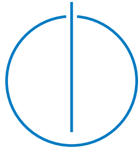


Technische Universität München
Lehrstuhl III - Datenbanksysteme

Schema Flexibility and Data Sharing in Multi-Tenant Databases

Diplom-Informatiker Univ.
Stefan Aulbach

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Florian Matthes

Prüfer der Dissertation:
1. Univ.-Prof. Alfons Kemper, Ph. D.
2. Univ.-Prof. Dr. Torsten Grust
(Eberhard-Karls-Universität Tübingen)

Die Dissertation wurde am 26.05.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.10.2011 angenommen.

Abstract

Hosted applications, operated by independent service providers and accessed via the Internet, commonly implement multi-tenancy to leverage economy of scale by consolidating multiple businesses onto the same operational system to lower Total Cost of Ownership (TCO). Traditionally, Software as a Service (SasS) applications are built on a software platform to allow tenants to customize the application. These platforms use traditional database systems as storage back-end, but implement an intermediate layer that maps the application data to generic structures inside the DBMS. We discuss these mapping techniques in the context of schema flexibility, such as schema extensibility and schema evolution. However, as the schema mapping has to be handled by the application layer, the DBMS degenerates to be a "dumb" data repository. Thus, a multi-tenant database system for SaaS should offer explicit support for schema flexibility. That implies that schemas can be extended for different versions of the application and dynamically modified while the system is on-line. Furthermore, when co-locating tenants, there is a potential for sharing certain data across tenants, such as master data and configuration data of the application.

We present FlexScheme, a meta-data model that is specially designed for multi-tenancy. It enables data sharing and schema flexibility by adding explicit support for schema extensibility and "Lights-out" Online Schema Evolution. We employ meta-data sharing where each tenant inherits the schema of the base application that can be extended to satisfy the individual tenant's needs. The meta-data sharing is complemented by master data sharing where the global data set is overridden by the individual changes of a tenant. We develop a light-weight schema evolution mechanism that, rather than carrying out costly data reorganizations immediately, lets the DBMS adaptively schedule the reorganization to not impact co-located tenants.

We implement a main-memory-based DBMS prototype that has specialized operators for data sharing and lights-out schema evolution. We show that, in the multi-tenancy context, both techniques effectively lower TCO by allowing a higher tenant packaging as well as better resource utilization. Furthermore, we focus on efficient storage mechanisms for FlexScheme and present a novel approach, called XOR Delta, which is based on XOR encoding and is optimized for main-memory DBMS.

Acknowledgements

Many people attended me during the time I was working on this thesis. Without their invaluable support, this project never completed.

I am grateful to my advisor Prof. Alfons Kemper, Ph.D., for all the insightful discussions, the feedback on publications and presentations, the encouragements, and the opportunity to pursue this thesis at his chair.

Dean Jacobs, Ph.D., is responsible for me to have the initial contact with Software as a Service. At this point, I would like to thank him for all the comprehensive discussions, phone calls, and e-mails.

For their help, the pleasant work atmosphere, the many discussions, and much more I would like to thank my colleagues: Martina Albutiu, Nikolaus Augsten, Ph.D., Veneta Dobreva, Florian Funke, Dr. Daniel Gmach, Prof. Dr. Torsten Grust, Andrey Gubichev, Benjamin Gufler, Sebastian Hagen, Stefan Krompaß, Dr. Richard Kuntschke, Manuel Mayr, Henrik Mühe, Prof. Dr. Thomas Neumann, Fabian Prasser, Dr. Angelika Reiser, Jan Rittinger, Dr. Tobias Scholl, Andreas Scholz, Michael Seibold, Jessica Smejkal, and Bernd Vögele. I particularly thank our secretary Evi Kollmann.

Several students offered their support for implementing the research prototype: Andreas Gast, Vladislav Lazarov, Alfred Ostermeier, Peng Xu, and Ursula Zucker. I am thankful for their work.

Finally, I thank Iris, my partner, and Ruth, my mother, for all their love, support and patience throughout the years.

Contents

1. Introduction **1**

1.1. Cloud Computing . . . 1

1.2. Software as a Service . . . 2

1.3. The Impact of Massive Multi-Tenancy . . . 4

1.4. Schema Flexibility of SaaS Applications . . . 4

1.5. Problem Statement . . . 7

1.6. Contributions and Outline . . . 8

2. Fundamental Implementation of Multi-Tenancy **11**

2.1. Basic Database Layouts for Multi-Tenancy . . . 11

2.1.1. Shared Machine . . . 11

2.1.2. Shared Process . . . 13

2.1.3. Shared Table . . . 13

2.2. Scalability and Efficiency . . . 14

2.2.1. Multi-Tenancy Testbed . . . 15

2.2.2. Experiment: DBMS Performance with Many Tables . . . 18

2.3. Multi-Tenancy Issues with Current DBMS . . . 21

2.3.1. Buffer Pool Utilization . . . 21

2.3.2. Meta-Data Management . . . 22

3. Extensible Schemas for Traditional DBMS **25**

3.1. Schema Mappings . . . 25

3.1.1. Schema Based Approaches . . . 26

3.1.2. Generic Structures . . . 27

3.1.3. Semistructured Approaches . . . 31

3.2. Case Study: The force.com Platform . . . 33

3.2.1. Meta-Data-Driven Applications . . . 34

3.2.2. Data Persistence . . . 34

3.3. Request Processing . . . 35

3.3.1. Chunk Tables . . . 35

3.3.2. IBM pureXML . . . 39

3.4. Chunk Table Performance . . . 41

3.4.1. Test Schema . . . 41

3.4.2. Test Query . . . 42

3.4.3. Transformation and Nesting . . . 43
3.4.4. Transformation and Scaling . . . 43
3.4.5. Response Times with Warm Cache . . . 44
3.4.6. Logical Page Reads . . . 44
3.4.7. Response Times with Cold Cache . . . 44
3.4.8. Cache Locality Benefits . . . 46
3.4.9. Additional Tests . . . 46
3.5. Schema Evolution on Semistructured Schema Mappings . . . 47
3.5.1. Microsoft SQL Server . . . 47
3.5.2. IBM DB2 . . . 52

4. Next Generation Multi-Tenant DBMS **55**
4.1. Estimated Workload . . . 56
4.2. Multi-Tenant DBMS Concepts . . . 56
4.2.1. Native Schema Flexibility . . . 57
4.2.2. Tenant Virtualization . . . 57
4.2.3. Grid-like Technologies . . . 58
4.2.4. Main Memory DBMS . . . 58
4.2.5. Execution Model . . . 59
4.3. Native Support for Schema Flexibility . . . 60
4.4. FlexScheme – Data Management Model for Flexibility . . . 61
4.4.1. Data Structures . . . 61
4.4.2. Deriving Virtual Private Tables . . . 64
4.4.3. Comparison to Other Models . . . 65
4.5. Applying FlexScheme to Multi-Tenant DBMS . . . 66

5. Data Sharing Across Tenants **69**
5.1. Data Overlay . . . 69
5.2. Accessing Overridden Data . . . 70
5.2.1. Basic Implementations . . . 71
5.2.2. Special Index-Supported Variants . . . 71
5.2.3. Data Maintenance . . . 74
5.2.4. Secondary Indexes . . . 75
5.2.5. Overlay Hierarchies . . . 75
5.2.6. Comparison to Other Delta Mechanisms . . . 78
5.3. Shared Data Versioning . . . 80
5.3.1. Pruning of Unused Versions . . . 80
5.3.2. Tenant-specific Reconciliation . . . 80
5.4. Physical Data Organization . . . 82
5.4.1. Snapshot Approach . . . 83
5.4.2. Dictionary Approach . . . 84
5.4.3. Temporal Approach . . . 85
5.4.4. Differential Delta Approach . . . 86
5.4.5. XOR Delta Approach . . . 88

