Supporting Multi-tenancy in Relational Database Management Systems for OLTP-style Software as a Service Applications

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Oliver Schiller

aus Kirchheim unter Teck

Hauptberichter:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Mitberichter:	Prof. Johann-Christoph Freytag, Ph. D.
Tag der mündlichen Prüfung:	23. September 2014

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2015

Acknowledgements

First of all, I want to thank my doctoral advisor, Prof. Bernhard Mitschang, for his support, his guidance and many interesting discussions. Through his guidance I learned a lot about conducting scientific research and his ideas gave me new insights into my work. Finally, he gave me the opportunity to work on this challenging topic in his research group.

Furthermore, I would like to say thank you to my colleagues during my time at Institute of Parallel and Distributed Systems at University of Stuttgart, namely (in alphabetical order) Andreas Brodt, Nazario Cipriani, Matthias Großmann, Christoph Gröger, Carlos Luebbe, Peter Reimann, Holger Schwarz, Stefan Silcher and Christoph Stach. It was a pleasure and a great time working together with you all!

Last but definitely not least, I want to express my sincere thanks to my girlfriend, my parents, my brother, and my friends for their continuous support, encouragement and patience during my work on this thesis.

Oliver Schiller

Dettingen, 20.03.2014

Contents

List of Acronyms

ACID	Atomicity, Consistency, Isolation, and Durability
IaaS	Infrastructure as a Service
I/O	Input/Output
LSN	Logical Sequence Number
MVCC	Multi-version Concurrency Control
NIST	National Institute of Standards and Technology
OLAP	Online Analytical Processing
OLAP	Online Transaction Processing
PaaS	Platform as a Service
RDBMS	Relational Database Management System
SaaS	Software as a Service
SLA	Service Level Agreement
SQL	Structured Query Language

TID	Tuple Identifier
TPC	Transaction Processing Council
WAL	Write-ahead Log
VM	Virtual Machine

Deutsche Zusammenfassung[1]

Es läutet an der Türe, der Postbote ist da. Er bringt einen Datenträger mit Arbeit; der Datenträger enthält ein Update der Textverarbeitungssoftware. Dieses muss baldmöglicht auf der unternehmenseigenen, vorhandenen IT-Infrastruktur, sprich 20 Arbeitsplatzrechnern, eingespielt werden.

1 Hintergrund

Das vorhergehende Szenario kam jahrelang, so oder in ähnlicher Form, im Rahmen des üblichen Vertriebs- und Nutzungsmodells von Software vor; die Software wurde vom nutzenden Unternehmen eigenständig betrieben und gepflegt. Über die Jahre hinweg hat sich dieses Modell jedoch hin zur Vereinfachung für die nutzenden Unternehmen verändert, z. B. durch automatische Updates der Software über eine Internetverbindung und externe, spezialisierte Dienstleister, die große Teile der administrativen Aufgaben übernehmen. Ein Trend hin zur reinen Nutzung der Software war unübersehbar vorhanden und auch gewünscht. Als Konsequenz hierauf wurde in jüngster Vergangenheit ein weiterer Schritt vollzogen und ein neues Modell, namens Software as a Service (SaaS), eingeführt. Im Rahmen von SaaS wird Software durch einen Dienstleister auf dessen Infrastruktur betrieben und gepflegt. Der Dienstleister bietet die Nutzung der Software über eine Breitbandverbindung an.

Das SaaS Modell bietet Vorteile für beide Seiten: den Dienstleister und dessen Kunden. Die Kunden benötigen weniger hauseigene IT-Infrastruktur und verringern dadurch damit verbundene administrative Aufwände. Dies führt weiterhin zu einem verringerten Risiko und erhöhter Flexibilität, da keine initialen Investitionen von Nöten sind und die Expansion des Unternehmens auf der IT-Seite schnell und zügig durch den Dienstleister erbracht werden kann. Zusätzlich orientieren sich die Kosten am tatsächlichen Bedarf, somit sind die Kosten auch gut überschaubar und oftmals geringer als die Kosten für eine vergleichbare traditionelle Lösung. Dem Dienstleister wird ermöglicht Skalen- und Konsolidierungseffekte auszunutzen. Dies erlaubt

[1]A summary of this thesis in german.

eine Reduzierung der Fixkosten, die für jeden Kunden anfallen. Dies fördert einerseits die Gewinnspanne und erlaubt andererseits den Dienst zu einem attraktiven Preis anzubieten. Letzteres ermöglicht eine Ausweitung des Marktes. Viele der genannten Vorteile sind insbesondere für kleine bis mittelgroße Unternehmen von Bedeutung. Für diese Unternehmen sind die Fixkosten und die Risiken einer traditionell aufgebauten Lösung oftmals nicht tragbar, oder traditionell aufgebaute Lösungen sind nur mit sehr eingeschränkter Funktionalität realisierbar.

2 Mandantenfähigkeit

SaaS ist eines der im Rahmen von Cloud Computing möglichen Diensterbringungsmodelle. Weitere Diensterbringunsmodelle sind Platform as a Service (PaaS) und Infrastructure as a Service (IaaS). Für jedes dieser Diensterbringungsmodelle wird *Mandantenfähigkeit* als eine essentielle Eigenschaft angesehen [Mell and Grance, 2009]. Mandantenfähigkeit beschreibt die Fähigkeit, mehrere Mandanten auf einer Resource konsolidieren zu können. Als Mandant versteht man hierbei eine datentechnisch und organisatorisch abgeschlossene Einheit, die den Dienst nutzt, z. B. ein Unternehmen. Wesentliche Anforderungen, welche es im Zusammenhang mit Mandantenfähigkeit zu berücksichtigen gilt, sind die gemeinsame Nutzung von Betriebsmitteln bei ausreichender Isolation sowie der effiziente Betrieb und die effiziente Verwaltung, auch bei einer großen Anzahl an Mandanten.

Im Rahmen von SaaS betrifft die Realisierung von Mandantenfähigkeit jede Schicht des Systemaufbaus und die jeweilig dazugehörenden Komponenten. Somit ergibt sich als eine Teilaufgabe einer SaaS Lösung üblicherweise die Implementierung von Mandantenfähigkeit in der Datenschicht mit den darin eingesetzten Datenbankmanagementsystemen. Die weite Verbreitung von Relationalen Datenbankmanagementsystemen (RDBMS) führt dazu, dass die Realisierung von Mandantenfähigkeit unter Verwendung von RDBMS von zentraler Bedeutung für die Realisierung vieler mandantenfähiger SaaS Lösungen ist.

Bisherige RDBMS Implementierungen bieten keine native Unterstützung für die Entwicklung und den Betrieb mandantenfähiger SaaS Anwendungen an. Aus diesem Grund benötigen existierende Lösungen eine zusätzliche Schicht über dem RDBMS, welche die notwendige Funktionalität realisiert. Beispiele hierfür sind Salesforce CRM, Intacct und Workday. Häufig implementiert diese zusätzliche Schicht vorhandene Funktionalität erneut, allerdings angereichert um Mandantenfähigkeit, z. B. die Schemaverwaltung. Hierdurch wird das Vermögen des RDBMS nicht mehr genutzt. Die Implementierung dieser zusätzlichen Schicht ist aufwändig und fehleranfällig, weiterhin fällt eine zusätzlich zu wartende Komponente an. Außerdem verliert eine solche Lösung an Optimierungspotential, z. B. verliert das RDBMS Information über die verwendeten Datentypen der Applikation. Zuletzt wird keine saubere Trennung der Verantwortlichkeiten für bestimmte Aufgaben eingehalten; die Datenverwaltung sollte in wesentlichen Teilen durch das RDBMS unterstützt werden. Eine native Unterstützung von Mandantenfähigkeit durch das RDBMS kann diese Nachteile aufheben.

3 Beiträge dieser Arbeit

Auf Grund der Reife und der weiten Verbreitung existierender RDBMS Architekturen ist deren Erweiterung um Funktionalität für mandantenfähige SaaS Anwendungen ein valider Ansatz. In dieser Arbeit wird dieser Ansatz verfolgt und wesentliche Erweiterungen einer auf Hintergrundspeicher basierten RDBMS Architektur für mandantenfähige SaaS Anwendungen mit OLTP Charakter beigetragen. Die hierfür zentralen Beiträge sind folgend aufgeführt:

- Zuerst wird eine Erweiterung vorgestellt, die als Grundlage zur Realisierung von Funktionalität für die Erstellung und den Betrieb mandantenfähiger SaaS Anwendungen dient. Hierzu werden Mandanten in das RDBMS als explizite Datenbankobjekte aufgenommen. Hiermit verbunden wird das Konzept *Tenant Context*, zu deutsch *Mandantenkontext*, eingeführt. Ein Tenant Context stellt ein logischer Container für die Ausführung von Operationen eines bestimmten Mandanten dar. Dies erlaubt eine leichtgewichtige Art der Virtualisierung im RDBMS. Basierend auf diesem Konzept wird eine Lösung zum Schemamanagement für mandantenfähige SaaS Anwendungen vorgeschlagen [Schiller et al., 2011b]. Die Lösung, genannt *TenantSchema*, erlaubt durch ein Vererbungsprinzip die Definition von Teilen des Anwendungsschemas, welche für alle Mandanten gleich sind. Hierbei wird auch die Definition eines gemeinsamen Datenbestandes berücksichtigt. Des Weiteren ermöglicht TenantSchema unabhängige, mandantenspezifische Anpassungen. Die Definition gemeinsam zu nutzender Resourcen führt zu einer Verringerung von Redundanz und dadurch zu einer erhöhten Skalierbarkeit. Des Weiteren wird eine zentrale Verwaltung des Anwendungsschemas ermöglicht.

- Jeder Mandant nutzt für jede Tabelle des Anwendungsschemas eine Tabelleninstanz, d. h. eine logische Menge an Tupeln. Die Konsolidierung einer Vielzahl von Mandanten auf einer RDBMS Instanz führt dazu, dass das RDBMS mit einer großen Anzahl an Tabelleninstanzen umgehen können muss. Die Abbildung dieser Tabelleninstanzen, d. h. derer Tupel, auf physische Speicherstrukturen, z. B. Heaps oder B-Bäume, kann Einfluss auf die Skalierbarkeit des Systems nehmen. Daher werden im Rahmen dieser Arbeit verschiedene Möglichkeiten zur Abbildung der logischen Tabelleninstanzen auf physische Speicherstrukturen vorgestellt. Es wird eine Evaluation verschiedener Varianten für zwei gängige physische Speicherstrukturen durchgeführt [Schiller et al., 2011a, 2012]. Hierbei liegt der Fokus auf den Vor- und Nachteilen hinsichtlich der Performanz bei der Konsolidierung einer Vielzahl kleinerer Mandanten[2]. Abschließend werden die Ergebnisse der Evaluation für eine sinnvolle Lösung in Verbindung mit TenantSchema genutzt.
- Nachfolgend wird die Architektur eines mandantenfähigen RDBMS Clusters für mandantenfähige SaaS Anwendungen addressiert. Ein besonderes Augenmerk wird hierbei auf eine sinnvolle Verarbeitungsgranularität von administrativen Operationen gelegt. Dies ist insbesondere von Bedeutung, falls eine Konsolidierung der Mandanten in eine physische Speicherstruktur erfolgt. In diesem Fall sind viele administrative Operationen nur noch aufwändig in selektiver Art und Weise, d. h. für

[2]Hiermit sind Mandanten gemeint, die lediglich eine geringe Verarbeitungs- und Datenkapazität benötigen.

eine Teilmenge der Mandanten, durchführbar. Zur Erhaltung von sinnvollen Verarbeitungsgranularitäten wird das Konzept der *Tenantspaces* eingeführt. Tenantspaces ermöglichen eine horizontale Partitionierung der Datenbank, die von einem Knoten des Clusters gespeichert und verarbeitet wird, d. h. Tenantspaces realisieren eine lokale horizontale Partitionierung. Hierbei sorgen Tenantspaces, dass die Partitionierung ausgerichtet an den Mandanten ist.

  Da sich Mandanten hinsichtlich ihrer Größe und ihrem Verhalten über die Zeit hinweg ändern, wird eine Veränderung der Zuordnung von Mandanten zu Tenantspaces notwendig, z. B. durch Aufteilen eines Tenantspaces in zwei neue Tenantspaces. Diese Form der Reorganisation muss nebenläufig zur Transaktionsverarbeitung erfolgen. Es werden hierzu verschiedene Ansätze hinsichtlich ihrer Eignung untersucht. Auf Basis dieser Untersuchung wird ein neuer Ansatz skizziert, der eine gute Eignung für den Anwendungsfall der Reorganisation eines Tenantspaces verspricht.

- In dem mandantenfähigen RDBMS Cluster stellt die Verteilung der Last eine wichtige Aufgabe dar, um Überlast- und Unterlastsituationen vorzubeugen. Zur Verteilung der Last muss eine globale Repartitionierung der Tenantspaces, d. h. Veränderung der Zuordnung von Tenantspaces zu Knoten des Clusters, möglich sein. Hierzu wird ein Migrationsalgorithmus namens *ProRea* vorgestellt. ProRea erlaubt die Verschiebung eines Tenantspaces von einem Knoten zu einem anderen Knoten des Clusters parallel zur Transaktionverarbeitung. ProRea ist ein mehrstufiger Algorithmus: zuerst werden die im Hauptspeicher des Quellknotens befindlichen Seiten in den Hauptspeicher des Zielknotens verschoben, daraufhin wird Transaktionsverarbeitung vom Quellknoten an den Zielknoten übergeben und zuletzt werden die restlichen noch nicht verschobenen Seiten vom Quellknoten auf den Zielknoten übertragen. Falls nach Übergabe der Transaktionsverarbeitung eine Transaktion Zugriff auf eine noch nicht übertragene Seite benötigt, wird diese synchron vom Quellknoten angefordert. In Verbindung mit Snapshot Isolation besitzt ProRea nur einen sehr geringfügigen Einfluss auf die parallel laufende Transaktionsverarbeitung, vorausgesetzt diese besteht im Wesentlichen aus kurz laufenden OLTP Transaktionen.

Abstract

The consolidation of multiple tenants onto a single RDBMS instance, commonly referred to as multi-tenancy, turned out being beneficial since it supports improving the profit margin of the provider and allows lowering service fees, by what the service attracts more tenants. So far, existing solutions create the required multi-tenancy support on top of a traditional RDBMS implementation, i. e., they implement data isolation between tenants, per-tenant customization and further tenant-centric data management features in application logic. This is complex, error-prone and often reimplements efforts the RDBMS already offers. Moreover, this approach disables some optimization opportunities in the RDBMS and represents a conceptual misstep with Separation of Concerns in mind. For the points mentioned, an RDBMS that provides support for the development and operation of a multi-tenant SaaS offering is compelling.

In this thesis, we contribute to a multi-tenant RDBMS for OLTP-style SaaS applications by extending a traditional disk-oriented RDBMS architecture with multi-tenancy support. For this purpose, we primarily extend an RDBMS by introducing tenants as first-class database objects and establishing tenant contexts to isolate tenants logically. Using these extensions, we address tenant-aware schema management, for which we present a schema inheritance concept that is tailored to the needs of multi-tenant SaaS applications. Thereafter, we evaluate different storage concepts to store a tenant's tuples with respect to their scalability. Next, we contribute an architecture of a multi-tenant RDBMS cluster for OLTP-style SaaS applications. At that, we focus on a partitioning solution which is aligned to tenants and allows obtaining independently manageable pieces. To balance load in the proposed cluster architecture, we present a live database migration approach, whose design favors low migration overhead and provides minimal interruption of service.

CHAPTER 1

Introduction

More and more applications and data evaporate off from local desktop computers and rise up into *the Cloud* which delivers them as hosted services. From the user's perspective, the hosted services allow the user to acquire software and hardware by a single click without any efforts, except for the money that the service provider debits from the user.

This new trend for the sales of IT yields new requirements for involved IT components in order to offer successful and profitable services. These requirements do not stop at RDBMS technology, which definitely represents one of the major players in the league of data management technology. To avoid falling off the Cloud, RDBMS technology has to meet these new requirements.

1.1 Cloud Computing

The last couple of years have seen widespread popularity of cloud computing for using and offering computing resources. According to National Institute of Standards and Technology (NIST) [Mell and Grance, 2009],

> “ *Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service.* ”

Put simply, cloud computing is a business model for pay-per-use computing over the network (most notably the Internet) in an efficient way. Thus, cloud computing changes the traditional access and use of IT. As opposed to traditional models in which a user manages, maintains and uses its IT on-premises, the user may now simply consume a suitable service offered by a service provider. The service provider then hosts the required computing resources to offer the service, typically to many service consumers.

1.1.1 Motivation

From the service consumer's perspective, the cloud computing model avoids costs to purchase the infrastructure required during periods of peak load, costs to run and maintain the infrastructure, and costs for staff which purchases infrastructure or manages IT staff. Instead, the service consumer only pays for its actual service consumption. For the service consumer, this model yields good scalability of its IT costs and transfers risk to the service provider. The enumerated costs now occur on the side of the service provider. The service provider, however, is highly specialized, amortizes its costs over many users and leverages economies of scale which ultimately results in far lower costs; the RAD lab estimates that cloud providers have lower costs by 75 to 80 % compared to data centers internal to companies [Armbrust et al., 2010].

The essential characteristics of cloud computing defined by NIST reduce additional overheads to a minimum. Thus, the service provider is able to pass the cost reduction partially to the consumer. In conjunction with the risk transfer from the consumer to the provider, the reduced costs lower barriers to entry for the consumer. The provider may attract new customers who were not able to afford traditional solutions or were unwilling to bear the related risk. All together makes cloud computing computing appealing from an economic point of view, for both: the provider and the consumer.

Despite of its business oriented nature, cloud computing is also about technology: technology that supports the cloud computing business model. Current technology allows for cloud computing in principle, but it does not always allow driving the business to its optimum. Hence, an important requirement is to have technology that supports the business best possible. This requirement comes with several technical challenges, just to mention some highlights: performance and management scalability, isolation of users on highly shared resources to fullfil their individual needs, high diversity of requirements among users, security, diversity of services and availability.

1.1.2 Service Delivery Models

The services offered by cloud computing range from infrastructure resources over platform resources to software resources. Accordingly, the service offerings are classified into three service delivery models: infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS). These three service models together form the cloud computing stack. Fig. 1.1 illustrates the cloud computing stack including the respective roles for the service delivery models and examples of delivered resources.

The top of the stack constitutes SaaS. SaaS offerings represent applications that target the end-user. For instance, salesforce.com[1] represents a popular customer relationship management application which is offered according

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

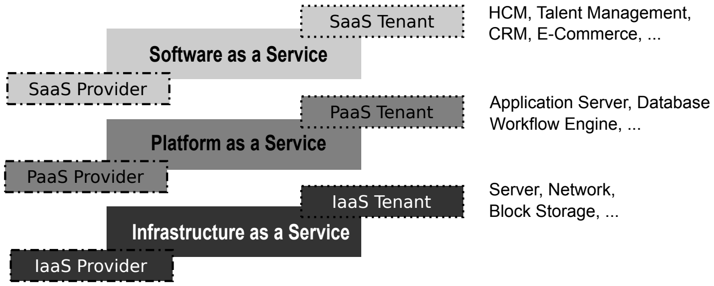

SaaS Tenant
HCM, Talent Management, CRM, E-Commerce, ...
Software as a Service
SaaS Provider
PaaS Tenant
Application Server, Database Workflow Engine, ...
Platform as a Service
PaaS Provider
IaaS Tenant
Server, Network, Block Storage, ...
Infrastructure as a Service
IaaS Provider

Figure 1.1: The cloud computing stack is formed by three different service delivery models which build on each other.

to the SaaS service model. The underlying layer of SaaS constitutes PaaS. Hence, SaaS offerings may be build on top of PaaS offerings. Thus, PaaS offers services to build, deploy and manage applications. For example, SQLAzure[2] represents a PaaS offering that provides databases as a service. Finally, at the bottom of the stack, IaaS offers the hardware that runs everything and the utilities to manage this hardware. One of the most prominent IaaS examples is Amazon EC2[3] which provides virtual machines.

1.1.3 Multi-tenancy

According to the NIST definition, effective resource pooling and sharing represents a critical factor for the success of cloud computing. Effective resource pooling and sharing increases the average resource utilization, by what total costs are reduced. Moreover, it reduces the number of resources which are required to fulfill the same demands as without resource pooling and sharing. A lower number of resources reduces management efforts, energy consumption, compute center floor space, cooling efforts and so forth. Thus, total costs are reduced again.

[2]`http://www.windowsazure.com/en-us/develop/net/fundamentals/cloud-storage`
[3]`http://aws.amazon.com/ec2`

A frequently used technique to improve resource sharing is *multi-tenancy*. Multi-tenancy avoids the need of a private system instance for each service consumer application. For this purpose, multi-tenancy consolidates several service consumer applications onto a single operational system instance, e. g., several consumer application databases onto a single RDBMS instance. At this, a service consumer application is considered a closed organizational unit referred to as *tenant*. In reverse, a tenant relates to a single application instance, at least logically. Thus, each tenant comes with its own users, own configuration and own data.

1.2 Multi-tenancy meets RDBMS technology

Data and therefore data management technology are integral parts for applications, also for cloud based applications. A very popular, wide-spread data management technology constitutes RDBMS technology. Its high proliferation is due to the list of features it offers: overall applicable, intuitive data model, transactions, concurrency control, performance, reliability, availability and more. Additionally, the current RDBMS technology bases upon more than a quarter century of engineering, which makes it a very mature technology.

As a consequence of its high proliferation, RDBMS technology indispensably gets confronted with multi-tenancy, i. e., the consolidation of multiple tenants onto a single RDBMS instance. A frequent scenario is that an application delivered as SaaS offering (SaaS application) consolidates multiple tenants onto a single RDBMS instance. In this case, the RDBMS instance is the data backend of the SaaS application. Current RDBMS implementations lack suitable support for tenant-aware data management required by SaaS applications, though. This is because the design of current RDBMS implementations had a different environment in mind; most RDBMS implementations were built to be used as data backend for a company's applications, usually in the company's own data center. That is, they assume a quite static environment tailored and restricted to one tenant. If the RDBMS is adopted as data backend of a multi-tenant application, thousands of tenants may access the same RDBMS instance. At this, each tenant uses the same application but slightly customized according to its individual needs. Thus, each tenant behaves slightly different and may change its behavior rapidly.

Despite of the lack of tenant-aware data management functionality, traditional RDBMS implementations have been successfully used as data backend in multi-tenant SaaS applications. To compensate for the lack of appropriate functionality, developers of multi-tenant SaaS applications eventually created an additional layer which provides tenant-aware data management functionality on top of the RDBMS. This yields several drawbacks which we subsequently discuss.

Each and every application has to implement its own tenant-aware data management solution on top of the RDBMS, which is complex, error-prone and expensive. As tenant-aware meta data management is typically not specific to a single application, a middleware solution compensates for this issue. Yet, building a tenant-aware data management solution on top of RDBMS requires re-implementing competencies an RDBMS already offers. For example, solutions on top of the RDBMS often implement a data dictionary similar to the data dictionary of the RDBMS [Aulbach et al., 2008; Weissman and Bobrowski, 2009], although multi-tenancy does not change the data model, i. e., the relational model remains. Moreover, these solutions must often parse, interpret and rewrite queries; RDBMS technology is specialized at this and performs these steps anyway. Hence, solutions on top of the RDBMS cause doubled efforts for implementation as well as for runtime, and they constitute another piece of software or system which has to be maintained. Moreover, if the RDBMS loses control and responsibility for the information about data, it also loses some optimization opportunities, e. g., suitable operators for comparison according to the data type and specialized index structures for certain data types.

To sum up, current solutions to tenant-aware data management with RDBMSs degrade the RDBMS to a stupid data bucket and omit to exploit the efforts an RDBMS offers. Based on this issue, we argue that native support of tenant-aware data management in RDBMSs tailored to the needs of multi-tenant SaaS application is compelling and profitable. Native support of tenant-aware data management in RDBMSs is also in line with Separation of Concerns, as it asserts that the RDBMS system remains the main authority for data management tasks.

1.3 Contributions and Outline

This thesis focuses on extending a traditional disk-based RDBMS architecture by functionality that supports tenant-aware data management for OLTP–style applications offered according to the SaaS delivery model. Since current RDBMS technology is wide-spread and represents a mature technology, we consider extending a traditional RDBMS architecture for the described use case a valid challenge that is beneficial in the near future.

The contributions of this thesis and the outline is as follows:

- Before venturing into the actual contributions, Chapter 2 gives further background information that describes the context of this thesis and forms a foundation on which subsequent discussions can stand.
- Thereafter, Chapter 3 starts with establishing the required infrastructure in an RDBMS to provide tenant-centric functionality natively. For this purpose, it introduces tenants as first class database objects and introduces the concept of a tenant context which determines the tenant's virtual database. Based on this, it contributes TenantSchema which constitutes a tenant-aware schema management tailored to the needs of multi-tenant SaaS applications. After describing TenantSchema from a conceptual point of view, implementation considerations follow.
- TenantSchema allows different strategies to map a tenant's tuples to physical storage. Chapter 4 evaluates several such strategies with respect to their scalability for large numbers of tenants. Based on the results of this evaluation, it finally proposes a storage layout for TenantSchema.
- Based on the previous results, Chapter 5 proposes a shared-nothing cluster architecture of a multi-tenant RDBMS for OLTP–style SaaS applications. At this, we focus on tenant management aspects and introduce a concept called *TenantSpaces*. TenantSpaces allow for a flexible horizontal partitioning of the database a certain cluster node serves, i. e., it represents a *local* partitioning approach. TenantSpace hereby align to the natural data partitioning induced by tenants. That is, TenantSpaces allow grouping tenants logically and physically. By

this, TenantSpaces allow to determine the degree of consolidation and the granule for customization, maintenance and administration. Finally, we consider approaches to reorganize TenantSpaces efficiently in parallel to query processing.

- In Chapter 6, we finally contribute a new live database migration approach called *ProRea* which is designed for the introduced shared-nothing multi-tenant RDBMS cluster. ProRea allows for a global repartitioning of TenantSpaces, i. e., it allows changing the assignment of TenantSpaces to cluster nodes by migrating a TenantSpace from one cluster node to another cluster node in parallel to query processing. ProRea is designed for minimal service interruption and low migration overhead, and it assumes multi-version concurrency control as concurrency control mechanism in order to provide snapshot isolation.
- Chapter 7 sums up the results of this thesis. It additionally lists open questions as future work.

CHAPTER 2

Background and Requirements

This chapter describes the context of this thesis and forms a foundation on which subsequent discussions can stand. For this purpose, it briefly reviews the evolution of data management and its requirements in cloud computing. At this, it motivates the adoption of an RDBMS as data backend in multi-tenant SaaS applications. After describing multi-tenancy models for traditional RDBMS implementations, this chapter contributes a list of requirements an RDBMS has to fulfill in order to facilitate the development and operation of multi-tenant SaaS applications. Finally, this chapter describes the system and environment conditions which we assumed for the contributions in this thesis.

2.1 Evolution of Cloud Applications and Cloud Data Management

Within cloud applications, as well as in traditional on-premise applications, data management technology represents an integral part. RDBMS technology undoubtedly is the most wide-spread data management technology used in traditional on-premise applications. Despite of its success in traditional on-premise applications, RDBMS technology was not the data management solution used in the beginnings of cloud computing. The big Internet companies which started offering applications over the Internet, e. g., Amazon, Google, or Yahoo, decided to build their own data management solutions, namely Amazon Dynamo [DeCandia et al., 2007], Google Bigtable [Chang et al., 2006], and Yahoo Pnuts [Cooper et al., 2008]. These systems are commonly referred to as *key-value stores*.

2.1.1 Key-Value Stores

The development of the mentioned key-value stores was motivated by the growing concern and experience that RDBMS technology lacks high scalability, availability, fault-tolerance and elasticity which represent important ingredients for successful cloud data management. Another issue often cited is that existing RDBMS implementations typically induce high administration efforts. These new key-value stores concentrated on eliminating these drawbacks: they scale out to hundreds or even thousands of nodes of commodity hardware, deal with node or network failures, and allow adapting to changed load situations by the capability to add or remove nodes dynamically. Moreover, they offer very good operability due to high automation and sophisticated autonomy.

To achieve the mentioned properties, key-value stores came with a limited set of features compared to RDBMS implementations; they offer simple data and data processing models, e. g., they limit access to one row at a time, use weak consistency models, e. g., eventual consistency [Vogels, 2008] or timeline consistency [Cooper et al., 2008], and often omit to provide access at attribute granularity. There is a broad range of applications that do well with these constraints. A major common characteristic of these applications is that they focus on one kind of object, e. g., web pages or messages. These applications access these objects by their primary key and only access one key at a time [DeCandia et al., 2007]. Furthermore, they require managing petabytes of data and typically use highly parallel data processing frameworks, e. g., Google map reduce[Dean and Ghemawat, 2008]. Crawling and indexing of web pages for Google Search [Chang et al., 2006] and Facebook's messaging system [Lakshman and Malik, 2010] are prominent examples.

With increasing popularity of the cloud computing paradigm, the cloud application landscape got wider and wider. As a consequence, the domains of cloud data management have evolved comparable to the data management domains in traditional on-premise applications. A common domain is data management for OLTP–style workloads; some examples are e-commerce, personal information management, customer relationship management, talent management, and accounting solutions. The functionality offered by key-value stores does not fit the needs of these applications. These applications typically access multiple items at once, often identified by different selection predicates. They come with larger and more complex data models which include several entities and relationships between them. In addition to that, they often require consistent, isolated access to multiple items, e. g., for transferring credits from one account to another account.

If an application developer decides to use a key-value store for such applications, it has to bite the bullet and implement the required functionality within the application itself. The implementation of such functionality, in particular consistency related functionality, is far from trivial and may turn application development into a nightmare. Agrawal et al. [2011c] gave a really nice and comprehensive example about updating bilateral friendship relationships, which actually gets an application development nightmare, in their cloud computing tutorial at EDBT 2011.

2.1.2 Evolving RDBMS Technology in Cloud Data Management

As a consequence of the previously described situation, existing RDBMS implementations emerged more and more for OLTP–style workloads in the cloud, despite of the previously mentioned issues with respect to scalability and availability. In essence, distributed transaction processing influences scalability and availability of an RDBMS implementation. Distributed transaction processing limits scalability and availability, as it requires synchronizing data access across multiple nodes in order to accomplish the ACID principle[Haerder and Reuter, 1983a]. For this purpose, it makes use of heavyweight, complex system components and protocols such as a central lock management, 2 Phase Commit [Weikum and Vossen, 2001] or Paxos [Lamport, 1998]. The synchronization required by distributed transactions causes overhead, e. g., it adds network latency to transaction processing. Moreover, if more nodes are involved in a distributed transaction, the probability of a failing node, and thus aborting of the transaction, increases. This lowers scalability and availability compared to a key-value store which only allows transaction-like processing at row granularity.

Hence, to accomplish good scalability and availability, the goal is to keep distributed transaction processing as low as possible. Data partitioning such that transactions rarely span partition boundaries accomplishes this goal. For this purpose, recent research into this topic covers suitable automatic partitioning solutions [Curino et al., 2010b; Pavlo et al., 2012; Stonebraker et al., 2007a] as well as manual, explicit partitioning solutions [Baker et al., 2011; Das et al., 2010; Tatemura et al., 2012] for RDBMS (or at least RDBMS–like) implementations running OLTP–style workloads.

Interestingly, multi-tenancy exhibits an explicit, natural partitioning of data, namely aligned to tenant boundaries. This natural partitioning lends itself in order to lower distributed transaction processing. Furthermore, in the small to medium business market, a tenant's capacity needs are usually small enough to be fulfilled by a single node [Yang et al., 2009]. Thus, if tenant boundaries are aligned to server boundaries, transactions of a single tenant run locally without synchronization across multiple nodes. This approach enables high scalability without giving up rich functionality. More exactly, it enables scalability with respect to the number of tenants.

In reality, a tenant's capacity demands may naturally exceed the capacity of a single machine. In this case, the system has to partition the tenant's data and redistribute the data across multiple nodes. Thus, another type of scalability is required: scalability with respect to the size of a tenant[1]. As already mentioned, research into appropriate techniques exists [Curino et al., 2010b; Pavlo et al., 2012; Stonebraker et al., 2007a]. With respect to multi-tenant RDBMSs, such techniques are important to deal with the few large tenants. However, it is at least equally important to keep in view the large number of small tenants.

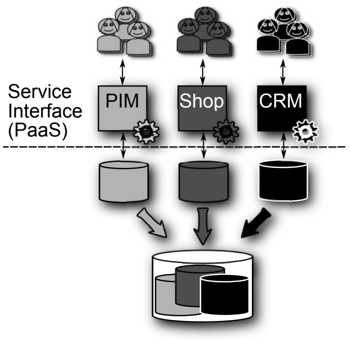
To conclude, RDBMS technology is entering cloud computing for the same reasons it gained wide-spread popularity in traditional applications. Nevertheless, certain domains in cloud computing require better support from RDBMS implementations to ease their adoption and to facilitate the development of applications. In this respect, multi-tenant data management is important. We consider a data management solution as multi-tenant if it is aware of tenants and shares underlying resources among them. In particular in the small to medium business market, multi-tenancy offers an opportunity for RDBMS technology as the natural data partitioning that is given by multi-tenancy allows avoiding distributed transaction processing quite elegantly in most cases.

2.1.3 Use Cases for Multi-tenant Data Management

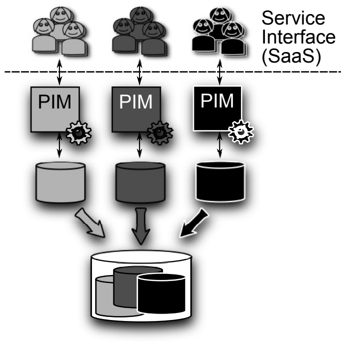
Multi-tenant data management occurs in two different use cases: *database consolidation* and *data backend in a multi-tenant SaaS application* [Reinwald, 2010].

Figure 2.1 illustrates these two use cases. Database consolidation (Fig. 2.1a) targets consolidating databases from different tenants onto a single RDBMS instance. Each tenant is assumed to use the RDBMS as backend for its own application. Thus, tenants use different data structures,

[1]By referring to the size of a tenant, we actually refer to its capacity demands.



(a) Database consolidation



(b) Data backend in a multi-tenant SaaS application

Figure 2.1: Two different use cases for multi-tenant data management.

have different access pattern, manage the schema on their own and execute administrative operations, e. g., schema upgrades, on their own. By contrast, if the RDBMS is adopted as data backend in a multi-tenant SaaS application (see Fig. 2.1b), multiple tenants access the RDBMS through the same multi-tenant application. Although most multi-tenant applications allow for customizations per tenant, the core data model and data access is very similar among tenants. In addition to that, the SaaS provider manages the major part of the schema and decides about most administrative operations.

These two use cases are apparently in line with the cloud computing service models PaaS and SaaS. If a provider offers data storage services according to the PaaS service model, database consolidation might be useful for efficiency. Alternatively, if a provider offers an application according to the SaaS model, consolidation of multiple tenants onto a single data backend instance might be useful for efficiency. Therefore, we refer to these two use cases as *PaaS use case* and *SaaS use case*.

There already exists research into multi-tenant, relational cloud data management for the PaaS use case [Curino et al., 2011a; Das et al., 2009; Zhou et al., 2011]. At first glance, the PaaS use case is no different from the SaaS use case for the RDBMS: a tenant in the SaaS scenario might be considered equal to a tenant in the PaaS scenario. However, the described use cases have different requirements and differ in the potential of sharing. The similarities between tenants in the SaaS use case entail higher potential of sharing than in the PaaS use case. To accomplish good scalability with increasing numbers of tenants, minimizing redundancies and decreasing fixed overheads is mandatory to exploit the offered potential. For example, schema redundancies are actually quite harmful with respect to main memory consumption [Aulbach et al., 2008; Jacobs and Aulbach, 2007]. The challenge to decrease fixed overheads gets more important the smaller the tenants and the larger the number of tenants. Additionally, scalability, implementation efforts and maintenance efforts are important aspects. For this reason, the RDBMS should offer functionality that eases the implementation and the maintenance of a multi-tenant SaaS application, e. g., by a central maintenance of the application schema while enabling per-tenant extensions.

This thesis focuses on the aspects related to the SaaS use case, i. e., the adoption of an RDBMS as data backend in a multi-tenant SaaS application. More specifically, this thesis focuses on native RDBMS support for multi-tenant SaaS application that run OLTP–style workloads. However, many of the discussed challenges and proposed solutions apply for the PaaS use case as well. In our opinion, functionality for the SaaS use case mainly represents an extension to functionality for the PaaS use case.

2.2 Multi-tenant Data Management with Traditional RDBMS

Commonly, SaaS providers building a multi-tenant application implement their own multi-tenant RDBMS. For this purpose, an additional layer on top extends an existing RDBMS implementation with multi-tenant data management in order to obtain a multi-tenant RDBMS.

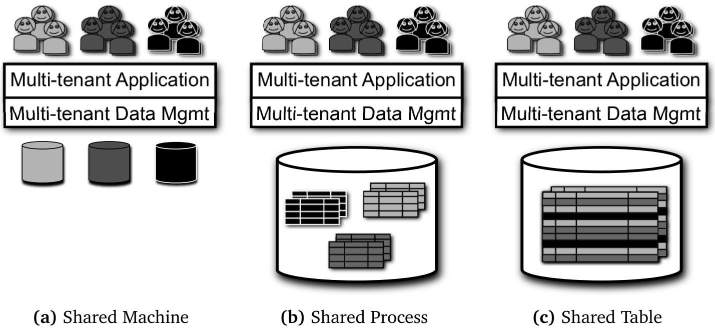


Figure 2.2: Illustration of different multi-tenancy models for RDBMS according to Jacobs and Aulbach [2007].

