Schemaänderungen, mit denen lediglich eine Anpassung des Data Dictionaries einhergeht. Zu diesem Typ gehören zum Beispiel das Anlegen einer neuen Tabelle oder das Hinzufügen von Spalten, die NULL-Werte enhalten darf. Auf der anderen Seite befinden sich die schwergewichtigen Operationen, die neben einer Änderung des Data Dictionaries auch eine Datenreorganisation erforderlich machen. Hierzu zählen die Änderung des Datentyps von Spalten oder das Hinzufügen von Attributen, deren Werte nicht undefiniert sein dürfen.

Aus diesen Fähigkeiten lässt sich eine mögliche Herangehensweise ableiten, mit deren Hilfe flexible Schemas implementiert werden können: alle unkritischen Operationen, d.h. alle diejenigen Operationen, die nur auf dem Data Dictionary arbeiten, werden unmittelbar durchgeführt, während Operationen, die eine Datenreorganisation, wie im Falle des neu hinzugefügen NOT-NULL-Attributs, oder eine Datenüberprüfung, falls der Datentyp eines Attributes geändert wird, erforderlich machen, erst zu einem späteren Zeitpunkt ausgeführt werden, an dem sich das System in einer Schwachlastphase befindet.

Darüber hinaus gibt es weitere Ansätze in modernen Datenbanksystemen, die speziell für flexible Schemas entwickelt worden sind. Beispielsweise beinhaltet Microsoft SQL Server 2008 eine Unterstützung für flexible Schemas, indem eine spezielle Speichermethode für dünn besiedelte Attribute (*Sparse Columns*) implementiert wird [ACGL+08]. Mittels IBM pureXML [pur] können auf ähnliche Weise flexible Schemas implementiert werden, indem man jedem Tupel einen Bereich für semistrukturierte Daten hinzufügt. Beide Varianten haben darüber hinaus den Effekt, dass sie Speicherplatz sparen, da der Aufwand für die Speicherung von NULL-Werten verringert wird.

4.1.2 *Tenant Swizzling* und *Working Set*-Modell

Anstatt Daten auf Pagelevel zu lesen, bietet es sich an, viel größere Datenblöcke auf einmal zu lesen. Der Umfang solcher Datenblöcke kann beispielsweise eine gesamte Buchungsperiode oder einen gesamten Mandanten umfassen. Für eine bessere Ressourcenauslastung empfiehlt es sich, die Datenrepräsentation im Arbeitsspeicher von der des Hintergrundspeichers zu entkoppeln. So ist gewährleistet, dass für jeden Speicherort die optimale Darstellung gewählt werden kann. Beim Transport der Daten vom oder zum Arbeitsspeicher müssen diese nun transformiert werden. Diese Transformation nennen wir *Tenant Swizzling*, angelehnt an das Pointer Swizzling in objekt-orientierten DBMS [KK95].

Auf die Objekte ist ein schneller Zugriff gewährleistet, solange diese nicht vom Hintergrundspeicher gelesen werden müssen. Ein solcher Zugriff wird nun – verglichen mit konventionellen DBMS – deutlich teurer, da die Daten erst vom Hintergrundspeicher gelesen, interpretiert und transformiert werden müssen. Ändert sich ein Objekt, so muss diese Transformation rückwärts durchlaufen werden, um das Objekt zu persistieren. Wird ein Write-Ahead-Log (WAL) geführt, ist eine solche Aktion jedoch nicht unmittelbar nach der Änderung erforderlich, sondern kann verzögert werden. Mit Tenant Swizzling können wegen der Entkopplung von Hauptspeicher- und Hintergrundspeicherrepräsentation ebenfalls flexible Schemas implementiert werden. Zudem kann die eingangs geforderte Hauptspeicherlokalität gewährleistet werden.