5.5. Evaluation 90
5.5.1. Overlay Operator 90
5.5.2. Physical Data Organization 95

6. Graceful On-Line Schema Evolution **101**
6.1. Schema Versioning 102
6.1.1. Versioning Concepts 103
6.1.2. Atomic Modification Operations 104
6.1.3. Applying Modification Operations 105
6.1.4. Version Pruning 106
6.2. Lightweight Physical Data Reorganization 107
6.2.1. Objectives 107
6.2.2. Evolution Operator 107
6.2.3. Physical Tuple Format 108
6.2.4. Access Behavior 109
6.3. Evolution Strategies 110
6.3.1. Immediate Evolution 110
6.3.2. Lightweight Evolution 111
6.3.3. Strategy Selection 112
6.4. Evaluation 112
6.4.1. Point-Wise Access 112
6.4.2. Queries 113
6.4.3. Modification Operation Chain Length 115
6.4.4. Summary 116
6.5. Related Work 116

7. Outlook and Future Challenges **119**

A. Appendix **121**
A.1. Associativity of Data Overlay 121

List of Figures

1.1. Software Development Priorities 2
1.2. Consolidation 3
1.3. Extensibility and Evolution for a Single Tenant 5
1.4. Extensibility and Evolution for Multiple Tenants 6

2.1. Basic Database Layouts 12
2.2. Shared Table Example 14
2.3. Multi-Tenancy Testbed Architecture 15
2.4. CRM Application Schema 16
2.5. Worker Action Classes 17
2.6. Schema Variability Results 20

3.1. Private Table Layout 26
3.2. Extension Table Layout 27
3.3. Universal Table Layout 27
3.4. Pivot Table Layout 29
3.5. Chunk Table Layout 30
3.6. Chunk Folding Layout 30
3.7. Sparse Columns Layout 31
3.8. pureXML Columns Layout 32
3.9. HBase Pivot Columns Layout 33
3.10. Query Execution Plan for Query $Q2_{pureXML}$ 40
3.11. Test Layout: Conventional 41
3.12. Test Layout: $Chunk_6$ 42
3.13. Query Execution Plan for Simple Fragment Query 43
3.14. Response Times with Warm Cache 45
3.15. Number of Logical Page Reads. 45
3.16. Response Times with Cold Cache 46
3.17. Response Times: Chunk Tables vs. Vertical Partitioning 47
3.18. Refined Worker Action Classes 48
3.19. SQL Server Performance 49
3.20. SQL Server Throughput 51
3.21. DB2 Performance 52

4.1. Multi-Tenant DBMS Cluster Architecture 57

4.2. Thread Model of the Multi-Tenant DBMS Instance 60
4.3. Example of Polymorphic Relation *R* 62

5.1. Data Overlay in SQL 70
5.2. Query Execution Plan Example 70
5.3. Overlay with Separate Indexes for Shared and Private Data 73
5.4. Overlay with Single Combo Index per Tenant 74
5.5. Overlay Hierarchy 76
5.6. Snapshot Approach 83
5.7. Dictionary Approach 84
5.8. Temporal Approach 85
5.9. Differential Delta Data Approach 86
5.10. XOR Delta Approach 88
5.11. Overlay Operator Response Times 91
5.12. Rebuild Times for Bit-Set and Combo-Index Rebuild 93
5.13. Roll-Up Query 94
5.14. Effects of Data Overlay on Queries 94
5.15. Query Execution Times 95
5.16. Space Requirements 97
5.17. Maintenance Execution Times 98

6.1. Pruning of Unused Fragment Versions 106
6.2. Evolution Operator Placement within Query Execution Plan 107
6.3. Physical Tuple Layout 108
6.4. Query Execution Plan with Evolution and Overlay Operators 110
6.5. Evolution Operator Performance for Point-Wise Accesses 114
6.6. Computational Overhead of Query Processing with Evolution 115
6.7. Modification Operations Influence 116

List of Tables

2.1. Schema Variability and Data Distribution 18
2.2. Experimental Results 20

1
Introduction

> *"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*
>
> (Mell and Grance, 2009)

With the advent of the *Ubiquitous Internet* a new trend has emerged: Cloud Computing. Cheap and reliable broadband Internet access allows businesses and individuals to communicate and collaborate in real-time across countries and even continents. Internet-based applications like *Google Apps* and *Microsoft Office Live!* gain popularity, as they enable easy collaboration and sharing of data around the globe, without the need for installing any software on the PC. Thus, applications and data are no longer maintained on individual PCs, but rather reside inside the *Cloud* of the Internet.

1.1. Cloud Computing

According to the NIST Definition of Cloud Computing (Mell and Grance, 2009, see above), Cloud Computing offers customers an on-demand self-service for accessing resources remotely. Depending on the service model, these are either hardware or software resources which are accessed via the Internet. The resources are allocated on demand, once a customer requests them, and are freed automatically after the user closes the session, without any human intervention from a service provider. Currently, three different models for Cloud Computing services are available.

Infrastructure as a Service (IaaS) A service provider pursuing this model offers on-demand access to infrastructural services, such as processing resources (e.g. virtual machines), storage, or other computing resources. An IaaS customer is able to deploy any application and run it without modification on the service provider's infrastructure. Within its designated environment, the customer has full control of all resources. Amazon EC2 (2011) is a typical representative of this service model.

Platform as a Service (PaaS) One abstraction level above of IaaS is the PaaS. A service provider offers a software development and runtime platform, where customers and/or

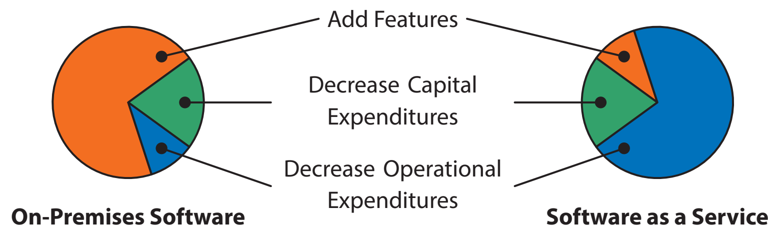


Figure 1.1.: Software Development Priorities

software vendors can develop and deploy applications. The development environment, including the programming language, is offered by the service provider. In the PaaS model, the consumers do not have any possibility to access underlying hardware resources. Instead, the service provider offers high-level APIs to interface with the infrastructure. As an example, the force.com Platform (Weissman and Bobrowski, 2009) offers a data persistence module for accessing a relational DBMS.

Software as a Service (SaaS) In this model, the service provider owns and operates an application that is accessed by the customers via Internet browser and/or Web Services. The customer can only access the application itself; all access to the underlying infrastructure is prohibited by the service provider. However, depending on the application, individual customizations may be possible.

All of the models have in common that the pricing of the services is on a Pay-per-Use basis, where service charges are due for only those resources that really have been consumed.

1.2. Software as a Service

From an enterprise perspective, accessing applications which are installed and maintained off-site is a well known practice. An *Application Hosting Provider* maintains a data-center where complex applications like ERP systems are installed. This way their customers do not need the technical expertise in their own staff, rather they pay the hosting provider a monthly fee based on the amount of consumed resources for administering the application. As the hosting provider only operates the infrastructure, the hosted application has to be licensed independently.

A very modern form of application hosting is Software as a Service (SaaS). Compared to traditional application hosting, where the service provider only maintains the infrastructure, the SaaS provider owns and maintains the application as well. SaaS customers typically access the service using standard Internet browsers or Web Services. SaaS services have appeared for a wide variety of business applications, including Customer Relationship Management (CRM), Supplier Relationship Management (SRM), Human Capital Management (HCM), and Business Intelligence (BI).

Design and development priorities for SaaS differ greatly from those for on-premises software, as illustrated in Figure 1.1. The focus of on-premises software is generally on adding features, often at the expense of reducing Total Cost of Ownership. In contrast, the

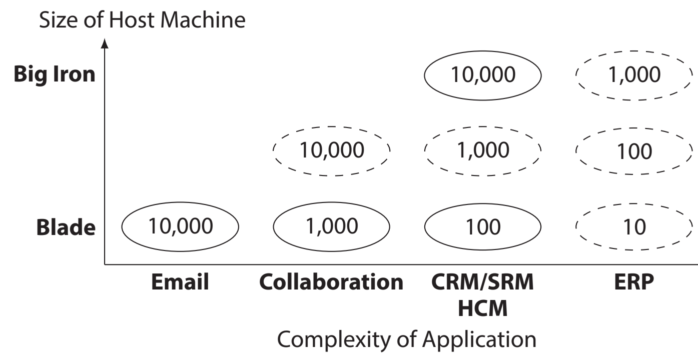


Figure 1.2.: Consolidation

focus of SaaS is generally on reducing *Total Cost of Ownership* (TCO), often at the expense of adding features. The primary reason for this is, of course, that the service provider, rather than the customer has to bear the cost of operating the system. In addition, the recurring revenue model of SaaS makes it unnecessary to add features in order to drive purchases of upgrades.