2.2.1 Multi-tenancy Models

There exist different approaches to implement a multi-tenant RDBMS, which differ in the adoption of the underlying RDBMS, in the implementation complexity and in the degree of sharing [Chong and Carraro, 2006; Frederick Chong and Wolter, 2006; Jacobs and Aulbach, 2007; Reinwald, 2010]. Jacobs and Aulbach [2007] classified different implementations of multi-tenant data management with traditional RDBMS implementations by the three general models. For each model industrial implementations or at least academical evaluations exist (we only name some examples):

Shared Machine. The tenants share a single machine, but each tenant uses a private RDBMS instance (see Fig. 2.2a, Soror et al. [2008]).

Shared Process. The tenants share a single RDBMS instance, but each tenant uses a private database or private set of tables (see Fig. 2.2b, SQLAzure [Microsoft], RelationalCloud [Curino et al., 2011a]).

Shared Table. The tenants share a single database instance and a single set of tables (see Fig. 2.2c, Salesforce.com [Weissman and Bobrowski, 2009], SuccessFactors [SuccessFactors, 2012], Electronic Contract Management [Kwok et al., 2008], Hui et al. [2009]).

Reinwald [2010] has refined this classification. However, for this discussion, the orginal, coarse-grained classification from Jacobs and Aulbach [2007] suffices.

As illustrated in Fig. 2.2, each model implements the multi-tenant RDBMS by means of an additional layer on top of the actual RDBMS implementation. Of course, this layer heavily depends on the used model. For example, tenant creation in Shared Machine requires providing a new RDBMS instance, whereas tenant creation in Shared Process only requires to create a new schema instance. Subsequently, we analyze the three models and their respective properties.

2.2.2 Shared Machine

To implement Shared Machine, several approaches with different abstraction levels exist. For example, machine virtualization allows using a private virtual machine for each tenant. The RDBMS instance of the tenant then runs within its virtual machine. Hence, isolation among tenants is guaranteed at the virtual machine level. Another approach is to start a private RDBMS instance for each tenant within the same operating system. In this case, each tenant uses a private RDBMS process; tenants are isolated at the process level of the underlying operating system.

All Shared Machine implementations eventually share resources beneath the RDBMS, but they do not share RDBMS resources among tenants. Therefore, each tenant comes with a considerable tenant footprint, e. g., the virtual machine footprint or the RDBMS instance footprint. Yet, as each tenant has its private RDBMS instance, Shared Machine offers high security, high isolation and extensive tenant-specific customization possibilities. Moreover, most technology adopted to accomplish Shared Machine already offers an comprehensive set of useful features such as live migration, load-balancing and high availability, e. g., VMware vSphere[2] or Xen Cloud Platform[3]. For the points mentioned, Shared Machine is mainly suitable for enterprise tenants that demand high security, require high customization possibilities, have high

[2]http://www.vmware.com/products/datacenter-virtualization/vsphere
[3]http://www.xen.org/products/cloudxen.html

performance requirements and are willing to pay for this. By contrast, Shared Machine is useless to consolidate thousands of small business tenant that access the RDBMS through a multi-tenant application. The fixed overheads per tenant by Shared Machine prevents an efficient and cheap service which eventually prevents attracting tenants.

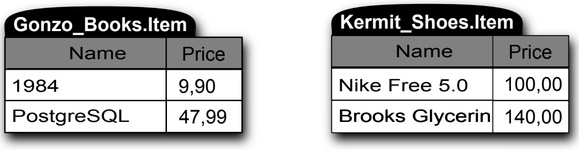
As opposed to Shared Machine, Shared Process and Shared Table are suitable for large numbers of small business tenants, as they share a single RDBMS instance among tenants. This thesis focuses on the consolidation of large numbers of smaller tenants onto a single RDBMS instance in order to achieve higher consolidation efficiencies. Therefore, the remainder of this thesis omits to take into account Shared Machine anymore but keeps in view Shared Process and Shared Table.

2.2.3 Shared Process

Shared Process shares a single database among tenants, but each tenant uses its own set of tables. Hence, although the application logic accesses the same tables for each tenant from a logical point of view, the access has to be routed to the tables that actually belong to the tenant. Therefore, Shared Process requires distinguishing the set of tables of one tenant from other tenants. For this purpose, the typical concepts available for structuring data, i. e., databases, schemas and tables, are sufficient. For instance, tenants may use private schemas but share a single database. Alternatively, tenant may use private tables but share a single schema. The degree of sharing eventually depends on the concrete implementation.

Figure 2.3 shows a simple example for a table *Item* that is defined in two different schemas for the tenants *Gonzo Books* and *Kermit Shoes*. An application component, further referred to as *query rewriter*, rewrites each table reference such that the according tenant's table is accessed. For example, we assume that the tenant *Gonzo Books* issues the following query:

```
SELECT * FROM Item
```


Gonzo_Books.Item

Name	Price
1984	9,90
PostgreSQL	47,99

Kermit_Shoes.Item

Name	Price
Nike Free 5.0	100,00
Brooks Glycerin	140,00

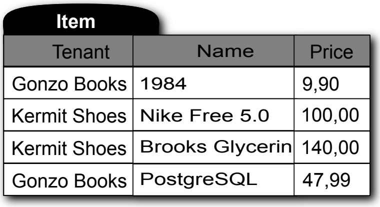
Figure 2.3: Example of the Shared Process approach.

In this case, the query rewriter replaces the referenced table *Item* by *Gonzo_Books.Item* such that the table of *Gonzo Books* is accessed. Hence, the resulting query is as follows:

```
SELECT * FROM Gonzo_Books.Item
```

The implementation of Shared Process, as illustrated, is quite simple. Shared Process offers high tenant-specific customization possibilities. It allows modifying a tenant's schema without influencing the schemas of other tenants. Moreover, as most RDBMS implementations allow administrative operations and definition of configuration parameters at the granularity of tables, tenant-specific administrative operations and tenant-specific configuration is feasible, e. g., tenant-oriented backup or tenant-specific buffer pools. Furthermore, Shared Process offers good isolation between tenants as they are physically segregated.

Research into multi-tenant RDBMS which focuses on the PaaS use case is mainly in line with Shared Process. In the PaaS use case, Shared Process offers a good trade-off between sharing and isolation. However, Shared Process also comes with an inherent drawback; it does not allow deeper sharing among tenants, e. g., sharing of meta data or sharing of indexes. Therefore, Shared Process prevents exploiting the potential of sharing in the SaaS use case (see Sec. 2.1.3).



Item

Tenant	Name	Price
Gonzo Books	1984	9,90
Kermit Shoes	Nike Free 5.0	100,00
Kermit Shoes	Brooks Glycerin	140,00
Gonzo Books	PostgreSQL	47,99

Figure 2.4: Example of the Shared Table approach.

2.2.4 Shared Table

Compared to Shared Process, Shared Table offers a higher degree of sharing since tenants even share the same set of tables. As tenants share the same set of tables, Shared Tables requires a technique to distinguish tuples of different tenants. For this purpose, each shared table obtains an additional attribute *tenant* that stores the respective tenant of each tuple, subsequently referred to as *tenant attribute*. Figure 2.4 shows an example of a shared table *Item* that stores tuples for the tenants *Gonzo Books* and *Kermit Shoes*.

A query rewriter sets the tenant attribute on storing a tuple of a tenant. For example, if the tenant *Gonzo Books* stores the tuple *('1984', 9,90)*, the query rewriter transforms this tuple to *(Gonzo Books, '1984', 9,90)*. Moreover, the query rewriter transforms the selection predicate of queries to retrieve only tuples whose tenant attribute equals the tenant to which a query belongs. For instance, we assume that the tenant *Gonzo Books* issues the following query:

```
SELECT * FROM Item
```

In this case, the query rewriter extends the selection predicate of the original query as follows:

```
SELECT * FROM Item WHERE Tenant = 'Gonzo Books'
```

To allow for efficient query processing, we recommend to add the tenant attribute in index definitions of the shared table. Thus, the RDBMS may use an index to retrieve tuples of one specific tenant. For example, if the table *Item* requires an index for the attribute *Price*, the index definition should actually be *(Tenant, Price)*. Certainly, there are multiple options to place the tenant attribute within the index definition. A discussion of its placement with respect to B-trees follows in Section 4.2.

Similarly, the constraints defined on a shared table typically also should include the tenant attribute. This is because the defined constraints apply for each tenant separately. For example, a unique constraint on the attribute *Name* of the table *Item* must not enforce unique names across all tenants, but it must enforce unique names for a single tenant.

Hence, the tenant attribute essentially implements namespaces. Each namespace relates to one specific tenant and allows segregating tenants among each other.

The Shared Table approach allows consolidating a large number of tenants onto one database instance. The number of tenants is not limited by the available main memory because the size of the meta data remains constant if a new tenant is created. The size remains constant because a tenant does not own dedicated database objects such as tables and indexes. However, this approach comes with lower security relative to Shared Machine or Shared Process as it stores data of different tenants physically intermingled. In addition to that, customization per tenant and operations per tenant are difficult to implement.

If a large number of small tenants access the RDBMS, the Shared Table approach may yield better utilization of the buffer pool than the Shared Process approach. This is because the Shared Process approach allocates dedicated pages for each tenant, whereas the Shared Table approach shares pages between tenants. Therefore, the Shared Process approach tends to have a higher internal fragmentation, especially if tenants only store few tuples in a table. A higher internal fragmentation yields a lower average storage utilization of pages which in turn may decrease the buffer pool utilization.

2.3 Requirements for a multi-tenant RDBMS for SaaS

The preceding part of this chapter shows that multi-tenancy is compelling for RDBMSs which are adopted in SaaS applications in order to accomplish high scalability and low fixed overheads per tenant. A multi-tenant RDBMS used as backend in a SaaS application has to be elastic, fault-tolerant and highly available. The nature of cloud computing mandates these requirements. A multi-tenant RDBMS has to meet additional requirements which are specific to multi-tenancy and thus centric to tenants. From the previous discussions, we derive and sum up these requirements:

Consolidation efficiency. A multi-tenant RDBMS has to support the consolidation of multiple tenants in an efficient manner. Consolidation is efficient if it is able to meet the needs of each tenant similar to without consolidation but, in the entire, carries on a cost benefit. For instance, consolidation may improve resource utilization or may enhance scalability by leveraging synergy effects among tenants.

To achieve highest efficiency, the RDBMS requires sharing system components among multiple tenants to improve their utilization and to amortize their costs across multiple tenants. For example, if tenants use private WAL instances that append log records to a private file at disk, the flush of a tenant's log records writes to a different location at disk than the flush of another tenant's log records. This results in many random I/O operations spread over the disk. By contrast, if all tenants use a single WAL instance that writes to a single location at disk, all log records are written sequentially and the flush of a tenant's log records may even piggy back the flush of another tenant.

Tenant-specific customization. Although tenants access the RDBMS through the same application, tenants may have individual needs. For this purpose, SaaS application are often customizable, e. g., model-based approaches similar to Jakob et al. [2007] are possible. For example, a tenant may want to extend the application core schema[4] of the SaaS application by additional attributes, may want to have

[4]We refer to the schema which the application defines without any tenant-specific customizations as *application core schema*.

a higher degree of fail safety by means of more replicas of its data or may want to have its data stored physically segregated from other tenants. The RDBMS should allow fulfilling these individual needs by tenant-specific customization possibilities. Of course, the more diverse the needs of the tenants are, the less potential for synergetic effects exists.

Tenant-oriented operability. Administrative and maintenance operations should run at the granularity of tenants or a set of tenants. For example, in a cluster environment the system should allow moving a tenant from one cluster node to another cluster node or should allow replicating a tenant's database. Of course, this requirement is tightly coupled to tenant-specific customization; tenant-oriented operability allows to run and customize administrative or maintenance operations according to the needs of a tenant or a group of tenants.

Isolation and security. Despite of the consolidation of multiple tenants onto the same system, a tenant's access has to be restricted to its own data. For this purpose, the RDBMS itself has to provide adequate access control mechanisms and security domains that account for tenants and prevent that a tenant is able to access data of another tenant. Access control mechanisms should be complemented by further security measures such as encryption or redundant supervision. Such measures establish security at the system level.

Another important isolation aspect is about performance. A multi-tenant RDBMS should support to cater for the service quality defined as service level agreements (SLA) in the service contract. If the defined SLAs include performance guarantees, e. g., throughput or latency, the RDBMS has to allow for performance isolation of tenants such that all SLAs are met and the total infrastructure is utilized near the optimum. For example, approaches are proactive or reactive limitations, such as limiting the allowed number of parallel running transactions per tenant or limiting the total resource consumption of a transaction, dynamic admission control and load-balancing. Such approaches naturally require appropriate auditing and monitoring of a tenant's actual resource consumption.

OLTP	OLAP
Pre-determined queries	Ad-hoc queries
Simple queries	Complex queries
Small found sets	Large found sets
Short transactions	Long transactions
Update/Select	Select (Read-only)
Real time update	Batch update
Detail row retrieval	Aggregation and group by
High selectivity queries	Low selectivity queries

Table 2.1: Differences in the workload characteristics of OLTP and OLAP French [1995].

2.4 Environment Conditions

The design of systems and related algorithms depend on the environment, i. e., the use case and system infrastructure, in which they are adopted. From this perspective, a clear understanding of the environment is important for successful system and algorithm design.

2.4.1 Application

The contributions in this thesis assume a multi-tenant SaaS application which runs OLTP–style workloads. OLTP is one of the two domains in which data management have been split, the other domain is OLAP. Table 2.1 shows the different workload characteristics of OLTP and OLAP. OLTP–style workloads are common in interactive application, e. g., e-commerce, personal information management and health care management.

Furthermore, we assume that the tenant may partially customize the application with respect to the data model, the business processes and the user interface. The tenant-specific customization is feasible by means of an architecture in which the customizations related to a certain tenant are described through meta data [Kim et al., 2014; Weissman and Bobrowski, 2009]. That is, the application actually interprets meta data to know its data

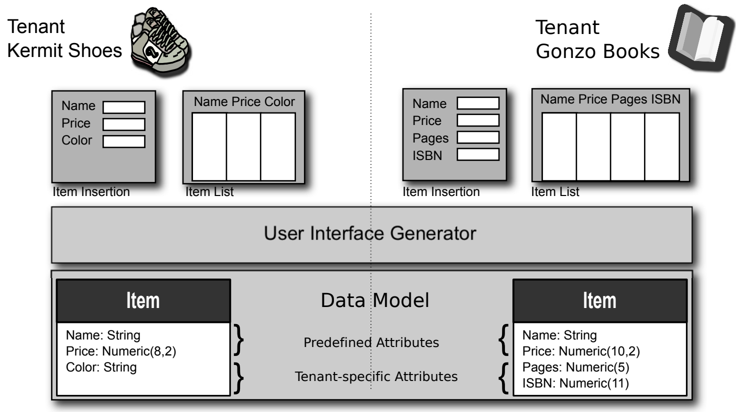


Figure 2.5: Example of a customization in a multi-tenant SaaS application.

model, its behavior and its user interface with respect to a certain tenant. Fig. 2.5 illustrates a simple example. In this example, a SaaS provider offers an online shop as a service. Thus, tenants may use this online shop to sell their goods. The online shop comes with a table *Item* that stores the items to sell. There exists a list view that lists all items and an insertion form that allows inserting a new item. The tenants Kermit Shoes and Gonzo Books have customized the online shop by adding additional attributes to the table *Item*. Kermit Shoes added a new attribute *Color* and Gonzo Books added two new attributes *Pages* and *ISBN*. Hence, the application has to interpret the meta data to generate an appropriate list view and an appropriate insertion form. A deeper discussion of the implementation of such an application is beyond the scope of this thesis.

2.4.2 Hardware Infrastructure

Furthermore, we assume commodity servers as hardware infrastructure. Nowadays, it is common to use many commodity servers to accomplish large, highly scalable systems. If the system requires more capacity, new commodity servers are simply added. If the system requires less capacity, commodity server are released or even shut down. Commodity servers are cheap, provide good quality and base upon open standards.

Note that the system may also run on commodity-like virtual servers instead of physical machines. For example, a SaaS provider that offers a multi-tenant SaaS application may run the related RDBMS on virtual machines offered by a IaaS provider. In this case, the SaaS provider takes advantage of Cloud Computing to offer its Cloud Computing offering.

2.5 Summary

An RDBMS implementation that supports multi-tenant OLTP–style SaaS offerings is definitely reasonable and feasible. A major assumption hereby is that tenants are relatively small, i. e., a tenant requires only few machines, in most cases even less than one machine. Under this assumption, consolidation improves average infrastructure utilization. Furthermore, high scalability for one tenant, i. e., a single logical application instance, is not required. As a consequence, efficient transaction processing without sacrificing features or performance is feasible. Yet, the actual challenge is to accomplish good scalability of performance, manageability and operability for large numbers of tenants.

The remainder of this thesis deals with this challenge by contributing to an RDBMS for OLTP–style multi-tenant SaaS applications.

CHAPTER 3

Tenant-awareness and Schema Management

The presented multi-tenancy models Shared Process and Shared Table share RDBMS resources among tenants by consolidating them onto a single RDBMS instance. For this purpose, Shared Process and Shared Table implement data isolation between tenants, tenant-aware schema management and further tenant-centric data management features in an additional layer on top of the RDBMS. This approach is complex, omits to exploit the capability of the RDBMS, disables some optimization opportunities in the RDBMS and represents a conceptual misstep with Separation of Concerns in mind.

This chapter presents a multi-tenant schema management approach inspired by our previously published work [Schiller et al., 2011]. To support multi-tenant schema management and other tenant-centric functionality, the RDBMS primarily has to be aware of tenants. For this purpose, Section 3.1 embarks upon presenting an integration of tenants into an RDBMS. Based on the integration of tenants into the RDBMS, Section 3.2 presents a multi-tenant schema management approach tailored to the needs of a multi-tenant SaaS application. Thereafter, Section 3.3 proposes a language extension to manage the introduced concepts, and Section 3.4 discusses the integration of the presented concepts in a typical RDBMS architecture. Finally, measurements using a preliminary implementation of the presented concepts conclude this chapter.

3.1 State of the Art of Schema Management for Multi-tenant SaaS applications

Modern RDBMS implementations provide a data catalog that stores information about data, i. e., meta data, and allows tailoring a database to the needs of a specific application. This data catalog roughly stores two different kinds of meta data: physical and logical. The physical meta data describes how data is stored on disk and what the available access paths are. The logical meta data expresses the application-specific data structures by means of the relational model, i. e., by tables, attributes and constraints.

In the SaaS use case, the contents of the data catalog look closely alike between tenants, especially with respect to the logical meta data. This is because the application that accesses the system is the same across all tenants. Thus, each tenant uses the same application core schema which only slightly differs between tenants by reason of per-tenant extensions.

Subsequently, we describe how Shared Process and Shared Table deal with this situation and what the related drawbacks of both approaches are.

3.1.1 Shared Process

Shared Process creates for each tenant a private instance of the application core schema. Therefore, Shared Process facilitates tenant-specific extensions of the application core schema by means of standard RDBMS functionality. Furthermore, the modification of a tenant's schema does not interfere with schemas of other tenants. Thus, Shared Process exploits the capability of the RDBMS with respect to schema definition and management.

However, the modification of the application core schema requires iterating over all existing instances and applying the corresponding modification for each instance. Current RDBMS implementations lack support for this task. Hence, a suitable implementation, either external or within the RDBMS, is required to manage the application core schema.

3.1.2 Shared Table

The basic principle of Shared Table is simple. Nevertheless, compared with Shared Process, Shared Table becomes quickly complicated if tenants want to extend a table of the application core schema according to their own needs, e. g., by adding new attributes. If tenants extend a table of the application core schema, the logical structure of the table may slightly differ from tenant to tenant. This calls for appropriate mapping techniques which map the differently structured logical tables of tenants onto a fixed shared table defined within the RDBMS.

A straightforward approach is to add tenant-specific attributes to the schema of the shared table. The schema of the shared table then represents the union of the tenants' logical table schemas. This approach scales poorly because it yields a wide sparse table, though. For instance, if each of 10,000 tenants defines five additional attributes, a tuple stores useful values only in a small fraction of the total number of attributes (only 5 of 50,000 attributes are specific to one tenant) while filling the remaining attributes with NULL values. Wide sparse tables occur often as a result of schema inflexibility [Agichtein and Gravano, 2003; Agrawal et al., 2001; Chen et al., 2012; Raman et al., 1998], and it is known that a wide sparse table tends to low performance if the attributes of a tuple are stored in position-based and row-oriented manner. The low performance is due to the overhead of storing NULL values [Beckmann et al., 2006a]. Research into wide sparse tables examines how to get rid of this overhead. A promising result represents the Interpreted Attribute Storage Format that is implemented in Microsoft SQL Server as Sparse Columns feature [Beckmann et al., 2006a]. As opposed to positional tuple representations, the tuple is represented by a list of key-value pairs in which the key references the related attribute definition in the system catalog. By this, the Interpreted Attribute Storage Format is able to avoid storing NULL values, but it prevents random attribute access optimizations as it has to retrieve the offset of a tuple at access time by scanning the whole tuple. Other research proposes to infer hidden schemas from the wide sparse table and define partial indexes [Stonebraker, 1989] which cover the inferred schemas [Chu et al., 2007]. This would eventually result in a similar approach as Shared Process since each tenant uses a dedicated set of indexes. In addition to the challenge of storing NULL values, another issue with wide sparse tables related to popular RDBMS implementations is the limited number of supported attributes per table, e. g., IBM DB2 V10.1 LUW supports a maximum of 1012 [IBM, 2012] and Oracle 11g a maximum of 1000 attributes per table [Oracle, 2008b].

To prevent such scalability and performance issues, other approaches that rely on mapping the tenants' logical schemas onto a *fixed generic* schema in the RDBMS have been adopted. Subsequently, we outline three popular techniques [Aulbach et al., 2008; Chong and Carraro, 2006]:

Pivot Table. This technique maps each cell value of the tenants' logical tables to one row of a single pivot table. To identify the cell of the logical table, it additionally stores the tenant, the row number and attribute number. That is, the Pivot Table describes the function: $(tenant \times rowno \times attributeno) \rightarrow value$. The data type of the value is typically flexible, e. g., varchar. Yet, typing is possible by maintaining one Pivot Table for each data type. Analogously, indexed and non-indexed attributes can be implemented.

This approach suffers from high runtime overhead because reassembling a tuple with n attributes requires $n-1$ joins. Moreover, the high amount of cell identification data compared to real application data may degrade the buffer pool hit ratio.

The Pivot Table approach focuses on wide sparse data sets with a very irregular distribution of NULL values. Hence, this approach omits to exploit that the total data set is partitioned by tenants into a limited number of narrow, dense subsets.

Universal Table. A Universal Table includes, in addition to the attributes of the core application schema, a preset number of custom attributes that enable storing tenant-specific attributes. If a tenant adds an attribute to its logical table schema, the attribute is mapped onto an unused custom attribute of that tenant.

The number of preset custom attributes is critical. It must be high enough to meet the needs of all tenants. Yet, if the maximum number of custom attributes is high, e. g., 200, but most tenants only require a small number of custom attributes, e. g., 5, there is again overhead from storing NULL values.

The preset custom attributes require flexible data types to allow storing arbitrary values. Therefore, the RDBMS loses any information about data types. Thus, appropriate type information requires to be maintained by the application itself.

Extension Table. This technique stores the attributes of the application core schema in a shared table and the tenant-specific attributes in a so-called extension table. A common surrogate ties the parts of the tuple across the tables. A tenant may possess an own extension table or it may share one extension table with others if their tenant-specific attributes are exactly identical.

This approach has the same scalability issues as described for the Shared Process approach due to the potentially high number of tables. To prevent this, all tenants may alternatively share one extension table that in turn adopts an approach such as Pivot Table or Universal Table. In this case, the same issues as described for these approaches apply.

There exist variants and hybrids of the two described approaches. Aulbach et al. [2008] propose a technique called Chunk Folding. This technique stores the tenant-specific attributes in a fixed set of so-called Chunk Tables. A Chunk Table is similar to a Pivot Table, but the Chunk Table stores a set of columns instead of only one column. The logical tables are divided into suitable chunks and distributed across the fixed set of Chunk Tables. Other related mapping techniques have been proposed and evaluated in the context of storing XML [Florescu and Kossmann, 1999].

3.1.3 Drawbacks of existing approaches

Shared Process and Shared Table both require an additional component on top of the RDBMS which conducts the multi-tenant schema management, e. g., iterating over multiple schema instances or storing the data type of a tenant-specific attribute.

This component is simpler for Shared Process than for Shared Table since Shared Process largely exploits the capability of the RDBMS. However, Shared Process comes with significant redundancy because each tenant obtains a dedicated instance of the application core schema. Therefore, Shared Table forfeits scalability which counts particularly for large numbers of tenants. Contrarily, Shared Table offers higher scalability, but the component on top of the RDBMS has to reimplement many of the features an RDBMS already offers. The component has to map the tenants' logical table structures onto the shared table defined in the RDBMS. For this purpose, it has to store and manage meta data about the tenants' table structures which may also limit the scalability and does not avoid redundancy of meta data and data between tenants per se.

To reduce the complexity of the implementation on top of the RDBMS, the RDBMS may take over some of the required functionality by means of SQL views and INSTEAD-OF triggers. Yet, SQL views that apply for all tenants and act as tenant filters on Universal Tables still require that the application maintains the tenants' logical table structures. Alternatively, per-tenant SQL views in conjunction with a generic table layout that keeps more of the data dictionary in the RDBMS, e. g., Extension Tables, could be adopted. Yet, this approach requires that the application manages the views and the triggers per tenant. For this purpose, the application has to maintain mappings between tenants, views and underlying tables by what per-tenant schema customization still requires considerable management efforts in the application.

Many of the presented techniques to achieve per-tenant schema extension for Shared Table target high flexibility at the schema level. To obtain this flexiblity, these techniques come with tuple reconstruction or null compression overhead. In the scope of multi-tenancy, lower flexibility suffices because a tenant's logical table structure is well-known and the corresponding data set is dense. A tenant-aware data catalog can help to exploit this fact for an efficient storage layout (see Sec. 4.4).

The discussed points demonstrate that a multi-tenant schema management which is natively integrated into the RDBMS is reasonable in order to achieve good scalability and simplify the development of multi-tenant SaaS applications.

3.2 Tenants as First-Class Database Object

A central goal to support multi-tenancy in an RDBMS is that tenants share physical RDBMS resources and database objects. Despite of the resource and object sharing, each tenant must seem to have individual resources and objects. From a conceptual view, each tenant requires virtual RDBMS resources and a virtual database that contains the objects that relate to the tenant and isolates it from other tenants. To establish this kind of virtualization, we introduce the idea of a *tenant context* and a *tenant context manager*, as illustrated in Fig. 3.1.

A tenant's context keeps all information to determine the tenant's virtual database. Furthermore, it includes information such that the RDBMS engine behaves according to the tenant's configuration. Thus, the tenant context represents an execution context for operations and transactions. For this reason, each transaction or operation is associated with the tenant context of the tenant that issued it. The RDBMS engine considers the associated tenant context to carry out the transaction or operation appropriately. For example, a replication component may route replication commands dependent on the tenant context of the query that caused the replication commands.

Different tenant contexts may reference the same objects, or they may reference objects that belong only to a certain tenant. Thus, the tenant contexts allow for flexbile sharing patterns. For example, a set of tenants may use the same WAL instance, whereas another set of tenants may use private WAL instances. Naturally, the WAL component has to be aware of

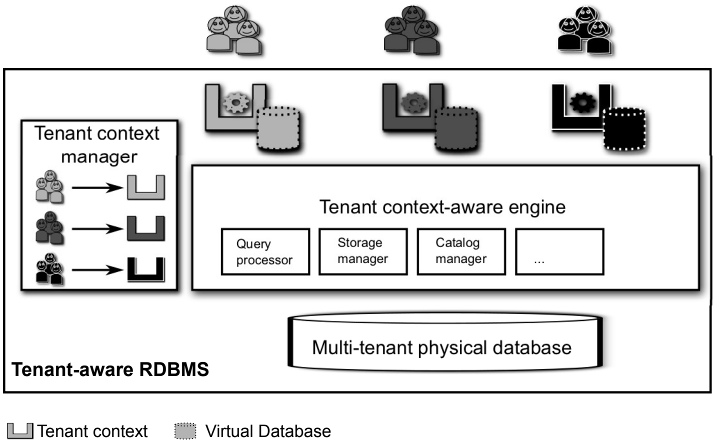


Figure 3.1: The tenant context manager maps tenants to their corresponing tenant context. The context creates a tenant's virtual database from and with the underlying physical components and database objects.

this fact and my require to cater for a distinction of entries that belong to different tenants.

The tenant context manager manages available tenant contexts, e. g., creation and destruction. Moreover, it serves as primary target for components of the RDBMS engine in order to retrieve the tenant context associated with the tenant that has issued the transaction or operation under concern.

To handle, maintain and identify tenant contexts, an RDBMS must know about tenants in the first place. Therefore, we introduce tenants as first-class database objects that identify a certain context. Furthermore, we assume that the RDBMS knows the tenant that performs a transaction or an operation. With this information, the RDBMS is able to process the transaction or operation sensitive to the tenant. Note that the transaction or operation of a specific tenant is restricted to its tenant context. Hence, the tenant context determines a logically closed container with inviolable boundaries that establishes system-level security and isolation between tenants.

3.3 TenantSchema— Native Tenant-aware Schema Management

To overcome the drawbacks mentioned for Shared Process and Shared Table, we subsequently present TenantSchema which is our multi-tenant schema management approach. TenantSchema is tailored to the needs of multi-tenant SaaS applications and integrated into the RDBMS. It allows maintaining an application core schema while enabling per-tenant schema extensions. With TenantSchema, the RDBMS completely maintains the meta data, such as data types of tenant-specific attributes. Furthermore, TenantSchema facilitates direct access to the RDBMS without the need of a complex component on top of the RDBMS and it seamlessly integrates into the concept of tenant contexts.

To achieve the objectives mentioned above, TenantSchema picks up the idea of SQL schemas as proposed by the SQL standard. SQL schemas represent namespaces that allow a user to group database objects logically. For example, schemas allow the user to segregate different applications or independent parts of an application that use the same database. TenantSchema adopts SQL schemas with a slightly different intention. It uses them to group objects that are redundant among tenants and to segregate objects of different tenants. For this purpose, TenantSchema comes with two different schema types: *virtual schema* and *tenant schema*. Furthermore, it defines a schema inheritance concept which allows deriving a schema from another schema. Thereby, a derived schema inherits the objects that are defined in the parent schema. TenantSchema prohibits modifying or removing inherited objects, but it allows extending and creating objects according to a defined set of rules.

Subsequently, we describe each schema type and the respective inheritance rules in detail. For this description, we primarily define a table definition and a table instance as follows:

Definition 3.3.1 *A* table definition $d_s(t) = (R, C)$ *represents the structure of a table t in schema s as a relation $R = (a_1, ..., a_n)$ of n typed attributes $a_1, .., a_n$ and a set C of constraints over the n attributes. A* table instance $i_s(t)$ *of a table*

t in schema s contains tuples according to $d_s(t)$ which are visible to schema s. Each table definition has at most one related table instance.

3.3.1 Virtual Schema

This schema type intends to define the application core schema which a tenant may extend according to its individual needs. Hence, a virtual schema describes the schema parts that are invariant between tenants.

Virtual schemas are arranged hierarchically. The hierarchy describes an inheritance relation. A virtual schema s_2 can inherit from another virtual schema s_1. s_2 hereby inherits all the database objects contained in s_1. Thus, an inheritance relation between two schemas also imposes an inheritance relation on the contained objects. That is, if s_1 contains $d_{s_1}(t)$, s_2 contains $d_{s_2}(t)$ which represents a child of $d_{s_1}(t)$ with respect to inheritance. To describe the relation between objects with respect to inheritance, we use the terms subsequently defined:

Definition 3.3.2 *The term $p(x)$ refers to the predecessor of x with respect to inheritance. That is, if x represents the child of y with respect to inheritance, $p(x) = y$ holds. Furthermore, $P(x)$ describes the set that contains the* chain of predecessors *starting from x, i. e., $P(x) = \{y_n \mid (x, y_1, \ldots, y_n, \ldots, y_m), p(x) = y_1, p(y_{i-1}) = y_i, 1 < i, n \leq m, \nexists y_{m+1} : p(y_m) = y_{m+1}\}$.*

TenantSchema allows extending inherited table definitions by new attributes and new constraints. The extension is expressed by a table definition extension which we define as follows:

Definition 3.3.3 *A table definition extension $e_{s_2}(t)$ of table t in schema s_2 is $((a_{n+1}, ...a_m), C_2)$ if $d_{s_1} = ((a_1, ..., a_n), C_1)$, $d_{s_2} = ((a_1, ..., a_n, a_{n+1}, ...a_m), C_1 \cup C_2)$ and $p(d_{s_2}(t)) = d_{s_1}(t)$, where C_2 is a set of constraints over the m attributes of $d_{s_2}(t)$.*

Descriptively, a table definition extension in schema s_2 represents the additional attributes and additional constraints that s_2 defines for the related inherited table definition.

In addition to extending inherited table definitions, TenantSchema also allows defining new tables in a derived virtual schema. Nevertheless, our concept mandates that the names are unique across the whole inheritance path.

TenantSchema however forbids modifying inherited table definitions. For example, renaming or reordering of attributes defined in the parent table definition is disallowed. This guarantees that index and constraint definitions remain valid for derived tables. Furthermore, newly created attributes always have to follow after the attributes that already exist. This constraint avoids intermingling attributes of different schemas which helps to stay consistent with application code, as the application may access the attributes by their position. In our opinion, an application should pursue a clean separation between data and its presentation anyway and, thus, the ordering of the attributes in the data layer should not be used to influence the ordering of the attributes in the presentation. In addition to that, TenantSchema mandates that the primary key stays consistent for a table definition across the inheritance path.

The virtual schema got its name because an application is unable to use a virtual schema for query processing. A virtual schema represents an abstract base schema which is redefined by tenant schemas. Therefore, a virtual schema does not require a table instance for a contained table definition. A table instance $i_{s_1}(t)$ related to a table definition $d_{s_1}(t)$ of table t in the virtual schema s_1 is only created on demand, i. e., if an administrator decides to insert tuples into t for s_1. If a table definition $d_{s_1}(t)$ has a related table instance $i_{s_1}(t)$, a schema s_2 with $p(s_2) = s_1$ contains $d_{s_2}(t)$ and a table instance $i_{s_2}(t)$; $d_{s_2}(t)$ is derived from $d_{s_1}(t)$ and $i_{s_2}(t)$ is derived from $i_{s_1}(t)$. Hence, $i_{s_1}(t)$ could be considered as default content of t for all schemas s, where $s_1 \in P(s)$. Note that a tenant is not allowed to change the contents of table instances in virtual schemas. We propose that an administrator of the SaaS provider which is responsible for the virtual schema maintains the related table instances.

If the table t is extended by $e_{s_2}(t)$, the table instance $i_{s_2}(t)$ is extended correspondingly, i. e., tuples of $i_{s_1}(t)$ are extended by the attributes defined by $e_{s_2}(t)$. We refer to this extension as *vertical table portion* $v_{s_2}(t)$. The attributes

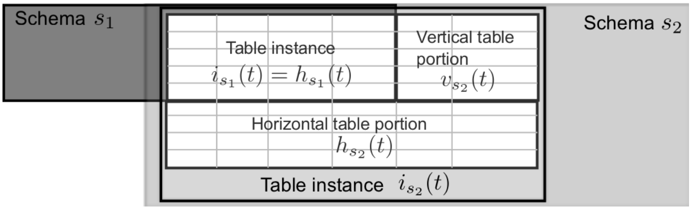


Figure 3.2: Table instance $i_{s_2}(t)$ visible to schema s_2 is composed of $i_{s_1}(t)$ inherited from schema s_1, $h_{s_2}(t)$ and $v_{s_2}(t)$.

in a vertical table portion are initially filled with their given default value or NULL if no default value is specified. The resulting table instance $i_{s_2}(t)$ naturally enables the insertion of tuples. We refer to the tuples which have been actually created within s_2 as *horizontal table portion* $h_{s_2}(t)$. The tuples that belong to the horizontal table portion can be modified and deleted. To sum up, a table instance actually consists of the combination of horizontal and vertical table portions. In general, a table instance $i_s(t)$ is assembled by the following recursive equation:

$$i_s(t) = (p(i_s(t)) \bowtie v_s(t)) \cup h_s(t) \tag{3.1}$$

Thus, in our example with $p(s_2) = s_1$, the table instance of t in s_2 is assembled by $i_{s_2}(t) = (i_{s_1}(t) \bowtie v_{s_2}(t)) \cup h_{s_2}(t)$. Figure 3.2 illustrates the assembly of a table instance by assuming that schema s_2 inherits from schema s_1.

The previous discussion considered only logical aspects, schema inheritance should however account for certain physical aspects as well, e. g., index definitions, mappings to storage containers, and so forth. Index definitions are a standard concept in RDBMS implementations. We propose that if a schema s_1 defines indexes for a table t of s_1, s_2 with $p(s_2) = s_1$ inherits those indexes as well as tables. Yet, as opposed to tables, the definition of an index is not expandable.