Kann aufgrund der Hauptspeichergröße nicht der gesamte Mandant in den Arbeitsspeicher geladen werden, dürfen nur diejenigen Teile geladen werden, die zum unmittelbaren *Working Set* des Mandanten gehören. Dieses ist je nach Applikation unterschiedlich und besteht beispielsweise aus der aktuellen Buchungsperiode. Wird ein Mandant aktiv, so werden die Daten des Working Sets mittels Tenant Swizzling vom Hintergrundspeicher geholt, interpretiert und online gebracht. Alle anderen Daten werden als archivierte Daten betrachtet und stehen nur noch lesend und mit erhöhter Zugriffszeit zur Verfügung. So wird gewährleistet, dass alle Daten eines Mandanten auf jeden Fall verfügbar sind und z.B. in analytischen Anfragen verwendet werden können. Sobald der Mandant deaktiviert wird, werden die Hauptspeicherstrukturen wieder persistiert.

Aus den Daten von Salesforce.com [McK07] lässt sich ableiten, dass selbst größere Mandanten selten mehr als 1 GB Daten in einer CRM-Applikation vorhalten, was eine sehr populäre, aber nicht die einzig mögliche Applikation für SaaS ist. Das hier vorgestellte Szenario erlaubt es demnach, eine respektable Anzahl von Mandanten auf ein entsprechend ausgestattetes System zu arrangieren, zumal das Working Set deutlich unter 1 GB liegen dürfte.

Aus Performancegründen kann weiterhin der Einsatz eines Bufferpools notwendig sein, wobei damit jedes Objekt doppelt im Hauptspeicher vorliegt, einmal objektorientiert und einmal im Hintergrundspeicherformat. Deshalb müssen beim Einsatz eines Bufferpools die Kosten für den erhöhten Hauptspeicherbedarf und die Vertragsstrafen bei nicht eingehaltenem SLA abgewogen werden. Einen solchen Ansatz verfolgt SAPs *liveCache*. Ist der Hauptspeicher ausreichend, um die Daten des gesamten Mandanten aufnehmen zu können, kann auf Bufferpools verzichtet werden.

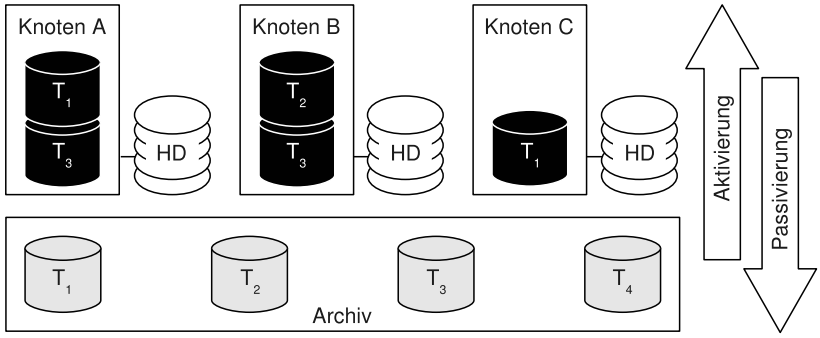

Abbildung 1: Multi-Tenancy DBMS

4.2 Grid-Infrastruktur und Near-Line-Storage

Ähnlich wie *Greenplum* [Gre] setzen wir auf eine Grid-Infrastruktur bestehend aus Knoten geringer Hardwarekomplexität. Gemäß Abbildung 1 bilden diese Knoten eine Farm von lose gekoppelten Shared-Nothing-Systemen, die Zugriff auf einen gemeinsamen Archivspeicher haben, dessen einzige Aufgabe es ist, passive Mandanten zu speichern; aktive Mandanten verwenden die lokalen Platten des Knotens. Für den Archivspeicher ist kein schnelles, und somit teures SAN erforderlich, ein Near-Line-Storage-System ist ausreichend. Ähnlich einem Cluster erhält man mit der Farm die Fähigkeiten zu Überbuchbarkeit und horizontalen Skalierbarkeit, bei gleichzeitig reduziertem Administrationsaufwand.