A well-designed hosted service reduces TCO by leveraging economy of scale. The greatest improvements in this regard are provided by a *multi-tenant architecture*, where multiple businesses are consolidated onto the same operational system. Multi-tenancy invariably occurs at the database layer of a service; indeed this may be the only place it occurs since application servers for highly-scalable Web applications are often stateless (Hamilton, 2007).

The amount of consolidation that can be achieved in a multi-tenant database depends on the complexity of the application and the size of the host machine, as illustrated in Figure 1.2. In this context, a tenant denotes an organization with multiple users, commonly around 10 for a small to mid-sized business. For simple Web applications like business email, a single blade server can support up to 10,000 tenants. For mid-sized enterprise applications like CRM, a blade server can support 100 tenants while a large cluster database can go up to 10,000. While the TCO of a database may vary greatly, consolidating hundreds of databases into one will save millions of dollars per year.

One downside of multi-tenancy is that it can introduce contention for shared resources (MediaTemple, 2007), which is often alleviated by forbidding long-running operations. Another downside is that it can weaken security, since access control must be performed at the application level rather than the infrastructure level. Finally, multi-tenancy makes it harder to support application extensibility, since shared structures are harder to individually modify. Extensibility is required to build specialized versions of enterprise applications, e.g., for particular vertical industries or geographic regions. Many hosted business services offer platforms for building and sharing such extensions (Salesforce AppExchange, 2010).

In general, multi-tenancy becomes less attractive as application complexity increases. More complex applications like Enterprise Resource Planning (ERP) and Financials require more computational resources, as illustrated in Figure 1.2, have longer-running operations, require more sophisticated extensibility, and maintain more sensitive data. More-

over, businesses generally prefer to maintain more administrative control over such applications, e.g., determining when backups, restores, and upgrades occur. More complex applications are of course suitable for single-tenant hosting.

1.3. The Impact of Massive Multi-Tenancy

Multi-tenancy makes it harder to implement several essential features of enterprise applications. The first is support for master data, which should be shared rather than replicated for each tenant to reduce costs. Examples of such data include public information about business entities, such as DUNS numbers, and private data about supplier performance. Data may be shared across organizations or between the subsidiaries and branches of a hierarchically-structured organization. In either case, the shared data may be modified by individual tenants for their own purpose and the DBMS must offer a mechanism to make those changes private to the tenant.

The second problematic feature is application modification and extension, which applies both to the database schema and the master data it contains. Such extensibility is essential to tailor the application to individual business needs, which may vary based on industries and geographical regions. An extension may be private to an individual tenant or shared between multiple tenants. In the latter case, an extension may be developed by an Independent Software Vendor (ISV) and sold to the tenant as an add-on to the base application. While a limited form of customization can be provided by configuration switches and wizards, more complex applications require the ability to modify the underlying database schema of the application.

The third problematic feature is evolution of the schema and master data, which occurs as the application and its extensions are upgraded. In contrast to most on-premise upgrades, on-demand upgrades should occur while the system is in operation. Moreover, to contain operational costs, upgrades should be "self-service" in the sense that they require the minimum amount of interaction between the service provider, the ISVs who offer extensions, and the tenants. Finally, it is desirable for ISVs and tenants to delay upgrades until a convenient time in the future. It should be possible to run at least two simultaneous versions of the application, to support rolling upgrades, however ideally more versions should be provided.

1.4. Schema Flexibility of SaaS Applications

Service providers make their SaaS applications more attractive by allowing a high level of customization. Besides very simple mechanisms like configuration switches, there are more complex mechanisms like adapting predefined entities or even adding new entities. Those modifications result in a high number of application variants, each individually customized for a particular tenant. Although most of the customizations may be small, managing the huge number of variants is a big challenge.

Furthermore, the service provider has to release new versions of the application on a regular basis. Tenants are then forced to upgrade on-the-fly to the most recent version

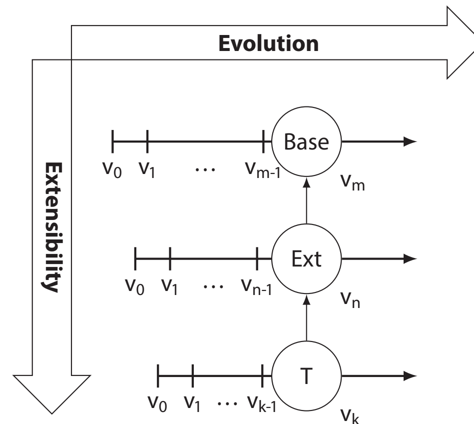


Figure 1.3.: Extensibility and Evolution for a Single Tenant

of the application. However, depending on the customizations performed by a particular tenant, that may not be possible. Currently, many service providers avoid this issue by simply limiting the customization possibilities.

In a nutshell, SaaS applications develop in at least two dimensions: extensibility and evolution. The *extensibility* dimension is made up of extensions to the common base application. These extensions may be developed individually by tenants themselves or by ISVs, and thus be shared between tenants. The *evolution* dimension tracks changes to the SaaS applications which are necessary to either fix issues with the application or to integrate new features. Evolution is not only required for the base application itself, but also for extensions.

Example 1.1: *Figure 1.3 shows a SaaS application for a single tenant which is made up of the following components: the base application* Base *in version v_m, an extension* Ext *of an ISV in version v_n, and the tenant-specific extension* T *in version v_k. The components are developed and maintained separately from each other, therefore the releases of new versions are not synchronized. There may be dependencies between the components as an extension may depend on a specific version of the base application or another extension. In the example,* T *in version v_k depends on* Ext *in version v_n and* Ext *in version v_n depends on* Base *in version v_m.*

Managing such a setup becomes challenging, as more tenants lead to more dependencies. Currently, SaaS applications avoid these issues by restricting the number of concurrently available versions: the service provider forces its customers to upgrade to the most recent application version as soon as it is released. However, for some tenants such a behavior may not be acceptable, because tenant-specific and ISVs' extensions have to be checked for compatibility. Before migration can take place, changes to extensions may be required. Affected tenants may accept to pay a higher service fee if they do not have to upgrade right away and can stay on a given version of the base application with which all their extensions work. Managing each tenant's application instance separately is not feasible for

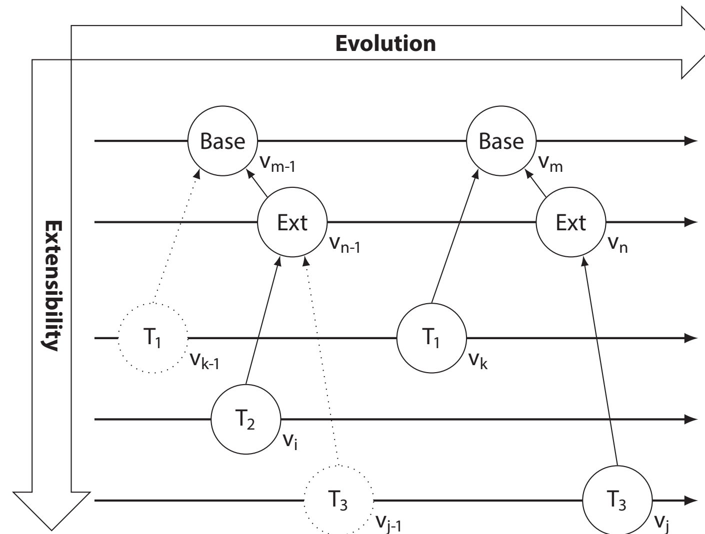


Figure 1.4.: Extensibility and Evolution for Multiple Tenants

the service provider, as the administration and maintenance costs would be similar to on-premise solutions multiplied by the number of customers of the service provider. As a consequence, a multi-tenant DBMS has to explicitly model evolution and extensibility of SaaS applications.