Note that TenantSchema omits to consider overriding inherited data, i. e., tuples of an inherited table instance, so far. Overriding inherited data is supposed to be useful functionality with respect to tenant-specific customization, though. Overriding inherited data essentially means that a tenant may modify a shared, inherited tuple, but its modification has to be invisible to other tenants. Copy-on-write for modification and ghost entries for deletion are concepts which allow implementing data overriding. Other research into tenant-aware schema management which happened parallel to ours takes into account data overriding by copy-on-write[1].

3.3.2 Tenant Schema

As opposed to virtual schemas, a tenant schema relates to a specific tenant, i. e., each tenant possesses an associated tenant schema that represents a part of its context. A tenant schema must inherit from a virtual schema. The same rules as described for a virtual schema with respect to extending and creating objects apply. From this perspective, a tenant schema behaves similar to a virtual schema, but there exist two important differences: a tenant schema enforces to have related table instances for each contained table definition, and a tenant schema is final with respect to inheritance, i. e., another schema cannot inherit from a tenant schema. Tenant schemas are not created explicitly, but the creation of a tenant automatically initiates the creation of an associated tenant schema. The creation of the tenant schema includes creating instances of all contained table definitions that are without a related table instance by inheritance.

Figure 3.3 illustrates a simple example of the presented concept. The upper part of the figure models the virtual schema *Shop* that defines a table *Item* and a table *Color* which additionally has a related table instance. The lower part of the figure illustrates two derived tenant schemas: the tenant schema *Kermit Shoes* on the left and the schema *Gonzo Books* on the right.

[1] Aulbach et al. [2011] propose a multi-tenant schema management concept which is similar to ours. Both contributions have been done independently and have been submitted to different conferences nearly at the same time. Their contribution concentrates on data overriding and schema evolution aspects in a main-memory RDBMS, whereas we concentrate on the implementation aspects and the benefits of sharing the application core schema. From our point of view, both contributions are important. Furthermore, we regard taking their results and evaluating how to implement data overriding and schema evolution in TenantSchema as extension of a traditional disk-based architecture an interesting direction for future work.

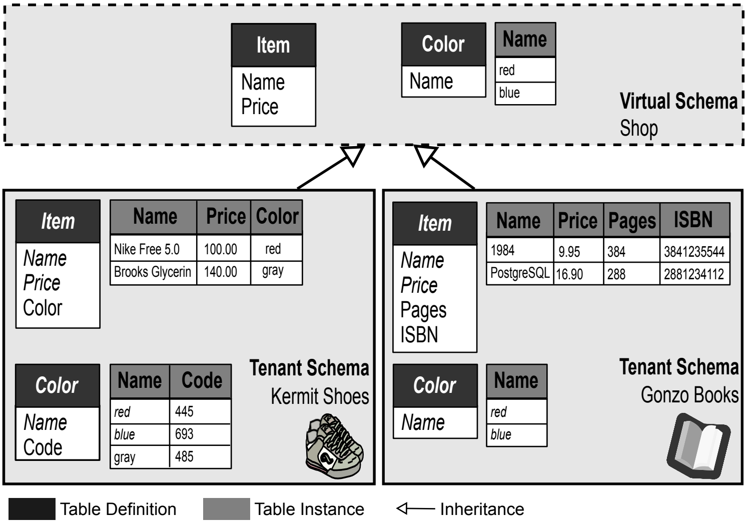


Figure 3.3: Illustration of our schema inheritance concept using the introduced e-commerce scenario. The italic parts represent parts which are inherited.

Kermit Shoes extends *Item* by an attribute *Color*, whereas *Gonzo Books* extends it by an attribute *Pages* and another attribute *ISBN*. *Kermit Shoes* additionally extends *Color* by an attribute *Code*. In addition, the tenant schemas contain the respective instances of *Item* and *Color*. Note that *Kermit Shoes* extends the inherited instance of *Color* (vertical table portion due to the additional attribute *Code*, and horizontal table portion due to the additional tuple which represents the color gray).

3.4 Language Extension

To manage schema hierarchies and tenants, we propose extending the SQL language by suitable statements. The following listing enumerates the statements that we devised to create schema hierarchies:

```
CREATE VIRTUAL SCHEMA <schemaname>
    [ INHERITS FROM <schemaname> ]
DROP VIRTUAL SCHEMA <schemaname>
```

The statement `CREATE VIRTUAL SCHEMA` creates a new shared schema with the given name. The optional clause `INHERITS FROM` specifies the virtual schema from which the newly created schema inherits. The statement `DROP VIRTUAL SCHEMA` drops the schema with the given name as well as derived schemas. As dropping a schema is obviously an operation which may have far-reaching consequences and is very harmful in case of misoperation, we recommend to add some secureness, e. g., by restricting dropping a schema if it still has descendants with respect to schema inheritance.

To address an object in a schema, the schema must prepend the name of the object. For example, the statement `CREATE TABLE Shop.Item` creates the table *Item* in the schema *Shop*.

To manage tenants, we propose the following statements:

```
CREATE TENANT <tenantname>
    [ SCHEMA INHERITS FROM <schemaname> ]
DROP TENANT <tenantname>
SET TENANT {<tenantname>|None}
```

The `CREATE TENANT` statement creates a new tenant with the given name. In addition to creating a tenant, the statement creates an associated tenant schema. The optional clause `SCHEMA INHERITS FROM` specifies a virtual schema from which the tenant's schema inherits. The `DROP TENANT` statement drops the given tenant as well as dependent objects, most notably the associated tenant schema.

In order to carry out operations within a tenant's context, the RDBMS requires to know the tenant that performs an operation. For this purpose, the `SET TENANT` statement explicitly causes the system to switch to the given tenant. Instead of explicitly setting the tenant, the authentication process may set the appropriate tenant by mapping users to tenants during connection establishment. Nevertheless, applications typically exploit a connection pool to avoid the overhead of re-establishing a connection. Thus, switching a tenant on an established connection is necessary. Our mechanism supports this requirement. To increase security, we envision an authentication mechanism that extends our approach, similar to proxy authentication in Oracle's database software [Oracle, 2012].

3.5 Implementation

The implementation of the previously introduced concepts requires in-depth interventions in an RDBMS implementation which affects several components. This section identifies affected components and briefly describes related implementation solutions.

3.5.1 Architecture Overview

Although there exists a variety of different RDBMS implementations, the core usually looks alike and typically consists of five components: process manager, client communication manager, relational query processor, transactional storage manager, shared components and utilities [Hellerstein et al., 2007]. Figure 3.4 depicts a simplified version of a typical RDBMS architecture according to Hellerstein et al. [2007]. Figure 3.4 additionally shows the extensions required for the implementation of tenant-awareness and TenantSchema: The catalog manager maintains tenants and schema hierarchies. The query parsing module within the relational query processor supports statements to manage tenants and schema hierarchies. The plan executor takes into account the tenant context with which a query is associated. For this purpose, it consults the tenant context manager which represents a new module that belongs to shared components and utilities.

3.5.2 Catalog Data Model

Figure 3.5 depicts the entities and the corresponding relationships which are relevant for the implementation of TenantSchema. This conceptual view of the data model is valid in the general case, but its mapping to a concrete implementation is highly specific. Our prototypical implementation was inspired by the catalog of PostgreSQL upon which our implementation

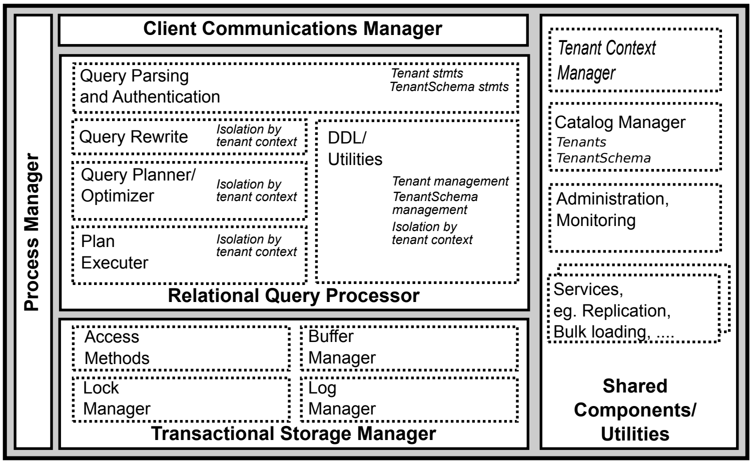

Process Manager
Client Communications Manager
Query Parsing and Authentication
Tenant stmts
TenantSchema stmts
Query Rewrite
Isolation by tenant context
DDL/ Utilities
Query Planner/ Optimizer
Isolation by tenant context
Tenant management
TenantSchema management
Isolation by tenant context
Plan Executer
Isolation by tenant context
Relational Query Processor
Access Methods
Buffer Manager
Lock Manager
Log Manager
Transactional Storage Manager
Tenant Context Manager
Catalog Manager
Tenants
TenantSchema
Administration, Monitoring
Services, eg. Replication, Bulk loading, ...
Shared Components/ Utilities

Figure 3.4: Blueprint of an RDBMS architecture including the extension for TenantSchema (italic).

bases. Figure 3.6 exemplifies the resulting catalog tables through a catalog excerpt of our e-commerce scenario.

The catalog table *Tenant* stores available tenants. They are identified by unique names and system-generated integer identifiers. According to our e-commerce scenario, Fig. 3.6 shows two entries for the tenants *Gonzo Books* and *Kermit Shoes*. For the sake of readability, we use names to indicate references to tuples. The system actually uses integer identifiers to store references.

The system table *Schema* stores the defined schemas. A schema entry stores the schema name, a unique identifier and its type, i. e., virtual or tenant. As a tenant schema always relates to exactly one tenant, it possesses the same identifier as its associated tenant. This approach avoids mapping steps from a tenant schema to its associated tenant and vice versa. A schema entry additionally stores the schema path up to the root level of the schema hierarchy. The system recursively traverses this path to gather and assemble the database objects visible in the schema (see Sec. 3.5.4). In accordance

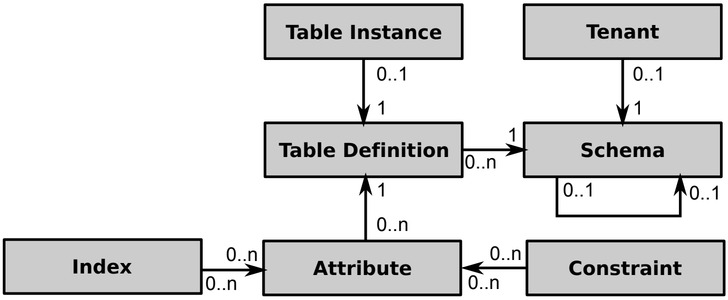


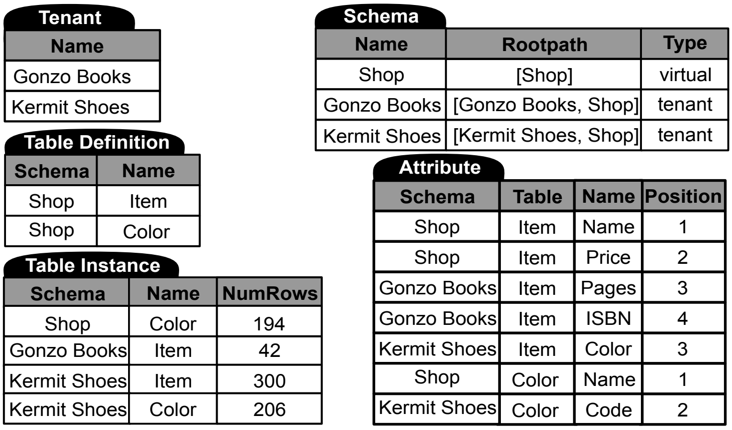
Figure 3.5: Simplified Catalog Model for TenantSchema (UML Notation).

with the tenants of our example, Fig. 3.6 shows two entries for the tenant schemas *Gonzo Books* and *Kermit Shoes*. The stored root path reflects that both schemas inherit from the virtual schema *Shop* which also has a related entry in *Schema*.

The system tables *TableDefinition* and *TableInstance* provide the basis to represent the extension of tables according to our schema inheritance concept. *TableDefinition* stores table definitions and *TableInstance* stores table instances. A common identifier ties a table definition and related table instances together. As each virtual schema or tenant schema may extend the lists of attributes, constraints and indexes of an inherited table definition, the system assembles the complete table definition by traversing the related schema inheritance path. In our example, the virtual schema *Shop* defines a table *Item* that has two attributes: *Name* and *Price*. The schemas *Gonzo Books* and *Kermit Shoes* respectively extend *Item* by additional attributes. Thus, *Attribute* contains corresponding entries: one entry for the attribute *Color* of *Kermit Shoes* and two entries for the attributes *Pages* and *ISBN* of *Gonzo Books*.

3.5.3 Main Memory Representation

Query processing accesses the catalog heavily since information stored in the catalog is required for authentication, parsing, and query optimization. Therefore, modern RDBMS implementations transfer the external format of



Tenant

Name
Gonzo Books
Kermit Shoes

Table Definition

Schema	Name
Shop	Item
Shop	Color

Table Instance

Schema	Name	NumRows
Shop	Color	194
Gonzo Books	Item	42
Kermit Shoes	Item	300
Kermit Shoes	Color	206

Schema

Name	Rootpath	Type
Shop	[Shop]	virtual
Gonzo Books	[Gonzo Books, Shop]	tenant
Kermit Shoes	[Kermit Shoes, Shop]	tenant

Attribute

Schema	Table	Name	Position
Shop	Item	Name	1
Shop	Item	Price	2
Gonzo Books	Item	Pages	3
Gonzo Books	Item	ISBN	4
Kermit Shoes	Item	Color	3
Shop	Color	Name	1
Kermit Shoes	Color	Code	2

Figure 3.6: Example of our data dictionary for the introduced e-commerce scenario.

the catalog into a quickly accessible network of main memory objects. This network has to be small so that the system can easily keep the catalog in main memory as much as possible. Therefore, the main memory layout requires preventing redundancy by sharing common parts between tenants.

Figure 3.7 depicts a simplified example of the proposed memory structures in order to hold the description for the table *Item* of our e-commerce scenario. The large boxes hold general information about a table. We refer to them as *table descriptors*. The table descriptor *Shop.Item* holds the information of the table definition that is invariant between tenants. Contrarily, the table descriptors *Gonzo Books.Item* and *Kermit Shoes.Item* hold information that is specific to the respective tenant schema. They reference the descriptor *Shop.Item* to access the common definition parts easily. Each table descriptor references an *attribute descriptor list* which holds information about related attributes. The lower part of Fig. 3.7 shows the related attribute descriptor lists. The attribute descriptor list of *Shop.Item* references two attribute descriptors that describe the structure of the two attributes *Name* and *Price*. The attribute descriptor list of *Gonzo Books.Item* additionally references

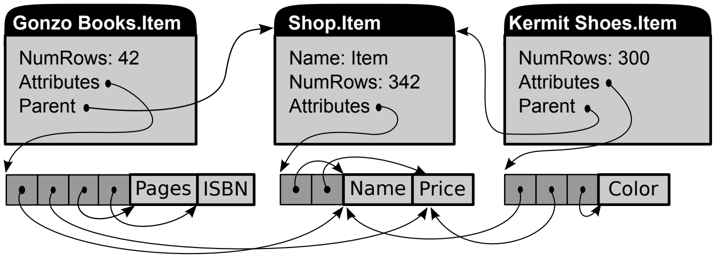


Figure 3.7: Main memory layout of table and attribute descriptors.

an attribute descriptor for the attribute *Pages* and *ISBN*. Analogously, the attribute descriptor list *Kermit Shoes.Item* references an attribute descriptor for the attribute *Color*.

3.5.4 Maintaining and Accessing the Catalog

To maintain the presented catalog, data definition operations have to consider the schema hierarchy and the tenant context within which they are executed. For example, the statement *ALTER TABLE ADD ATTRIBUTE* must ensure that the name of the new attribute is unique across the inheritance path.

Accessing a table entails constructing a table descriptor in main memory. If the accessed table is inherited from a virtual schema, the system assembles relevant parts by traversing the related schema inheritance path. Algorithm 3.1 shows the construction of a table descriptor in pseudo code. The algorithm recursively steps through the schema inheritance path until it finds a descriptor of the desired table. After finding a descriptor, it goes back through the schema inheritance path and creates an appropriate chain of descriptors, as explained in Sec. 3.5.3.

The shown algorithm may require more catalog lookups than comparable, standard algorithms to build a table descriptor because it has to traverse the schema inheritance path. In practice, this overhead is negligible. The descriptors which relate to a virtual schema are likely to exist in main memory since they are frequently accessed as multiple tenants share them. As a

Algorithm 3.1 Retrieval of a table descriptor. $tableId$ is the identifier of the table under concern, $rootPath$ is the inheritance path for the schema the table belongs to (see Fig. 3.6 for an example).

```
1:  function $getTableDescriptor(tableId, rootPath)$
2:      $tabDescr \leftarrow queryTabDescrCache(tableId, rootPath.first)$
3:      if $\nexists tabDescr$ then
4:          $tabDescr \leftarrow buildBaseTabDescr(tableId, rootPath.first)$
5:          if $\nexists tabDescr \wedge \exists rootPath.next$ then
6:              $parent \leftarrow getTableDescriptor(tableId, rootPath.next)$
7:              $tabDescr \leftarrow buildDerivedTabDescr(parent, rootPath.first)$
8:              $PutTableDescrInCache(tabDescr)$
9:              return $tabDescr$
10:         end if
11:     else
12:         return $tabDescr$
13:     end if
14:     return not found
15: end function
```

consequence, the shown algorithm returns directly after the first recursion in most cases.

3.5.5 Synchronizing the Catalog

To ensure that tables are not dropped or modified during the execution of a statement that references them, each statement acquires appropriate table-level locks. TenantSchema requires a locking approach that accounts for the relationship of tables according to the given schema inheritance. The modification of a table definition $d_{s_1}(t)$ has to prevent the concurrent modification of table definitions $d_{s_2}(t)$ with $s_1 \in P(s_2)$. The same goes the other way, if a table definition $d_{s_2}(t)$ is modified, the related table definitions $d_{s_1}(t)$ with $s_1 \in P(s_2)$ have to be protected against parallel modifications.

To accomplish the described behavior, the lock manager has to support a multi-granularity scheme for table-level locks [Gray, 1978]. The granularity of the table-level locks follows the inheritance relation of tables which the inheritance relation of the schema hierarchy imposes. The table-level lock for t in s_1 implicitly includes the table-level lock for t in s_2 if $s_1 \in P(s_2)$. For this purpose, if a statement wants to acquire a table-level lock on t in s_2, the lock manager records an absolute lock on t in s_2 and intentional locks on t in all schemas s_1 where $s_1 \in P(s_2)$, i. e., all related predecessors with respect to inheritance. The statement naturally obtains the requested lock only if it does not conflict with existing locks (absolute or intentional) on t or on predecessors. The described scheme requires additional locks to the actual lock on the table. However, our use case generally entails a flat schema hierarchy. Thus, locking a table requires recording only few additional locks.

If a catalog entry is modified, the system requires to update the related main memory structures. For this purpose, old entries are invalidated and discarded. New entries which reflect the modified version are created during the next access. On invalidating a table descriptor, the system takes care of invalidating dependent table descriptors in deeper levels of the schema hierarchy, i. e., the invalidation process traverses the descendents and invalidates related entries as well.

3.5.6 Storage Layout and Processing

The storage layout used to the store the table portions and the related processing do not depend on TenantSchema. Nevertheless, there exists a canonical implementation which we briefly outline.

Table portions may be considered as tables which have a certain relationship to each other. Based on this consideration, a natural approach is to store each table portion in private data structures as usually done for tables. The query planner then has to assemble the table portions appropriately to create the table instance for a certain tenant. For this purpose, the query planner has to account for the assembly rule of table instances given by Eq. 3.1 and the schema inheritance path (similar to Alg. 3.1). The query planner traverses the schema inheritance path from the actual tenant schema over virtual schemas to the root virtual schema. During this traversal, it gathers all available table portions of the table under concern. Finally, it combines the horizontal table portions using the union operator. Vertical table portions are included by joining them with the table instance of the predecessor schema. For efficient processing, techniques for efficient vertical composition, as used in Stonebraker et al. [2005], and pruning of table portions, e. g., by using bloom filters as in Chang et al. [2006], are usable. Hence, the processing for operations associated with a tenant schema is obviously simple and efficient.

Contrarily, operations which target shared data, i. e., data that relates to virtual schemas, may require high run-time efforts. This is because an insertion or modification of a tuple related to a virtual schema has to take into account all schemas that inherit from the virtual schema under concern, e. g., to check an unique constraints or to extend vertical table portions suitably. However, we assume that the data related to virtual schemas rarely changes by what the higher efforts are bearable in the general case.

Note that the described approach just describes one feasible solution. There exists a variety of possibilities to store the tuples of different table portions. The subsequent chapter discusses alternatives that focus on higher scalability with respect to the number of tenants.

3.6 Evaluation

To evaluate the effectiveness of TenantSchema with respect to scalability, we implemented a preliminary version of TenantSchema which allows defining a base schema and derived tenant schemas and focuses on evaluating the benefits of sharing among tenants. This preliminary version was implemented in PostgreSQL 8.4.

Subsequently, we present the results of an experiment to measure main memory consumption and lookup time of the data catalog. In doing so, we draw a comparison between Shared Process, Shared Table and TenantSchema.

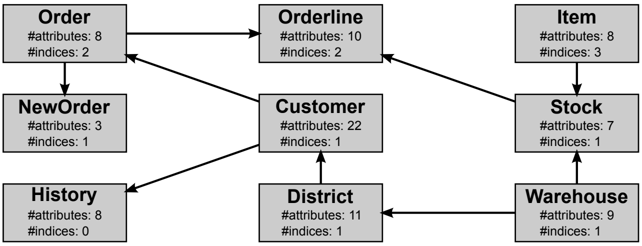


Figure 3.8: Core application schema of the testbed. The test extends this schema per tenant by a given number of custom attributes.

3.6.1 Testbed

The TPC-C schema [Transaction Processing Performance Council, 2012] depicted in Fig. 3.8 represents the application core schema for the test. The schema comprises 9 tables, 12 indexes and 86 attributes. The test distributes a given number of custom attributes evenly over the tables *Item*, *Customer*, *District*, and *Warehouse*. For example, 32 custom attributes per tenant entails 8 custom attributes per table. The custom attributes are of type *varchar*.

For Shared Process, the test creates for each tenant a dedicated schema instance by mapping it onto a SQL schema whose name includes the related tenant, e. g., *shop_tenant23*. Thereafter, the test defines custom attributes in each schema. For Shared Table, we use the Universal Table approach; all tenants share a single set of tables which is defined in a single schema. The test defines this schema with custom attributes and extends each table by an attribute to store the tenant. For TenantSchema, the test creates the base schema depicted in Fig. 3.8 and derived tenant schemas. Finally, the test defines custom attributes in each tenant schema.

After generating the schemas, the test creates a random sequence of explain queries that we call *explain sequence*. An explain query runs through all query processing steps except for the actual execution of the generated plan. Thus, an explain query is well suited to estimate data catalog characteristics. The explain sequence contains exactly one explain query for each table and each tenant. An explain query projects all attributes of the related table.

Depending on the approach, the explain queries slightly differ. Explain queries for the Shared Process approach qualify a tenant's tables by placing the name of the associated schema in front of table names. Explain queries for the Shared Table approach obtain a selection predicate according to the related tenant. For TenantSchema, the test places a suitable *SET TENANT* statement in front of the explain query in order to set the related tenant.

The test executes the generated explain sequence two times in a row. Thereafter, it shutdowns the DBMS and restarts it with another database. The test switches between systems and databases according to the approaches that are evaluated. In between, it drops file system caches of the Linux kernel. Hence, the first execution of the explain sequence uses a cold cache, whereas the second execution uses a warm cache. The test executes the explain sequences sequentially over a single session.

The test reports for each explain query the end-to-end execution time as difference from the time of issuing the explain query to the time of retrieving the results. After executing a sequence, our test additionally reports the resident main memory consumption of the corresponding PostgreSQL Backend using *smaps* in the proc-interface of Linux. Hence, each test run measures main memory consumption two times.

For our tests, we ran the databases on a Dell Optiplex 755 that was equipped with an Intel Core2 Quad Q9300 CPU running at 2.50 GHz and 4 GB of main memory. We stored the database and its write-ahead log on two striped 250 GB SATA 3.0 GB/s hard drives spinning at 7.200 RPM. The test machine ran a 64 bit 2.6.31 Linux kernel (Ubuntu release 9.10 Server). The client machine on which we ran our test tools was equipped with four Dual Core AMD Opteron 875 CPUs running at 2.2 GHz and 32 GB of main memory. The operating system was a 64 bit 2.6.9 Linux Kernel (CentOS release 4.8). This machine was connected to the database machine over a 1 GBit/s ethernet network. We used the default database configuration generated by PostgreSQL, except the size of the buffer pool, which we increased to 1024 MB. The data dictionary tables totally fit in buffer pool.

	Shared Process		Shared Table		TenantSchema	
Cust. Attr.	1000	10000	1000	10000	1000	10000
0	193	2.041	4	4	15	106
32	236	2.308	4	4	43	387
64	267	2.586	4	4	70	654
128	329	3.114	5	5	123	1197

Table 3.1: PostgreSQL Backend's main memory consumption in MB with different numbers of custom attributes per tenant and different tenant cardinalities (1000, 10000).

3.6.2 Experimental Results

Table 3.1 lists the main memory consumption of the previously presented test cases. Each test case ran three times. Table 3.1 reports the highest measured memory consumption of the three runs after executing the generated explain sequence the second time. The measured values after the first execution of the query sequence are nearly identical, though. This is because the required meta data was completely loaded during the first execution. Thus, the second execution just queries the data dictionary cache, but it does not build additional structures.

The measurements evidence that the main memory consumption of Shared Table does not depend on the number of tenants but only on the number of custom attributes. That is evident as the meta data for custom attributes is naturally shared among all tenants. The amount of additional meta data on increasing the number of custom attributes is fairly small. Contrarily, Shared Process and TenantSchema consume considerably more memory by a higher number of tenants and custom attributes. The memory consumption of both approaches increases almost identical for higher numbers of custom attributes. We expected this behavior because both keep custom attributes dedicated for each tenant.

TenantSchema consumes little main memory for 10000 tenants without customization. Yet, it consumes more main memory by higher degrees of customization per tenant. Consequently, the scalability of TenantSchema mainly depends on the customization requirements. TenantSchema consumes considerably less main memory than Shared Process. For 10000 tenants, the difference amounts to approximately 1900 MB, independent of the number of custom attributes. Shared Process requires considerably more main memory since each additional tenant requires a new instance of the application core schema. The shown result approves the effectiveness of sharing the application core schema. Previous research into the main memory consumption of Shared Process in different RDBMS implementations approve its high main memory consumption [Jacobs and Aulbach, 2007].

Interestingly, our measurements show that an additional attribute requires approximately 800 Bytes of main memory, although an entry in the system table *Attribute* has an average size of 107 Bytes. Thus, an attribute consumes at least seven times more main memory than the size of its external representation. As typical use cases had only smaller amounts of meta data, PostgreSQL adopts main memory structures optimized for fast access, but not for size. In our use case, however, the volume of meta data becomes considerably large. The implementation of the main memory structures has to take large meta data volumes into account. Note that even if the access to a single structure may suffer by optimizing it for size, the overall performance may increase due to the lower main memory consumption of the data catalog.

Fig. 3.9 reports the accumulated lookup times for the second execution of the explain sequence, i. e., against a warm data dictionary cache. The lookup times of Shared Process and of our approach increase more than linearly with the number of custom attributes. As our approach maintains a smaller data dictionary, it does not degrade as much as Shared Process. Hence, for 10000 tenants and 64 custom attributes per tenant, our approach takes about 0,7 milliseconds per explain query as opposed to Shared Process that takes about 7 milliseconds per explain query.

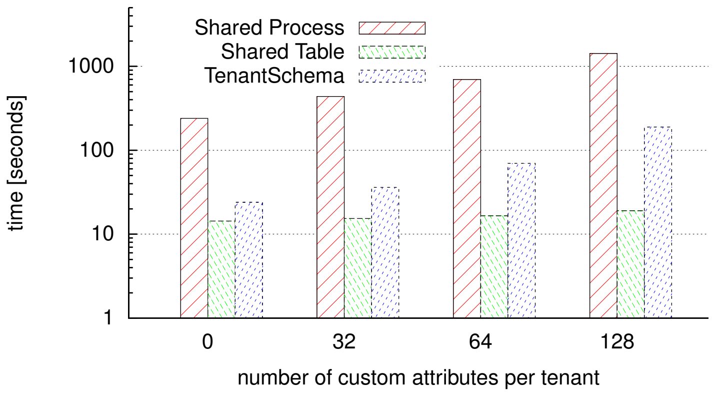


Figure 3.9: Accumulated data dictionary lookup times of querying each table for each tenant (90000 queries) on a warm data dictionary cache.

3.6.3 Discussion

Shared Table outperforms TenantSchema in the presented measurements. However, this naive comparison compares apples and oranges. Shared Table requires multi-tenant schema management on top of the RDBMS. The additional management layer on top of the RDBMS causes extra runtime overhead, which needs to be added to the performance figures of Shared Table for a fair comparison. Moreover, an implementation of multi-tenant schema management on top of the RDBMS does not avoid redundancy per se and is difficult to implement, especially tenant-specific schema extensions are difficult to implement. Yet, even without the need of tenant-specific schema extensions, TenantSchema provides a benefit as it simplifies application development; it avoids an additional data access component on top and it enables system-level tenant isolation. This benefit is paid for by increased main memory consumption compared to Shared Table, but still on a similar scale.

In comparison with Shared Process, our measurements show that sharing the application core schema may considerably decrease main memory consumption and lookup times of the data catalog. Of course, lookup times of the data catalog only form a small fraction of total query time. For instance, taking the 7 milliseconds for 10000 tenants and 64 custom attributes of Shared Process and assuming an average runtime of 140 milliseconds for a simple query, the lookup times only form 5 %. The actual performance benefit of TenantSchema is the moderate main memory consumption of the data catalog while providing multi-tenant schema management. Low main memory consumption of the data catalog is important as the main memory consumption of the data catalog limits available space for the buffer pool, which ultimately impacts overall query performance. For instance, assume that a machine with 32 GB of main memory runs our test case for 10000 tenants and 128 custom attributes. In this case, the Shared Process approach would leave 29 GB for query processing, whereas our approach would leave 31 GB, which is about 7 % more. In practice, the benefit is likely to be higher as applications tend to have a larger schema than the schema used in our experiments (larger means more tables, attributes and indexes). Coelho et al. [2011] has evaluated the relational schema of 512 applications based on MySQL and PostgreSQL, they state 34 tables and about 9 attributes per table in average. Simply scaling up our results to those numbers yields expected savings of about 7 GB by TenantSchema compared to Shared Schema.

Note that the discussed issues also apply to pre-compiled queries. Pre-compiled queries avoid the overhead of parsing and planning during runtime and, thus, decrease accesses to the data dictionary. However, storing all the required meta data for each tenant in the plans or storing separate plans for each tenant of the pre-compiled queries yields similar scalability issues as in Shared Process. Thus, pre-compiled queries may also benefit from TenantSchema.

3.7 Summary and Outlook

By introducing tenants as first-class database objects, the RDBMS becomes aware of tenants. Combined with the concept of a tenant context, which glues together the tenant's view of the database, the RDBMS provides then the necessary infrastructure to support multi-tenant data management functionality.

TenantSchema makes use of this infrastructure in order to facilitate multi-tenant schema management. For this purpose, TenantSchema offers different schema types for different challenges. Virtual schemas intend to describe application core schemas; a virtual schema may inherit from another virtual schema to specialize the application core schema for a specific domain. Tenant schemas that inherit from virtual schemas enable schema isolation and per-tenant schema extension. These concepts facilitate the central maintenance of the application core schema and reduce meta data and application data redundancy between tenants.

To conclude, TenantSchema eases the development of multi-tenant SaaS applications while keeping in view scalability.

There exist several open questions and challenges to improve TenantSchema for productive use. The modifications of shared meta data and application data comes with multiple challenges, particularly if TenantSchema is adopted in a distributed environment. The RDBMS, probably supported by a suitable tool, should allow the provider to roll out such modifications incrementally, one tenant by one tenant, and should support the provider in resolving conflicts. We consider the design of a suitable solution for this challenge as an important direction for future work. In this context, we envision to extend and adapt approaches such as the PRISM workbench for graceful schema evolution [Curino et al., 2008] to the needs of a multi-tenant SaaS application.

CHAPTER 4

Storage Concepts and their Performance

The previous chapter presents TenantSchema which facilitates schema management tailored to the needs of a multi-tenant SaaS application. TenantSchema uses the concept of table portions to describe the data parts which multiple tenants share and data parts which a tenant uses private. The proposed canonical solution to map the logical table portions to physical storage uses dedicated physical data structures for each table portion (see Sec. 3.5.6).

A consequence of this solution is that a large number of tenants yields a large number of physical data structures. Moreover, the number of data structures increases with more tenants, which eventually yields scalability issues. This chapter embarks upon describing these scalability issues and discusses two alternative approaches which consolidate table portions into one data structure to improve scalability. Thereafter, this chapter analyzes the most promising approach in comparison with private data structures, assuming

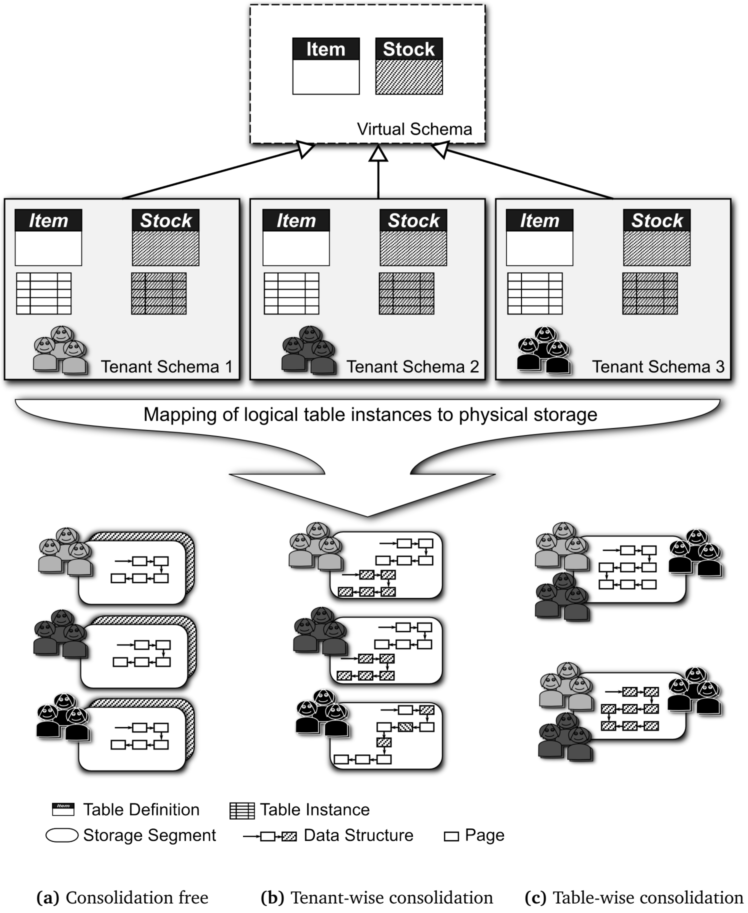
B-trees and heaps, which are wide-spread data structures in modern RDBMS implementations. Finally, this chapter presents an alternative approach to map table portions to physical storage structures in TenantSchema. This alternative approach relies on the results of the previous analysis.

4.1 General Approaches to Store a Tenant's Tuples

Efficient access and handling of table data mandates appropriate *data structures* with related *access paths*. To implement such data structures, total database storage requires to be organized in reasonable chunks with different granularities. Therefore, modern RDBMS implementations manage database storage as a set of *segments* which are divided into pages of fixed size. Segments are organized in storage containers that are usually called *table spaces* and abstract from underlying physical characteristics. Multiple segments can be assigned to one table space. A segment allocates and deallocates *extents* within the table space to which it is assigned; an extent is a set of pages that are physically contiguous. An extent map records associated extents of a segment. Pages within a segment are logically co-located and numbered in ascending order; the number addresses the page within the segment.

A segment usually stores only one data structure, e. g., a B-tree or heap, which in turn relates to a single table instance. This approach implements efficient table-oriented access paths. For example, a scan of all tuples of a table instance only requires scanning the related segments and thus only reads tuples of the table under concern. Furthermore, this approach enables maintenance and customization at data structure or table space level, e. g., backup and recovery of a single table space or physical clustering of a table instance.

The introduction of tenants increases the number of table instances considerably compared to traditional OLTP–style applications. Therefore, scalability with respect to the number of table instances represents a requirement which a multi-tenant RDBMS has to meet. For this purpose, the consolidation of multiple table instances or rather table portions into a single segment or even a single data structure is feasible. This is because the consolidation



(a) Consolidation free **(b)** Tenant-wise consolidation **(c)** Table-wise consolidation

Figure 4.1: Three approaches to map the private table portions of tenants to physical storage structures.

reduces fixed overheads induced by segments or data structures. However, such consolidation approaches also bear some challenges with respect to access path efficiency, customization, isolation and functionality centric to a table instance.