Zwecks besserer Verfügbarkeit wird eine Master/Slave-Replikation auf Mandantenebene eingesetzt, was nicht notwendigerweise bedeutet, dass zwei auf demselben Master aktive Mandanten auch auf demselben Slave allokiert sind. Da man aus Performancegründen auf Random-I/O auf der Logdatei verzichten möchte, darf nicht für jeden Mandanten eine eigene Logdatei geführt werden, sondern nur genau eine pro Knoten. Für die Replikation auf Mandantenebene muss nun ein *partielles Log Shipping* implementiert werden, das nur die Logeinträge des jeweiligen Mandanten an die Replikate weiterleitet. Da gemäß der eingangs aufgestellten Anforderungen eine Transaktion niemals die Grenzen von Mandanten überschreiten kann, ist ein solches partielles Log Shipping möglich, ohne dass benachbarte Mandanten davon betroffen sind. Zudem befinden sich bei der Verwendung von einem Container pro Mandant, wie es in Abschnitt 3.2 beschrieben ist, niemals Daten von verschiedenen Mandanten auf einer gemeinsamen Speicherseite. Ob synchrones oder asynchrones Log Shipping zum Einsatz kommt, hängt von dem mit dem Mandanten vereinbarten SLA, aber auch von der räumlichen Verteilung der Replikate ab: Innerhalb eines Data Centers erfolgt synchrone Replikation, während Replikation für Desaster Recovery in ein entferntes Data Center besser asynchron erfolgt.

Bei der Aktivierung eines Mandanten werden der Master und ein oder mehrere Slaves instantiiert, indem der Datenbestand vom Archivspeicher geladen wird. Soll zur Laufzeit ein zusätzlicher Slave gestartet werden, so lädt der Knoten den – mittlerweile veralteten Datenbestand – vom zentralen Archivspeicher und macht einen Roll-Forward mit Hilfe der Logdaten des Masters. Dies setzt voraus, dass eines der beteiligten Replikate einen

Logdatensatz bereithält, der sich auf die im zentralen Archivspeicher abgelegte Datenausprägung bezieht. Fällt ein Knoten der Farm aus, so muss sichergestellt werden, dass alle Mandanten, für die der ausgefallene Knoten die Masterrolle übernommen hat, wieder einen eindeutigen Master zugewiesen bekommen. Beim Recovery nach einem Fehlerfall kann auch die Kopie auf dem gemeinsam genutzten Archivspeicher genutzt werden.

Solange ein Mandant passiv ist, kann dieser beliebigen Knoten zugewiesen werden, so dass eine statische Lastverteilung ohne großen Aufwand realisiert werden kann. Ist der Mandant hingegen aktiv, so wird die dynamische Lastverteilung dadurch realisiert, dass der überlastete Master seine Führungsrolle an einen der beteiligten Slaves abgibt. Da ein Slave keine Anfragelast zu schultern hat, kann damit die Last feingranular auf Mandantenebene verteilt werden. Allerdings schränkt dieses Verfahren die Wahl des Zielknotens auf diejenigen Knoten ein, die ein Replikat des betroffenen Mandanten besitzen. Sollte die Umschaltung der Rollen nicht den gewünschten Effekt erzielen, so bietet sich an, ein Replikat auf einem neuen Slave zu erstellen und diesen dann als Master zu wählen. Anschließend kann die Anzahl der vorgehaltenen Kopien wieder reduziert werden. Mit dieser Erweiterung kann eine Migration auf einen neuen Knoten zur Laufzeit durchgeführt werden. Ebenfalls können der Farm neu hinzugefügte Knoten sofort genutzt werden.

5 Zusammenfassung und Ausblick

Mit dieser Arbeit wurde ein Anforderungskatalog für den Einsatz von DBMS in Multi-Tenancy-Anwendungen und SaaS skizziert. Einige der Forderungen lassen sich mit unmodifizierten konventionellen DBMS durch den Einsatz von generischen Strukturen, wie beispielsweise Chunk Tables, bereits jetzt realisieren. Einige unserer Arbeiten beschäftigen sich mit der Speicherung dieser generischen Strukturen mittels verschiedener Speichermethoden, wie konventionelle Tabellen, Chunk Tables, XML und dünn belegte Tabellen.