Example 1.2: *Figure 1.4 depicts a SaaS application used by three tenants. It shows how the SaaS application and its extensions develop according to the dimensions extensibility and evolution. In this example, the base application has evolved from version v_{m-1} to version v_m and the extension has evolved from version v_{n-1} to version v_n. Tenants T_1 and T_3 have already migrated to the new version of the base application and the extension, but Tenant T_2 is still using the old version of the base application and the extension, because the tenant-specific extension of T_2 has not been checked for compatibility with the new version of the extension and the base application yet.*

Even with a high degree of customization, big parts of the SaaS application and the ISV extensions can be shared across tenants. As the previous example shows, there is a high potential for *data sharing* as certain data can be shared across tenants. Such data may include master data or catalogs. By applying data sharing techniques, more tenants can be packed onto the existing infrastructure.

Example 1.3: *In Figure 1.4, Tenants T_1 and T_3 share version v_m of the base application* Base*. If the application would be managed separately for each tenant, the version v_m of the base application* Base *would require double amount of resources, as it would be part of the application instance of Tenant T_1 and part of the application instance of Tenant T_3. Once the tenant-specific extensions of Tenant T_2 have been checked for compatibility with*

the new version of the base application and the extension, Tenant T_2 can migrate to the new version of the base application and the extension. After this migration has happened, there is even more potential for data sharing, as all three tenants share the common base application in version v_m and Tenants T_2 and T_3 share the common third-party extension Ext *in version v_n.*

Moreover, shared data needs to be update-able, as already existing entries may have to be overwritten by tenants. Instead of replicating all data for each tenant, a small delta per tenant can be used if only a low percentage of the shared data is modified by the tenant.

From time to time, new common data becomes available, e.g., when new catalog data is released or in case of application upgrades. For a tenant, it is important to always have a consistent snapshot of the shared data. If shared data could change over time without the knowledge of the tenants, they would get unpredictable results. Therefore, there must be a migration path for each individual tenant when common data changes. Tenants are allowed to either follow the service provider's upgrade cycle or follow their own migration path by staying on a certain version.

The bottom line is that the multi-tenancy features—extensibility, evolution and data sharing—are closely related. A multi-tenant DBMS needs an integrated model to capture these features. There already exist models which capture extensibility, like the object-oriented concept of inheritance (Khoshafian and Abnous, 1990), and there are models for capturing the evolution of an application (Curino et al., 2009), but there is currently no integrated model that captures both dimensions and data sharing together. In this thesis, we therefore propose such a model, called FlexScheme.

1.5. Problem Statement

The previous sections show that there are two important design goals for multi-tenant databases. On the one hand, a high level of consolidation must be achieved in order to significantly lower TCO; on the other hand, the SaaS application must be attractive to the customer, and thus schema flexibility mechanisms have to be available to allow customizability of the application.

These features cannot be easily implemented in a traditional DBMS and, to the extent that they are currently offered at all, they are generally implemented within the application layer. Traditional DBMSs do not offer mechanisms for schema flexibility, even worse, they prohibit continuously evolving schemas, as schema-modifying DDL operations may negatively affect service availability due to expensive data redefinition phases. As an example, force.com does its own mapping from logical tenant schemas to one universal physical database schema (Weissman and Bobrowski, 2009) to overcome the limitations of traditional DBMSs. However, this approach reduces the DBMS to a 'dumb data repository' that only stores data rather than managing it. In addition, it complicates development of the application since many DBMS features, such as query optimization, have to be re-implemented from the outside. Instead, a next-generation multi-tenant DBMS should provide explicit support for extensibility, data sharing and evolution.

1.6. Contributions and Outline

This thesis addresses schema flexibility and data sharing in the context of multi-tenant applications. Due to the high level of consolidation in the SaaS context, combined with a constantly evolving database schema, the database workload of a multi-tenant SaaS application can fundamentally differ from traditional on-premise application workloads.

In **Chapter 2** we discuss the basic setup of traditional DBMSs without schema flexibility when used as a back-end for multi-tenant SaaS services. Based on the results of experiments with our multi-tenant CRM Testbed which simulates a typical SaaS application workload, we argue that traditional DBMSs suffer from performance degradation in a typical SaaS setup with a high consolidation level. Parts of this chapter have been presented at BTW Conference 2007 (Jacobs and Aulbach, 2007) and at SIGMOD Conference 2008 (Aulbach, Grust, Jacobs, Kemper, and Rittinger, 2008).

Chapter 3 presents sophisticated schema mapping techniques to allow schema extensibility on traditional DBMSs. Our novel approach called *Chunk Folding* is compared to already existing schema mapping techniques. Our performance analysis measures the impact of schema mapping on request processing. Furthermore, we evaluate vendor-specific mechanisms of some DBMS products which allow for limited support of schema evolution. The results have been presented at SIGMOD Conference 2008 (Aulbach, Grust, Jacobs, Kemper, and Rittinger, 2008) and SIGMOD Conference 2009 (Aulbach, Jacobs, Kemper, and Seibold, 2009a).

In **Chapter 4** we propose our special-purpose DBMS which is optimized for a multi-tenant workload, as it is typically expected in a Cloud Computing environment. We introduce features like native schema flexibility which is handled by our data model called *FlexScheme*. Our prototype is optimized for a low TCO by leveraging the advantages of scaling-out on a cluster of commodity hardware servers. The prototype concept has been presented at BTW Conference 2009 (Aulbach, Jacobs, Primsch, and Kemper, 2009b) and FlexScheme was presented at ICDE Conference 2011 (Aulbach, Seibold, Jacobs, and Kemper, 2011).

The *data sharing* component of our prototype is presented in **Chapter 5**. We describe our approach of accessing shared data which has been overridden by individual tenants. When accessing shared data, our specialized *Overlay* operator merges the shared data with the tenant's additions on-the-fly. Furthermore, we discuss various physical representations for shared data which support data versioning as well. Parts of this work have been presented at ICDE Conference 2011 (Aulbach, Seibold, Jacobs, and Kemper, 2011).

A specialized query plan operator for *Schema Evolution* is presented in **Chapter 6**. We propose a method for graceful on-line schema evolution, where the data is evolved on-access, rather than pre-emptive as in current DBMSs. Our approach enables schema evolution without service outages, by deferring the costly physical data redefinition. Finally, we evaluate several strategies for our lazy schema evolution approach.

Chapter 7 concludes this thesis and outlines future challenges.

Previous Publications

Parts of this thesis have been previously published at the following peer-reviewed conferences.

BTW 2007 Dean Jacobs and Stefan Aulbach. *Ruminations on Multi-Tenant Databases.* In Kemper et al. (2007), pages 514 – 521.

SIGMOD 2008 Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. *Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques.* In Wang (2008), pages 1195 – 1206.

BTW 2009 Stefan Aulbach, Dean Jacobs, Jürgen Primsch, and Alfons Kemper. *Anforderungen an Datenbanksystem für Multi-Tenancy- und Software-as-a-Service-Applikationen.* In Freytag et al. (2009), pages 544 – 555.

SIGMOD 2009 Stefan Aulbach, Dean Jacobs, Alfons Kemper, and Michael Seibold. *A Comparison of Flexible Schemas for Software as a Service.* In Çetintemel et al. (2009), pages 881 – 888.

ICDE 2011 Stefan Aulbach, Michael Seibold, Dean Jacobs, and Alfons Kemper. *Extensibility and Data Sharing in Evolving Multi-Tenant Databases.* In Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE), April 11 – 16, 2011, Hannover, Germany, pages 99 – 110.

2

Fundamental Implementation of Multi-Tenancy

For SaaS applications, the implementation of multi-tenancy is an inherent requirement as discussed in the previous section. Database vendors give a lot of recommendations on how to implement multi-tenancy atop of their products, but the concrete implementation decision depends on the application domain.

This chapter presents a set of fundamental implementation techniques for the database layer of a SaaS application, which do not per se offer schema flexibility. They serve as a basis for more sophisticated approaches, which support schema flexibility as well.

2.1. Basic Database Layouts for Multi-Tenancy

At first glance, we introduce three different approaches for implementing multi-tenancy atop of existing DBMSs: Shared Machine, Shared Process, and Shared Table. Each of these approaches differently affects TCO, as they directly influence the level of tenant consolidation. The approaches are increasingly better at resource pooling, but they increasingly break down the isolation between tenants, weaken security and affect resource contention.