In the following three sections, we outline three approaches to store private table portions of tenants, i. e., table portions that relate to tenant schemas. Fig. 4.1 illustrates these different approaches. First, Section 4.1.1 discusses private storage structures for each table portion and for each tenant, illustrated by Fig. 4.1a. We refer to this approach as *consolidation free*. Second, Section 4.1.2 discusses variants which consolidate table portions into a single segment or even a single data structure, but limited to a single tenant, illustrated by 4.1b and referred to as *tenant-wise consolidation*. Finally, Section 4.1.3 considers variants which consolidate table portions from different tenants into one data structure but limited to a single data structure type of a single table, illustrated by 4.1c and referred to as *table-wise consolidation*.

For the discussion of the three approaches, we limit ourselves to two common data structures: *heaps* and *B-tree*. A heap stores tuples in arbitrary order and enables a sequential scan access path. A B-tree represents a tree-based data structure that provides an index based access path. Heaps and B-trees are both used as primary table storage. B-trees are additionally used as secondary indexes that store references to tuples in a heap or a primary B-tree. A variant of the B-tree, called B^+-tree, is arguably the most common indexing technology in modern RDBMS implementations [Weikum and Vossen, 2001]. The subsequent discussion also applies to B^+-trees although we always refer to B-trees.

4.1.1 Consolidation Free Storage of a Tenant's Tuples

This approach maps a tenant's table portions to private data structures, i. e., private heaps and private B-trees, which are in turn stored in private segments. Thus, this approach segregates a tenant's data from other tenants quite well.

This approach has some drawbacks with respect to scalability, though. Large numbers of tenants lead to hundreds of thousands of segments and data structures. A high number of data structures tend to result in many lowly utilized pages due to internal page fragmentation, which ultimately degrades buffer pool utilization. The same issue arises with respect to extents if they are assigned dedicated to one specific segment and thus to a specific tenant and table. Smaller page sizes and smaller extent sizes may partially circumvent the described effect. Yet, very small pages and small extents also degrade performance: more I/O requests, higher relative overhead of page headers, large extent maps and more expensive buffer pool maintenance. In addition to that, underlying structures such as a file system or a raid system dictate the smallest useful size of pages and extents. For efficiency, the page and extent size has to align to sizes of organization forms in these structures, e. g., the block size of the file system or the stripe size of the raid system.

Moreover, the auxiliary data structures required to maintain and manage data structures often also limit scalability. This is because these data structures typically entail a fixed overhead, e. g., a meta data page, or a minimum amount of allocated space. A large number of segments therefore yields severe additional overheads as the fixed overhead or minimum allocated space per segment is multiplied by the number of segments. For example, if an auxiliary data structure occupies at least one page having a size of 8 KB, 10000 tenants each using 10 segments may yield a total consumption of about 780 Mb only by the auxiliary data structures. The major portion of the auxiliary data structures stays in main memory since they are frequently accessed. Hence, a large number of smaller segments with dedicated auxiliary data structures may considerably reduce available main memory space for query processing.

4.1.2 Tenant-wise Consolidation

A natural approach to improve scalability is to consolidate segments and data structures. The consolidation of segments decreases their number, but it does not reduce the number of data structures. The main advantage is an improved extent utilization and the opportunity to consolidate certain auxiliary structures. However, if a segment stores pages from multiple tables, there is no distinction which pages belong to a table anymore. Thus, table scans are more expensive because they either require scanning the whole segment, i. e., data of all tables, or have to be replaced by equivalent index scans (only if there exist an appropriate index, of course). Moreover, maintenance operations, e. g., reclustering, have to cope with the whole segment. An additional data structure that associates pages with a table may circumvent this issue, e. g., by establishing a chain of pages through next pointers, as usually implemented for B-trees within one level.

Existing RDBMS implementations already implement partial segment consolidation strategies to cope with many small tables. For example, Microsoft SQL Server uses mixed extents which are shared by up to eight segments. A new segment allocates pages from mixed extents until it grows to the point that it consumes eight pages. Thereafter, it allocates private extents [Microsoft, 2013].

Another strategy is to consolidate all table portions of a tenant into a single heap. This strategy intermingles tuples of tables having different access patterns within one page. This fact may degrade paging behavior and thus the buffer pool utilization. Moreover, it suffers from the same drawbacks with respect to table scans as described for the consolidation of segments.

Regarding B-trees, the diversity of B-trees with respect to the index keys and their data types prevents a reasonable consolidation of several B-trees into a single B-tree.

This approach offers good isolation as each tenant uses its private pages. However, its benefit is small as the number of segments, data structures and related auxiliary structures still increases with the number of tenants.

4.1.3 Table-wise Consolidation

As opposed to the previously described approaches, table-wise consolidation consolidates tuples of different tenants into a single data structure. In doing so, each data structure only stores tuples of horizontal table portions of tenant schemas that relate to the same table of the application core schema, i. e., whose related table definitions are descendants of the same table.

Regarding heaps, table-wise consolidation gives up the opportunity to

	Isolation	Scalability	Customization	Table Scan
Consolidation Free	+	– –	++	++
Tenant-wise	+	–	+	–
Table-wise	– –	++	–	– –

Legend: ++ very good, + good, – bad, – – very bad

Table 4.1: Differences of three approaches to store a tenant's tuples.

scan a single tenant's tuples efficiently. In fact, scanning the heap would access orders of magnitude more data than required, i. e., in average about number of tenants more data. In this respect, table-wise consolidation typically behaves even worse than tenant-wise consolidation assumed that the number of tenants is higher than the number of tables. However, table-wise consolidation puts data with similar access patterns into one page. Although tenants may customize an inherited table, the major part of the application logic that accesses the table is likely to remain the same, at least with respect to the core application schema. Hence, the access patterns on a table of the core application schema are likely to stay similar across tenants.

A similar strategy is feasible for B-trees. A B-tree which is defined on a table in a virtual schema can simply index all related table portions in derived tenant schemas.

Table-wise consolidation is extremely scalable because the number of data structures does not depend on the number of tenants, at least with respect to the tables and indexes defined by the application core schema. For tables or indexes that a tenant defines within its tenant schema, there exist still the same scalability issues as described.

4.1.4 Conclusions

Table 4.1 roughly summarizes the main differences of the three presented approaches. Consolidation free and table-wise consolidation are the two extreme sides with respect to scalability. Tenant-wise consolidation is closer to consolidation free (assumed that the number of tenants is larger than the number of tables). This is because the pure consolidation of data structures of a single tenant into a single segment does not enhance scalability considerably. Thus, we argue that the potential benefit of tenant-wise consolidation is usually low for the targeted use cases. Tenant-wise consolidation introduces significant compromises regarding scan performance, paging behavior and maintenance operations. We believe that table-wise consolidation is a natural consolidation approach having good potential to improve scalability. Therefore, we skip considering tenant-wise consolidation in the remainder of this thesis and concentrate on consolidation free and table-wise consolidation.

Note that table-wise consolidation (as described) does not consolidate tables or indexes which a tenant defines in its tenant schema. If tenants define a lot of new tables and new indexes, this aspect requires to be considered as well. This case may require additional measures, as described for Microsoft SQLServer. In this thesis, we skip a deeper consideration of this aspect.

Although not explicitly stated in the previous discussion, consolidation free and table-wise consolidation can naturally make sense side by side. Many small tenants which rarely scan tables and have low requirements with respect to customization or tenant-oriented functionality call for table-wise consolidation. If tenants are large and have special requirements, consolidation free definitely has its strengths. Of course, in addition to choose a certain approach per tenant, it is even feasible to choose a certain approach per table, or even per data structure type of a table. For example, tenants may use private heaps but share a common B-tree.

The following two sections provide a more detailed performance and scalability evaluation of table-wise consolidation for B-trees and heaps.

4.2 Consolidation into a Shared B-tree

This section provides an analytical performance comparison of a table-wise consolidated, shared B-tree (see Sec.4.1.3) with a private B-tree for each tenant and each table (consolidation free), as published in Schiller et al. [2012]. For this purpose, we embark upon discussing the properties of two natural approaches to consolidate tenants into a shared B-tree and recommend one approach, namely *Partitioned B-tree*, as generally applicable approach. The analytical comparison bases upon an approximation model for the behavior of a B-tree which assumes uniformly random insertion of tuples. This approximation model allows estimating the expected difference of nodes for a Partitioned B-tree and equivalent private B-trees, assuming uniformly random insertions of tuples over all tenants. Note that the term *equivalent private B-trees* refers to the private B-trees that are generated by the same sequence of key operations as the shared B-tree. Finally, we analyze the impact of the obtained results with regard to the buffer pool utilization and discuss the analytical results from a practical perspective.

4.2.1 Two Consolidation Approaches

To consolidate tuples of multiple tenants into a single B-tree, a tenant's tuples require a unique label. For this purpose, an additional attribute that stores the identifier of the tenant to which a tuple belongs is suitable. This attribute, subsequently referred to as *tenant attribute*, represents a system attribute which is logically separated from the attributes defined by the application (see Sec. 4.4). This attribute has to be included in the index key definition of the B-tree to allow searching for tuples of one specific tenant. Subsequently we refer to this key as *tenant key*.

Two natural ways exist to place the tenant key: either ahead or behind the index keys defined by the application. If the tenant key precedes the application index keys, the tenant key partitions the B-tree according to tenants. This effectively yields a *Partitioned B-tree* [Graefe, 2003]. The Partitioned B-tree stores tuples of a tenant physically together and possesses subtrees which mainly contain tuples of one tenant. Only pages in the periphery of such a subtree contain tuples of other tenants, as illustrated in Fig. 4.2. By contrast, adding the tenant key behind the application index keys distributes a tenant's tuples arbitrarily across the whole B-tree. For the remainder, we refer to this approach as *Tenant Behind B-tree*.

In a Tenant Behind B-tree, scanning multiple tuples of a certain tenant requires jumping arbitrarily through the B-tree which entails access of multiple pages that only store few tuples of the tenant. Hence, regarding range scans, the Partitioned B-tree is obviously advantageous due to high locality of a tenant's index tuples.

However, the Partitioned B-tree may even favor simple index lookups (range scans of width one). The reason is that tenants often differ in their activity with respect to time, i. e., , regarding a certain point in time, some tenants are active, whereas some tenants are inactive. Using a Tenant Behind B-tree, the expected number of buffer pool pages that contain index tuples of inactive tenants is higher than using a Partitioned B-tree, assuming uniform data access of tenants. The buffer pool therefore contains more data that is unlikely to be accessed, which naturally decreases buffer pool hit ratio and ultimately degrades overall performance.

The access patterns of tenants may highly skew, i. e., a tenant mainly accesses very small amounts of its total key range. If all tenants have the same access skew and the same key range, the Tenant Behind B-tree may perform better for index lookups. In this case, locality for the same index key of different tenants is desired. However, this advantage erodes fast if several tenants are inactive. Moreover, the case that all tenants have the same access skew and the same key range occurs quite rarely and is difficult to forecast, as tenants constitute individual, closed organizations.

To sum up, the Partitioned B-tree provides high locality of a tenant's index tuples. By contrast, the Tenant Behind B-tree provides high locality for the same index keys of different tenants, which we regard rarely beneficial. Thus, we conclude that the Partitioned B-tree has its points as generally applicable approach and focus on this approach for our further discussion.

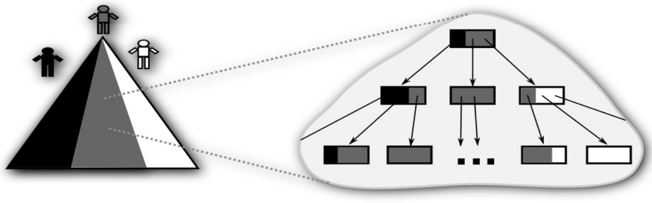


Figure 4.2: Illustration of a Partitioned B-tree.

4.2.2 Foundations

Since the proposal of B-trees in Bayer and McCreight [1972], many different variants have established, most notably the B$^+$-tree. With the different variants, different definitions and models of their behavior have emerged. To obtain a consistent view, this section embarks upon defining concepts which are required for the following discussions. Furthermore, it reviews models for the the behavior of B-trees and explains how we use them for the subsequent performance considerations.

4.2.2.1 B-tree and its Variants

A *B-tree* [Bayer and McCreight, 1972] is a balanced tree in which all leaves have the same distance to the root. In a B-tree of *order* k, each node has at most $2k$ entries and $2k+1$ children. All nodes except for the root node have at least k entries and $k+1$ children. The root node has at least 1 entry and 2 children. We number the levels of a B-tree in ascending order starting with 1 from the bottom level to the root level (top). Hence, the root level is numbered as h, if h is the height of the B-tree.

In contrast to a *B-tree*, a *B$^+$-tree* stores all key-value pairs in the leaves which are called *buckets*. The nodes in levels above are called *inner nodes*. They store only key-pointer pairs which route a search to the suitable bucket. Thus, the structure of the inner nodes is identical to a B-tree whose leaves are the buckets of the B$^+$-tree. B$^+$-trees are wide-spread since they increase the order, i. e., k, and come with very good scan behavior, given that the buckets provide links to their neighbors. However, a consideration of B$^+$-trees is slightly more complicated since the maximum number of entries in the buckets might differ from the maximum number of entries in inner nodes. Moreover, the split strategy of buckets and inner nodes might differ. This differences in the split strategy require considering some special cases. For this reason, we limit ourselves to B-trees in order to simplify our discussion. However, the results hold for B$^+$-trees as well due to the same structure in higher levels which is our main focus in the subsequent discussion.

An *N-key random B-tree* is built by N successive random insertions. A random insertion inserts a new key with equal probability in all possible intervals, comprising the intervals between keys and the two open intervals before the left-most key and following the right-most key.

A famous result from Yao [1978] states that the nodes at the bottom level in an N-Key random B-tree reach a steady-state with an asymptotic average storage utilization that approximates to $\ln(2) \approx 69.3$ %. Yao's results additionally state that the storage utilization converges fast to the given asymptotic limit for B-trees of high order, i. e., a large number of children per node. Another important result in this context is that the split probability in the bottom level is independent of N for large N. The split probability in the bottom level is $1/(2k \ln(2))$ [Wright, 1985]. Note that Eisenbarth et al. [1982] formalized the method used by Yao, called *Fringe Analysis*. Baeza-Yates [1989] applied Fringe Analysis to B^+-trees and obtained essentially the same results.

4.2.2.2 Approximation Model

Although it has not been mathematically proven yet, it is plausible to assume that levels above the bottom level in B-trees behave similar to the bottom level itself [Wright, 1985]. This follows from the assumption that insertions into the level directly above the bottom level are equally likely to hit any of the intervals in that level. This assumption arises out of the equal probability of a split for each inserted entry in the bottom level. The same argumentation applies recursively to levels further above. However, this argumentation is critical for the root level and the level underneath of the root level. These levels are too transient because they contain too few keys in order that the average storage utilization converges near to the asymptotic value. Thus, it is only safe to assume $u_l = \ln(2)$ for $l \leq h-2$, where u_l is the average storage utilization in level l of B-tree with height h. We refer to this assumption as *steady-state assumption*.

From this assumption additionally follows the probability for a split in level l, $l \leq h-2$ in a B-tree of height h. The split probability in level l is given by the probability that a new entry is inserted in level l and a split occurs. The probability that a new entry is inserted in level l is the probability that a split occurs in level $l-1$. This statement holds recursively until $l = 1$, i. e., l

is the bottom level. At the bottom level the insertion probability is naturally 1. Thus, the split probability of a split in level l is given by $1/(2k\,ln(2))^l$.

Yao's approach does not allow to estimate the average storage utilization of the root level, as it does not allow determining the probability for a certain number of entries in the root node. This is because Yao's model uses expected values for the number of nodes with a certain number of entries in the bottom level. These expected values take into account all variations of the bottom level for a B-tree that contains a given number of total entries. But they do not state anything about the structure above the bottom level, which prevents to determine the split probability of nodes in this level. However, this information is required to keep track of the node utilization above the bottom level, levels further above, and ultimately the root level. A fully exhaustive simulation of all possible ways to build an N-key random B-tree theoretically represents an alternative. Yet, as the number of states in such a simulation grows exponentially with the B-tree order, such a simulation blows computable complexity for higher B-tree orders.

Driscoll et al. [1987] approximates the number of children of the root node in a B-tree. For B-trees of low order, they present several measurements which match well with their approximation. We tried their approach for higher B-tree orders ($k > 50$), which are common nowadays. Unfortunately, it seems that their approximations yield a larger error for larger B-tree orders.

Against this backdrop, we use an approximation from Langenhop and Wright [Langenhop and Wright, 1988, 1989] to estimate the expected storage utilization of the root level and the level underneath of the root level in an N-Key random B-tree. They adopt the *restricted occupancy problem* to estimate the probability that a new entry in a B-tree level causes a split. The number of combinations for the restricted occupancy problem $S_k(b, s-bk)$, i. e., how many possibilities exist to place $s-bk$ balls in b bins with limited capacity k, equals to the possible configurations for a level with b nodes and s entries in a B-tree of order k. This is because per definition a node in a B-tree of order k has at least k entries and at most $2k$ entries. Therefore, if a level has b nodes and s entries, bk entries have to be uniformly distributed across the b nodes and $s-bk$ entries are placed freely across the b nodes of which each each has a remaining capacity of k.

Based on this model, Langenhop and Wright [1988, 1989] approximate that the probability of a certain node having exactly $2k$ entries is $S_k(b-1, s-bk-k)/S_k(b, s-bk)$, by assuming that all configurations for b nodes and s entries are equally likely. This is because $S_k(b-1, s-bk-k)$ is the number of configurations for b nodes such that one certain node has $2k$ entries. If a certain node has $2k$ entries, only $b-1$ nodes contain free space and thus only $s-(b-1)k-2k = s-bk-k$ entries are placed freely. By multiplying the probability that a certain node has $2k$ entries with the probability that an insertion hits this node, which is $(2k+1)/(s+b)$ (as there are $s+b$ intervals that the new entry may hit), we obtain the split probability for this node. The split probability does not depend on the concrete node. Thus, each node is equally likely to split (b times the same split probability). Hence, the probability of a split due to insertion of a new entry in a B-tree level with b nodes and s entries is given by

$$P_{split_k}(b,s) = b\,\frac{(2k+1)}{s+b}\,\frac{S_k(b-1, s-bk-k)}{S_k(b, s-bk)}.$$

For simplicity, we subsequently assume a constant k and thus skip the dependency on k in the formulas.

The steady state assumption for levels l, $l \leq h-2$, and the approximation of Langenhop and Wright in remaining levels above allow defining an approximation model. For this purpose, a triple (h, b, s) addresses all B-trees having height h, $h \geq 2$, and b nodes and s entries in the level $h-1$. Using the steady-state assumption and the split probability, it is easy to derive transition probabilities from one triple to another when a new entry is inserted only by regarding the origin triple. The probability that triple (h, b, s) transitions to triple (h', b', s'), $h \geq 2$, due to insertion of an entry is

$$P((h,b,s) \rightarrow (h',b',s')) =$$

$$\frac{1}{(2k\,\ln(2))^{h-2}} \begin{cases} 1 - P_{split}(b,s) & \text{for } (h,b,s) \rightarrow (h,b,s+1) \\ P_{split}(b,s) & \text{for } ((h,b,s) \rightarrow (h,b+1,s) \wedge b \leq 2k \vee \\ & (h,2k+1,s) \rightarrow (h+1,2,2k)). \end{cases}$$

The first case of the equation considers an insertion without split, whereas the second case considers an insertion that leads to a split of the node in which the entry is inserted. The second case also takes into account an insertion that leads to a split of the root node. In this case, the height increases and the new level $h-1$ therefore has exactly two nodes and $2k$ entries.

If we limit the maximum height of B-trees, the number of triples is finite, which allows to determine a probability matrix for the transition from one triple to another triple. Hence, our model directly maps to a markov model. Thus, we are able to calculate $P_N((h,b,s))$ which denotes the probability of B-trees having a total of N entries, height h, b nodes and s entries in the level $h-1$, where $h \geq 2$ and thus $N \geq (2k+1)$. The probability of the triples after the Nth insertion (P_N) is given by multiplying the triple probabilities of the initial state with the transition probability matrix $N-(2k+1)$ times. The initial state is $P_{2k+1}((2,2,2k)) = 1$ which corresponds to B-trees directly after the first split of the root node, i. e., the smallest B-tree with $h = 2$. The resulting probabilities of triples allow calculating the expected average storage utilization of the root level and level underneath of the root level:

$$u_h(N) = \sum_{(h,b,s)} P_N((h,b,s)) \frac{b-1}{2k} \tag{4.1}$$

$$u_{h-1}(N) = \sum_{(h,b,s)} P_N((h,b,s)) \frac{s}{b2k}. \tag{4.2}$$

Note that the average node utilization of level $h-1$ for a triple (h,b,s) is given by $s/(b2k)$. For the root level h, the number of entries is one less than the nodes in the level $h-1$, i. e., $b-1$. Thus, the utilization of the root node is $(b-1)/(2k)$. Note that calculating the average storage utilization of the root node for $h = 1$ is trivial but requires to be considered as special case.

4.2.3 Analysis

Regarding the efficiency of consolidating all tenants into a single Partitioned B-tree, two factors gear into each other: *sharing* of pages and *caching* of pages. Sharing of pages may increase average page utilization and, thus, reduce the total number of pages. For example, if the private B-trees come with an average root page utilization of 30 %, whereas the Partitioned B-tree ensures 60 %, the Partitioned B-tree requires half the pages to store the same number of entries. The tighter packing of information yields the opportunity to reduce disk accesses. This reduction of disk accesses is accomplished if shared pages are cached in a main memory buffer pool and locality between tenants occurs. Locality between tenants occurs if different tenants require the same page within a short time interval so that the page stays in the buffer pool. To sum up, the reduction of pages by sharing them and the access probability of shared pages constitute central factors when estimating the efficiency of consolidation.

To estimate the effectiveness of sharing, we primarily approximate the total expected difference of the number of nodes between the Partitioned B-tree and equivalent private B-trees. For this purpose, we approximate the excepted difference of the number of nodes per level (starting from the leaves). Thereafter, we discuss the relevance of these results regarding the expected caching behavior.

Our analysis uses for all private B-trees the same integral height. An exact analysis would require considering partitions according to possible heights of private B-trees. Thereafter, the impact of consolidating a partition into the Partitioned B-tree has to be regarded. However, the number of such partitions is pretty low (at maximum the height of the Partitioned B-tree) and the consolidation of two B-trees having different heights yields at most one additional entry per level. Therefore, the impact of this restriction is negligible, but it simplifies the analysis.

Furthermore, we assume that B-trees are built by a key insertion sequence of size N that is uniformly random across tenants and their respective key range. The table in Fig. 4.2 explains and summarizes the notations that we use frequently during the following analysis.

4.2.3.1 Sharing of Pages

In this section, we compare the number of nodes of the Partitioned B-tree and equivalent private B-trees in a level-wise manner starting from the bottom

p	Partitioned B-tree.
B	Set of equivalent private B-trees.
$u_l(N)$	Exp. node utilization in level l of a B-tree with N entries.
$u_{p,l}(N)$	Exp. node utilization in level l of Partitioned B-tree with N entries.
$u_{B,l}(N)$	Exp. node utilization in level l of equivalent private B-trees with N entries.
$h_p(N)$	Height of Partitioned B-tree (we skip N if dependency is clear).
$h_B(N)$	Height of equivalent private B-trees (we skip N if dependency is clear).
$V_l(N)$	Exp. number of nodes in level l of a B-tree with N entries.
$V_{B,l}(N)$	Exp. sum of nodes in level l of equivalent private B-trees with N entries.
$V_{p,l}(N)$	Exp. number of nodes in level l of Partitioned B-tree with N entries.
$\delta_l(N)$	Exp. difference of nodes between the Partitioned B-tree and equivalent private B-trees in level l and with N entries (i. e. $V_{p,l}(N) - V_{B,l}(N)$).

Table 4.2: Notations used for the comparison of a single Partitioned B-tree with equivalent private B-trees.

level. We deduce for each level l the expected difference $\delta_l(N)$ between the number of nodes in the Partioned B-tree $V_{p,l}(N)$ and the number of nodes in equivalent private B-trees $V_{B,l}(N)$.

The Partitioned B-tree requires the tenant key to distinguish tuples of different tenants (see Sec. 4.2.1). As typical B-tree implementations use index pages of fixed size, the additional tenant key reduces the Partitioned B-tree order by

$$f_r = \textit{tenant key size}/(\textit{tenant key size} + \textit{index tuple size}). \quad (4.3)$$

The capacity of a node that additionally has to store the tenant identifiers of index tuples thus reduces from $2k$ to $2k(1 - f_r)$. This reduction gets particularly large if index tuples are small, e. g., an additional tenant key of 2 bytes degrades the node capacity by 12.5 % for index tuples having a size of 14 bytes. Therefore, compression of the tenant key is highly recommended. For this purpose, prefix compression [Bayer and Unterauer, 1977] constitutes a promising approach since the tenant key builds a prefix of the whole index key.

With prefix compression, only nodes that contain tuples of different tenants come up with reduced capacity. In a Partitioned B-tree, there exist at most

$|B|-1$ transitions from one tenant to another tenant in each level. Thus, each level has at most $|B|-1$ nodes that contain tuples of different tenants and therefore have reduced capacity. Note that an implementation that strips common prefixes of entries and stores them only once in a dictionary within the node [Bhattacharjee et al., 2009] reduces the overhead even more. With such an implementation, the space overhead of the tenant key reduces in lower levels, in which rarely more than two tenants share a single node, to some bytes (about 8-12 bytes). Therefore, by applying the steady-state assumption, we approximate $V_{B,l}(N) \approx V_{p,l}(N)$ for $l \leq h_B - 2$ in case of fixed page size and prefix compression (see Fig. 4.3).

From $V_{B,h_B-2}(N) \approx V_{p,h_B-2}(N)$ directly follows that the level $h_B - 1$ has nearly the same expected number of entries in the Partitioned B-trees and the private B-trees. As we regard the case of many tenants (> 1000), we take $|B| > 2k(1-f_r)$ as a given, i. e., the number of tenants is larger than the maximum number of entries in a B-tree node. It follows that $h_B - 1 \leq h_p - 2$, by what the steady-state assumption holds for the Partitioned B-tree, but not for the private B-trees. The expected number of nodes in level l is given as the expected number of nodes in $l-1$ divided by the expected average number of children of a node in level l ($2k\ \ln(2)+1$ for $l \leq h-2$). Thus, $V_{p,h_B-1}(N) = \frac{V_{p,h_B-2}(N)}{2k\ \ln(2)+1}$ holds in the Partitioned B-tree. With $V_{p,h_B-2}(N) \approx V_{B,h_B-2}(N)$, we obtain

$$\delta_{h_B-1}(N) \approx \frac{V_{B,h_B-2}(N)}{2k\ \ln(2)+1} - V_{B,h_B-1}(N) \qquad (4.4)$$

For level h_B, i. e., the root level of the private B-trees, the expected number of entries differs between the Partitioned B-tree and equivalent private B-trees. The difference in the expected number of children, i. e., nodes in level h_{h_b-1}, leads to this discrepancy. In the Partitioned B-tree, the number of children of level h_B is given by $V_{p,h_B-1} = V_{B,h_B-1}(N) + \delta_{h_B-1}(N)$. Prefix compression loses its effectiveness in this level, by what we expect a maximum capacity of $2k(1-f_r)$ for nodes in this level. Thus, the expected average number of children of a node is $2k(1-f_r)u_{p,h_B}(N)+1$. In the private B-trees, each tenant has exactly one root node (level h_B). Therefore, $V_{B,h_B} = |B|$ holds. Consequently, the expected difference of nodes in level h_B between

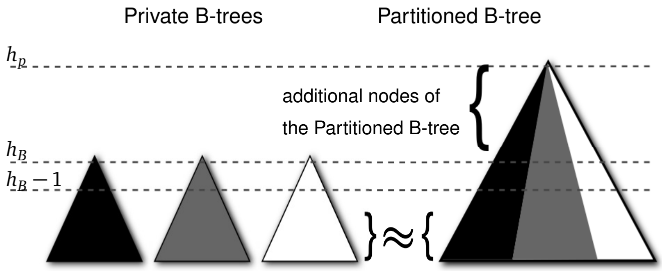


Figure 4.3: Compared to private B-trees, the Partitioned B-tree requires additional nodes between level h_B and h_p.

the Partitioned B-tree and equivalent private B-trees is

$$\delta_{h_B}(N) \approx \frac{V_{B,h_B-1}(N) + \delta_{h_B-1}(N)}{2k(1-f_r)u_{p,h_B}(N)+1} - |B| \tag{4.5}$$

So far, we omitted to consider additional nodes that the Partitioned B-tree possesses above level h_B. The number of these nodes depends on the expected number of nodes in level h_B, since the nodes in level h_B represent the leaves of the partial Partitioned B-tree between h_p and h_B, as illustrated in Fig. 4.3. The expected number of nodes in level h_B is $V_{p,h_B}(N) = |B| + \delta_{h_B}(N)$ and each node in level l, $h_p > l > h_B$, has at least $k(1-f_r)$ entries. Since in a B-tree of order $k(1-f_r)$ with N entries, $\frac{N-1}{k(1-f_r)}$ is an upper bound for the total number of nodes, we estimate an upper bound of expected nodes in level h_B and above by

$$\delta_{\geq h_B}(N) \lesssim \delta_{h_B}(N) + \frac{|B| + \delta_{h_B}(N) - 1}{k(1-f_r)} \tag{4.6}$$

The approximation model which is described in Sec. 4.2.2.2 allows computing the results for Eq. 4.4 – 4.6. However, we have not explained so far how to get $V_{B,l}$. By reason of uniformly random insertions across the key range, each tenant has the same expected number of nodes in each level of its private B-tree. Thus, $V_{B,l}(N) = |B| \sum_{i=0}^{N} Pr(E=i)V_l(i)$. The random variable E denotes that the private B-tree of a tenant has E entries. Since a Pólya urn [Eggenberger and Pólya, 1923] models the process of insertion with respect to the number of entries in a private B-tree, the random variable E has a beta-binomial distribution. Assume an urn that contains red balls for key intervals of a certain tenant and black balls for all key intervals of other tenants. Each ball (interval) is uniformly randomly chosen by definition. After choosing a ball, another ball of the same color is added. This process is analogous to choose a key interval for insertion of a new entry and add this entry afterwards. In our scenario, the beta-binomal can be estimated quite well by a poison distribution.

We wrote a small program that calculates results for Eq. 4.2 – 4.6 by using the described approximation model. The calculation assumed a fixed page size of 8 KB and a tenant identifier size of 2 bytes which leads to reduced B-tree order of $k(1 - f_r) = 4000/(4000/k + 2)$ (see Eq. 4.1). The results of Eq. 4.4 did not show a significant difference of nodes. We argue that the node utilization in the level underneath of the root level oscillates around the limiting value and converges quite fast for considered orders. We have observed this behavior while filling a B-tree randomly.

Figure 4.4 shows the results of Eq. 4.6, i. e., an upper bound for the difference between the expected number of nodes in the Partitioned B-tree and equivalent private B-trees in levels l, $l \geq h_b$, for two different B-tree orders and 1000 tenants. Given that Eq. 4.4 does not show a significant difference of nodes, it is not suprising that the graphs are almost linear. The visible minimum of the curve in both cases indicates that the heights of private B-trees increase from height 2 to height 3 (the transition from height 1 to height 2 lays on the y-axis due to the resolution of the figure). Note that a negative value of the difference means that the Partitioned B-tree requires less nodes in levels h_B and above compared to private B-trees. In both cases, the major part of the graph is negative which favors the Partitioned B-tree. The average of the absolute expected difference of nodes between the transition from height 1 to height 2 and the transition from height 2 to height 3 approves the first estimation. The average difference of nodes is about -268 for $k = 50$ and -252 for $k = 100$. Note that after the transition from height 2 to height 3, the behavior of the root levels mainly reiterates, but on a larger scale with respect to the number of entries, which is because

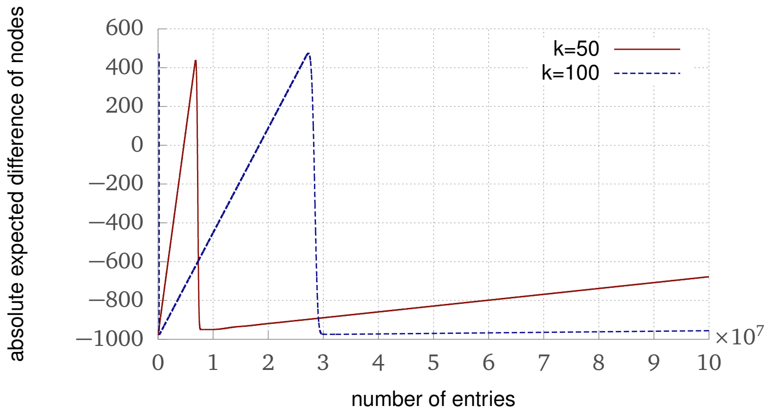


Figure 4.4: Approximation of upper bound for the difference between the expected number of nodes in the Partitioned B-tree that are in levels equal to root levels of private B-trees and above and the expected number of root nodes in private B-trees. The graphs show the difference for two different B-tree orders (k), assuming a fixed page size of 8 KB and 1000 tenants.

of the increased height. Note that the graph does not reflect the change of height of the Partitioned B-tree. The height of the Partitioned B-tree does not influence the difference of nodes, as the difference just takes into account the number of nodes in level h_B and above. The number of these nodes smoothly increases even if the height of the Partitioned B-tree increases.

4.2.3.2 Caching of Nodes

We assume that a B-tree node has the same size as a buffer pool page.

Our previous consideration concludes that $V_{B,l}(N) \approx V_{p,l}(N)$ for $l \leq h_B - 1$. Note that even if there is a small difference between the number of nodes in the Partitioned B-tree and the number of nodes in equivalent private B-trees in these levels, this difference will not become noticeable with respect to overall performance, given that nodes within a level are uniformly accessed. The number of nodes in these levels is already quite high. Thus, a node in these levels have low access probability and, thus, tend to vanish from the buffer pool anyways. This is different to nodes in level h_B and above. These nodes are more frequently accessed and, thus, tend to stay in the buffer pool. Hence, it is mandatory to pack information in these nodes as tightly as possible.

Our consideration additionally states that a Partitioned B-tree requires in average less nodes in level h_B and above, assuming uniformly random insertion of keys. The absolute gain seems very small, about 252 nodes for a B-tree order of $k = 100$ and $|B| = 1000$ tenants, i. e., about $1/4$ of $|B|$. Yet, as these nodes tend to be frequently accessed and therefore usually stay in the buffer pool, the node reduction is quite useful in this case, as it allows caching more information, i. e., more keys, in the same amount of main memory.

Despite of this advantage, from the perspective of a certain tenant, the Partitioned B-tree has a longer search path relative to equivalent private B-trees (about 2 or 3 nodes). Yet, if these additional nodes stay in main memory, the overhead boils down to searching within these nodes, which is negligible compared to accessing the disk. Thus, the additional upper levels of the Partitioned B-tree must stay in main memory to get performance that is comparable to private B-trees. As usual cache management strategies replace rarely accessed nodes, e. g., the LRU strategy, tenants have to access the Partitioned B-tree frequently to ensure that nodes in upper levels stay in the buffer pool. Naturally, the Partitioned B-tree wins at the time when most nodes in h_B stay in main memory.

For our further comparison, we assume that an accessed node stays time period t after its last access in buffer pool, thereafter it vanishes. We express t as number of accesses of the Partitioned B-tree or rather equivalent private B-trees. We assuming the Independent Reference Model [Coffman and Denning, 1973] for nodes in one level of a B-tree. The probability that a node is buffered is the same as the probability that the node was referenced within the previous t accesses of the B-tree, which is $P_{\leq t} = 1 - (1 - \beta_p)^t$. At that, β_p is the access probability of a node, which is the same for all nodes in a level since we assume uniform access. Therefore, β_p is the reciprocal of the the number of nodes in this level. Hence, the expected number of buffered

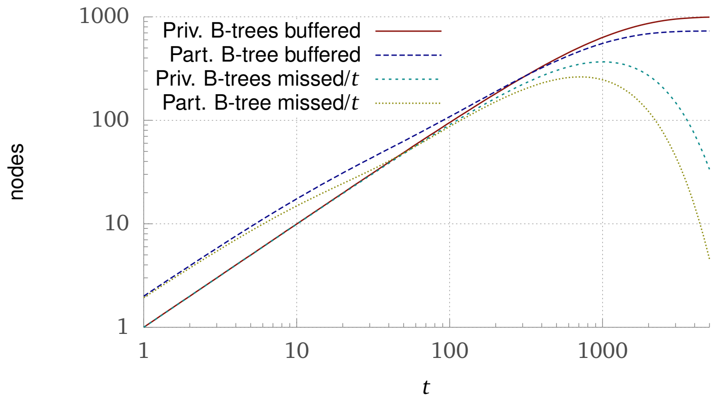


Figure 4.5: Comparison of average caching behavior between Partitioned B-tree and equivalent Private B-trees for nodes in level h_B and above. The parameter t expresses the time that a node stays in the buffer pool after its last reference by means of total accesses of the B-tree index. This comparison assumes 1000 tenants and a B-tree order of 100.

nodes for a given t and level l is given by $V_l P_{\leq t} = V_l(1-(1-1/V_l)^t$, and the expected number of node misses in a level per t accesses is given by $t(1-P_{\leq t})$. We use these two formulas to compare the cache behavior of the root level in the private B-trees, i. e., level h_B, with the corresponding level and the levels above in the Partitioned B-tree for the previously derived average case. For the Partitioned B-tree, we have to apply the formula separately to each level l with $l \geq h_B$ and sum up the results.