Andere Anforderungen stehen erst mit einem modifizierten Datenbankkernels bereit, wie beispielsweise Schemaänderungen zur Laufzeit. Einige der in Abschnitt 3.2 genannten Modifikationen wurden bereits in MaxDB 7.7 [Maxb] implementiert [Maxa].

Ein großes Forschungsvorhaben ist die Umsetzung eines DBMS, wie es in Abschnitt 4 vorgestellt worden ist. Dabei wird insbesondere ein Augenmerk auf die Administration der Farm mittels eines Adaptiven Controllers gelegt, der den Zustand aller Knoten und aller Mandanten überwacht und auf die Einhaltung vereinbarter SLAs achtet. Neben proaktiver Aktivierung eines Mandanten spielen auch Migration und Lastverteilung eine große Rolle. Der adaptive Controller soll zudem in der Lage sein, die (Hoch-) Verfügbarkeit der Mandantendaten in Einklang mit den SLAs zu garantieren.

Weitere Fragestellungen beschäftigen sich mit dem Fall, dass ein Mandant die Grenzen einer einzelnen Maschine sprengt. Anstatt die Daten zu partitionieren und somit ein synchrones System zu erhalten, könnte durchaus auch der Einsatz von relaxierten Konsistenzmodellen, wie es beispielsweise *PNUTS* [CRS+08] bietet, gerechtfertigt sein. In einem solchen asynchronen System geht man das Risiko von inkonsistenten Daten ein, und hofft darauf, dass diese Inkonsistenzen behoben werden können.

Literatur

[ACGL+08] Srini Acharya, Peter Carlin, César A. Galindo-Legaria, Krzysztof Kozielczyk, Pawel Terlecki und Peter Zabback. Relational Support for Flexible Schema Scenarios. In *VLDB*, Seiten 1289–1300, 2008.

[AGJ+08] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper und Jan Rittinger. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *SIGMOD Conference*, Seiten 1195–1206, 2008.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes und Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver und Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *VLDB*, Seiten 1277–1288, 2008.

[DB2] Data Integration and Composite Business Services, Part 3: Build a Multi-Tenant Data Tier with Access Control and Security. `http://www.ibm.com/developerworks/db2/library/techarticle/dm-0712taylor/`.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall und Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, Seiten 205–220, 2007.

[GKS+08] Daniel Gmach, Stefan Krompass, Andreas Scholz, Martin Wimmer und Alfons Kemper. Adaptive Quality of Service Management for Enterprise Services. *TWEB*, 2(1), 2008.

[Gra03] Goetz Graefe. Sorting And Indexing With Partitioned B-Trees. In *CIDR*, 2003.

[Gre] Greenplum. `http://www.greenplum.com/`.

[GT07] Heike Gursch und Werner Thesing. No-Reorganization Principle – Data Storage Without I/O Bottlenecks. `http://maxdb.sap.com/training/internals_7.6/reorgfree_EN_76.pdf`, 2007.

[JA07] Dean Jacobs und Stefan Aulbach. Ruminations on Multi-Tenant Databases. In *BTW*, Seiten 514–521, 2007.

[KK95] Alfons Kemper und Donald Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB J.*, 4(3):519–566, 1995.

[LMP+08] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim und Sang-Woo Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *SIGMOD Conference*, Seiten 1075–1086, 2008.

[Maxa] MaxDB – Internals. `http://maxdb.sap.com/training/#internals`.

[Maxb] MaxDB – The SAP Database System. `http://maxdb.sap.com/`.

[McK07] Todd McKinnon. Plug Your Code in Here – An Internet Application Platform. `www.hpts.ws/papers/2007/hpts_conference_oct_2007.ppt`, 2007.

[pur] XML Database - DB2 pureXML. `http://www.ibm.com/db2/xml/`.

[Sim] Amazon SimpleDB. `http://aws.amazon.com/simpledb/`.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem und Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, Seiten 1150–1160, 2007.

[WEEK04] Martin Wimmer, Daniela Eberhardt, Pia Ehrnlechner und Alfons Kemper. Reliable and Adaptable Security Engineering for Database-Web Services. In *ICWE*, Seiten 502–515, 2004.