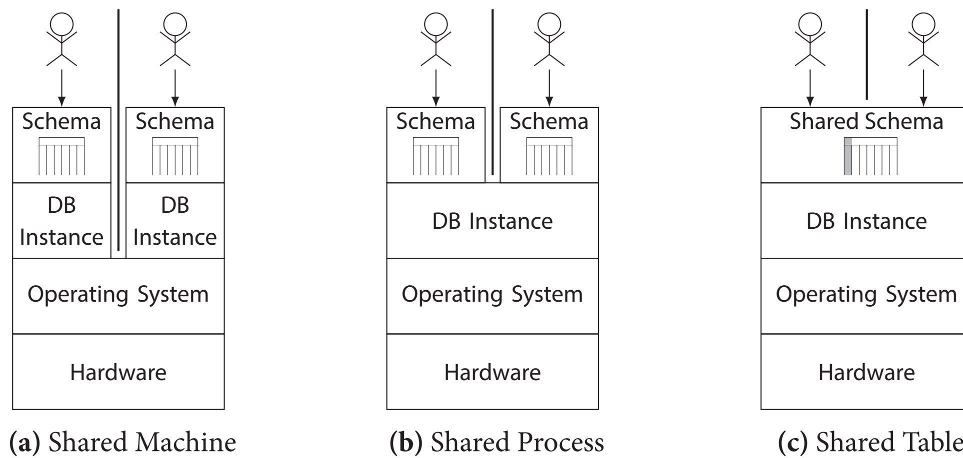
2.1.1. Shared Machine

The *Shared Machine* approach as shown in Figure 2.1a is the easiest way to implement multi-tenancy atop of existing database systems. With this approach, each tenant gets its own DBMS instance. There are two implementation variants, depending on the usage of hardware virtualization techniques. When not using virtualization, there is one DBMS process per tenant running on a shared Operating System (OS) instance on a shared host. In the other case, the host runs a hypervisor that virtualizes the hardware resources. Above this hypervisor, each tenant has its own OS instance running the DBMS process. Figure 2.1a shows the variant without virtualization.

This approach is popular in practice because it does not require modifying the implementation of the DBMS. Furthermore, it does not reduce tenant isolation, especially if each DBMS instance is run in its own virtual machine.

However, the lack of resource pooling is the major drawback of this approach. Recent DBMS instances have a memory footprint of 55 MB to 273 MB[1], depending on the database

[1] cf. Jacobs and Aulbach (2007)



(a) Shared Machine **(b)** Shared Process **(c)** Shared Table

Figure 2.1.: Basic Database Layouts

system used. In combination with virtualization, the size of the footprint increases by the amount of resources needed by the hypervisor and the OS. Even for large scale hardware, this approach cannot handle more than tens of active tenants per server. In addition, each DBMS instance requires its own connection pool on each application server, so database connections will not be shared among tenants. To improve this situation, the operating system might offer mechanisms to share connections among co-located DBMS instances.

Since each tenant has either its own DBMS instance or even its own OS instance, there is a high level of isolation between tenants. Tenant data is placed in one or more table-spaces which are exclusively allocated to the appropriate DBMS instance. Unauthorized data access across tenants is prohibited by the OS or the hypervisor. Furthermore, individual DBMS instances allow for I/O load balancing by distributing tenants across different backing disks. With this approach, each tenant is visible to the OS as a process, or as a virtual machine to the hypervisor, respectively, thus the scheduler of the OS or the hypervisor can be leveraged for avoiding resource contention.

Collocating data in a table-space on a per-tenant basis allows for easy migration from one host to another. While a tenant is off-line, the DBMS instance can be migrated by simply moving the data to the new host; for online migration, modern database systems provide mechanisms for data migration like Oracle Streams (Oracle Corporation, 2002).

For upgrading the application, each tenant has to be processed individually. This introduces a high administrative overhead during the upgrade process but allows for having different application revisions for different tenants. Furthermore, the individual upgrade process places downtime only on single tenants, whereas the application remains available for the other tenants.

Sharing of common data can be added by having a separated DBMS instance for shared data which is accessed by the application via a separate connection pool. Tenant-specific overwriting or adding common data with such a solution must be handled inside the application. Since each tenant has its own DBMS process, there is no possibility for meta-data sharing.

2.1.2. Shared Process

In order to get better resource utilization, the *Shared Process* approach can be used. As Figure 2.1b shows, each tenant gets its own set of tables within a DBMS instance which is shared across multiple tenants. For most DBMSs, there is no difference whether a tenant gets its own schema or not, since database schemas are usually implemented using a lightweight prefixing mechanism.

All tenants using the same DBMS instance share the resources of that instance. In this setup, the resource allocation to the tenants has to be controlled by the DBMS instance, which weakens the resource contention compared to the shared machine approach. This also affects tenant isolation, since the DBMS instance is responsible for guaranteeing tenant isolation. User rights and authorization rules have to be maintained carefully so that tenants can access only their individual data.

For better isolation, it is useful to place each tenant in its individual physical table-space. Doing so allows for easy migration as in the shared machine case, as well as the possibility of doing I/O load balancing across different storage back-ends. For tenant migration, similar mechanisms as in the shared machine scenario can be used.

The shared process approach increases resource sharing across tenants. One of the preeminent factors is the reduced memory consumption per tenant due to co-locating multiple tenants into one single DBMS instance. Furthermore, the connections from the application server to the DBMS can be pooled across tenants. However, in such a setup, the DBMS looses control about access permissions: the application server has to connect as super-user in order to access the data of all co-located tenants, thus weakening the security mechanisms of the DBMS. Additionally, resource contention increases, because the shared process approach suffers from a bad buffer-pool utilization if lots of tables are used simultaneously. We discuss this issue in detail in Section 2.2.2.

As with the shared machine approach, application upgrades have to process each tenant individually. However, the administrative overhead is lower since only one DBMS instance has to be managed.

Common data can be shared by placing the data in a separate schema of the DBMS instance. As with the previous approach, tenant-specific changes to the common data must be handled inside the application. Although no DBMS supports this at the moment, there is a high potential of sharing the meta-data across tenants. Since the schemas of tenants are very similar to each other, there is a high level of redundancy in the schema description.

2.1.3. Shared Table

Figure 2.1c shows the *Shared Table* approach where all co-located tenants not only share the same DBMS instance, but also the same set of tables. To distinguish entries from various tenants, each table has to be augmented by an additional column Tenant which identifies the owner of a particular tuple. An example of such an augmentation can be found in Figure 2.2. Every application query is expected to specify a single value for this column.

From a resource pooling perspective, this approach is clearly the best of the three approaches. Its ability to scale-up is only limited by the number of tuples a DBMS instance

Account			
Tenant	Aid	Name	...
17	1	Acme	...
17	2	Gump	...
35	1	Ball	...
42	1	Big	...

Figure 2.2.: Shared Table Example

can hold. As with the shared process approach, the connection pool can be shared across tenants, but then the application server needs to connect as super-user, again.

However, there are several significant problems with this approach. Since the data of all tenants are intermingled inside the DBMS instance, several issues arise with such a setup. First, tenant migration requires queries against the operational system instead of simply copying table-spaces. Second, for an application upgrade, all tenants have to migrate to the new version at the same time. Third, the isolation between tenants is further weakened. From a security perspective, there is a need for row-based access control if security should be pushed down into the DBMS instance, but not all available DBMSs support these mechanisms. The fourth and biggest problem with this approach is that queries intended for a single tenant have to contend with data from all tenants, which compromises query optimization. In particular, optimization statistics aggregate across all tenants and table scans go across all tenants. Moreover, if one tenant requires an index on a column, then all tenants have to have that index as well.

Since all tenants are packed in one single set of tables, there is no need for meta-data sharing at all, however access to common data can be enabled by putting these data in a separate schema. Write access to common data still has to be handled inside the application. As only one DBMS instance has to be managed, the administrative overhead is as low as in the shared process approach.