Figure 4.5 shows the results of the described approach using our example of 1000 tenants and $k = 100$. The graphs show the number of nodes that are kept in the buffer pool as well as the number of misses for t references. As expected, the Partitioned B-tree looses for low numbers of references. The reason is that the additional upper levels of the Partitioned B-tree also contribute to the number of misses in this case. Recall that a low number of references actually implies a higher probability that nodes vanish from the buffer pool. This disadvantage of the Partitioned B-tree diminishes fast. For $t > 30$, the misses of the Partitioned B-tree are on a level with the misses of the private B-trees. However, the Partitioned B-tree requires a small amount of additional nodes in the buffer pool (about 10) to accomplish the same number of misses. These additional nodes are the additional nodes of the Partitioned B-tree above level h_B. The Partitioned B-tree starts outperforming the private B-trees around $t = 200$. At this point, the Partitioned B-tree requires the identical buffer space, but it produces less misses.

4.2.4 Practical Considerations

Our analysis reports a better expected buffer pool hit ratio of the Partitioned B-tree compared to equivalent private B-trees assumed that the acessed B-tree nodes in upper levels are cached long enough in the buffer pool, which is usually given if they are accessed frequently.

Yet, the total savings which our analysis indicates seem small. In our example of 1000 tenants, the Partitioned B-tree saves about 250 pages near the root level, which sums up to 2 MB for a page size of 8 KB. Thus, if frequently accessed, the Partitioned B-tree yields similar savings of buffer pool space. The savings are negligible compared to typical buffer pool sizes (8 GB, 32 GB or even 64 GB). Apparently, saving 2 MB of 8 GB seems not worth mentioning. However, typical application schemas may easily reach 50 tables with at least one primary key index [Coelho et al., 2011], which are accessed frequently. In this case, the total savings are about 100 MB in average, for 10000 tenants even 1 GB, which is indeed a huge benefit. Another important benefit is that the Partitioned B-tree improves the buffer pool hit ratio and therefore reduces the disk I/O utilization.

Note that the Partitioned B-tree may also perform worse than equivalent private B-trees if it is infrequently accessed and, thus, the B-tree nodes tend to vanish from the buffer pool. According to our example in Fig. 4.5, the Partitioned B-tree may require about 10 pages more buffer pool space to obtain the same number of misses than equivalent private B-trees. This leads to about 4 MB more buffer pool space for 50 indexes and 1000 tenants, again assumed a fixed page size of 8 KB. This is quite negligible compared to the potential benefit.

Our consideration focuses on the average case, i. e., the case that all sizes of private B-trees or at least all sizes of a specific height are equally likely. In real application scenarios, this assumption does not hold necessarily. An unfavorable case for the Partitioned B-tree is that each tenant has a root node utilization which is between $ln(2)$ and 1. In our example, this would entail an expected difference of about 400 nodes, i. e., 400 additional nodes for the Paritioned B-tree. On the other side, for applications that mainly focus on small tenants, the benefit of a Partitioned B-tree tends to be larger than in the average case. According to our previous example, if all tenants store uniformly distributed up to 50000 entries, the expected reduction of nodes in the Partitioned B-tree is about 800 nodes for 1000 tenants. This is about three times more than our general results.

Note that our consideration assumes uniform index access across tenants. In practice, skew with respect to the activity of tenants is likely. High skew in the activity of tenants tends to decrease the benefit of the Partitioned B-tree, because parts of the Partitioned B-tree are rarely accessed. Thus, access locality of these parts gets worse by what the Partitioned B-tree is unable to play out its strengths. This is in line with a rarely accessed Partitioned B-tree, as shown in Sec. 4.2.3.2. Hence, it is mainly useful to consolidate tenants that show similar activity patterns.

We conclude this section with a pragmatic discussion regarding other sequences than uniformly random key insertion. Johnson and Shasha [1989] evaluated the expected storage utilization of B-trees with different mixtures of insertions and deletions. Their evaluation shows a similar shape and limiting value of the node utilization as in the case of pure insertions for less than 40 % of deletions. Thus, we presume that, in this case, the Partitioned B-tree comes with similar characteristics as our analysis shows.

Ordered insertion of keys constitutes another interesting and frequent use case. In this case, the utilization of nodes converges to 0.5. Therefore, the worst case from the perspective of the Partitioned B-tree gets worse relative to uniformly random insertions. In the worst case, the Partitioned B-tree doubles the required nodes in upper levels compared to private B-trees. Therefore, we presume that the average benefit of Partitioned B-tree shrinks for ordered insertions. Note, the discussed aspect of small tenants still holds for these sequences.

4.3 Consolidation into a Shared Heap

To get insights about the performance characteristics of consolidating tenants into a single heap, this section presents a comparison of two approaches: a private heap for each tenant and a single, shared heap for all tenants. The presented comparison mainly encompasses our previously published work in Schiller et al. [2011a]. As opposed to the analytical performance evaluation of consolidating tenants into a single B-tree, this evaluation uses measurements of executing different database workloads.

4.3.1 Testbed

At the time of this evaluation, a standard database benchmark or open source database benchmark implementation that accounts for multi-tenancy did not exist. Therefore, we developed an own tenant-aware benchmark, called *MTBench*, that mimics an online shop scenario and allows executing corresponding database workloads.

4.3.1.1 Tenant-aware Benchmark: MTBench

MTBench builds on the well-known TPC-C benchmark [Transaction Processing Performance Council, 2012]. Yet, to keep in view multi-tenant SaaS applications, MTBench's query mix is based on common query types occurring in SaaS and web applications [Aulbach et al., 2008; Urdaneta et al., 2009].

MTBench provides two tools: the data generator *MTBenchDataGen* and the workload simulator *MTBenchLoadSim*.

Transactions	Number and Access of Selected Tuples per Table									Workloads	
	Warehouse	District	Customer	Item	Stock	Order	Orderline	NewOrder	History	*Read*	*Write*
Login Update			1u							–	12.5%
Login			1r							12.7%	–
Random Items				20r						14.0%	14.1%
Price Items				20r						7.0%	7.0%
Category Items				20r						10.4%	10.3%
Item Detail				1r						52.3%	51.7%
Order	1r	1ur	1r	5–15r	5–15u	1i	5–15i	1i		–	1.2%
Availability				5–15r	5–15r					1.3%	–
Order Status						5r	25–75			0.8%	0.8%
Payment	1u	1u	1ur						1i	–	1.2%
Payment Status	1r	1r	1r							1.2%	–
Delivery			1u			1u	5–15u	1d		–	1.2%
										100%	100%
Stock Level					Sr					$\frac{Tenants}{GenerationUnit}$	

Table 4.3: The table shows the number and access of selected tuples (read, update, insert, delete) for each transaction. The two right-most columns show the distribution of transactions for the read and write workload. Stock Level is executed for each tenant once per *Generation Unit* of other transactions (0.14% for 100, 1.4% for 1000 and 14% for 10000 tenants relative to the number of other transactions).

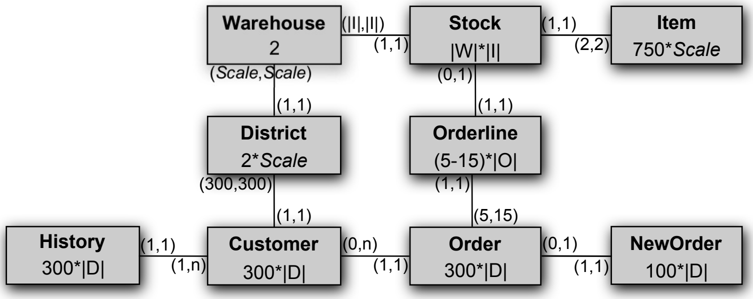


Figure 4.6: MTBench generates data for the depicted schema. The initial per-tenant table cardinalities base upon the parameter *Scale*.

MTBenchDataGen generates data according to the schema depicted in Fig. 4.6 and given parameters such as the number of tenants and the parameter *Scale* which determines the initial per-tenant cardinalities of tables. The boxes depicted in Fig. 4.6 include the per tenant-cardinality of the respective table.

After loading the generated data, MTBenchLoadSim allows executing a transaction mix on this data. For this purpose, MTBenchLoadSim offers two different transaction mixes: *read* and *write*. Table 4.3 summarizes the access characteristics of the respective transactions as well as their distribution with respect to the two workloads. All transactions, except for Price Items, Category Items and Stock Level, access tuples by their primary key. Price Items selects the first items whose price exceeds a randomly chosen price in ascending order. Category Items selects items per randomly chosen category. Stock Level accesses all stock tuples of a tenant.

MTBenchLoadSim uses the card deck approach that TPC-C proposes to ensure a given transaction distribution. The arguments of a transaction, most notably the tenant, are chosen randomly. All distributions are uniform. After executing a transaction, MTBenchLoadSim reports its end-to-end execution time as difference from the time of issuing the transaction to the time of retrieving the results. We refer to this time as *response time*.

4.3.1.2 Test Configuration

Using MTBenchDataGen, we generated three datasets: 100 tenants with Scale 100, 1000 tenants with Scale 10 and 10000 tenants with Scale 1. We loaded the generated data sets into six different databases: three databases with a shared heap per table and three databases with a private heap for each tenant and each table (i. e., the heap is partitioned according to the tenants). Subsequently, we refer to the former case as *Shared Heap* and to the latter case as *Private Heap*.

Thereafter, we created suitable non-partitioned B-tree indexes to support the query mix of MTBench. For Shared Heap, each table, except for the History table, got a B-tree index on the primary key. Furthermore, we created two additional B-tree indexes on the Item table (columns price and category), one additional B-tree index on the Order table (column customer) and one additional B-tree index on the Orderline table (column item). By this, all queries of the transaction mixes are supported by an index. For Private Heap, we omitted the indexes on the Warehouse and District table, because they contain only a few tuples per tenant which makes an index useless in this case. Each index obtained the tenant identifier as first index key. Moreover, we clustered each table by its primary key index, except for the Item table, which we clustered by its category index.

According to the described method, we generated databases for two different RDBMS implementations: PostgreSQL 8.4 and IBM DB2 10.1 LUW. PostgreSQL 8.4 does not allow to create a partitioned heap with a non-partitioned B-tree index. For this reason, we implemented a simple, preliminary partitioning solution, which is totally geared towards our use case. For this purpose, each tenant uses private files whose names are extended by the corresponding tenant identifier. The system knows about the files it has to access by the `SET TENANT` statement introduced in Sec. 3.4. This solution was designed for minimal overhead in order to allow for a clean performance analysis of storage model characteristics without impact of data dictionary efforts or complicated partitioning-related logic. For DB2, we used the facility to create range partitioned tables with non-partitioned B-tree indexes. Each table of the benchmark schema is range partitioned using the tenant identifier as partitioning key such that a partition includes exactly one tenant. The B-tree indexes are defined to span all partitions.

For load simulation, our test uses 8 database sessions in parallel. At this, MTBenchLoadSim distributes the number of tenants evenly over the sessions. For instance, for 1000 tenants, MTBenchLoadSim assigns the first 125 tenants to the first session, the second 125 tenants to the second session and so forth. Thus, we get rid of lock contention within one tenant which is beneficial for our considerations, as contention within one tenant may distort our measurement with respect to storage model characteristics. We started each test on a freshly booted machine. After a warm-up phase of 30 minutes, we measured a time window of one hour. We repeated each test three times. As the results were consistent, we only report the first run.

4.3.1.3 Hardware and System Configuration

We ran the databases on a Dell Optiplex 755 that was equipped with an Intel Core2 Quad Q9300 CPU running at 2.50 GHz and 4 GB of main memory. We stored the database and its write-ahead log on two striped 250 GB SATA 3.0 GB/s hard drives spinning at 7.200 RPM. The test machine ran a 64 bit 2.6.31 Linux kernel (Ubuntu release 9.10 Server). MTBenchLoadSim ran on a machine that was equipped with four Dual Core AMD Opteron 875 CPUs running at 2.2 GHz and 32 GB of main memory. The operating system was a 64 bit 2.6.9 Linux Kernel (CentOS release 4.8). The client machine was connected to the database machine over a 1 Gbit/s ethernet network.

We adapted the default configuration of PostgreSQL by setting a buffer pool size of 1024 MB, 128 MB of working memory, 12 checkpoint segments and a completion target of 0.8[1]. Furthermore, we disabled any automatic table analysis or reorganization tasks. We used the default page size of 8 KB.

We adapted the default configuration of DB2 by setting a buffer pool size of 1024 MB. Furthermore, we enabled the automatic memory manager of DB2 and disabled any automatic table analysis or reorganization tasks. The default page size was set to 8 KB. The tablespaces to store the data were

[1]`http://www.postgresl.org`

configured as database managed storage with an extent size of 4 and a prefetch size of 0; file system caching was enabled.

4.3.2 Performance Study

This section embarks upon first observations and remarks. Thereafter, it presents and interprets the results of running the two workloads of MT-BenchLoadSim for Shared Heap and Private Heap on PostgreSQL and DB2. For comparison, we took the 99th percentile of response times of Shared Heap as limit, and we compared the number of transactions of Shared Heap and Private Heap whose response time was below this limit. We always use Shared Heap as baseline and applied this calculation to each transaction type separately.

4.3.2.1 First Observations and Remarks

In the PostgreSQL case, all Shared Heap variants lead to a total database size of about 16.5 GB. For Private Heap, the total database size was slightly larger and increased with the number of tenants. For 10000 tenants, Private Heap occupied about 1.5 GB more than Shared Heap. This is explained by internal page fragmentation and overheads by auxiliary data structures such as the free space map.

IBM DB2 LUW essentially showed the same behavior as PostgreSQL. For Shared Heap, the total database size was about 16.2 GB independent from the number of tenants. Private Heap consumed about 17.2 GB for 100 tenants, 18 GB for 1000 tenants and 25.1 GB for 10000 tenants. Hence, each tenant and, thus, each additional data partition needed about 800 Kb. This is because IBM DB2 LUW allocates space in extent sizes. Thus, each data partition occupies at least as much pages as the defined extent size. Moreover, each data partition comes with an extent map that additionally allocates as much pages as the defined extent size. Thus, a smaller extent size may reduce the overhead per partition, i. e., using an extent size of 2 instead of 4 would halve the overhead per data partition.

Furthermore, the indexes grew between 6-10 % from Shared Heap to Private Heap. We assume that IBM DB2 LUW requires storing a partition identifier for each index entry in order to reference the data segment in which the related tuple is stored. This additional partition identifier yields larger index tuples and thus larger indexes. The relative increase naturally depends on the average size of the index tuples. Note that our implementation in PostgreSQL keeps off this identifier as it assumes that the tenant identifier itself identifies the related data segment.

After running first sniff tests to analyze the test cases for abnormal behavior, the test case for 10000 tenants and Private Heap on IBM DB2 LUW showed an unbearable performance loss. As opposed to other test cases, this case turned out to be heavily CPU-bound. The snapshot facility of IBM DB2 LUW showed a significant increase of lock waits, package cache utilization and catalog cache utilization compared to Shared Heap. Thus, we assume that the partitioning logic combined with the high number of partitions causes significant computation overheads when optimizing and running a query. Due to the short-running transactions which MTBenchLoadSim executes, these overheads lead to a significant overall performance degradation. For 100 and 1000 tenants, we also observed a slightly higher cpu utilization in Private Heap compared to Shared Heap. Yet, in our opinion, this slightly higher cpu utilization should not distort the results with respect to the interpretation of the characteristics of the two different storage concepts.

Note that although the table partitioning facility of IBM DB2 LUW allows to implement our Private Heap use case, it was actually designed for OLAP scenarios. In OLAP scenarios, table partitioning is often used to partition data according to certain time intervals, e. g., months. This allows running reports over a given time interval very efficiently since the RDBMS requires accessing only those partitions which contain data for the given time interval. This works for partitioning of tenants as well. Nevertheless, the number of data partitions between the traditional use case and our Private Heap use case is likely to differ about one or two orders of magnitude. Moreover, in the multi-tenancy use case, a query accesses exactly one partition since it belongs to exactly one tenant (except for administrative queries, of course). By contrast, in the traditional use case, a query may span multiple partitions

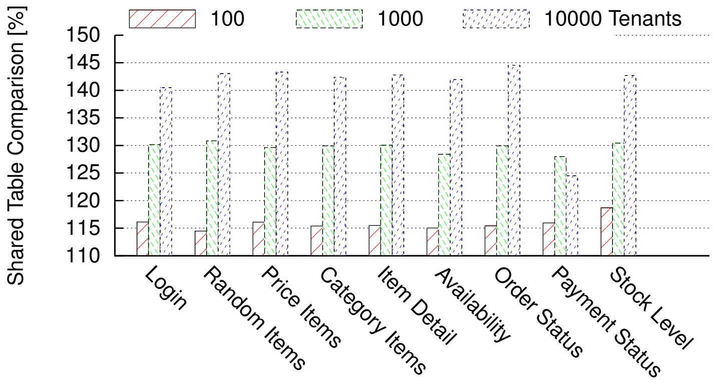


Figure 4.7: Relative comparison of Private Heap to Shared Heap on PostgreSQL for running the read workload of MTBenchLoadSim.

which are implicitly given by a selection predicate which requires complex planner logic. To sum up, the traditional use case requires more complicated logic and is optimized for a smaller number of partitions compared to the multi-tenancy use case. We assume this as the reason for the previously described behavior.

4.3.2.2 Read Workload

PostgreSQL Figure 4.7 shows that Private Heap outperforms Shared Heap for the read workload. Recall that we use the results of Shared Heap as baseline, i. e., a figure above 100 % means that Shared Heap performs worse than Private Heap, whereas a figure below 100 % means that Shared Heap performs better. The reason for the advance relies on the execution of Stock Level. As all tuples of a tenant's stock table are accessed during this transaction, Private Heap executes a heap scan. Contrarily, Shared Heap executes an index scan. The index scan is slower because it fetches index pages and heap pages. As additional negative side effect, index scans cause more random I/O requests than heap scans. Thus, the average wait time of I/O requests is higher for Shared Heap. For 100 tenants, we measured 42 ms compared to 39 ms for Private Heap. The lower average wait time of I/O requests accelerates page reads, which enhances Private Heap.

The measurements show that the performance advance of Private Heap grows with the number of tenants. This is because an index scan has to search for the start of the scan. Thus, there exists an absolute search overhead for each index scan. As the fraction of Stock Level executions increases by the number of tenants, the accumulated search overhead gets higher for more tenants, although the amount of accessed heap pages remains almost identical.

The performance trend of Payment Status obviously deviates from other transactions. The reason is the low per-tenant cardinality of the Warehouse and District table which Payment Status accesses. Due to the low cardinality of these tables, Shared Heap stores the tuples of all tenants into a few pages, e. g., two pages for the Warehouse table. Contrarily, Private Heap requires for each tenant a dedicated page. Hence, Private Heap tends to a higher internal fragmentation which decreases page utilization and, ultimately, buffer pool utilization. This effect gets more intense with increasing numbers of tenants. The measurements of the buffer pool hit ratio for the Warehouse and District table confirm this. For example, in Private Heap, the buffer pool hit ratio for heap pages of the Warehouse table decreases from 12% for 100 tenants to 2% for 10000 tenants.

DB2 Figure 4.8 shows that Private Heap outperforms Shared Heap for 100 tenants, although the total size of indexes was about 8 % larger. Thus, an implementation of Private Heap tailored to the use case would perform even better (as shown for PostgreSQL). For 1000 tenants, Private Heap heavily degrades compared with Shared Heap. This is different from the measurements using PostgreSQL. We explain this behavior by an increasing number of extent map pages that IBM DB2 LUW loads into the buffer pool in order to access the data partitions. To inspect the buffer pool contents, IBM DB2 LUW comes with the `db2pd -pages` utility. In several random samples of `db2pd -pages`, the buffer pool contained thousands of extent map pages

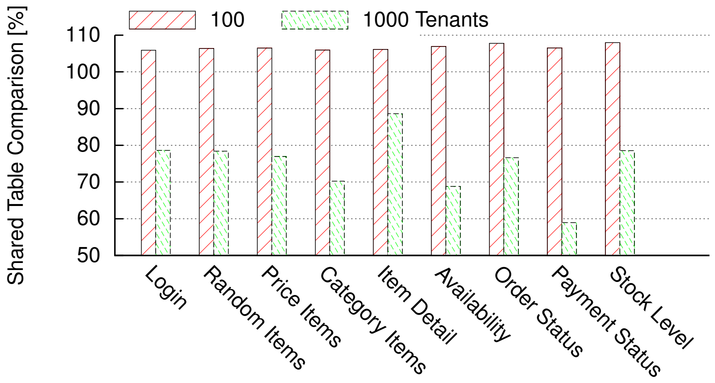


Figure 4.8: Relative comparison of Private Heap to Shared Heap on IBM DB2 LUW for running the read workload of MTBenchLoadSim.

for Private Heap and 1000 tenants. Contrarily, for Shared Heap, the number of extent map pages was always less than 50. This is no surprise as Private Heap uses a private extent map for each data partition. Thus, put simply, the number of extent map pages in the buffer pool increases with the number of accessed data partition. The high number of extent map pages in the buffer pool yields lower data pages in the buffer pool and thus reduces hit ratio compared to Shared Heap. This explains the performance degradation from 100 to 1000 tenants.

Despite of this side effect, the relative comparison of the change from 100 to 1000 tenants shows the same pattern as in the PostgreSQL case. Payment Status differs in the performance trend relative to other transaction. As previously, the reason is the low per-tenant cardinality of the Warehouse and the District table.

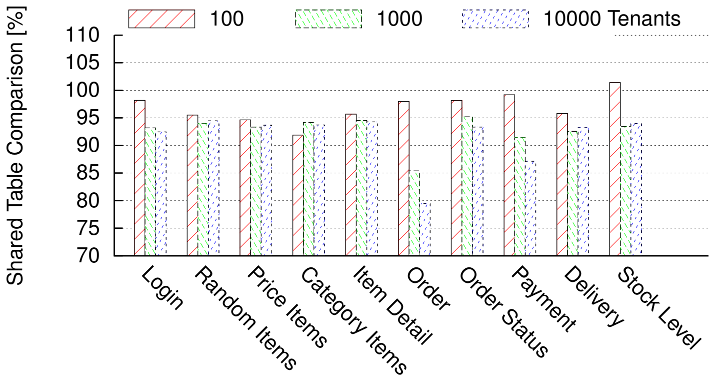


Figure 4.9: Relative comparison of Private Heap to Shared Heap on PostgreSQL for running the write workload of MTBenchLoadSim.

4.3.3 Write Workload

PostgreSQL Figure 4.9 shows the results for running the write workload on PostgreSQL. Private Heap looses its advance for this workload. For Payment, the measurements show the same trend as for Payment Status of the read workload. In addition to Payment, Order deviates significantly from the behavior of other transactions by degrading highly for larger numbers of tenants. This is because Private Heap has to spread insert requests of different tenants into their respective pages. Contrarily, Shared Heap may consolidate insert requests of different tenants into one page. This is quite beneficial for moderate insert frequencies. In our moderate write workload, the Order and the Orderline table of a tenant are accessed infrequently. Therefore, for Private Heap, the heap pages of these tables represent probable victims while searching space for allocations of other pages. Hence, such pages are often written out without being full. This leads to a lower probability of finding a page that has enough free space and is already in the buffer pool. Moreover, the buffer pool manager generally tends to write out more pages. The measured I/O characteristics confirm this result, e. g, for 10000 tenants,

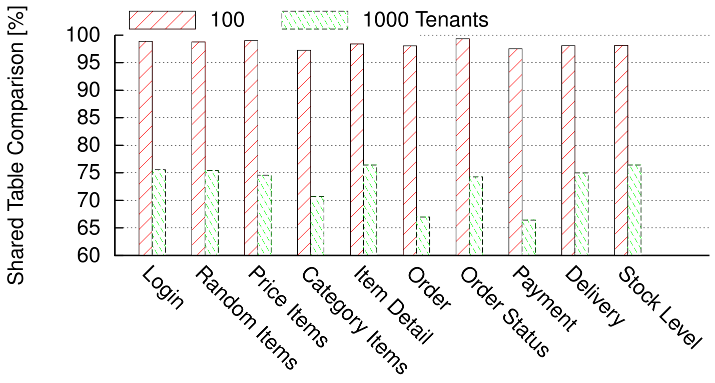


Figure 4.10: Relative comparison of Partitioned to Non-Partitioned Table on IBM DB2 LUW for running the write workload of MTBenchLoadSim.

the write/read ratio increases from about 0.7 for Shared Heap to 1.3 for Private Heap, although totals are nearly identical.

For 100 tenants, some transactions, mainly write transactions, e. g. Login and Order, degrade less than other transactions. The reason is contention with respect to page access. This effect diminishes for larger numbers of tenants, as the average response time of transactions gets a bit shorter and the benefit of write consolidation prevails.

DB2 Figure 4.10 shows the results for running the write workload on DB2. Similar to the read workload, the overall performance degrades considerably from 100 to 1000 tenants. This is again explained by the higher number of extent map pages required to access the data partitions. However, as in the Postgresql case, Order and Payment deteriorates more compared with other transactions from 100 to 1000 tenants. The reasons for this behavior are the same as in the PostgreSQL case.

4.3.4 Practical Considerations

The measurements essentially fulfill the expectations. The benefit of Shared Heap with respect to performance depends on the number of tenants and the table workload. A clear benefit of Shared Heap, which the measurements show, is the reduced internal page fragmentation which reduces the pages required to store the data of the tenants. The relative reduction of pages is $1 - \frac{0.5}{p}$ if the last page a tenant consumes is half full, which is assumed to be the average case, and p is the average number of allocated pages of a tenant. With this consideration, it clearly follows that the page reduction is only relevant for low numbers of allocated pages per tenant. However, for low numbers of allocated pages per tenant, the advantage of consolidation reduces when taking into account efficient data access. Private Heap requires only a simple table scan access path for efficient access, whereas Shared Heap requires an index-based access path. Hence, in order that Shared Heap makes sense, the size of the index has to be smaller than the savings through consolidation. For example, assume that 1000 tenants fill one page with 50 tuples which yields a half full page. In this case, a consolidation may save 500 pages, but a related b-tree index having an average fanout of 200 would require about 255 pages. Hence, the total savings are only 245 pages, i. e., about 25 %. If we assume a page size of 8 KB, the savings yield about 2 MB for 1000 tenants and 20 MB for 10000 tenants, which is negligible. Hence, the consolidation mainly pays off if the application requires several small tables or tenants tend to require even less than a half page, e. g., lookup tables that implement enumerations.

As opposed to Private Heap, Shared Heap may fulfill insertions of different tenants by modifying a single page, which increases insertion performance. In return, this results in higher contention for page access. To estimate this effect, we also ran some tests having a higher fraction of write transactions (15% Order transactions). Even in this case, consolidating insertions amortizes higher contention. Contention only affects performance significantly if the workload is extremely write-intensive or executes long-running transactions frequently, which anyway contradicts to a OLTP-style scenario.

A clear advantage of Private Heap is the physical segregation of tenants.

Thus, tenant-oriented bulk operations are feasible. Moreover, table scans only require accessing pages which actually contain tuples of the tenant under concern. However, note that a table which is clustered according to the tenants yields the same benefit. Clustering is often required anyway for efficient range scans, e. g., needed in parent-child rollups.

To summarize, Private Heap and Shared Heap are both feasible options whose suitability depends on the actual table workload and the number of tenants. For lower number of tenants (about 100 or less) Private Heap is definitely the way to go because the performance benefit is negligible in comparison with the loss of flexibility and customization opportunities. However, for large numbers of tenants, internal page fragmentation, overheads of auxiliary data structures and distributing insertions over many heap pages may degrade query processing considerably. In this case, Shared Heap plays out its strengths.

4.4 Adoption of Results for TenantSchema

The previous analysis indicates that there are data structure access patterns which benefit from the consolidation-free approach and there are data structure access patterns which benefit from table-wise consolidation. This opens up a large space of configurations which may even change over time, e. g., if the application introduces a new function that uses a certain index more often. As a result, the configuration becomes very complex. For this reason, it is discussable whether it is useful at all to decide about consolidation for each data structure separately.

4.4.1 Discussion

From our point of view, to decide for each B-tree index separately whether tenants should use private B-trees or share a single Partitioned B-tree does not come with much benefit. According to our analysis, for large numbers of tenants, the Partitioned B-tree provides in most cases a benefit and only rarely introduces a penalty with respect to performance, however, a negligible penalty. Nevertheless, for low numbers of tenants, the Partitioned B-tree

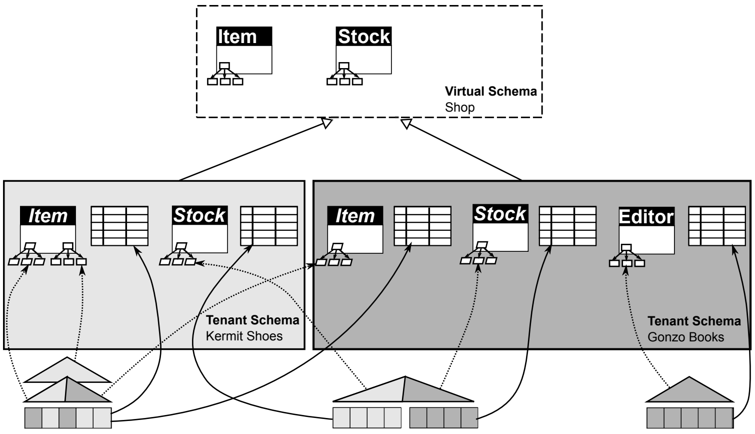


Figure 4.11: Storage Mapping Example for TenantSchema.

comes with no benefit, but it complicates isolation and customization. Thus, we argue that the only relevant decision is whether a tenant should use its private B-trees or whether a tenant should share Partitioned B-trees with other tenants.

For now, we assume that all tenants share Partitioned B-trees with other tenants. Hence, a B-tree index which is defined in a virtual schema s_1 on table t uses a single Partitioned B-tree to index all tuples of the horizontal table portion $h_{s_2}(t)$, where s_2 is a tenant schema and $s_1 \in P(s_2)$, i. e., s_1 is a predecessor of s_2. Fig. 4.11 illustrates this fact for the B-tree indexes which the virtual schema *Shop* defines on *Item* and *Stock*. Additional B-tree indexes which a tenant schema defines are mapped to a private B-tree, illustrated by the additional B-tree index which *Kermit Shoes* defines on *Item*. To separate a tenant's index tuples in private B-tree, we refer to Sec. 5, which presents an appropriate concept.

From our analysis, we conclude that during the definition of a table t in a virtual schema s_1 it makes sense to specify whether table portions $h_{s_2}(t)$, where s_2 is a tenant schema and $s_1 \in P(s_2)$, use private heaps or share a single heap.

Other table portions than the discussed ones use private data structures as described in Sec. 3.5.6. For example, additional tables which a tenant defines are mapped to private data structures, e. g., *Editor* which is defined by *Gonzo Books*. Note that shared data among tenants and private data of tenants is physically segregated since table portions that relate to virtual schema are stored in dedicated data structures. This physical segregation of data improves flexibility and performance with respect to replication and migration. For example, migration or replication of a tenant's data structures does not require including shared data.

4.4.2 Implementation of Shared Data Structures

To specify during creation of a table whether the related horizontal table portions in derived tenant schemas use private heaps or share a single heap, the `CREATE TABLE` statement takes an additional, optional argument `PARTITIONED BY TENANTS`. If this argument is given, each related horizontal table portion in tenant schemas uses a private heap. If the argument is omitted, the table instances are stored in a single heap file, i. e., the table storage is non-partitioned by tenants.

If the storage for a table is non-partitioned by tenants, each tuple stores the tenant to which it relates as a system attribute in order to distinguish the tuples of different tenants. We refer to this attribute as *tenant attribute*. The tenant attribute is part of the tuple header and thus transparent to the tenant. This yields a clean separation of system-relevant data and tenant data. In practice, a size of 4 bytes is probably sufficient to address all tenants uniquely. The overhead caused by this additional attribute depends on the average tuple size, similarly as described for B-tree indexes in Sec. 4.2.3.1. Note that if the table storage is partitioned by tenants, the tenant attribute is not needed to be stored since it is derivable from the context.

Each B-tree index which is defined for a table in a virtual schema automatically includes the tenant attribute as leading key for the index key definition.

Note that this is independent from the decision whether table storage is non-partitioned or partitioned by tenants.

All insertion operators set the tenant attribute to the tenant of the tenant context in which they are executed. Furthermore, the operators that carry out heap scans only return tuples that relate to the tenant of the current tenant context. The same holds for operators that carry out index scans. For this purpose, an index access operator includes the tenant identifier of the tenant context within it is executed as index search key. Naturally, maintenance operations that are triggered by the administrators may scan all tuples, e. g., to build a B-tree index or cluster a heap.

The system naturally interprets a tuple according to the schema of the related tenant.

4.5 Summary and Outlook

The introduction of tenants into an RDBMS yields a new dimension which has to be taken into account when storing tuples physically. For low numbers of tenants, private data structures for each tenant and schema give a good compromise between flexibility, isolation, performance and scalability. However, for large numbers of tenants, this approach leads to scalability issues which prevent to gain optimal performance. Our analysis of B-trees and heaps show that lowly utilized pages represent a central performance problem for large numbers of tenants since lowly utilized pages tend to deteriorate main memory utilization. This problem arises for the data structures itself as well as for auxiliary structures required to manage the data structures.

To improve page utilization for low numbers of tenants, the consolidation of multiple tenants into one data structure, i. e., one B-tree or one heap, represents a feasible option. For multi-tenant SaaS applications, the tables defined by the application core schema lend themselves for consolidation at the physical level, as they provide similar access pattern between tenants. To conclude, the benefit of data structure consolidation is comparable to machine consolidation but on another scale.

The consolidation of multiple tenants into a single data structure also comes with multiple drawbacks with respect to customization, maintenance and flexibility. The whole database has to be partitioned in reasonable chunks aligned to tenant boundaries. The size and composition of a reasonable chunk is determined by different criteria, e. g., a tenant's individual needs and the load portion needed to move in order to adapt to dynamic workload. The following section deals with this requirement in the context of a cluster architecture. In a cluster architecture, a feasible data partitioning and distribution approach is mandatory to run the infrastructure efficiently and dynamically.

The consolidation of multiple tenants into a single data structure calls for revisiting the standard implementation of the data structure and exploit the knowledge about tenants. For example, very efficient tenant identifier compression schemes are imaginable or an adaption of the ordered insertion heuristic used in B-trees.

Our approach improves scalability issues that arise due to large numbers of tenants which yield large numbers of table instances that relate to the same application core schema. However, if tenants define a lot of new tables, new scalability issues arise. This scenario equals to the requirements of the PaaS scenario, in which tenants use quite diverse tables with respect to structure and access patterns. The avoidance of unnecessary overheads in this scenario, e. g., by reducing auxiliary structures, is another important direction for future work.

CHAPTER 5

Cluster Architecture and Tenant Management

The capacity of a single machine does not suffice in all cases to cope with all tenants. In this case, scaling out, i. e., increasing the capacity of the infrastructure by additional machines, represents a common approach. Nowadays, a SaaS provider may easily implement this approach; it may simply demand new machines from an IaaS provider. Nevertheless, the used software, e. g., the RDBMS, has to be ready to run in a such a dynamic and distributed infrastructure.

Therefore, we propose an architecture sketch of a multi-tenant RDBMS cluster for OLTP-style SaaS applications, which targets such a dynamic, distributed infrastructure and takes into account multi-tenancy. A central requirement for such a system is to have an *efficient* granularity for customization and for administrative tasks like replication and load-balancing. Highly shared storage structures omit to provide such a granularity, for which we begin with introducing *TenantSpaces*.

5.1 TenantSpaces

TenantSpaces allow the user to partition tenants into isolated, reasonably manageable groups. Therefore, TenantSpaces facilitate group-oriented administration and allow exploiting similar behaviors and requirements of tenants. For example, a SaaS provider which serves tenants of different activity periods throughout the day may group tenants according to their activity. In doing so, the service provider may easily run maintenance operations only for the tenants which are currently inactive, which minimizes impact on query processing of active tenants. Moreover, grouping tenants according to their activity increases locality in shared structures between tenants.

5.1.1 Design and Implementation

TenantSpaces represent containers for tenants, i. e., each TenantSpace may contain multiple tenants, and a tenant belongs exactly to one TenantSpace, as illustrated in Fig. 5.1. A tenant is assigned to a TenantSpace during the creation of the tenant. The assignment may change due to the reorganization of a TenantSpace, as discussed in Sec. 5.1.2 and Sec. 5.3.

TenantSpaces allow limiting the degree of sharing, i. e., only tenants within one TenantSpace are allowed to share storage structures. Hence, a TenantSpace induces at least one heap per table and one B-tree per defined B-tree index. Therefore, a TenantSpace is physically independent from other TenantSpaces with respect to storage structures adopted to store and index tuples. Hence, TenantSpaces partition the whole system into physically independent horizontal slices that could be processed in bulk-oriented (at least page-oriented) manner, instead of requiring tuple-oriented access. Bulk-oriented processing of a TenantSpace is important for efficiency, particularly in a distributed system, e. g., the migration of a TenantSpace from one node to another and the replication of a TenantSpace.