2.2. Scalability and Efficiency

From a scalability perspective, the shared machine approach has severe drawbacks compared to the other two approaches. A typical DBMS has a memory footprint of several 100 MB, so even big machines can accommodate only a few tenants. However, due to its high level of tenant isolation, it may be useful in circumstances where security concerns are paramount. For better multi-tenant efficiency the latter two approaches look more promising. We therefore analyse the performance of currently available database systems as a back-end for multi-tenant applications when either the shared process or the shared table approach is used. Our multi-tenancy testbed, that simulates a multi-tenant CRM application, serves as a benchmark for the behavior of the DBMS instance as it handles more and more tables. Conventional on-line benchmarks such as TPC-C (Transaction Processing Performance Council, 2010) increase the load on the database until response time goals for various request classes are violated. In the same spirit, our experiment varies the number of tables in the database and measures the response time for various request classes.

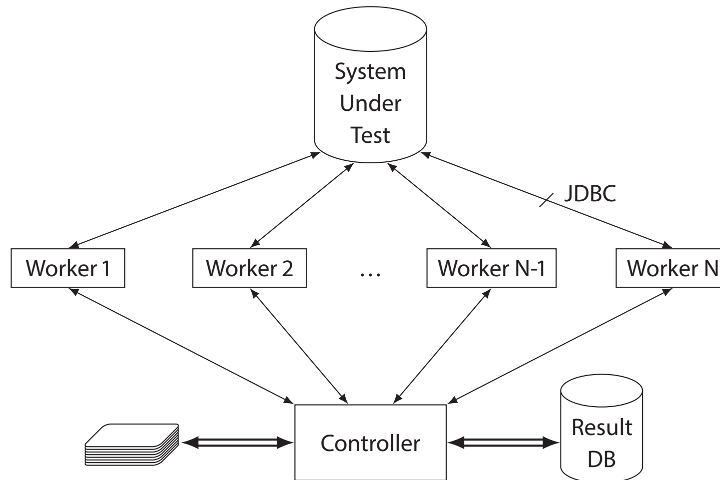


Figure 2.3.: Multi-Tenancy Testbed Architecture

2.2.1. Multi-Tenancy Testbed

Many SaaS applications have been developed leveraging already existing DBMS technology by introducing an intermediate layer enabling multi-tenancy and schema flexibility. Furthermore, some DBMS vendors provide integrated mechanisms for facilitating the deployment of multi-tenancy applications on their products.

For experimenting with various multi-tenant database implementations, we developed a configurable testbed with a well-defined workload simulating the OLTP component of a hosted CRM service. Conceptually, users interact with the service through browser and Web Service clients.

The testbed does not actually include the associated application servers, rather the clients simulate the behavior of those servers. The application is itself of interest because it characterizes a standard multi-tenant workload and thus could be used as the basis for a multi-tenant database benchmark.

Components

The testbed is composed of several processes as shown in Figure 2.3. The system under test is a multi-tenant database running on a private host. It can be configured for various schema layouts and usage scenarios. A worker process engages in multiple client sessions, each of which simulates the activities of a single connection from an application server's database connection pool. Each session runs in its own thread and gets its own connection to the target database. Per request the response time of the database is measured and reported to the controller. Multiple workers are distributed over multiple hosts.

The controller task assigns actions and tenants to workers. Following the TPC-C benchmark (Transaction Processing Performance Council, 2010), the controller creates a deck of *action cards* with a particular distribution, shuffles it, and deals cards to the workers. The controller also randomly selects tenants, with an equal distribution, and assigns one to each card. Finally, the controller collects response times and stores them in a result da-

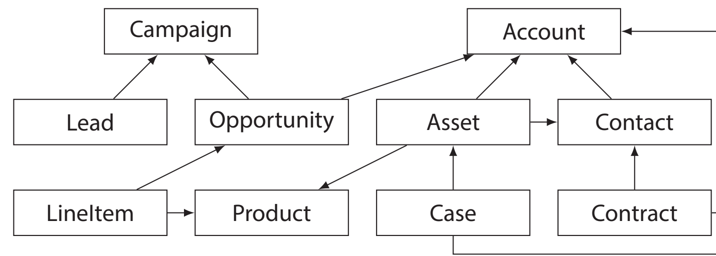


Figure 2.4.: CRM Application Schema

tabase. The timing of an action starts when a worker sends the first request and ends when it receives the last response.

Database Layout

The base schema for the CRM application contains ten tables as depicted in Figure 2.4. It is a classic DAG-structured OLTP schema with one-to-many relationships from child to parent. Individual users within a business (a tenant) are not modeled, but the same tenant may engage in several simultaneous sessions so data may be concurrently accessed. Every table in the schema has a tenant-id column so that it can be shared by multiple tenants.

Each of the tables contains about 20 columns, one of which is the entity's ID. Every table has a primary index on the entity ID and a unique compound index on the tenant ID and the entity ID. In addition, there are twelve indexes on selected columns for reporting queries and update tasks. All data for the testbed is synthetically generated.

Worker Actions

Worker actions include CRUD operations and reporting tasks that simulate the daily activities of individual users. The reporting tasks model fixed business activity monitoring queries, as they may be found on an application dashboard, rather than ad-hoc business intelligence queries, and are simple enough to run against an operational OLTP system. These ad-hoc reports perform queries with value aggregation and parent-child roll-ups. Worker actions also include administrative operations for the business as a whole, in particular, adding and deleting tenants. Depending on the configuration, such operations may entail executing DDL statements while the system is on-line, which may result in decreased performance or even deadlocks for some databases. The testbed does not model long-running operations because they should not occur in an OLTP system, particularly one that is multi-tenant.

To facilitate the analysis of the experimental results, the worker actions are grouped into classes with particular access characteristics and expected response times. Lightweight actions perform simple operations on a single entity or a small set of entities. Heavyweight actions perform more complex operations, such as those involving grouping, sorting, or aggregation, on larger sets of entities. The list of action classes (see Figure 2.5) specifies the distribution of actions in the controller's card deck. In contrast to TPC-C, the testbed does not model client think times.

Select Light (50%): Selects all attributes of a single entity or a small set of entities as if they were to be displayed on an entity detail page in the browser.

Select Heavy (15%): Runs one of five reporting queries that perform aggregation and/or parent-child-roll-up.

Insert Light (9.59%): Inserts one new entity instance into the database as if it had been manually entered into the browser.

Insert Heavy (0.3%): Inserts several hundred entity instances into the database in a batch as if they had been imported via a Web Service interface.

Update Light (17.6%): Updates a single entity or a small set of entities as if they had been modified in an edit page in the browser. The set of entities is specified by a filter condition that relies on a database index.

Update Heavy (7.5%): Updates several hundred entity instances that are selected by the entity ID using the primary key index.

Administrative Tasks (0.01%): Creates a new instance of the 10-table CRM schema by issuing DDL statements.

Figure 2.5.: Worker Action Classes

The testbed adopts a strategy for transactions that is consistent with best practices for highly-scalable Web applications (Kemper et al., 1998; Jacobs, 2005). The testbed assumes that neither browser nor Web Service clients can demarcate transactions and that the maximum granularity for a transaction is therefore the duration of a single user request. Furthermore, since long-running operations are not permitted, large write requests such as cascading deletes are broken up into smaller independent operations. Any temporary inconsistencies that result from the visibility of intermediate states must be eliminated at the application level. Finally, read requests are always performed with a weak isolation level that permits unrepeatable reads.

Database Drivers

To adapt the testbed to a variety of possible DBMS back-ends, the access to the database is handled by database drivers. These DBMS-specific drivers transform a request for a specific worker action into proprietary code. The database drivers access the DBMS by either a JDBC request or by calling the API of the product-specific library. For each worker session, one separate driver is instantiated. Each request which has been received by the worker triggers a database action by the driver.

Each driver is optimized for the supported DBMS and can thus leverage specialized techniques and features of the DBMS. This is necessary to exploit vendor-specific extensions to the standard feature set of a DBMS, such as XML features or special optimizations for sparse data.