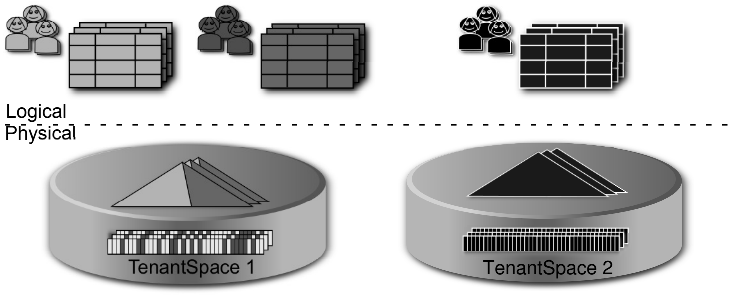


Figure 5.1: Illustration of TenantSpaces.

TenantSpaces enable group-oriented functionality and abstract from the underlying storage organization. To accomplish this, the system has to take TenantSpaces as processing granules, i. e., common maintenance and administrative operations use a TenantSpace as additional argument, e. g., migration, replication or clustering. For instance, the concept of TenantSpaces allows migrating thousand tenants within one TenantSpace to another node, instead of migrating only one of thousand tenants. TenantSpaces additionally reduce the problem size of optimization algorithms required for load-balancing, since they provide a more coarse-grained optimization unit.

To conclude, TenantSpaces represent a partitioning approach that is tailored to the needs and characteristics in multi-tenant SaaS scenarios. TenantSpaces establish clear boundaries of physical sharing and they provide suitable granularity for administrative tasks such as load-balancing and replication, which is required to implement scalability and availability.

Note that TenantSpaces are similar to tablespaces. However, they manage a different granularity, namely tenants instead of storage segments; assigning a tenant to a tablespace would not be intuitive and would not be correct from a semantical perspective. So far, we associate a TenantSpace with exactly one dedicated tablespace. For commodity servers that use one directly attached storage device, this approach is usually quite adequate, as a distinction between table data, index data or log data makes no sense. However, TenantSpaces do not require to be associated in one-to-one relationship with tablespaces. TenantSpaces may actually combine multiple tablespaces. In this case, TenantSpaces enforce that tenants are equally distributed and aligned across multiple tablespaces. Finally, TenantSpaces come with operations that are specific to multi-tenancy, as we subsequently discuss.

5.1.2 Reorganization Operations

During the lifetime of a service, tenants change: they grow, they shrink, they change their behavior or they even stop using the service. To adapt to these changes, the system requires reorganizing TenantSpaces, i. e., it requires changing the number of TenantSpaces and the assignment of tenants. For instance, the system may observe that a TenantSpace becomes too large or too hot for useful load-balancing. As a result, it may decide to split up the TenantSpace into two equally sized TenantSpaces that come with nearly equal load.

Against this backdrop, we propose the following three operations to reorganize TenantSpaces:

- split one TenantSpace into multiple new TenantSpaces,
- merge multiple TenantSpaces into one TenantSpace,
- move one tenant from one TenantSpace to another TenantSpace.

Note that these operations are intended to reorganize a TenantSpace only locally, i. e., limited to a single node. Hence, none of the proposed operations allows moving one tenant or TenantSpace from one cluster node to another cluster node. This functionality is decoupled from TenantSpace reorganization and discussed in the subsequent chapter.

5.2 System Architecture

The design of our system bases on a shared-nothing cluster architecture. The cluster comprises multiple commodity servers with directly attached storage devices. Each server runs a single RDBMS instance. All instances together serve the multi-tenant database which is partitioned across the cluster nodes. Each tenant server thereby serves multiple TenantSpaces or

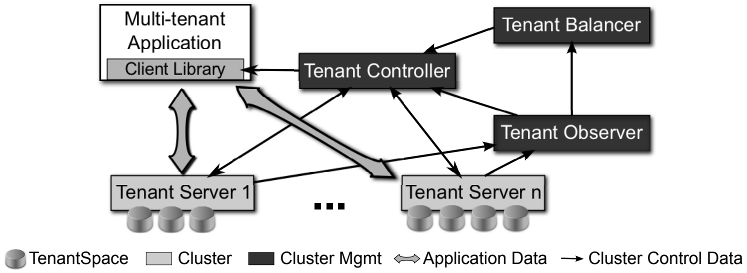


Figure 5.2: System architecture of a multi-tenant RDBMS for OLTP-style SaaS applications.

replicas of TenantSpaces. TenantSpaces are aligned to server boundaries which in turn ensures that tenants are aligned to server boundaries. Hence, a single node is able to process a tenant's queries.

The system architecture consists of the building blocks illustrated in Fig. 5.2. The main building blocks are: *tenant controller*, *tenant observer*, *tenant balancer* and *tenant server*. In the subsequent section, we describe the responsibility of the respective building block.

5.2.1 Tenant Server

The tenant server is the working horse with respect to query processing. A tenant server serves multiple TenantSpaces and processes queries issued by related tenants. For this purpose, the tenant server is built according to all previously discussed concepts; the tenant server is tenant-aware, supports TenantSchema, supports different storage concepts for tenants and supports TenantSpaces.

Furthermore, the tenant server supports replication to enhance load-balancing and availability. We envision a combination of synchronous multi-master replication [Kemme and Alonso, 2000; Wu and Kemme, 2005] and asynchronous master-slave replication. Consequently, there are few synchronous replicas on cluster nodes which are connected by low-latency links and additional asynchronous replicas on other cluster nodes, probably in another data center, e. g., in another availability zone if Amazon EC2 is used.

5.2.2 Tenant Controller

The tenant controller is a central component that stores the structural state of the system with respect to tenants, TenantSpaces and tenant servers. Hence, the tenant controller knows the available TenantSpaces and corresponding replicas, and it keeps track which tenant server currently serves which TenantSpaces and which replicas. Finally, it knows the association between tenants and TenantSpaces. To implement the tenant server, a highly available coordination system for distributed systems, e. g., Zookeeper [Hunt et al., 2010], is useful. Google Bigtable [Chang et al., 2006] or Solr [Kuć, 2013] are examples that adopt such a system.

The client application queries the tenant controller for information about the tenant servers related to a specific tenant. Note that the tenant is given by the `SET TENANT` statement. Thereafter, the client application issues data processing queries of the tenant under concern to one of the tenant servers which serves a synchronous replica of the TenantSpace that contains the tenant. Hence, the tenant controller does not take part in the data flow of query processing which enhances scalability of the tenant controller. Moreover, the client library (see Fig. 5.2) caches the returned state information about TenantSpaces and servers locally. The tenant server itself reports stale information, by what the client library is forced to query the tenant controller for fresh information.

In addition to the tasks mentioned, the tenant controller additionally manages the virtual schemas and caters for the replication of the virtual schemas across all tenant servers. In smaller installations, the modification of the virtual schemas is feasible as distributed transaction that involves all tenant server. However, larger installations require more sophisticated concepts and tools that support the incremental rollout of a new version of virtual schemas (see the discussion in Sec. 3.7).

5.2.3 Tenant Observer and Balancer

The tenant observer and the tenant balancer work highly intermeshed. Together, they provide an autonomic manager that implements the MAPE (monitor, analyze, plan, execute) loop for autonomic systems [Kephart and Chess, 2003].

For this purpose, each tenant server sends locally gathered and aggregated statistics about tenants and TenantSpaces to the tenant observer such as the number of transactions, average response times, number of logical reads and writes, number of physical reads and writes, used CPU time, average I/O times and so forth. The statistics are sent as part of heartbeat messages which each tenant server periodically sends anyway. The tenant observer is able to recognize node failures or network partitioning by missing heartbeats. The data gathered by the tenant observer provides the basis for the tenant balancer. The tenant balancer analyzes this data in order to ensure that the system meets the SLAs upon which the service provider agreed with tenants. In doing so, it additionally aims to utilize available resources optimally to maximize the gain of the provider. Based on the analysis, the tenant balancer places tenants within suitable TenantSpaces, reorganizes TenantSpaces and migrates TenantSpaces from one node to another. For example, it may split a TenantSpace into two TenantSpaces because the TenantSpace has become too large to run maintenance or administration operations efficiently. Basically, the tenant balancer plans these operations which the tenant servers actually execute.

Of course, the algorithms for the analyze, plan and execute phase come with many challenges, e. g., optimization of used resources, consolidation of different workloads, prediction of workloads, combinatorial optimization algorithms complexity, service availability, and more. The analyze and plan phase yields an optimization problem with multiple weighted objectives. There is a lot of ongoing research into such and similar problems, e. g., by Elmore et al. [2011a], Curino et al. [2011a], and Sousa and Machado [2012]. We feel confident that the solutions proposed for these problems are also useful for the implementation of the tenant balancer. Hence, we consider an evaluation of the mentioned solutions regarding their applicability for the tenant balancer and, ultimately, the design of the tenant balancer a central direction for future work. In the remainder of this thesis, we focus on operations required during the execute phase, namely TenantSpace reorganization operations and TenantSpace migration operations. These operations are challenging since they require minimal impact on query processing, particularly with respect to performance and availability.

5.3 Consideration of Online TenantSpace Reorganization

TenantSpace reorganization has to run *online*, i. e., in parallel to concurrently running transactions, in order to guarantee availability. This is because tenants may not suspend their activity throughout the day. Even if a tenant suspends its activity, it cannot be excluded that the tenant nevertheless issues some queries. The system may take out the replica which is reorganized from query processing such that only the other replicas take part in query processing. However, even in this case, the replica still gets the updates from other replicas in order to stay synchronous, which is comparable to concurrently running transactions.

Online TenantSpace reorganization for shared data structures is challenging because it induces a physical reorganization of shared heaps and shared B-trees, i. e., it requires moving tuples from one physical location to another physical location. Contrarily, a tenant's private data structures do not need to be modified, which allows processing them as a whole, often without any physical movement. Yet, even if a physical movement of the private data structures is required, it moves chunks at the granularity of pages or extents whose physical location is typically resolved by an additional mapping table anyway. Thus, the physical movement of private structures is simple to implement and usually does not impact the way how transactions access the data. Therefore, our considerations concentrate on shared data structures.

5.3.1 General Characteristics and Challenges

TenantSpace reorganization breaks down into multiple reorganization tasks, each task related to one table. The reorganization of one table does not influence the physical data structures of other tables, i. e., the reorganization of a table is independent of other tables. Therefore, we limit ourselves considering the reorganization of a single table, i. e., the related shared heap and shared B-trees that reference tuples within this shared heap.

Splitting and merging causes new sets of shared data structures: splitting induces at least one new heap and one new B-tree per defined B-tree index, and merging loses at least one heap and one B-tree per defined B-tree index. Accordingly, the reorganization approach has to allow moving tuples between data structures. This prevents the adoption of any approach that assumes in place operations.

Moving one heap tuple at a time and building the related new, reorganized B-tree indexes by insertion of the tuple is inefficient due to many random I/O operations. Therefore, it is desirable to build the new, reorganized B-tree indexes asynchronously by algorithms that provide sequential access patterns. A central challenge for such algorithms is that most RDBMS implementations use physical references within indexes, i. e., the tuple identifier (TID) which is stored as reference to a tuple is actually its physical location (at least with respect to pages). Thus, moving a heap tuple across pages requires updating the corresponding references in the indexes. Searching the reference is expensive, and updating references of the old, original B-tree indexes in place makes no sense because these indexes are eventually reorganized anyway. Thus, deferring the update of the references till reorganization of the indexes is recommended. For this purpose, the introduction of a temporary physical-to-physical mapping table which maps the old physical location of a tuple to its new physical location is usable [Sockut and Iyer, 2009]. The reorganization populates this physical-to-physical mapping table by entering a suitable mapping entry when moving a heap tuple. The mapping entry maps the old TID of the tuple within the original heap to the new TID within the reorganized heap. The old indexes are therefore still usable. Index access simply has to resolve the new TID by querying the physical-to-physical mapping table using the old TID. A hash table or a B-tree indexed by the old TID is therefore an adequate data structure for such a mapping table.

Another challenge is the impact of reorganization on the query processing performance. A suitable algorithm for TenantSpace reorganization has to be efficient in terms of overall performance and impact. However, reorganization operations may last long if they do not impact query processing severely. For this purpose, the algorithm should always allow forecasting the impact on query processing during the next reorganization step. This requirement is very important to be able to counterbalance high demands of query processing by slowing down reorganization operations.

To obtain insights about online TenantSpace reorganization, we subsequently consider existing and promising concepts to reorganize the shared heap of a table and related B-tree indexes. As online reorganization is needed in other highly available systems as well, e. g., for clustering, a broad range of approaches to database reorganization already exists. Sockut and Iyer [2009] surveys existing approaches and concepts comprehensively.

5.3.2 Shared Heap Reorganization

We subsequently consider two approaches which differ in the transition from the original to the reorganized data. The first approach reorganizes the whole heap until new transactions may access the reorganized heaps (heap-oriented heap reorganization). Contrarily, the second approach makes reorganized data available at page granularity, i. e., source pages are reorganized one by one and the reorganized tuples are directly accessed instead of the original tuples (page-oriented heap reorganization).

5.3.2.1 Heap-oriented Heap Reorganization

This approach bases upon fuzzy reorganization that is a widely used as a general concept for different reorganization task, e. g., structural schema changes, clustering and index reorganization. Fuzzy reorganization as a general concept is well explained in Sockut and Iyer [2009], which also surveys the different occurrences of fuzzy reorganization.

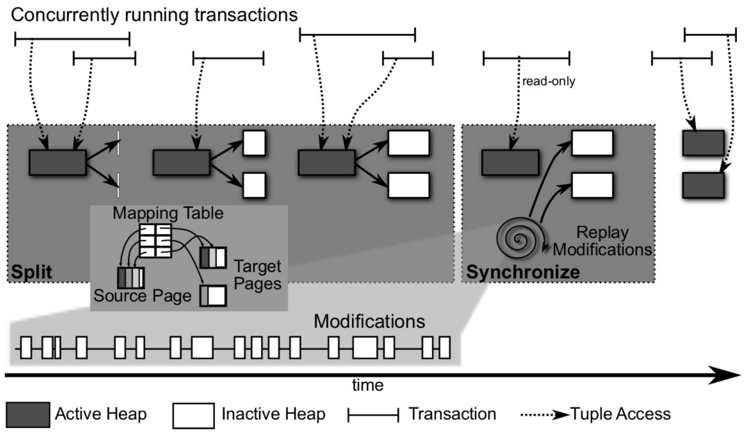


Figure 5.3: Illustration of heap-oriented heap reorganization using the split operation as example.

Løland [2007] explains how to carry out a horizontal split and merge of a table using fuzzy reorganization, which is exactly what we require for reorganizing the heaps during TenantSpace reorganization. Fig. 5.3 illustrates how fuzzy reorganization runs in order to split one heap into two new heaps. The algorithm runs in two phases: *Split* and *Synchronize*. Split scans all tuples of the source heap and stores them in the corresponding target heaps. The corresponding target heap of each tuple is determined by the tenant attribute (see Sec. 4.4). As already explained, the reorganization task maintains a physical-to-physical mapping table to keep track to which location a tuple is moved. Before Split starts, the reorganization task captures the current write ahead log[1] position. After Split has finished, all modifications (redo operations) stored in the write ahead log are replayed on the target heap in order to synchronize the target heaps with the source heaps. During Synchronize, update transactions are blocked. Finally, the mapping of the table and TenantSpace to its heap is changed. Thus, transactions now use the

[1]We assume that the system has a write ahead log. Naturally, there are other variants which work as well, e. g., explicit recording of modifications of the table under concern.

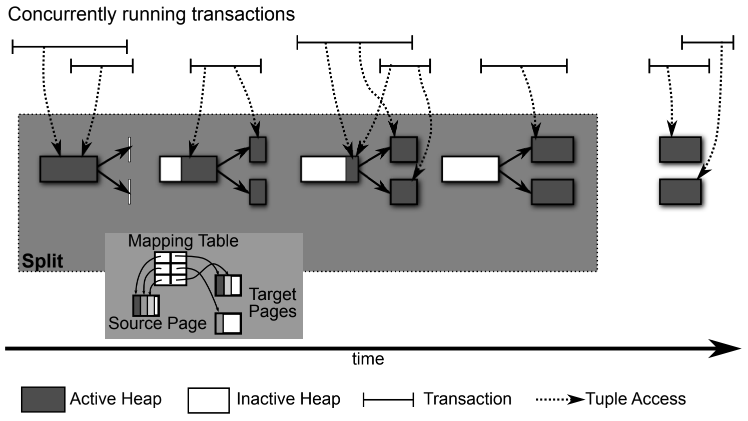


Figure 5.4: Illustration of page-oriented heap reorganization using the split operation as example.

new heaps. This last step may require some additional measures to ensure serializability, e. g., by aborting all transactions that are still accessing the source heap [Løland, 2007].

5.3.2.2 Page-oriented Heap Reorganization

As opposed to heap-oriented heap reorganization, page-oriented heap reorganization makes the reorganized data visible for concurrently running transactions one page at a time. IBM's IMS Version 9 uses a similar approach to reorganize HALDB partitions [IBM, 2011, pp. 382-391]; Omiecinski et al. [1992] propose a similar approach for clustering.

Fig. 5.4 illustrates this approach using the split operation as example. After splitting a page from the original heap and writing the corresponding tuples into the reorganized heap pages, concurrently running transactions access the tuples under concern directly at the reorganized heaps. In contrast to heap-oriented heap reorganization, this approach does not require replaying any modifications; it still requires a physical-to-physical mapping table, though.

This approach comes with a limitation: it assumes that the shared heap is never scanned during normal query processing of tenants, but tuples are only accessed by their TID. The TID contains the page number of the tuple. Therefore, the TID allows determining whether the page in which the tuple is stored is already reorganized, assumed that the reorganization task maintains a progress counter that stores the highest page number it has reorganized. If the reorganization task processes pages according to the order of the physical addressing, the access to the tuple only has to compare the current progress counter with the page referenced by the TID in order to determine whether it has to access the source heap or the corresponding target heap. In the latter case, it requires resolving the physical location of the tuple in the target heap by querying the physical-to-physical mapping table which is incrementally build during the split. This approach works as well for updating tuples. The insertion of a tuple lends itself to be carried out on the corresponding target heap.

5.3.2.3 Evaluation

We preliminarily implemented heap-oriented heap split and page-oriented heap split in PostgreSQL. We refer to Schiller [2011] and Mälzer [2013] for details about the implementations. Based on these implementations, we ran some simple, rough measurements. We omit to go into details of the measurements and their setup because we regard the measured, absolute numbers not as representative since both implementations are very preliminary, still offer optimization potential and are not really comparable. However, we use results of these simple measurements for illustration purposes in our subsequent discussion. We explain the trends we have seen so far and argue that these trends still hold for the optimized case.

Fig 5.5 illustrates the average transaction response times of parallel running transactions during heap-oriented and page-oriented heap split. Heap-oriented heap reorganization comes with an almost uniform impact on query processing during Split. Split mainly causes sequential read and write I/O operations for copying the database. The physical-to-physical mapping table, which we implemented as a B-tree, is gradually filled from left to right. Thus,

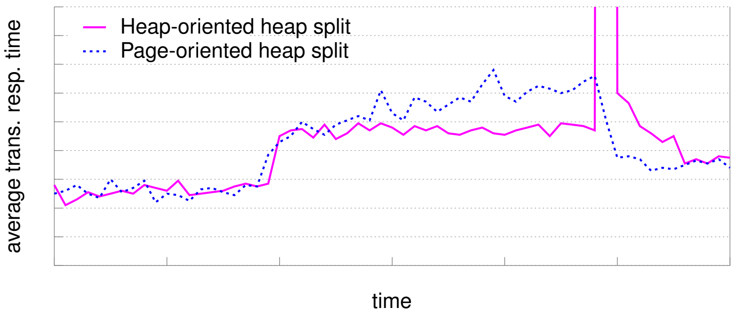


Figure 5.5: Illustration of the performance behavior for heap-oriented and page-oriented heap split.

the physical-to-physical mapping table only causes rarely additional write I/O operations. Hence, the impact on query processing is controllable by controlling the throughput of reorganization.

After Split, Synchronize follows. It is difficult to forecast how much and how long this phase impacts parallel query processing. The impact certainly depends on the volume of recorded modifications which in turn depends on the write percentage of the workload and the length of Split. The length of Split is highly variable even for the same table size given that a control-theoretic approach limits the throughput of Split to control its impact on parallel running transactions (in our tests the synchronize phase took about 10 seconds; the log volume was about 0.1 % of the database). The peak shown in Fig. 5.5 is caused as transactions are blocked during Synchronize. The bottleneck during Synchronize is definitely not reading the write ahead log but applying the changes. Applying the changes causes random I/O operations to load the pages of the physical-to-physical mapping table, which is accessed in order to map the old TIDs in the log entries to the new TIDs used in the target heaps, and to load the pages of the target heaps that require being modified. Note that using the write ahead log during Synchronize may require reading a lot of data which is not needed. This issue occurs if all TenantSpaces share one write ahead log, which is recommended for performance reasons [Chang et al., 2006].

After the reorganization has finished, the performance is still quite bad and requires some time to achieve a good level. At the time of switching the heaps, the buffer pool still contains many pages of the old heap, but transaction processing requires now pages from the new, reorganized heaps. Thus, these pages are now loaded into the buffer pool until a stable hit ratio is achieved. To conclude, the transition from the original data to the reorganized data is very harsh without control because the buffer pool contents require being swapped within a short time period.

Page-oriented heap reorganization also causes sequential read and write I/O operations for copying the database, exactly as heap-oriented heap reorganization does. As opposed to heap-oriented heap reorganization, page-oriented heap reorganization degrades query processing more and more during reorganization. This is because query processing has to access the physical-to-physical mapping table. The size of the physical-to-physical mapping table and the number of accesses to the physical-to-physical mapping table increases with the amount of reorganized data. Thus, the pages of the physical-to-physical mapping table vanish many other pages from the buffer pool. Nevertheless, its size could be estimated quite well and thus also its impact. If the RDBMS is able to provide enough free main memory for the physical-to-physical mapping table, the impact of accessing the mapping table is nearly constant and negligible. Our mapping table implementation is straightforward, by what it may reach a considerable size (in our tests it reached about 10 % of the database size). Yet, we feel confident that an intelligent compression approach, e. g., prefix compression combined with delta compression and variable length encoding, is very effective, since all TIDs that refer to the same page start with the same page number and the positions of tuples within a page are ascending. Assumed that the physical-to-physical mapping table compresses to about 40 % of its size and a mapping entry consists of two TIDs, each having 6 bytes, the physical-to-physical mapping table would require only about 50 MB for 10^7 tuples (10000 tuples for each of 1000 tenants), which is only about 5 % of the table size given that the average size of a tuple is 100 bytes.

Page-oriented heap reorganization interferes more with query processing than heap-oriented heap reorganization since it has to synchronize its activity with the activity of query processing by means of exclusive locks or latches. However, the required locks or latches are very short, thereby conflicts that limit parallelism are rare. In our preliminary measurements, we did not observe any contention issues.

Note that the transactions still have to access the physical-to-physical mapping table after successful heap reorganization. Therefore, the performance after the heap reorganization is worse than before the heap reorganization. This performance degradation holds until the B-tree indexes are split as well.

Although we used the split operation as example during our previous discussion, the discussed points apply similarly to merging TenantSpaces and moving a tenant from one TenantSpace to another TenantSpace.

5.3.3 Shared B-tree Reorganization

As soon as a heap is reorganized, related B-tree indexes require being reorganized as well. The recreation of B-tree indexes from scratch would require rescanning the whole heap and sorting the tuples according to the index key, which is undoubtedly expensive. Thus, scanning the leaves of the existing B-tree indexes in order to build the new indexes is recommended; the existing B-tree indexes are smaller than the new heaps and already provide the correct order of the tuples. To use the existing B-tree indexes, index reorganization has to replace the old TID references by new ones using the physical-to-physical mapping table built during heap reorganization.

The fuzzy reorganization approach works for B-tree indexes exactly as described for heaps (Omiecinski et al. [1992] being an example) and comes with the same characteristics. A page-oriented approach as described for heaps is difficult to establish since the search key does not allow determining whether the pages related to the search are already reorganized. An alternative uses the index key to keep track of the reorganization progress. In doing so, a boundary value for the index key allows the system to determine whether the tuples related to the search key are already reorganized or

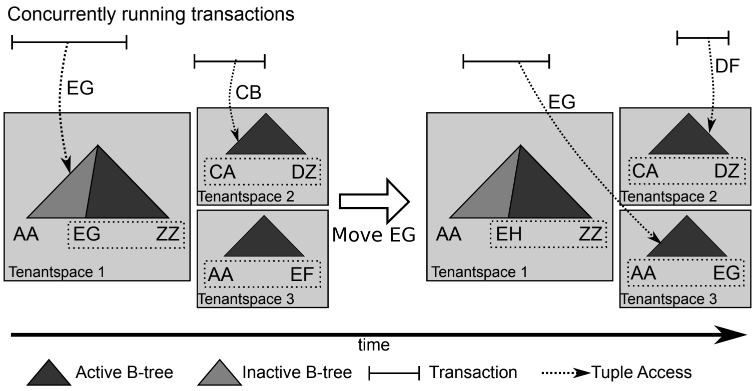


Figure 5.6: Illustration of shared B-tree reorganization that reorganizes one key at a time.

not [Omiecinski, 1988; Zou and Salzberg, 1998]. Naturally, it is feasible to process a whole leaf page at once. In this case, the highest key found within the page represents the new boundary value[2]. This approach is comparable to a page-oriented approach, yet an index key determines the boundary value which indicates the progress of the reorganization.

Our use case inevitably leads to the consideration of partial indexes [Stonebraker, 1989] whose covered key ranges change according to the progress of the reorganization. Fig. 5.6 illustrates this approach using the split operation as example. The illustration shows the split of TenantSpace1 into two new TenantSpaces, namely TenantSpace2 and TenantSpace3. Each TenantSpace contains a B-tree index. Each index is associated with a predicate that determines its responsible range. At the beginning, the original index covers the whole range and the target indexes naturally are empty. Each key is reorganized at a time, in the example the key EG is moved from the original TenantSpace1 into the target TenantSpace3. For each reorganized key, the lower bound of the predicate associated with original index is incremented

[2]For simplicity, we assume unique B-tree indexes only. In non-unique B-tree indexes, the reorganization of a single key may actually require reorganizing of multiple entries, possibly even spread over multiple pages.

to the reorganized key (as exclusive boundary), and the upper bound of the predicate associated with the corresponding target index is incremented to the reorganized key. Query processing takes into account the predicates and consult the suitable index. Note that Stonebraker [1989] assumes that a partial index is only usable if its associated predicate completely contains the search predicate. Regarding point-wise index lookups, exactly one of the partial indexes is usable. In case of searches over a range, the range predicate requires being broken up at the boundary value. As a result, two indexes are required in order to process the query. This induces an overhead (two index lookups instead of one), but at a certain point of time this overhead only concerns a single tenant and only queries whose range predicate includes the boundary value. In practice, we regard the overhead therefore negligible.

5.3.4 Conclusions

5.3.4.1 Applicability of Considered Approaches

From our previous discussion, we conclude that page-oriented heap reorganization combined with key-oriented (page-oriented) B-tree reorganization represents a feasible approach to reorganize TenantSpaces. This approach smoothly changes the contents in the buffer pool during reorganization. As opposed to other approaches, this entails well controllable behavior regarding the performance impact. Furthermore, it avoids redundant work since it omits to require additional synchronization steps by means of replaying modifications.

Although page-oriented heap reorganization is a feasible approach for TenantSpace reorganization, we are not fully convinced of this approach. The physical-to-physical mapping table may become large and impact query processing over a long time period. In our opinion, tenants provide a useful, smaller granularity for mainly independent units of work. However, a reorganization approach that processes each tenant separately is usually expensive because, as opposed to the presented approaches, it runs totally tuple-based, i. e., it fetches each tuple of a tenant separately. Typically, this yields random I/O operations, except if the heap is clustered according to an index and the reorganization algorithm uses the order of tuples which the index imposes. Based on these constraints, we subsequently outline an algorithm, whose implementation and evaluation represents an interesting direction for future work.

5.3.4.2 Sketch of a Tenant-aware Reorganization Approach

This reorganization approach is tuple-oriented and runs in iterations, i. e., one iteration for each tenant. It is actually comparable to an iterative application of the move operation, which moves exactly one tenant from one TenantSpace to another TenantSpace. The reorganization of one tenant runs exactly as described for the B-tree index reorganization based on partial indexes, i. e., partial indexes are used to transition to the reorganized state gradually. In doing so, the reorganization approach starts with the clustering B-tree index. As opposed to pure B-tree index reorganization, this approach reorganizes the related heap tuples in parallel, i. e., if the reorganization task reorganizes the index entries for a certain key, it additionally reorganizes the heap tuples which are referenced by the index entries under concern.

To enable access by other indexes, a physical-to-physical mapping table is still necessary. Other B-tree indexes are reorganized by the described method based on partial indexes. Thereafter, the physical-to-physical mapping is unnecessary. Thus, the volume of the mapping table is limited to one tenant rather than the complete TenantSpace (recall that the described steps are carried out for a single tenant). A smaller physical-to-physical mapping table is naturally advantageous since less main memory is required. Moreover, the direct impact[3] of reorganization for a single tenant is shorter.

The performance of this approach highly depends on the correlation between the order of the entries in the clustering B-tree index and the order of the tuples in the heap: the higher the correlation, the lower the random I/O operations. Note that this algorithm costs the same as an equivalent clustering which has to be periodically done anyway. Hence, TenantSpace reorganization might ride piggyback on a clustering run. Note that although

[3]The direct impact is without the impact which may occur due to the additional I/O operations required for reorganization.

we discussed this sketch in the scope of TenantSpace reorganization, it also represents a viable approach for other reorganization tasks, such as clustering or compaction. Finally, the algorithm is quite flexible; it also allows processing multiple tenants in a row which might be interesting if tenants are very small in order to exploit sequential access.

5.4 Summary and Outlook

TenantSpaces represent a concept that partitions the database stored at a single node aligned to tenants. Thus, TenantSpaces allow creating manageable storage granules even in the case of deeply shared storage structures. Furthermore, they provide clear boundaries of sharing and they allow grouping tenants according to their behavior and their needs. TenantSpaces hereby allow for seamless configurations between the two extremes: all tenants share one TenantSpace or each tenant is assigned to its private TenantSpace.

Local and horizontal partitioning of shared storage structures is particularly important in a distributed, dynamic environment in order to accomplish load-balancing and availability in an efficient and flexible manner. Multiple systems already base upon this concept, e. g., Google Bigtable [Chang et al., 2006]. In contrast to these systems, TenantSpaces keep in view tenants since they result in a horizontal partitioning that aligns to tenants across all storage structures.

The repartitioning, i. e., the reorganization of TenantSpaces, is a fundamental operation that has to run in parallel to transaction processing. Our considerations of existing reorganization approaches conclude that an approach that incrementally makes reorganized data visible to transaction processing has desired characteristics: it is efficient since it does not yield work that is done redundantly and it provides a well controllable behavior.

Our considerations of TenantSpace reorganization has led to a sketch of a new algorithm which uses tenants as unit of work to reduce the maximum overhead during reorganization. Although we discussed this algorithm in the scope of TenantSpace reorganization, it is useful for other reorganization tasks as well, e. g., clustering and schema modification. Thus, we consider the implementation and evaluation of this algorithm as an important task for future work.

The tenant balancer, which we introduced in our presented architecture to a multi-tenant RDBMS cluster for SaaS applications, represents a critical component whose design is important as future work.

In the next chapter, we consider another important operation which the cluster has to provide: live TenantSpace migration, i. e., the migration of a TenantSpace from one cluster node to another cluster node in parallel to query processing.

CHAPTER 6

Live Database Migration

Efficient load-balancing constitutes a central competency to guarantee good resource utilization and to ensure that the infrastructure meets SLAs upon which the service provider and the tenants agreed. In this connection, a primitive operation represents the migration of load from one cluster node to another cluster node, e. g., to avoid overload situations or to reduce energy consumption. In our presented multi-tenant RDBMS cluster for OLTP-style SaaS offerings, the tenant balancer decides about the migration of a TenantSpace from one node to another node; the tenant balancer decides when to migrate, which TenantSpace to migrate and where to migrate the TenantSpace. However, it only initiates the migration, but it does not know how to migrate a TenantSpace. The actual migration of the TenantSpace is done by the cluster nodes itself.

This chapter concentrates on the question how to migrate a TenantSpace from one cluster node to another. As services nowadays run 24/7, the migration must not interrupt normal processing, which means the migration has to run *live*. Moreover, migration requires running efficiently and should impact query processing only lowly. *ProRea* which is our live database migration approach, as published in Schiller et al. [2013], is designed to fulfill these requirements for OLTP-style workloads that use snapshot isolation as concurrency control mechanism. This chapter presents the basic concept of ProRea. It analyzes costs and trade-offs of ProRea and outlines an implementation of ProRea. Thereafter, results of measurements underpin the good performance of ProRea compared to comparable live database migration approaches. Finally, considerations with respect to log-based recovery during migration complete this chapter.

Note that it does not matter whether we migrate a TenantSpace or a database. According to our definition of a TenantSpace, a TenantSpace is indeed comparable to a database from the perspective of migration. Thus, for the remainder of this chapter, we use the more general term database to emphasize that ProRea is not only useful in our proposed architecture and scenario but also in other live database migration scenarios.

6.1 State of the Art in Database Migration

Subsequently, we describe existing techniques for migration specific to databases. An implementation of these techniques usually uses and exploits features of the RDBMS or is even tightly integrated into the RDBMS. We start with *Stop and Copy* as a straightforward approach and go on with live database migration approaches.

6.1.1 Stop and Copy

Stop and Copy shuts down the database, flushes it to disk, copies the database from the source to the destination, and, finally, restarts it at the destination. Obviously, this approach is straightforward and very efficient but also has some intrinsic drawbacks.

First of all, the database is offline during migration. For example, to copy a database having 2.5 GB over a 1 Gbit/s ethernet requires at least 20 seconds, assumed full network transfer speed. Yet, co-located tenants on the same machine or other machines may observe severe degradation when copying full speed ahead. Limiting transfer speed actually solves this issue but lengthens service interruption. In addition to that, the database runs with a cold buffer pool after migration which naturally impacts performance. For the points mentioned, live migration approaches that minimize service interruption and performance impact are desired.

6.1.2 Live Migration

Live migration is a topic which fascinated academia during the last decades. In this connection, live migration of processes was a really popular topic; Milojičić et al. [2000] provides a comprehensive survey on live process migration. Nevertheless, process migration has not taken a significant role in practical use. In contrast, live virtual machine migration sees increasing practical use in recent years, e. g., in Xen VMM [Barham et al., 2003], probably due to the increasing proliferation of virtual machine technology.

In principle, live migration of virtual machines enables live migration of relational databases as well; a virtual machine may simply wrap a corresponding RDBMS instance. However, virtual machines come with huge footprints which entail large fixed overheads and thus contradict to the goal of minimizing fixed overheads per tenant, as already discussed in Sec. 2.2. Furthermore, virtual machine migration and process migration do not know the concepts of the RDBMS. They concentrate on another, more heavyweight level; they transfer whole processes or virtual machines instead of only a database. Thus, they do not exploit useful characteristics specific to the RDBMS, e. g., optimized buffer pool replacement algorithms or characteristics of the concurrency control mechanism.

The points stated above initiated research into live migration of relational databases. A common practice for live database migration is what we refer to as fuzzy migration since it is similar to fuzzy backup and fuzzy reorganization [Sockut and Iyer, 2009]. The core principle hereby is exactly the same as explained for heap-oriented heap reorganization during the considerations of TenantSpace reorganization in Sec. 5.3. A migration process creates and transfers a fuzzy dump of the database to the destination. Thereafter, it transfers recorded modifications of the database which occurred while dumping and transferring and applies them on the database image at the destination. The previous step may run repeatedly in hope that the volume of modifications that accumulates during each step decreases. Finally, the migration process shuts down the database at the source, transfers the last set of recorded modifications, applies them on the database image at the destination, and restarts the database at the destination. Now, the database at the destination is able to process transactions. This approach is proactive as it copies the data to the destination before switching transaction processing.

Recent research applied this technique on different system architectures and evaluated different aspects [Das et al., 2011; Sean Barker et al., 2012; Umar Farooq Minhas et al., 2012]. This approach is quite simple and applicable by adopting standard database tools. However, it also has some considerable drawbacks: redundant work at the source and the destination, higher network traffic compared to stop and copy, blocked updates during replay of changes recorded in the last step, and transaction load stays at the source until the whole data is transferred, which is undesired for out-scaling.

To overcome these drawbacks, Elmore et al. [2011c] proposed a reactive approach, called *Zephyr*. Reactive approaches switch transaction processing from the source to the destination before they copy the data. If a transaction requires a page, it has to pull the page synchronously from the source.

ProRea represents a hybrid approach, which is similar to Zephyr, but uses *pro*active and *rea*ctive measures to minimize the number of page faults and to improve buffer pool handling at the source. Moreover, ProRea bases upon snapshot isolation, whereas Zephyr bases upon strict 2-phase locking. Nowadays, snapshot isolation became very popular; major RDBMS implementations including Microsoft SQL Server, Oracle, MySQL, PostgreSQL, Firebird, H2, Interbase, Sybase IQ, and SQL Anywhere support snapshot isolation. During migration, snapshot isolation allows for simple synchronization between transactions that run concurrently at the source and the destination.

The synchronization concept of ProRea never aborts read transactions and only rarely aborts write transactions at the source due to migration.

6.2 ProRea

Live database migration techniques, especially techniques which are tightly integrated into the RDBMS, depend on the adopted concurrency control mechanism. For this reason, we briefly review the principle of snapshot isolation on which ProRea relies. In this context, we also summarize the central environment conditions for which ProRea is designed. Subsequently, we explain the algorithmic building blocks of ProRea. Finally, a semi formal proof that the presented algorithmic building blocks fulfill snapshot isolation follows.

6.2.1 Snapshot Isolation and other Environment Conditions

For a short period of time, ProRea allows running transactions concurrently at the source and at the destination. The measures required to ensure isolation during this period depend on the used concurrency control mechanism. ProRea assumes snapshot isolation [Berenson et al., 1995a], which enforces the following two rules [Normann and Ostby, 2010]:

1. A transaction T reads the latest committed tuple version that exists at the time T started, i. e., it never sees changes of concurrently running transactions.
2. A transaction T aborts if a concurrent transaction of T has already committed a version of a tuple which T wants to write (*First Committer Wins*).

In practice, snapshot isolation is implemented by multi-version concurrency control (MVCC) [Reed, 1978]. Typical MVCC implementations maintain several versions of one tuple in a chronologically arranged *version chain*. This version chain is traversable and may span multiple pages. A page provides information where to find the previous or later version of each tuple it contains. Thus, the access of a specific version of a tuple may require following its version chain, which may cause multiple page accesses.

For instance, PostgreSQL uses a non-overwriting storage manager and a forward chaining approach that allows traversing all tuple versions starting from the oldest version. For this purpose, a tuple stores a reference to its successor with respect to time. By contrast, Oracle's database software updates tuples in place and stores older versions in so-called undo segments. In doing so, it uses a backward chaining approach that allows traversing tuple versions starting from the newest version. The InnoDB storage manager in MySQL and the solution in MS SQL Server use similar approaches.

Instead of the First Committer Wins rule, typical implementations enforce a slightly different rule, referred to as *First Updater Wins* rule. This rule prevents a transaction from modifying a tuple if a concurrent transaction has already modified it, usually by means of locking, preferably tuple-level locking. If the first updater commits and releases locks, the waiter aborts. If the first updater aborts and releases locks, the waiter may modify the tuple. Hence, the First Updater Wins rule eventually leads to the same effect as the First Committer Wins rule.

We focus on snapshot isolation as it avoids most concurrency anomalies, but it provides higher concurrency compared to strict 2-phase locking. For this reason, snapshot isolation became popular in recent years and most major RDBMS implementations nowadays support snapshot isolation. Note, the concurrency control only affects one step of ProRea which could be easily replaced for another concurrency control mechanism, e. g., Elmore et al. [2011c] proposed a strategy for concurrently running transactions at the source and the destination that bases upon strict 2-phase locking which could be applied as well.

In summary, the environment conditions on which the design of ProRea bases are:

- an RDBMS cluster with shared nothing system architecture and database location transparency, as presented in Sec. 5.2,
- snapshot isolation based on MVCC and the First Updater Wins rule as concurrency control mechanism (as previously discussed),

- OLTP workloads that mainly consist of short-running transactions (in line to our use case, see Sec. 2.4),
- and reliable, order-preserving, message-based communication channels having exactly-once semantics.

6.2.2 Basic Concept

ProRea runs in five successive phases: 1. *Preparation*, 2. *Hot Page Push*, 3. *Parallel Page Access*, 4. *Cold Page Pull* and 5. *Cleanup*. Figure 6.1 outlines the interaction of the source and the destination in each phase. First, Preparation initializes migration at the source and the destination. Thereafter, Hot Page Push proactively pushes all hot pages, i. e., pages in the buffer pool, from the source to the destination. During Hot Page Push, each transaction that runs modification operations on an already transferred page sends suitable modification records to the destination. The destination applies the modification records to be consistent with the source. Next, the source hands over transaction processing to the destination. From now, the destination processes newly arriving transactions. During Parallel Page Access, transactions run concurrently at the source and at the destination. If a transaction at the destination requires a page which has not been transferred yet, it synchronously pulls the page from the source. After the last transaction at the source completes, Parallel Page Access transitions to Cold Page Push. Cold Page Push pushes the data which has not been transferred yet, the potentially cold data, to the destination. Finally, migration finishes with cleaning up used resources.

During migration, we use the concept of page ownership to synchronize page access between the source and the destination. If a node owns a page, it has the current version. Moreover, only the node that owns the page may modify it. Note that the ownership concept is a low-level concept that ensures consistent, synchronized page access across the source and the destination. On the contrary, transaction-level synchronization is accomplished by snapshot isolation which builds upon multi-version concurrency control and tuple-level locks.

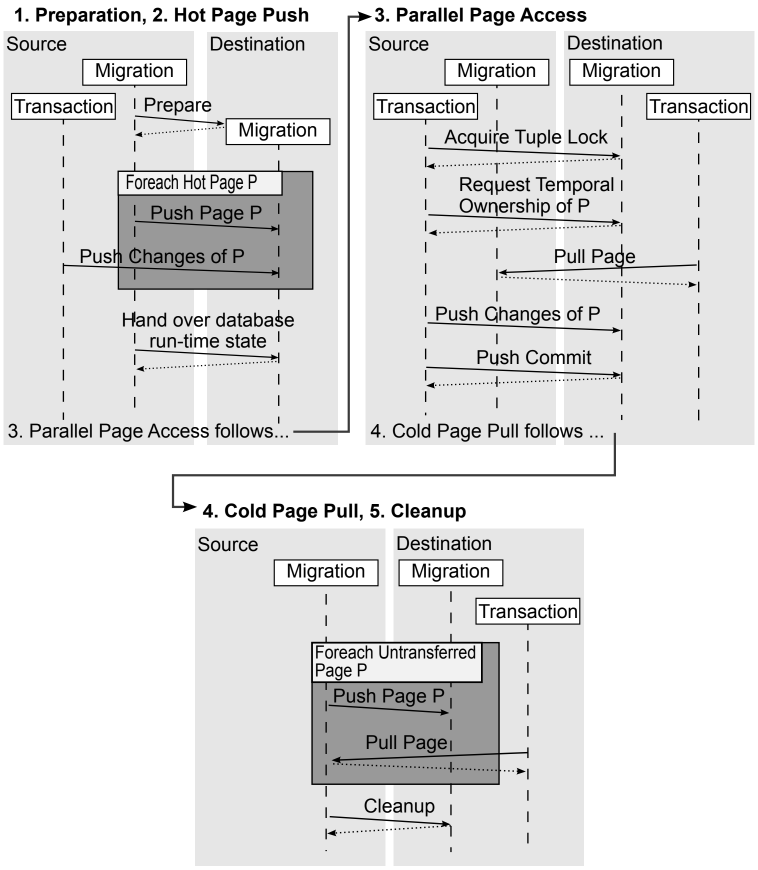


Figure 6.1: Sequential diagram for the algorithmic phases of ProRea.

Subsequently, we dive into the details of each phase using the notations listed in Tab. 6.1.

6.2.2.1 Preparation and Hot Page Push

To prepare migration, the source sets up local data structures and the migration infrastructure, e. g., processes and communication channels. Thereafter, it sends an initial message to the destination. This message includes the meta data of the database, e. g., table definitions, index definitions, user and role definitions and so forth. From this information, the destination creates an empty database and sets up its local migration infrastructure. Finally, it acknowledges preparation which triggers transitioning to Hot Page Push. Note that, as already mentioned, the preparation for the migration of a TenantSpace runs identically. Only the meta data which is sent in the preparation message may differ; the preparation message has to include all the tenant-specific meta data of tenants that are assigned to the TenantSpace.

Hot Page Push transfers each page $P \in \mathscr{D}_m \cap \mathscr{B}_s$, i. e., the hot pages, to the destination and changes $o(P)$ to *destination*, i. e., the destination obtains the ownership of P. The destination inserts the retrieved page in the buffer pool using standard buffer allocation methods.

If a transaction accesses an already transferred page P for modification, it requests *temporal ownership* for P. As the destination does not run transactions of $\mathscr{D}_m$ during Hot Page Push, the request simplifies to changing $o(P)$ to *temporal ownership* at the source. However, to maintain a consistent image of P at the destination, each modification operation of P creates a modification record which it sends to the destination. The modification record contains all information required to synchronize the image of P at the destination. In practice, page-level logical operations, i. e., physiological operations, lend themselves. If the RDBMS already creates such records for logging or replication, it simply has to capture records for P and ship them to the destination. The destination applies the retrieved record on P which ensures that the image of P at the destination is consistent to the image at the source. Note that the order-preserving property of the communication channel ensures that the modification records arrive at the destination in

$\mathscr{D}_m$	The database to be migrated.
$\mathscr{T}_s/\mathscr{T}_d$	Transactions at the source/destination.
$\mathscr{B}_s/\mathscr{B}_d$	Buffer pool of the source/destination.
$o(P)$	Ownership of page P.
$S(T)$	Start timestamp of transaction T.
$C(T)$	Commit timestamp of transaction T.

Table 6.1: Notations.

order of their creation. Thus the modification records at the destination are applied in the correct order.

The set of contained pages in $\mathscr{B}_s$ may change while the migration process scans $\mathscr{B}_s$ for hot pages; some pages in $\mathscr{B}_s$ give way to new pages. Therefore, $\mathscr{B}_d$ does not necessarily contain the actual set of hot pages of $\mathscr{D}_m$ after the described pass over $\mathscr{B}_s$. To obtain a higher similarity, further passes over $\mathscr{B}_s$ that transfer hot pages of $\mathscr{D}_m$ which are not transferred yet may be useful. Our current implementation simply scans the buffer pool two times. We have not considered more sophisticated termination conditions so far. Nevertheless, we assume that the change of the difference of transferred pages in two consecutive passes represents a good start. If the change converges to zero, there is no noteworthy progress anymore with respect to obtaining a higher similarity.

After the last pass, Hot Page Push hands over transaction processing to the destination. To ensure valid transaction processing, the destination requires the same *database run-time state* as the source. The database run-time state consist of the state information that is globally visible such as currently running transactions, the identifier of the next transaction, and the lock table. Hence, it includes everything such that newly incoming transactions for $\mathscr{D}_m$ run at the destination exactly as they would run at the source. The source adds the database run-time state to a handover message. To obtain a consistent state at the source and the destination, the source freezes the database run-time state while preparing the handover message. Consequently, the source prevents starting new transactions, modifying $\mathscr{D}_m$ and completing running transactions. The destination delays starting new transactions until it has received the handover message and has taken over the contained database run-time state. Directly after handover, the source and the destination run transactions on $\mathscr{D}_m$ concurrently, with which Parallel Page Access deals.

Existing and newly arriving clients have to be informed about the new location of the database. For this purpose, the migration task updates the cluster management instance with the new location of $\mathscr{D}_m$. In addition to that, the source notifies clients which want to access $\mathscr{D}_m$ about the new location of $database$. Thus, the client library is able to connect to the destination directly, without querying the cluster management. This approach enables updating the cluster management asynchronously.

6.2.2.2 Parallel Page Access

As transactions on $\mathscr{D}_m$ run concurrently at the source and the destination, Parallel Page Access requires synchronization between both of them in order to comply to the concurrency control protocol. Subsequently, we briefly describe the algorithm to achieve snapshot isolation, but we defer discussing its correctness to a separate section (see Sec. 6.2.3).

In order to complete successfully, transactions which already run at the source may require accessing pages that have been already transferred to the destination. The destination could have modified the page already, which is why the source does not necessarily have the current image of the page. To take into account this situation, the source accesses a tuple using the steps which algorithm 6.1 shows in pseudo code notation. The listing just shows the case of reading and updating a tuple. Note that the algorithm assumes that there is an appropriate latch on the page that prevents that other transactions modify the page or that the migration task transfers the page under concern.

The source primarily acquires an exclusive lock for the tuple which it wants to modify. If the source obtains the tuple lock locally, it also acquires the lock at the destination. Thus, the destination has the global view with respect to acquired tuple locks. Acquiring the lock may fail if the destination already locked the tuple under concern. Note that it is essential to acquire the lock at

Algorithm 6.1 Access of a tuple in page P at the source.

```
1: if $access = read$ then
2:     $read(P, tuple)$
3: else
4:     if $acquireTupleLock(tuple) = failed$ then        ▷ Request tuple lock.
5:         $abort$
6:     end if
7:     if $o(P) = source$ then
8:         $(P, Rec_{mod}, status) \leftarrow update(P, tuple)$        ▷ $Rec_{mod}$ is the modification record.
9:     else        ▷ Destination already owns page $P$.
10:        $res \leftarrow requestTemporalOwnership(P)$
11:        if $includesPage(res)$ then        ▷ Response may include current version of $P$.
12:            $P \leftarrow getPage(res)$
13:        end if
14:        $(P, Rec_{mod}, status) \leftarrow update(P, tuple)$
15:        $returnOwnership(P, Rec_{mod})$
16:        if $status = failed$ then
17:            $abort$
18:        end if
19:    end if
20: end if
```

the destination even if the source still owns the ownership. This is because the ownership of the page may change due to a request of the page from the destination, although the source holds the tuple lock. In this case, the destination has to know about the tuple lock to discover potential update conflicts of concurrently running transactions.

If the source has already transferred the page, it requests *temporal ownership* of the page from the destination. The destination tracks temporal ownership of the source and sends an appropriate response. The response includes the current image of the page if a transaction at the destination already modified the page. After updating the page, the source returns ownership to the destination and piggybacks the corresponding modification record. The source proactively returns ownership since the probability that several transactions of the source access the same page is low, in the case of short running transactions (more details follow in Sec. 6.3).

Note that the methods which actually read or update the tuple (line 2, 8 and 14) may require following the version chain of the tuple, which yields accesses of tuple version in other pages. These accesses run similar to the previous description.

Furthermore, if a transaction completes at the source, it sends a message that includes its completion status to the destination. Therefore, the destination knows about completion of all transactions, which it requires to release locks related to the transaction and ensure valid snapshots for new transactions.

Tuple access at the destination is simpler (see algorithm 6.2). Transactions at the destination primarily check the ownership of a page when accessing it. If it already owns the page, no additional efforts are necessary. If the source still owns the page, the destination pulls the page from the source. If the source temporarily obtained ownership of the page, the destination waits until the source returns the ownership.

6.2.2.3 Cold Page Pull and Cleanup

Cold Page Pull starts as soon as the last transaction at the source completes. Analogous to Parallel Page Access, the destination pulls pages which it re-

Algorithm 6.2 Access of a tuple in Page P at the destination.

```
1: if $o(P) = source$ then
2:     $pullFromSource(P)$                      ▷ Page has not been transferred yet.
3: end if
4: if $o(P) = temporalsource$ then
5:     $waitForOwnership(P)$                    ▷ Wait till source returns ownership.
6: end if
7: if $access = read$ then
8:     $read(P, tuple)$
9: else
10:    $acquireTupleLock(tuple)$
11:    $(P, Rec_{mod}, status) \leftarrow update(P, tuple)$
12:    if $status = failed$ then
13:        $abort$
14:    end if
15: end if
```

quires but which have not been transferred yet from the source. To finish migration, the source additionally pushes pages which have not been transferred during Hot Page Push or as response of a pull request from the destination, i. e., pages which we consider as cold.

Finally, after the destination owns all pages, migration cleans up used resources and completes by a handshake between the source and the destination.

6.2.3 Snapshot Isolation during Parallel Page Access

During Parallel Page Access, concurrency control has to span the source and the destination since both of them run transactions concurrently. For this reason, the concurrency control measures are different to the standard measures done without migration. Note that the standard measures are totally sufficient during Hot Page Push and Cold Page Pull.

In an RDBMS that promotes the ACID principle, concurrency control must work properly even during migration. To foster the trust in the correctness of ProRea, we subsequently present a semi-formal proof that Parallel Page

Access ensures snapshot isolation. For this purpose, we assume the following prerequisites: Snapshot isolation runs correctly without migration (Prereq. 1). The source and the destination access the same pages as they would without migration (Prereq. 2). The ownership concept works correctly (Prereq. 3). Recall that the ownership concept synchronizes page access. The ownership concept ensures that the node which owns the page has its current version and this node is the only one which is allowed to modify the page.

Lemma 6.2.1 *During Parallel Page Access, a transaction $T_s \in \mathcal{T}_s$ sees the latest committed tuple version that existed when T_s started.*

Proof: As the source does not start new transactions anymore, $C(T_d) > S(T_s), \forall T_d \in \mathcal{T}_d$ holds. Therefore, T_s never has to see tuple versions created by T_d. Hence, the latest committed tuple version which T_s has to see must exist at the source and is therefore visible to T_s. □

Lemma 6.2.2 *During Parallel Page Access, a transaction $T_d \in \mathcal{T}_d$ sees the latest committed version of a tuple t that existed when T_d started.*

Proof: We prove this by contradiction. Let us assume that there exists the latest committed tuple version v_{T_s} of transaction T_s, $C(T_s) < S(T_d)$, and v_{T_s} is not visible to T_d. Prereq. 1 and prereq. 3 directly yield that T_s had to run at the source, thus $T_s \in \mathcal{T}_s$.

If T_s modified pages P_1 and P_2 with $o(P_1) = source$ and $o(P_2) = source$ to write v_{T_s} into P_1, T_d pulls page P_1 (or P_2 in case of a forward version chain). Thus, contrary to the assumption, it has to see v_{T_s} (either directly or, in case of a forward chain, by following the version chain to P_1 which T_d also pulls in this case).

If T_s modified a page P_1 and P_2 with $o(P_1) = temporalsource$ and $o(P_2) = temporalsource$ to write v_{T_s} into P_1, T_s returns ownership to the destination with modification records that ensure that the images of P_1 and P_2 are consistent at the source and the destination. As the modification records are applied before the ownership transitions to the destination, T_d waits until its image of P_1 is consistent to the source. Thus, contrary to the assumption, it has to see v_{T_s} (either directly or by following the version to P_1 for whose consistency T_d again waits).

The remaining cases are analogous to the previous cases.

Thus, in all cases, T_d sees v_{T_s}. □

Lemma 6.2.3 *During Parallel Page Access, a transactions $T_s \in \mathscr{T}_s$ aborts if the version v_{T_s} of a tuple t which it wants to create conflicts with an already existing version v_{T_d} of t created by a concurrent transaction $T_d \in \mathscr{T}_d$.*

Proof: T_d acquires an exclusive tuple lock for t in order to write v_{T_d} into page P. If T_s is unable to obtain the lock at the destination since T_d still holds the lock, T_s aborts. If T_s acquires the lock (which implies that T_d has committed or aborted), it eventually gets the current image of P that contains v_{T_d} in the response to the ownership request (analogous to Lemma 2). Hence, T_s sees v_{T_d} and, thus, aborts. □

Lemma 6.2.4 *During Parallel Page Access, a transaction $T_d \in \mathscr{T}_d$ aborts if the version v_{T_d} of a tuple t which it wants to create conflicts with an already existing version v_{T_s} of t created by $T_s \in \mathscr{T}_s$ with $C(T_s) > S(T_d)$.*

Proof: If T_s still runs at the source, it still holds an exclusive lock for t at the destination in order to write v_{T_s}. The exclusive lock semantics ensure that T_d is unable to get the lock for t. Thus T_d has to recognize the conflict and wait for completion of T_s. As T_s notifies the destination about its completion, strict 2-phase locking ensures that T_d gets the lock for t not until it is able to inspect the state of T_s. If T_s has committed, T_d aborts. Note that this also applies if T_s has already completed at the time T_d acquires the tuple lock. □

Theorem 6.2.5 *ProRea ensures snapshot isolation during Parallel Page Access.*

Proof: From Lemma 6.2.1 and Lemma 6.2.2 together with Prereq. 1 follows that a transaction $T \in \mathscr{T}_s \cup \mathscr{T}_d$ sees the latest committed tuple versions that existed when T started. Thus, rule 1 of snapshot isolation holds during Parallel Page Access.

Prereq. 1 implies that local conflicts, i. e., conflicts between transactions at the same node, are recognized. Lemma 6.2.3 ensures that a transaction

$T_s \in \mathscr{T}_s$ aborts, if there exists a potential conflict with a transaction $T_d \in \mathscr{T}_d$. Lemma 6.2.4 ensures that a transaction $T_d \in \mathscr{T}_d$ aborts if it conflicts with a transaction $T_s \in \mathscr{T}_s$ and $C(T_s) > S(T_d)$. Hence, a transaction will successfully commit only if its updates do not conflict with any updates performed by transactions that committed since the transaction under concern started. As consequence, rule 2 of snapshot isolation holds during Parallel Page Access.

Thus, Parallel Page Access meets rule 1 and rule 2 of snapshot isolation. □

Note that Lemma 6.2.3 is more restrictive than required to accomplish snapshot isolation. If transaction T_s observes a conflict with a still running transaction T_d, it aborts even if T_d eventually fails (which T_s does not know). Thus, transactions at the source do not wait for tuple locks or check completion status of other transactions at the destination during Parallel Page Access. From the perspective of T_s, the abort constitutes a pessimistic decision since it aborts although it has the chance to commit successfully (in most cases a chance with quite low probability). We argue that this design decision simplifies ProRea considerably. This design decision prevents synchronizing lock releases from the destination back to the source. In addition to that, it prevents synchronizing completion of transactions from the destination to the source. Finally, it prevents more complicated deadlock detection and management since a deadlock cycle can never span the source and the destination.

6.3 Analysis

This section analyzes run-time costs and provides design rationales for ProRea.

6.3.1 Hot Page Push

Hot Page Push requires a shared latch on a page while transferring it, which blocks writers of the page and thus ensures a physically consistent copy of the page. Yet, the shared latch is very short if pages are transferred asynchronously. The same holds for creating and shipping the modification records. Note that modification records are often created anyway for logging or replication. Therefore, we regard the overhead at the source as negligible.

This is different at the destination since it inserts received pages and applies received modification records. If the pages under concern are still in $\mathscr{B}_d$, the load caused by applying the modification records tends to be lower than transaction load at the source, with which the destination should be able to deal anyway. In addition to that, the destination has to insert pages into $\mathscr{B}_d$ which is assumed uncritical if the destination has enough available buffer space. If the destination has to free buffer space or has to write out received pages, Hot Page Push may cause severe load at the destination. We regard this an important issue in down-scaling scenarios that requires further evaluations.

6.3.2 Handover

Handover transaction processing from the source to the destination represents a critical part since it blocks most operations on $\mathscr{D}_m$. The length of the blocking period mainly depends on the transfer time of the handover message which in turn depends on its size and the network latency. Recall that the handover message transfers the database run-time state. During our tests, the main part of the database run-time state consisted of the lock table and the ownership map. Thus, we limit ourselves considering the size of the lock table and the ownership map. A lock table with 500 lock identifiers each having 64 bytes requires 32 Kb. A lock table with 500 entries approximates to 50 concurrent transactions each holding 10 locks. This size is reasonable (perhaps slightly overestimated), as we assume simple, short-running transactions, which only update few tuples. The ownership map requires two bits for the state of each page. Thus, for a page size of 8 Kb and a database size of 5 GB, the total size of the ownership map approximately amounts to 160 Kb. Under these prerequisites, the total size of the handover message is about 200 Kb which requires less than 3 ms transfer time in a latency-free 1 Gbit/s ethernet network at half speed. In practice, network latency adds to the transfer time. Yet, even if latency adds additional 10 ms, the transfer of the handover message is short. Thus, the blocking period during handover is short and thus has a low impact.

6.3.3 Parallel Page Access

Parallel Page Access entails synchronization between the source and the destination. The synchronization principles from the source to the destination and the inverse way differ; pages are synchronized eagerly from the source to the destination, whereas the inverse way is done lazily. The lazy synchronization is justified as eager synchronization of *all* modifications from the destination to the source is obviously unnecessary. The inverse way behaves differently because the destination requires the current image of the page for future transactions. Thus, eager synchronization of pages from the source to the destination is useful.

The lazy synchronization from the destination to the source causes that the destination returns the complete page image of a modified page if the source asks for its ownership. This is because, at the time the source requires the page, the destination need not have the previous page image anymore. Therefore, it is unable to create the corresponding modification records. It may indeed search the log (if existent) for the corresponding records, but this approach is more complicated than simply returning the page and may even require additional disk I/O.

The source directly returns the ownership of the page after its modification. It might be objected that the source may keep the ownership such that other transactions at the source may also modify the page without requesting the ownership from the destination again. However, Parallel Page Access is short and transactions only update few tuples. The probability that multiple transactions at the source require writing the same page is in general low and will further decrease in the future. Hence, page ownership would eventually return to the destination without additional accesses at the source in most cases. For this reason, it is useful to take the chance to create modification records, send them directly to the destination and piggyback return of ownership.

The total synchronization overhead during migration highly depends on the write share, page contention and network latency. A high write share and high page contention increase the probability that a transaction at the source and a transaction at the destination desire to write the same page. For example, if all transactions insert tuples in ascending order, many transactions want to write to the same page. In this case, the source and the destination require transferring the page back and forth. However, Parallel Page Access is short, as transactions are short. Therefore, the amount of pages that is transferred back and forth should be small.

6.3.4 Cold Page Pull

During Cold Page Pull, two operations affect overall performance: page push from the source to the destination and page pull by the destination.

The former operation requires reading a huge part of the database at the source and writing it at the destination. Although the operations cause mainly sequential disk I/O, their impact is usually too high to run them full speed ahead. Therefore, limiting the throughput of these operations is essential to limit their impact on overall performance. Sean Barker et al. [2012] shows how the overall performance depends on the throughput of reading and writing the database. Based on these results, they propose a control-theoretic approach to limit the performance impact due to migration. Their results also hold for pushing the pages during Cold Page Pull.

If a transaction at the destination pulls a page from the source, the access time of the page includes network transfer time and network latency. Let us assume that the network overhead adds 1 ms. Compared to a disk access which may easily require 10 ms on a moderately utilized machine, the network overhead is low. However, compared to a buffer pool access which may take about 0.2-0.5 ms, the network overhead is high. As ProRea transfers the hot pages during Hot Page Push, a page fault typically entails a disk access at the source why we consider the relative degradation by the network overhead as tolerable.

6.4 Indexes and Buffer Management

6.4.1 Index Migration

Note that our previous description of ProRea is not limited to a certain storage structure. For this reason, ProRea works for an index exactly as for the primary storage if the index uses MVCC, as described in 6.2.1, in order to serialize access to its tuples.

Alternatively, an index may index all visible versions of a tuple in the primary storage. An index access then implies an access of the looked up tuple in the primary storage to determine its visibility. In this case, locks for index tuples are unnecessary and only latches are required to ensure the physical integrity of index pages. PostgreSQL being an example of such an implementation. As the ownership concept used in ProRea ensures the physical integrity of pages, ProRea is usable for this kind of index implementation as well.

6.4.2 Improved Buffer Handling

Common buffer pool replacement strategies require to know the access history of a page, e. g., usage counters, to make the right decisions which buffer pages to replace. To safe this information, the migration task also transfers the related meta data of a page. Naturally, all phases of ProRea maintain the meta data suitably. For example, if the source requests temporal ownership for a page, the destination also increments the usage counter of the page when granting ownership.

In a Shared Process multi-tenancy model, multiple databases, i. e., multiple tenants, share available buffer pool space. Either all databases share the whole buffer pool space equally or each database obtains a reserved amount of buffer pool space dedicated to it. In both cases, if a database frees buffer pool space, other databases will be able to allocate more space. For example, if ten databases equally share 4 GB of buffer pool space and one is shutdown, the remaining nine database may allocate about 11 % more buffer pool space.

For this reason, ProRea frees $\mathscr{B}_s \cap \mathscr{D}_m$ directly after Parallel Page Access, i. e., after all transactions at the source have completed.

To free the buffer pool space, dirty pages require being written out, which causes considerable random disk I/O. However, if a page has been already transferred to the destination and the source does not change the page anymore, it is safe to skip the page immediately from the buffer pool without writing it out. From a conceptual view, the transfer of the page replaces writing out the page to disk. Note that this does not violate durability; the recovery protocol ensures durability.

6.5 Implementation

We built a prototypical implementation of ProRea integrated into PostgreSQL 8.4 [PostgreSQL Global Development Group, 2012]. This section outlines the main implementation challenges and decisions. Although the implementation is specific to PostgreSQL, the main implementation concepts and solutions which we subsequently discuss are transferrable to other implementations as well.

6.5.1 Process Model and Communication

PostgreSQL uses a process-based approach to enable parallel processing. There exists one parent server process, called *Postmaster*, which forks new processes for different tasks, e. g., query processing, buffer pool cleaning and online reorganization. For query processing, the Postmaster waits for incoming connections and forks for each incoming connection a new process, called *Backend*. The client of the incoming connection then communicates with the newly forked Backend. The Backend accepts incoming commands from a client and carries out the related action. For example, a common command is to execute an SQL query.

The migration of a database using ProRea requires forking multiple new processes. Figure 6.2 shows a process instance model during migration.

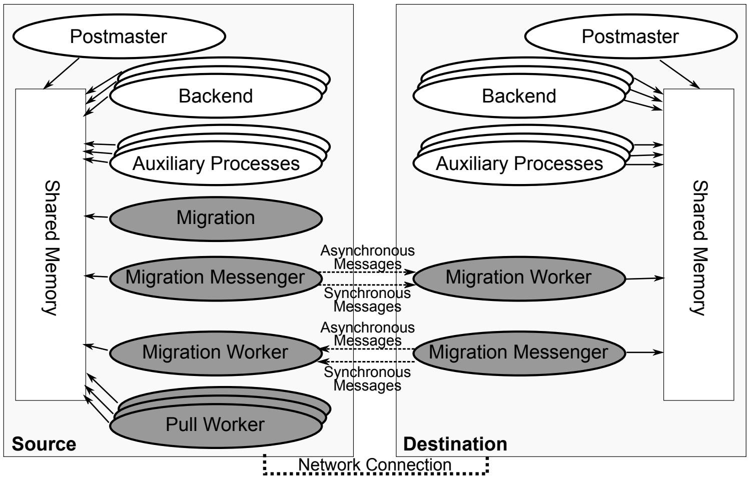


Figure 6.2: Process instance model during the migration of a database using ProRea (the grey-shaded, rounded boxes represent processes which relate directly to migration).

The Backend which is called *Migration* runs and coordinates the migration of a database. It is forked through an incoming connection by a client. The client then starts the migration utility function with needed arguments such as the destination node. The migration utility function then runs through the phases of ProRea, i. e., it mainly pushes pages from the source to destination.

The *Migration Worker* is a Backend that runs at the source and the destination respectively. It is forked during the preparation of migration. The Migration Worker processes commands which the respective opposite node sends during migration. For this purpose, the Migration Worker accepts commands that relate to migration exclusively. PostgreSQL uses a command identifier character (1 byte) to identify the command. Accordingly, a free character (M) introduces a new class of commands that relate to migration exclusively. The remaining payload includes further details about the command with respect to migration, e. g., the migration command type or the contents of a page.

In order to communicate with the Migration Worker at the opposite node, i. e., sending commands to the Migration Worker, each node implements a centralized communication mechanism using an additional process that is called *Migration Messenger*. The Migration Messenger corresponds to the Migration Worker at the respective opposite node: the Migration Messenger at the source and Migration Worker at the destination produce a communication channel from the source to the destination, and the Migration Messenger at the destination and the Migration Worker at the source implement the reverse way (illustrated in Fig. 6.2).

Other processes communicate with the Migration Messenger via shared memory. For this purpose, the Migration Messenger provides a buffer into which a process may to put a command (protected by a semaphore). The Migration Messenger periodically flushes this buffer, i. e., sends its contents to the corresponding Migration Worker. The provided communication channel is asynchronous. The asynchronous communication works well for pushing pages and modification records. However, ownership requests or pulling pages requires synchronous communication, i. e., the sender has to wait until the response arrives and, thus, the message has to be transferred and processed as fast as possible. For this purpose, the Migration Messenger also provides a synchronous communication channel. The synchronous communication channel uses the same buffer as the asynchronous communication channel. However, if a senders puts a message into the buffer that requires synchronous communication, the buffer is directly flushed. Furthermore, the sender starts sleeping. The receiving Migration Worker searches the received contents for synchronous messages and processes them directly. It returns the response by using the asynchronous communication channel followed by a flush. The opposite Migration Worker in turn searches the contents for a synchronous response. If it finds a synchronous response, it wakes up the corresponding process.

The Migration Worker obviously represents a bottleneck since it serially processes all migration commands. Therefore, it is feasible to hand over certain processing tasks to other processes in order to accomplish parallel processing. This approach is particularly useful for commands that tend to require waiting for I/O operations, e. g., transactions at the destination that require pulling pages. For this reason, the source runs a pool of so-called *Pull Workers*. Instead of processing a pull request by itself, the Migration Worker only serves as multiplexer for pull request, i. e., it passes received pull requests to an available Pull Worker which actually processes the request.

Note that the described functional decomposition and distribution of responsibilities is also usable in other RDBMS implementation that base upon threads instead of processes.

6.5.2 Ownership Map and Synchronization

Each segment of $\mathscr{D}_m$ obtains a separate ownership map. The ownership map is a simple bitmap that uses two bits for each page in order to keep track of its state. The page number determines the location of the state bits. The states are actually: `Source`, `Destination`, `Source_Temporal`, `Destination_Dirty`. Except for `Destination_Dirty`, the states are self-explanatory from the previous description of ProRea and actually relate to the ownership concept. `Destination_Dirty` is slightly different; in addition to the ownership of the page, it indicates that the destination has modified the page and the source does not have the newest version. This state is required to determine whether the response to a temporal ownership request of the source has to include that page.

The bitmap is divided into pages of fixed size and uses the standard buffer pool interface. As the pages of the bitmap are frequently accessed they usually stay in the bufferpool. Nevertheless, it is recommended to pin the pages in the bufferpool to avoid unnecessary overheads while seeing into the state of a page. Note that the bitmap needs to be WAL-logged to deal with system failures appropriately (see Sec. 6.7 for a more detailed discussion about system failures during migration).

To synchronize access to the state of page, we use the latches that are used to synchronize access to the buffer which contains the page. This approach is no restriction so far; changing or inspecting the state of a page in most cases yields accessing it anyway.

6.5.3 Transaction and Migration Processing

PostgreSQL's access to a page in a heap or an index was adapted to account for the ownership of a page. The major modifications are included while reading a page into the buffer and directly after acquiring a latch for a buffer. The former case only concerns the destination. If the page has not been transferred, the destination simply demands the page from the source using the synchronous communication channel previously explained. This is actually comparable with reading a page from any storage medium and thus integrates seamlessly into the existing implementation. The latter case, i. e., after acquiring a latch for a buffer, is definitely more difficult to keep in view. It required identifying all locations within PostgreSQL's source code that acquires latches for accessing the heap or B-tree during query processing. After acquiring the latch, the code has to check the state of the page and continue according to the current state as described for ProRea.

The modification records used to synchronize already transferred pages are the records which are created for the WAL. If necessary, i. e., if the page under concern is already transferred, we simply copy the related, created modification records and transfer them using the asynchronous communication channel provided by the Migration Messenger. Naturally, creation and transfer of the modification records require holding an exclusive latch on the page. The order-preserving property of the communication channel ensures that the records arrive in the same order as they have been created. At the destination, the standard mechanism provided to apply a WAL record is used for the received record. As PostgreSQL uses almost the same methods for applying WAL records as for transaction processing (including concurrency control), this approach works fine for ProRea. Note that other systems may use special methods for recovery that are not allowed to run in parallel with query processing which prevents using the standard recovery methods as described.

The created modification records sometimes cover more than a single page. For instance, a record for the split of a B-tree page references two pages. Thus, to successfully apply a record, the destination may require to fetch a page if not all of the pages referenced by the record have been transferred already. To avoid the overhead of fetching the pages, our implementation attaches the pages which are referenced by the modification record but have not been transferred yet to the contents of the modification record message.

PostgreSQL uses tuple locks which are listed directly at the tuple. Thus, a transaction checks the current lock state of a tuple by accessing its tuple header. If a transaction thereby determines that it has to wait for the release of a tuple lock, it actually waits for the completion of the transaction that holds the tuple lock which the tuple header also references. Note that other MVCC implementations use similar concepts, e. g., Oracle's database software and MySQL, since this approach avoids holding large numbers of tuple locks in main memory. As a consequence of this approach, we are able to omit synchronizing tuple locks explicitly between the source and the destination. The synchronization methods by means of transferring pages and modification records automatically cater for passing the tuple locks back and forth between the source and the destination. In other words, if a node has the current version of a page it also knows the current state of tuple locks with respect to the page. Our prototypical implementation does not support the case that multiple transactions hold a shared lock on a tuple, since this case requires additional measures. However, this case never happens during snapshot isolation; it is a specially supported feature, e. g., if the user explicitly locks tuples. To support this case, explicit synchronization for tuple locks from the source to the destination, as described, is required.

During Cold Page Push, reading the pages at the source and writing them at the destination may thrash the buffer pool. To avoid thrashing the buffer pool, the source and the destination uses a ring buffer that overlays the buffer pool. A feature we have not implemented, but which is definitely worth to consider, is writing a consecutive set of pages at once and filling the gaps between pages transferred during Cold Page Pull by the pages transferred during Hot Page Push or pulled on demand. By filling a gap, a fully sequential write of multiple pages is feasible. This approach reduces the number of dirty pages in the buffer pool by negligible additional costs.

6.6 Experimental Evaluation

We implemented a simple testbed to evaluate the run-time characteristics of our prototypical implementation. For comparison with a purely reactive approach, we used an implementation without Hot Page Push and optimized buffer pool handling, which we refer to as *Pure Rea*.