Schema Variability	Number of instances	Tenants per instance	Total tables
0.0	1	10,000	10
0.5	5,000	2	50,000
0.65	6,500	1 – 2	65,000
0.8	8,000	1 – 2	80,000
1.0	10,000	1	100,000

Table 2.1.: Schema Variability and Data Distribution

2.2.2. Experiment: DBMS Performance with Many Tables

With the multi-tenancy testbed we measured the performance of traditional relational DBMSs as they handle more and more tables. In order to programmatically increase the overall number of tables without making them too synthetic, multiple copies of the 10-table CRM schema are created. Each copy should be viewed as representing a logically different set of entities. Thus, the more instances of the schema there are in the database, the more *schema variability* there is for a given amount of data. The testbed is configured with a fixed number of tenants—10,000—, a fixed amount of data per tenant—about 1.4 MB—, and a fixed workload—25 client sessions.

The schema variability takes values from 0 (least variability) to 1 (highest variability) as shown in Table 2.1. For the value 0, there is only one schema instance and it is shared by all tenants, resulting in 10 total tables. At the other extreme, the value 1 denotes a setup where all tenants have their own private instance of the schema, resulting in 100,000 tables. Between these two extremes, tenants are distributed as evenly as possible among the schema instances. For example, with schema variability 0.65, the first 3,500 schema instances have two tenants while the remaining ones have only one.

Testbed Configuration

For this experiment, the multi-tenancy testbed was configured as follows.

1. Each run has a duration of one hour, where the first 10 minutes (ramp-up phase) and the last 5 minutes (ramp-down phase) have been stripped. Thus, the performance is reported within a 45 minute steady-state interval.

2. The action classes are distributed as stated in Figure 2.5.

3. Each worker has a set of sessions, where each session manages a physical connection to the database. In our setup we used 5 workers with 5 sessions each. This simulates an application server with 25 connections within a connection pool. Since some JDBC drivers have optimizations when opening a lot of concurrent connections to the same database, we distributed the load across 5 workers. Furthermore, with this setup, we were able to stress the DBMS optimally.

We measured the response time of each query. Each worker uses an internal stopwatch to determine the runtime of a request. The response time is end-to-end, thus it contains submitting the request across the network, request processing within the DBMS, data transfer across the network, and data output at the client. We accessed all attributes of each result set entry to avoid unpredictable results if the DBMS-specific JDBC driver implements lazy fetching, where the driver avoids unnecessary network operations.

Results

The experiment was run on an IBM DB2 database server with a 2.8 GHz Intel Xeon processor and 1 GB of memory. The database server was running a recent enterprise-grade Linux operating system. The data was stored on an NFS appliance that was connected with dedicated 2 GBit/s Ethernet trunks. The workers were placed on blade servers with a 1 GBit/s private interconnect.

IBM DB2 allocates 4 KB of main memory per table within the database heap, as soon as the table is accessed for the first time. This memory is never freed. Additionally, there are references to file descriptors, etc. which are not part of that memory. The experiment was designed to exploit this issue. Increasing the schema variability beyond 0.5 taxes the ability of the database to keep the primary key index root nodes in memory. Schema variability 0.5 has 50,000 tables, which at 4 KB per table for DB2 consumes about 200 MB of memory. The operating system consumes about 100 MB, leaving about 700 MB for the database buffer pool. The page size for all user data, including indexes, is 8 KB. The root nodes of the 50,000 primary key indexes therefore require 400 MB of buffer pool space. The buffer pool must also accommodate the actual user data and any additional index pages, and the dataset for a tenant was chosen so that most of the tables need more than one index page.

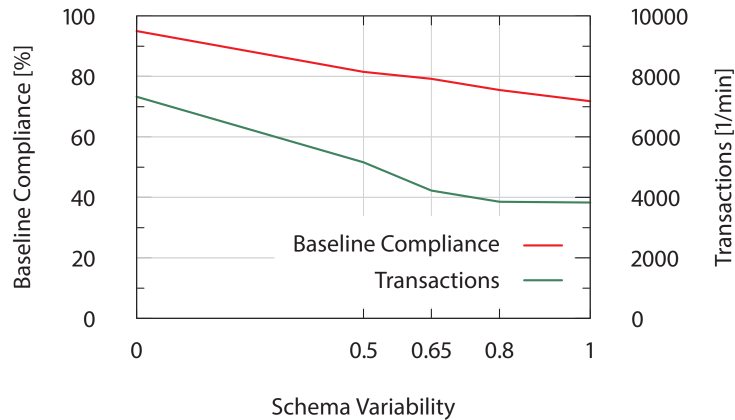
The raw data collected by the controller was processed as follows. First, the ramp-up phase during which the system reached steady state was stripped off. Then roll-ups of the results were taken across 30 minute periods for an hour, producing two runs. This process was repeated three times, resulting in a total of six runs. The results of the runs were consistent and so only the first run is reported for each value of the schema variability; see Table 2.2.

The first line of this table shows the *Baseline Compliance*, which was computed as follows. The 95% quantiles were computed for each query class of the schema variability 0.0 configuration: this is the baseline. Then for each configuration, the percentage of queries within the baseline were computed. The lower the baseline compliance, the higher the percentage of queries whose response time is above the baseline. Per definition, the baseline compliance of the schema variability 0.0 configuration is 95%. Starting around schema variability 0.65 the high response times are no longer tolerable. The baseline compliance is also depicted in Figure 2.6a. The second line of Table 2.2 is the database throughput in actions per minute, computed as an average over the 30 minute period. The throughput is also depicted in Figure 2.6a.

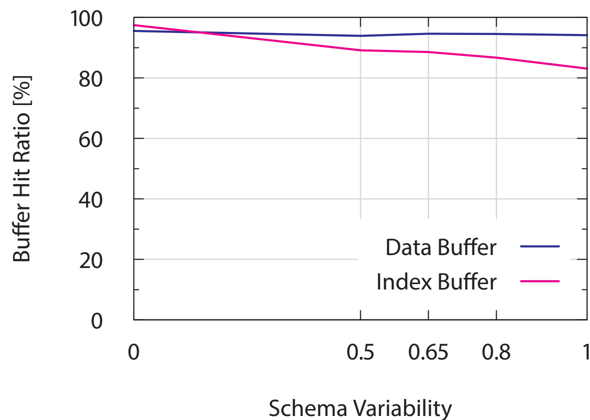
The middle part of Table 2.2 shows the 95% quantiles for each query class. For the most part, the response times grow with increasing schema variability. We hypothesize that the exceptions occur for the following reasons. First, for low schema variability, there is more

Metric			Schema Variability				
			0.0	0.5	0.65	0.8	1.0
Baseline Compliance		[%]	95.0	81.5	79.2	75.5	71.8
Throughput		[1/min]	7,325.60	5,162.30	4,225.17	3,852.70	3,829.40
95% Response Time	Select Light	[ms]	370	766	747	846	1,000
	Select Heavy	[ms]	2,226	1,677	1,665	1,959	2,375
	Insert Light	[ms]	4,508	2,031	2,620	3,020	2,005
	Insert Heavy	[ms]	8,530	10,128	13,383	16,681	9,718
	Update Light	[ms]	428	1,160	1,403	1,719	2,049
	Update Heavy	[ms]	679	1,393	1,524	1,777	2,096
Bufferpool Hit Ratio	Data	[%]	95.53	93.89	94.58	94.54	94.12
	Index	[%]	97.46	89.13	88.57	86.69	83.07

Table 2.2.: Experimental Results



(a) Baseline Compliance and Database Throughput



(b) Buffer Hit Ratios

Figure 2.6.: Schema Variability Results

sharing among tenants and therefore more contention for longer running queries and tuple inserts. Since the heavyweight select queries do aggregation, sorting, or grouping, multiple parallel query instances have impact on each other. The query execution plans show that these queries do a partial table scan with some locking, so the performance for this class degrades. For insert operations, the database locks the pages where the tuples are inserted, so concurrently running insert operations have to wait for the release of these locks. This effect can be seen especially for the lightweight insert operations. Second, there is a visible performance improvement for the insert operations at schema variability 1.0, where the database outperforms all previous configurations. We hypothesize that this behavior is due to the fact that DB2 is switching between the two insert methods it provides. The first method finds the most suitable page for the new tuple, producing a compactly stored relation. The second method just appends the tuple to the end of the last page, producing a sparsely stored relation.