6.6.1 Test Environment

6.6.1.1 Testbed Implementation

The schema of our testbed consists of one table and one index: the *Customer* table of the TPC-C benchmark and an index for the primary key. Based on this schema, we generated three databases, each having a total size of about 5.1 GB.

The load simulator of the testbed runs a mix of simple select and update transactions, each accessing a single tuple by its primary key. The load simulator takes parameters for the access pattern, the read/write ratio, and the number of transactions to be issued per second (tps). The access pattern parameter specifies the percentage of transactions that access a certain percentage of the data, e. g., 80 % of transactions access 20 % of the data and 20 % of transactions access 80 % of the data. Within the resulting data ranges, transactions access data uniformly random. We refer to the respective access pattern by *transaction percentage/data percentage*, e. g., 80/20.

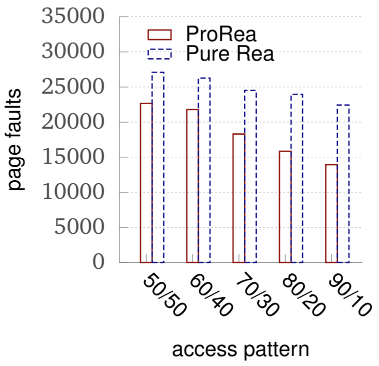
The load simulator begins each run with a ramp up phase that starts a new worker thread every 30 seconds till it reaches 25 threads. Each thread connects to all databases. The total warm up period before migration starts is 70 minutes. After this period latencies and throughput turned out to be quite stable for our test cases. The load simulator distributes the transactions uniformly random across the configured databases and available worker threads. Hence, all databases have to serve the same share of load. At the beginning, the source serves all three databases. After 70 minutes, one database is migrated from the source to the destination. Thus, after migration, the source serves two databases and the destination serves one database.

To limit the impact of pushing the pages during Cold Page Pull, we throttled transfer throughput to 4 Mb/s. Moreover, we ran three passes for each test, whereas each pass started on freshly booted machines. As the results of each pass were similar and each pass would lead to the same conclusions, we only report the results of the first pass.

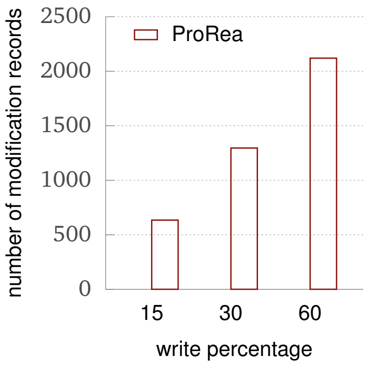
6.6.1.2 Test Systems

For our tests, we used two Dell Optiplex 755, one for the source and the other for the destination. The machines were equipped with an Intel Core2 Quad Q9300 CPU running at 2.50 GHz and 4 GB of main memory. We stored the database and its log on two striped 250 GB SATA 3.0 GB/s hard drives spinning at 7.200 RPM. The test machine ran a 64 bit 2.6.32 Linux kernel (Ubuntu release 10.04 Server), configured without swap space. The client machine on which we ran our test tools was equipped with four Dual Core AMD Opteron 875 CPUs running at 2.2 GHz and 32 GB of main memory. The operating system was a 64 bit 2.6.18 Linux Kernel (CentOS release 5.8). All machines were connected over a 1 GBit/s ethernet network.

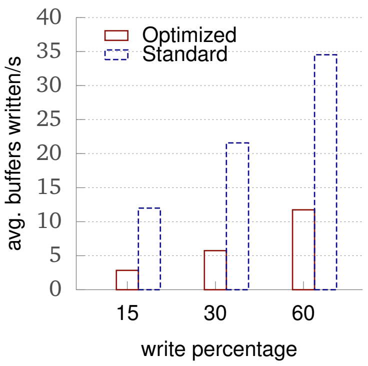
PostgreSQL's buffer pool was configured to 1.5 GB at the source and the destination. Autovacuuming, autoanalyze and checkpoints have been disabled.



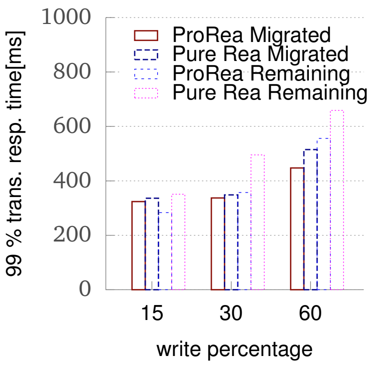
(a) Total page faults.



(b) Modifications sent.



(c) Avg. buffers written per second.



(d) 99 % transaction response times.

Figure 6.3: Fig. 6.3a compares the page faults of ProRea with a purely reactive approach, called Pure Rea. The different data access patterns entail increasing buffer pool hit ratio from left to right (80/20 means that 80 % of transactions access 20 % of the data). Fig. 6.3b shows the number of modification records which ProRea sends dependent on the write percentage. Fig.6.3c compares the standard buffer handling strategy of PostgreSQL with our optimized strategy (Sec. 6.4.2) during migration. Fig. 6.3d shows the 99 % quantile of transaction response times during migration. Fig. 6.3b, 6.3c and 6.3d show the access pattern 80/20.

6.6.2 Measurements

To estimate the page fault reduction of ProRea with respect to different buffer pool hit ratios, we started with tests that count the page faults for different access patterns during migration. The results presented in Fig. 6.3a show that the number of page faults decreases with higher buffer pool hit ratio (from left to right) for ProRea and Pure Rea. This is because higher buffer pool hit ratios yield lower numbers of different pages that are accessed during the period of migration. Obviously, the number of page faults caused by ProRea reduces more relative to Pure Rea. ProRea produces about 16 % less page faults for 50/50 and about 38 % less page faults for 90/10. Thus, regarding page fault reduction, ProRea has it strengths for workloads with good buffer pool hit ratio.

ProRea sends modification records during Hot Page Push to synchronize already transferred pages. During our test, the number of modification records scaled almost linearly with the write percentage, as shown in Fig. 6.3b. Compared to the number of reduced page faults, the number of modification records was considerably lower (about 8200 less page faults relative to Pure Rea and about 1300 additional modification records for a write percentage of 30 % and access pattern 80/20). Hence, ProRea reduces the total number of messages sent relative to Pure Rea. Despite of the message reduction, ProRea often increases the total amount of data sent across the network. This is because modification records are typically considerably larger than page pull requests. Our current implementation sends the whole modified tuple. Therefore, the additional amount of data depends on the average tuple size, which was about 320 bytes. For example, the test for 30 % write percentage transferred approximately 420 Kb additionally, which we regard negligible compared to the total database size (5.1 GB). Moreover, the transfer of modification records is less time critical than processing page faults, as it typically does not lead to wait periods for transactions. Recall that the modification records are transferred asynchronously.

To estimate the effectiveness of the optimized buffer pool handling (see Sec. 6.4.2), we measured the average number of buffers written per second at the source during migration. Fig. 6.3c shows the results for the standard buffer handling strategy of PostgreSQL and for our optimized buffer handling strategy. The results evidence that the optimized buffer handling strategy reduces the number of buffers written at the source. Naturally, the optimized buffer handling strategy mainly takes effect for higher write percentages. Moreover, its effect is limited to the allocated space of the database that is migrated. Anyway, it is a simple but effective optimization. For example, in our tests, it reduces the average number of buffers written per second from 22 to 6 for a write percentage of 30 %. Consequently, the average number of disk I/O operations at the source reduced by more than 10 %.

The reduction of disk I/O operations is in line with the 99 % quantiles of transaction response times during migration, as Fig. 6.3d depicts for ProRea and Pure Rea. The transaction response time is the end-to-end execution time as difference from the time of issuing the query to the time of retrieving the results. The difference between the 99 % transaction response times for the remaining databases of ProRea and Pure Rea is obvious and increases with the write percentage. With respect to the 99 % transaction response times of the migrated database, ProRea comes with a small advantage compared to Pure Rea, about 4 % to 9 % shorter 99 % transaction response times, although Pure Rea has to cope with more page faults. This is because the higher number of page faults did not cause a severe penalty in our tests since the network connection between the nodes in our test environment was very fast (< 0.1 ms latency).

Finally, we ran a test which mimics a typical outscaling scenario. In this scenario, the transaction throughput increases from 100 to 150 transactions per second within a time window of 10 minutes. Note that the load is still equally distributed across the three databases ie the number of issued transactions per second with which the source has to deal remains the same, before and after migration. The load was generated according to the access pattern 80/20. The write percentage was configured to 30 %. Three minutes after starting with increasing load, migration starts. Fig. 6.4 depicts the average transaction response time using a moving average over one minute. About 2 minutes after start of migration, the migration task transitions to Cold Page Pull. As a result, the response times reduce significantly. This is because of the buffer handling that releases load at the source. As the load further increases, the response times increase as well, until the load reaches

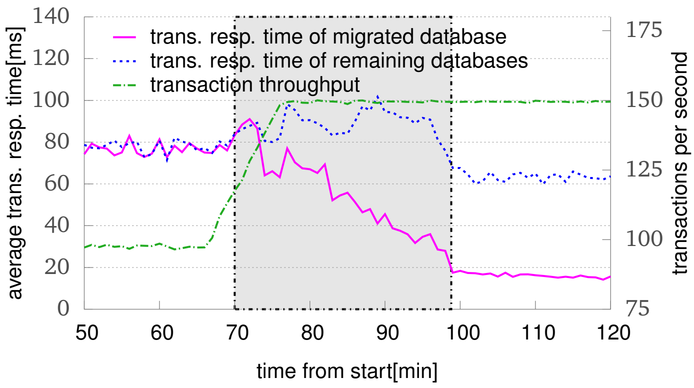


Figure 6.4: Outscaling scenario that increases transaction throughput from 100 transactions per second to 150 transactions per second. Each data point is calculated as the moving average over the past minute. The grey-shaded box represents the migration window.

its maximum. Thereafter, the graph shows that the response times of the migrated database decrease with ongoing migration. This is because more and more data is at the destination which is why the number of page faults reduces and, thus, the destination gets more and more independent from the source. The response times of the remaining databases degrade between 5 and 25 % compared to the response times before migration, although load at the source does not increase. This degradation results from reading pages to push them to the destination. After migration, the response times improve due to the lower total data volume with which the source has to deal. In another test, we ran the same scenario but limited transfer throughput during Cold Page Pull to 2 Mb/s. In this case, the average response times do not deteriorate more than 10 %, but the migration time almost doubles, i. e., the total time required for migration is about 54 minutes instead of 28 minutes. Optimally, a dynamic throughput adaption, as proposed by Sean Barker et al. [2012], should be used to limit the impact on transaction processing but carry out migration as fast as possible.

6.7 System Failures

Recovery from system failures represents a core functionality in a RDBMS implementation to guarantee the ACID principle. Database migration using ProRea faces recovery with a new challenge: the consistent state of the database can be distributed over the source and the destination in the event of a system failure. This section provides a conceptual discussion about dealing with system failures that occur during database migration with ProRea. This discussion focuses on showing that recovery from system failures is feasible without degrading migration performance significantly.

For this discussion, we assume a log-based recovery approach that uses typical log replay techniques to restore the latest transaction consistent database state before the system failure. Log-based recovery is undoubtedly the most common recovery approach in current RDBMS implementations.

6.7.1 General Reflections

To ensure common consensus about the current migration phase even in the event of a system failure, the two-way handshake protocol presented by Elmore et al. [2011c] is reasonable for ProRea as well. It asserts permanence of the current phase by logging the sending and receiving of messages related to phase transition.

The handover of the database run-time state does not require to be synchronized with the update of the database location at the cluster management (different to Elmore et al. [2011c]). This is because the source notifies clients about the new location of the database. Thus, the source may update the cluster management instance asynchronously using the same two-way handshake protocol as used for phase transition. Migration certainly can only finish after this update.

In the event of a system failure during Prepare or Hot Page Push, the abort of the migration represents the most obvious choice. The work done so far is low and the abort of the migration only requires cleaning up few data structures, e. g., meta data entries. A significant advantage of aborting is flexibility because the source still owns solely the consistent database image.

Moreover, the source and the destination can decide to abort the migration alone due to the common consensus about the migration phase. Hence, if the destination fails, the system can decide to start another migration from the source to an alternative destination without waiting for the failed destination. In Parallel Page Access and later phases, the consistent database image spans the source and the destination, thereby both of them have to be alive for successful recovery. This case offers the continuation of the migration which requires restoring its state. The subsequent sections discuss how to enable restoring the state of migration.

6.7.2 Logging

Just as the database state, the database log gets distributed over the source and the destination after handover. The source logs only updates of page P if $o(P) = source$, i. e., if it actually owns P. The destination logs updates of P if $o(P) = destination$ or $o(P) = temporalsource$. The latter case is accomplished because the destination logs applying modification records retrieved from the source after handover. The source and the destination independently maintain the latest log sequence number (LSN) (see Sec. 6.7.3) which a transaction $t_s \in \mathscr{T}_s$ caused at the source and the destination respectively. Before t_s is allowed to commit, the source and the destination flush their respective local log up to the respective latest LSN of t_s.

The described logging approach ensures that all log entries belonging to transaction t_s are flushed, partially at the source and partially at the destination. This is sufficient for a system which runs a selective redo approach; PostgreSQL being an example of such a system.

An Aries-style [Mohan et al., 1992] recovery approach however mandates a complete redo that repeats all changes of a page which are in the log in chronological order (referred to as repeating history), independent of the completion status of the related transaction. Therefore, the previously sketched logging approach additionally requires flushing log entries related to a page before its transfer to the destination. During Hot Page Push, an explicit flush of log entries is actually not required because all transactions still run at the source and the handover of the database run-time state flushes the log due to the phase transition anyway. Hence, only Parallel Page Access is affected. As Parallel Page Access is short, we consider the resulting additional overhead as negligible.

6.7.3 Log Sequence Numbers

The LSNs increase independently at the source and the destination, why they only provide local chronological ordering of log entries. Yet, ProRea implies that all log entries related to a page P at the source are chronologically before all log entries related to P at the destination. This is because, after hand over of the database run-time state, the first transfer of a page P to the destination irreversibly transfers the responsibility for logging modifications of P to the destination. Hence, with respect to P, arranging the log entries from the source before the log entries from the destination creates a partial order that relates entries of P in chronological order, which suffices for page-oriented redo and page-oriented undo. A special marker LSN allows indicating a reference to the latest log entry at the source. After the receipt of a page, the destination resets the LSN recorded in the page to this special marker LSN. During recovery, a special marker LSN in a page indicates that the destination has to apply all existing log records for the page.

Recall that a transaction $t_s \in \mathcal{T}_s$ creates log entries at the source and at the destination during Parallel Page Access. For transaction-oriented logical undo, the chronological order of log entries caused by t_s has to be preserved. The previously induced partial order does not preserve this order. For this reason, additional measures are necessary to enable transaction-oriented logical undo. For instance, overlaying a logical order over the log entries caused by t_s is one approach. The identifiers which establish the order are attached to all log entries caused by t_s (at the source and the destination). For this purpose, t_s maintains the identifiers, e. g., by a counter that is incremented for each log entry, and standard messages of the ProRea protocol piggy back the identifiers for related log entries to the destination. The source and the destination create independent backward chains for t_s (each log entry has a reference to the chronologically previous log entry of t_s). To establish the chronological order of log entries caused by t_s, the two chains of the source and the destination are merged by taking into account the created identifiers, i. e., by sorting the log entries using the identifiers. Alternatively, logical LSNs (at least partially) allow relating a LSN to the node which produced it, e. g., as done in Speer and Kirchberg [2005] for similar purposes. The relation of a LSN to the node which produced it allows building a chronologically ordered chain per transaction. Nevertheless, the source and the destination have to know the respective last log entry before they are allowed to create a new log entry for t_s to retain a chronologically ordered backward chain of log entries caused by t_s. In order that the source and the destination know the last log entry, the log entries require being shipped and applied synchronously from the source to the destination.

6.7.4 Durability of Migration State

The goal is to preserve most work of the migration in the event of a system failure, but without severe impact on its run-time performance. The two obvious, extreme approaches are: (I) discard all work done and start the migration from scratch or (II) flush each retrieved page to disk and log its receipt. The first approach has no run-time efforts, but it loses most work done. The second causes a considerable amount of disk I/O requests, which yields a performance loss during run-time. These approaches are naturally far from the desired goal.

Therefore, we propose an approach in which the buffer pool manager at the destination creates a log entry for the successful completion of writing out a page. The corresponding log entry is not forced to stable storage immediately, but it will be flushed to disk prior to or with the subsequent commit entry. Logging the successful write of a page is sometimes anyway done to minimize redo efforts. In our case, it allows determining which pages have been transferred and have been successfully written to stable storage. Hence, only pages that have been transferred but not written to stable storage require being retransferred during recovery, which represents a tolerable loss of work.

After all pages have been transferred to the destination, the destination runs a fuzzy checkpoint [Gray and Reuter, 1993], which eventually creates a page consistent database image at the destination. Thereafter, the source is allowed to purge the database.

6.7.5 Recovery

During analyze of the log, the destination creates an ownership map from the log entries the buffer pool manager has written. The destination transfers this ownership map to the source. Thereafter, the source starts redo for the pages it still owns. The destination does the same for the page it owns. If the destination finds a log entry related to a page it does not own, it has to fetch the page from the source. Naturally, prior to this, the source has to apply all redo log entries for this page. Thereafter, the undo pass starts. Each node undoes transactions it has started. If the failure occurred during Parallel Page Access, this may require to ship undo logs or pages from one node to the other and vice versa.

If only one node fails, only the failed node has to recover. However, in any case, the ownership map has to be transferred from the destination to the source to ensure common consensus about the ownership of a page.

6.7.6 Conclusions

Although the previous discussion only briefly considers recovery in the event of system failure, it shows that recovery is feasible without significant runtime overheads. Parallel Page Access represents the most critical phase; it may require additional measures to ensure appropriate ordering of log entries. However, these measures are limited to Parallel Page Access, which is assumed to be short anyway.

Log-based recovery is a discussable point by itself in highly available system environments. In such environments, replicas of a database lend themselves for recovery. Lau and Madden [2006] has shown that failover and rebuild is efficiently feasible. Furthermore, highly available system environments offer an interesting opportunity. If the source fails, the migration may continue from a replica. The destination simply has to ship the ownership map to the replica.

Even if recovery does not run log-based, the database log might be useful for other features, at least partially. For example, Talius et al. [2012] adopts the undo log to implement point in time recovery without the need of restoring a whole database image. Such functionality is quite appealing in system environments that allow for rare database backups, as highly available systems usually do. Hence, the previously discussed points hold for a broader range than only recovery.

6.8 Summary and Outlook

In our presented architecture of a multi-tenant RDBMS for OLTP-style SaaS workloads, live TenantSpace migration is a compelling operation for efficient load-balancing carried out by the tenant balancer. Live TenantSpace migration allows satisfying increasing capacity needs of a TenantSpace by moving it to a node with higher capacity. Furthermore, live TenantSpace migration allows minimizing costs by moving TenantSpaces physically closer together during periods of low load. Yet, live migration constitutes a challenging task because it must not interrupt service processing or degrade service performance significantly.

ProRea meets this challenge well. By combining proactive and reactive measures to transfer the TenantSpace from one node to another node, it provides efficient live migration, assumed snapshot isolation. The proactive measures in conjunction with the improved buffer pool handling entail less page faults and less disk I/O and, ultimately, less migration overhead compared to a purely reactive approach.

With respect to ProRea, we consider one central direction for future work. Although snapshot isolation became quite popular, it does not avoid all concurrency anomalies. Recent research into serializable snapshot isolation shows that guaranteeing true serializability retains most of the performance benefits of snapshot isolation [Cahill et al., 2008a]. A first implementation of serializable snapshot isolation in a production RDBMS release, namely

PostgreSQL, underpins these results [Ports and Grittner, 2012a]. These results reveal a new direction for future work: the redesign and clean reimplementation of ProRea to guarantee serializable snapshot isolation during migration.

With respect to live database migration for our use case, an evaluation focusing on downscaling is mandatory. ProRea unburdens the source very fast, as it switches transaction load from the source to the destination early. This behavior is particularly useful to migrate from an (almost) over-utilized cluster node to a lowly utilized cluster node. However, for downscaling it is more desired to keep the load as long as possible at the source in order to reduce the time window which utilizes the destination by migration and transaction processing concurrently. For this reason, we imagine that another approach with respect to transferring pages and handover load is useful. An approach that starts with proactive measures and finishes with reactive measures might be an interesting alternative. This approach could primarily push all cold pages from the source to the destination. During this period, the source records all pages which have been transferred and which have been modified thereafter. Next, it pushes hot pages and hands over load from the source to the destination. Handover includes the list of modified pages. In the next phase, the source pushes modified pages again and the destination reactively pulls modified pages on demand. We regard the implementation and evaluation of such an approach an interesting direction for future work.

With respect to the general infrastructure, the design of the tenant balancer as autonomous controller, which decides when to migrate, which database to migrate and where to migrate, represents an interesting and demanding work. Such decisions require taking into account typical capacity planning and placement criteria: load patterns, the growth of a TenantSpace, the associated penalty if a tenant's SLAs are violated, diurnal cycles of a tenant's load and so forth. Against this background, we consider an evaluation of existing capacity planning and placement methods with regard to their customization for migration decisions an interesting direction for future work.

CHAPTER 7

Conclusions

For SaaS application providers, it is mandatory to take into account multi-tenancy in order to create a competitive offering. Multi-tenancy allows the SaaS application provider to reduce capital and operational expenditures by the consolidation of multiple tenants onto a single physical resource. Multi-tenant SaaS applications often rely on an RDBMS as data backend. Therefore, an RDBMS with native multi-tenancy support simplifies application development and allows fulfilling central multi-tenancy requirements: consolidation efficiency, tenant-specific customization, tenant-oriented operability and isolation. This thesis contributes to such an RDBMS for OLTP-style workloads.

By integrating tenants as first-class database objects and introducing the idea of a tenant context, we established the necessary infrastructure to create multi-tenancy features in an RDBMS. Based on this infrastructure, TenantSchema, which is our multi-tenant schema management approach, improves consolidation efficiency by avoiding schema redundancy between tenants. Yet, it still supports tenant-specific customizations in isolated manner without affecting other tenants. In addition to that, it also keeps in view data sharing between tenants.

For many small tenants, the consolidation of multiple tenants into a single data structure turned out beneficial with respect to query performance, but this approach comes with disadvantages regarding tenant-oriented bulk operations. Our concept of TenantSpaces allows for a partitioning which aligns to tenant boundaries and which helps to control the degree of sharing and the possibilities of tenant-oriented operations and customizations.

Such a partitioning solution which aligns to tenant boundaries is particularly important in a distributed environment. In our proposed shared-nothing cluster architecture, TenantSpaces offer an appropriate granule for elastic workload management since they are aligned to server boundaries. For elastic workload management, our presented live TenantSpace migration approach, called ProRea, represents an eminently important operation that comes with minimal service interruption and low overheads, given that applications provide an OLTP-style workload.

To conclude, our presented multi-tenancy support facilitates the development and operation of a multi-tenant OLTP-style SaaS application. The presented features are integrated into a traditional disk-based RDBMS architecture. Thus, this approach represents a non-disruptive innovation which is feasible and definitely represents a benefit, but it also comes with certain limitations. The discussion about how to store a tenant's tuples shows a central issue: the format used in main memory is in line with the format used on external storage. In our scenario, this limitation is painful as it complicates using the main memory as efficient as possible. Emerging main-memory RDBMSs get rid of this limitation and provide further advantages. However, multi-tenant OLTP-style applications usually have a significant amount of rarely accessed data by what a main-memory only system would waste resources, e. g., energy. Thus, we envision hybrid architectures with smart loading techniques from an external medium for a tenant's data. Nevertheless, our findings are useful for such systems as well since they have to deal with similar challenges, e. g., schema redundancy and live migration, and thus also benefit from solutions discussed in this thesis.

List of Figures

List of Tables

Bibliography

E. Agichtein and L. Gravano. Querying Text Databases for Efficient Information Extraction. In *Proc. of ICDE*, pages 113–124, 2003.

D. Agrawal, S. Das, and A. El Abbadi. Big Data and Cloud Computing: Current State and Future Opportunities. In *Proc. of 14th EDBT/ICDT'11*, pages 530–533, 2011c.

R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In *Proc. of VLDB '01*, pages 149–158, 2001.

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A Vew of Cloud computing. *Commun. ACM*, 53:50–58, 2010.

S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *Proc. of SIGMOD Conf.*, pages 1195–1206, 2008.

S. Aulbach, M. Seibold, D. Jacobs, and A. Kemper. Extensibility and Data Sharing in evolving multi-tenant databases. In *Proc. of ICDE*, pages 99–110, 2011.

R. A. Baeza-Yates. Expected behaviour of B^+-trees under random insertions. *Acta Informatica*, 26:439–471, 1989.

J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. of CIDR*, pages 223–234, 2011.

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of SOPS*, pages 164–177, 2003.

R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173–189, 1972.

R. Bayer and K. Unterauer. Prefix B-trees. *ACM Trans. Database Syst.*, 2: 11–26, 1977.

J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. *Proc. of ICDE*, 2006a.

H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of SIGMOD Conf.*, pages 1–10, 1995a.

B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat. Efficient Index Compression in DB2 LUW. *Proc. VLDB Endow.*, 2:1462–1473, 2009.

M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proc. of SIGMOD Conf.*, pages 729–738, 2008a.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proc. of Symposium OSDI*, pages 15–15, 2006.

L. J. Chen, P. A. Bernstein, P. Carlin, D. Filipovic, M. Rys, N. Shamgunov, J. F. Terwilliger, M. Todic, S. Tomasevic, and D. Tomic. Mapping XML to a Wide Sparse Table. In *Proc. of ICDE*, pages 630–641, 2012.

F. Chong and G. Carraro. Architecture Strategies for Catching the Long

Tail. Microsoft Corp. Website, 2006. URL http://msdn.microsoft.com/en-us/library/aa479069.aspx.

E. Chu, J. Beckmann, and J. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *Proc. of SIGMOD Conf.*, pages 821–832, 2007.

F. Coelho, A. Aillos, S. Pilot, and S. Valeev. A Field Analysis of Relational Database Schemas in Open-source Software. In *Proc. of DBKDA*, pages 9–15, 2011.

E. G. Coffman and P. J. Denning. *Operating systems theory*. Prentice-Hall Englewood Cliffs, N.J.,, 1973.

B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. of VLDB Endow.*, 1:1277–1288, 2008.

C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proc. of VLDB Endow.*, 3:48–57, 2010b.

C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *Proc. of CIDR*, 2011a.

C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the PRISM workbench. *Proc. VLDB Endow.*, 1:761–772, 2008.

S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: an elastic transactional data store in the cloud. In *Proc. of HotCloud Conference*, 2009.

S. Das, D. Agrawal, and A. El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *Proc. of SOCC*, pages 163–174, 2010.

S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. of VLDB Endow.*, 4:494–505, 2011.

J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, 2008.

G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proc. of SIGOPS*, pages 205–220, 2007.

J. R. Driscoll, S.-D. Lang, and L. A. Franklin. Modeling B-Tree Insertion Activity. *Inf. Process. Lett.*, 26:5–18, 1987.

Eggenberger and Pólya. Über die Statistik verketteter Vorgänge, Zeitschrift für angewandte Mathematik und Mechanik. *Zeitschrift für angewandte Mathematik und Mechanik*, 1:279–289, 1923.

B. Eisenbarth, N. Ziviani, G. H. Gonnet, K. Mehlhorn, and D. Wood. The theory of fringe analysis and its application to 23 trees and b-trees. *Information and Control*, 55:125 – 174, 1982.

A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Towards an Elastic and Autonomic Multitenant Database. In *Proc. of NetDB Workshop*, 2011a.

A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proc. of SIGMOD Conf.*, pages 301–312, 2011c.

D. Florescu and D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical report, INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE, 1999.

G. C. Frederick Chong and R. Wolter. Multi-Tenant Data Architecture. Microsoft Corp. Website, 2006. URL `http://msdn.microsoft.com/en-us/library/aa479086.aspx`.

C. French. One size fits all" database architectures do not work for DSS. *ACM SIGMOD Record*, 24(2):449–450, 1995.

G. Graefe. Sorting and Indexing with Partitioned B-Trees. In *Proc. of CIDR*, 2003.

J. Gray. Notes on Data Base Operating Systems. In *An Advanced Course of Operating Systems*, pages 393–481, 1978.

J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

T. Haerder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15:287–317, 1983a.

J. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting Database Applications as a Service. In *Proc. of ICDE*, pages 832–843, 2009. doi: 10.1109/ICDE.2009.82.

P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, volume 10, 2010.

IBM, editor. *IMS – Administration Guide: Database Manager Version 9*. IBM, 2011.

IBM. DB2 10.1 LUW Infocenter – SQL and XML Limits, 2012.

D. Jacobs and S. Aulbach. Ruminations on Multi-Tenant Databases. In *Proc. of BTW Conf.*, pages 514–521, 2007.

M. Jakob, O. Schiller, H. Schwarz, and F. Kaiser. flashWeb: Graphical Modeling of Web Applications for Data Management. In *ER (Tutorials, Posters, Panels & Industrial Contributions)*, pages 59–64, 2007.

T. Johnson and D. Shasha. Utilization of B-trees with Inserts, Deletes and Modifies. In *Proc. of Symp. PODS*, pages 235–246, 1989. ISBN 0-89791-308-6.

B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of VLDB*, pages 134–143, 2000.

J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36:41–50, 2003. ISSN 0018-9162.

H. Kim, S. Choi, H. Choi, E. LEE, K. Kang, S. Song, and H. Lim. Metadata Driven Software Architecture, Feb. 6 2014. URL `http://www.google.com/patents/US20140040861`. US Patent App. 13/747,648.

R. Kuć. *Apache Solr 4 Cookbook*. Packt Publishing, Limited, 2013.

T. Kwok, T. Nguyen, and L. Lam. A Software as a Service with Multi-tenancy Support for an Electronic Contract Management Application. *IEEE Intl.*

Conf. on Services Computing, 2:179–186, 2008.

A. Lakshman and P. Malik. Cassandra A decentralized structured storage system. *Operating systems review*, 44(2):35, 2010.

L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, 1998.

C. Langenhop and W. E. Wright. Certain Occupancy Numbers via an Algorithm for Computing Their Ratios. *SIAM J. Discrete Math.*, 1(3):360–371, 1988.

C. Langenhop and W. E. Wright. A Model of the Dynamic Behavior of B-Trees. *Acta Inf.*, 27(1):41–59, 1989.

E. Lau and S. Madden. An integrated approach to recovery and high availability in an updatable, distributed data warehouse. In *In Proc. VLDB*, pages 703–714, 2006.

J. Løland. *Materialized view creation and transformation of schemas in highly available database systems*. PhD thesis, PhD thesis, Norwegian University of Science and Technology (NTNU), Norway, 2007.

H. Mälzer. Implementierung einer Online-Reorganisation von Tenantspaces. Diplomarbeit, 2013.

P. Mell and T. Grance. The NIST Definition of Cloud Computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

Microsoft. SQL Azure. http://msdn.microsoft.com/en-us/library/ee336230.aspx.

Microsoft. SQL Server 2008 R2 — Understanding Pages and Extents. MSDN Library, 03 2013. URL `http://msdn.microsoft.com/en-us/library/ms190969(v=sql.105).aspx`.

D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32:241–299, 2000.

C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17: 94–162, 1992.

R. Normann and L. T. Ostby. A theoretical study of 'Snapshot Isolation'. In

Proc. of the 13th ICDT, 2010.

E. Omiecinski. Concurrent storage structure conversion: from B+ tree to linear hash file. In *Proc. of 4th ICDE*, pages 589–596, 1988.

E. Omiecinski, L. Lee, and P. Scheuermann. Concurrent file reorganization for record clustering: A performance study. In *Proc. of 8th ICDE*, pages 265–272. IEEE, 1992.

Oracle. Oracle Database SQL Language Reference 11g (11.1) – Oracle Compliance with FIPS 127-2, 2008b.

Oracle. Oracle Database JDBC Developer's Guide and Reference 11g (11.1) – Proxy Authentication, 2012.

A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proc. of SIGMOD Conf.*, pages 61–72, 2012.

D. R. K. Ports and K. Grittner. Serializable Snapshot Isolation in PostgreSQL. In *Proc. of VLDB*, 2012a.

PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org`, 2012.

R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proc. of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 28–31, 1998.

D. P. Reed. NAMING AND SYNCHRONIZATION IN A DECENTRALIZED COMPUTER SYSTEM. Technical report, Cambridge, MA, USA, 1978.

B. Reinwald. Database support for multi-tenant applications. In *Proc. of IEEE Workshop on Information and Software as Services*, 2010.

B. Schiller. Tenantspace – Ein Raum für Mandanten. Diplomarbeit, 2011.

O. Schiller, A. Brodt, and B. Mitschang. Partitioned or Non-Partitioned Table Storage?
Concepts and Performance for Multi-tenancy in RDBMS. In *Proc. of SEDE Conf.*, 2011a.

O. Schiller, B. Schiller, A. Brodt, and B. Mitschang. Native support of multi-tenancy in RDBMS for software as a service. In *Proc. of EDBT Conf.*, pages 117–128, 2011b.

O. Schiller, A. Brodt, and B. Mitschang. On Consolidating Many Tenants into a Shared B-tree Index. In *Proc. of SEDE Conf.*, 2012.

O. Schiller, N. Cipriani, and B. Mitschang. ProRea – Live Database Migration for Multi-tenant RDBMS with Snapshot Isolation. In *Proc. of EDBT Conf.*, pages 117–128, 2013.

Schiller et al. Native Support for Multi-tenancy in RDBMS for Software as a Service. In *Proc. of EDBT Conf.*, pages 117–128, 2011.

Sean Barker et al. "Cut Me Some Slack": Latency-Aware Live Migration for Databases. In *Proc. of EDBT Conf.*, 2012.

G. H. Sockut and B. R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41:14:1–14:136, 2009.

A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *Proc. of SIGMOD Conf.*, 2008.

F. R. C. Sousa and J. C. Machado. Towards Elastic Multi-Tenant Database Replication with Quality of Service. In *Proc. of International Conference on Utility and Cloud Computing*, UCC '12, pages 168–175, 2012.

J. Speer and M. Kirchberg. D-ARIES: A Distributed Version of the ARIES Recovery Algorithm. In *ADBIS Research Communications*, 2005.

M. Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18:4–11, 1989.

M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *Proc. of VLDB Endow.*, VLDB '05, pages 553–564, 2005.

M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proc. of VLDB Endow.*, pages 1150–1160, 2007a.

SuccessFactors. Distinctive Cloud Technology Platform.

http://www.successfactors.com/cloud/architecture/, March 2012.

T. Talius, R. Dhamankar, A. Dumitrache, and H. Kodavalla. Transaction Log Based Application Error Recovery and Point In-Time Query. *Proc. of PVLDB*, 5(12):1781–1789, 2012.

J. Tatemura, O. Po, and H. Hacgümüş. Microsharding: a declarative approach to support elastic OLTP workloads. *SIGOPS Oper. Syst. Rev.*, 46:4–11, 2012.

Transaction Processing Performance Council. TPC Benchmark C, Standard Specification, Revision 5.10.1, 2012.

Umar Farooq Minhas et al. Elastic Scale-out for Partition-Based Database Systems. In *International SMDB Workshop , ICDE Workshops*, 2012.

Urdaneta et al. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks*, 53(11):1830–1845, 2009.

W. Vogels. Eventually Consistent. *Queue*, 6:14–19, 2008.

G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2001. ISBN 9781558605084. URL `http://books.google.de/books?id=U1mucyGy13MC`.

C. D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *Proc. of SIGMOD Conf.*, pages 889–896, 2009.

W. E. Wright. Some average performance measures for the B-tree. *Acta Informatica*, 21:541–557, 1985.

S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. In *Proc. of ICDE*, pages 422–433, 2005.

F. Yang, J. Shanmugasundaram, and R. Yerneni. A Scalable Data Platform for a Large Number of Small Applications. In *Proc. of CIDR*, 2009.

A. C.-C. Yao. On Random 2-3 Trees. *Acta Inf.*, 9:159–170, 1978.

W. Zhou, G. Pierre, and C. Chi. CloudTPS: Scalable Transactions for Web Applications in the Cloud. *IEEE Transactions on Services Computing*, 2011.

C. Zou and B. Salzberg. Safely and efficiently updating references during on-line reorganization. In *Proc. of VLDB*, pages 512–522, 1998.

All web links were last followed on February, 2014.

Curriculum Vitae

Oliver Schiller	
Date of birth:	December 1th, 1980
Place of birth:	Kirchheim unter Teck, Germany
Nationality:	German

07/2007 – 03/2013	Research staff member at Institute of Parallel and Distributed Systems (IPVS), Universität Stuttgart, Germany
11/2006 – 04/2007	Diploma thesis, Universität Stuttgart, Germany *Entwicklung eines CAWE-Werkzeuges für die modellbasierte Entwicklung von Webanwendungen*
10/2001 – 04/2007	Studies in Computer Science Universität Stuttgart, Germany Degree: Diplom-Informatiker (Dipl.-Inf.)
09/2000 – 06/2001	Military Service at Bundeswehr, Donauwörth and Ulm
09/1987 – 07/2000	Secondary School at Schlossgymnasium, Kirchheim unter Teck, Germany Degree: Abitur
09/1985 – 07/1989	Primary School at Grundschule, Kirchheim unter Teck-Nabern, Germany