The last two lines of Table 2.2 show the buffer pool hit ratio for the data and the indexes. As the schema variability increases, the hit ratio for indexes decreases while the hit ratio for data remains fairly constant. Inspection of the query plans shows that the queries primarily use the indexes for processing. The hit ratios are also depicted in Figure 2.6b.

2.3. Multi-Tenancy Issues with Current DBMS

As the previous experiment shows, there are two main factors which affect the scalability of traditional DBMSs when sharing a DBMS instance across tenants: buffer pool utilization and meta-data management. Furthermore, there is a third issue—online schema changes—which negatively affects the DBMS scalability. In the following we elaborate these issues and present short-term and long-term solutions.

2.3.1. Buffer Pool Utilization

Typically, the physical storage of a modern DBMS consists of several *data segments* which are slotted into *pages*. Each segment contains a header with information about the segment, such as high-watermarks and free-space information of the pages. A relation is usually stored across a set of such data segments, making a segment uniquely belonging to one particular relation. Thus, each page in a segment stores the tuples belonging to exactly one single relation.

With an increasing number of tables, the number of underutilized pages increase as well, leading to a lower buffer hit ratio for data pages. The previous experiment is very index-centric and therefore does not have a low buffer utilization for the data pages. However, other scenarios, where the workload is more data-centric, may show a different behavior.

Tree-based indexes are stored using the same mechanisms as described above. The pages do not contain data tuples, instead they contain one single node of the B-Tree. A very important property of a B-Tree is its balanced architecture combined with its high fan-out keeping the height of the tree very small. For guaranteeing fast index look-ups most DBMSs keep the root node of the B-Tree in the buffer pool, as soon as it has been accessed once. If the root node is kept in the buffer pool, the look-up costs are dramatically reduced

for subsequent look-ups in the same index. In the previous experiment, this effect becomes visible with a schema variability factor greater than 0.5. At this point, the DBMS is no longer able to keep all root pages of the primary key index in the buffer pool, thus leading to a severe performance degradation.

For the short term, the problem of underutilized buffer pools can be ameliorated by sharing tables across tenants. However, as discussed before, this lowers tenant isolation and simultaneously increases the risk of resource contention. For the long term, it should be possible to have tuples of various relations in the same page. However, this may negatively affect the paging behavior of current DBMSs and thus decreases the overall query performance. With the advent of terabyte-scale main-memories, this issue can be addressed by replacing disk-based DBMSs with main-memory-based DBMSs.

2.3.2. Meta-Data Management

As discussed above, each table which is known to the database system consumes at least 4 KB of main memory, as soon as it has been accessed once. This memory block contains the structural description of the table as well as the information for the physical access like file descriptors. Furthermore, since DB2 allocates this memory not inside the buffer pool, but inside the database heap, these per-table memory blocks are not affected by the DBMS's paging mechanism. Thus, once a table has been opened, its memory block is never removed until the whole database has been closed. For example, if 100,000 tenants are co-located on one DBMS instance and each tenant has 10 tables, then 4 GB of main memory are occupied by these memory blocks.

As a short term solution, the application of the shared table approach decreases the number of tables and thus decreases the memory requirements. However, this will lower tenant isolation. For the long term, there should be a schema inheritance mechanism, where the same structural description can be used by multiple table instances. As co-located tenants share the same application, this way, common parts of the schema can be factored out.

A second aspect is the way how schema redefinitions are handled. If an `ALTER TABLE` statement changes the data type of a particular attribute, the DBMS may need to perform data reorganizations or at least checks if all values in the affected column are compliant with the new data type. Such an operation may have severe effects on the query performance[2]: during the reorganization, the table becomes unavailable, thus lowering the SaaS application's availability. In the SaaS world this is crucial, as schema modification operations are part of the tenants' self-service and may be quite frequent.

Some DBMSs already address these issues. In the simplest form, checks and reorganizations can be avoided if "compatible" changes are specified in the `ALTER TABLE` statement. Such compatible changes are, for example, adding attributes, renaming attributes, or changing the data type of an attribute to a more generic data type, such as `VARCHAR(50)` to `VARCHAR(100)`. In such a scenario, the `ALTER TABLE` statement only changes the meta-data. However, the `ALTER TABLE` statement may gain an exclusive lock on the data dictionary, thus preventing concurrent requests from reading data dictionary entries.

[2] A detailed discussion of this issue can be found in Chapter 3.

Other mechanisms, like Oracle's Online Data Reorganization and Redefinition feature (Oracle Corporation, 2005), keep the affected table online: at first, a delta table is created in the same shape as the source table. During the reorganization process, all updates to the source table are redirected to the delta table. Then a target table is created that reflects the structural changes to the source table made by the `ALTER TABLE` statement. Next, the data is copied from the source table to the target table. Finally, the delta table is merged into the target table and source and delta tables are deleted. This procedure does not guarantee non-interruptible availability, but it reduces the downtime to a sub-second time window. However, it cannot be performed without the intervention of a DBMS administrator and are thus not suitable for SaaS setups.

3

Extensible Schemas for Traditional DBMS

The previous chapter shows that traditional DBMSs are not able to handle a multi-tenant workload out of the box. Besides the issues with buffer pool utilization and meta-data management, the DBMSs do not have the notion of tenants, thus they cannot leverage this knowledge for, e.g., query optimization.

Up to now, the fundamental implementation techniques as presented in the previous chapter do not offer any mechanisms for schema flexibility. In this chapter, we extend them with mechanisms for extensibility and evaluate, to what extent these implementations are suitable for schema evolution.

Currently available SaaS applications are composed of sophisticated persistence layers in the application to map the multi-tenant application to the relational world. Thus, the persistence layer needs to be built to overcome the issues with buffer pool utilization and meta-data management. We present schema mapping techniques which address this issue by mapping multiple single-tenant logical schemas in the SaaS application to one multi-tenant physical schema.

The presented schema mapping techniques are suitable for schema extensibility. Furthermore, some of them natively store semi-structured data inside the DBMS and thus have limited support for schema evolution.

3.1. Schema Mappings

We introduce the schema mapping techniques for multi-tenancy with a running example that shows various layouts for Account tables of three tenants with IDs 17, 35, and 42. Tenant 17 has an extension for the health care industry, while tenant 42 has an extension for the automotive industry; tenant 35 has no extension.

The most basic technique for implementing multi-tenancy is to add a tenant ID column (Tenant) to each table and share tables among tenants (cf. Section 2.1.3). This approach provides very good consolidation, but nevertheless, it does not support schema extensibility. As a result of the latter, it cannot represent the schema of our running example and is not further discussed here. This approach is generally taken by conventional Web applications, which view the data as being owned by the service provider rather than the individual tenants. As this approach is very limited, only simpler services on the left side of Figure 1.2 without the need for schema flexibility make use of it.

$Account_{17}$

Aid	Name	Hospital	Beds
1	Acme	St. Mary	135
2	Gump	State	1042

$Account_{35}$

Aid	Name
1	Ball

$Account_{42}$

Aid	Name	Dealers
1	Big	65

Figure 3.1.: Private Table Layout

We discuss several categories of schema mapping techniques, depending on the physical storage layout. All of these have in common that they support some aspects of schema flexibility. However, the experimental evaluation of these approaches shows, that some of them are more suitable for schema evolution than others. Furthermore, the approaches try to avoid stressing the meta-data budget of the DBMS by transferring actual meta-data into stored meta-data. This means that some of the meta-data has to be stored outside the DBMS to actually interpret the data tuple. In the example figures, gray columns denote such stored meta-data.

3.1.1. Schema Based Approaches

The first category of schema mapping techniques store the data in fully structured physical relations. That means, that the logical single-tenant schemas of the SaaS applications are mapped to fully-structured physical relations in the DBMS. As a consequence, the DBMS has the full schema information of the stored data.

Private Table Layout

The most basic way to support extensibility is to give each tenant their own private tables, as shown in Figure 3.1. Since the meta-data is entirely managed by the database, there is no overhead for meta-data in the data itself (i.e., gray columns are missing in the figure). However only moderate consolidation is provided since many tables are required. This approach is used by some larger services on the right side of Figure 1.2 where a small number of tenants can produce sufficient load to fully utilize the host machine.

Extension Table Layout

Figure 3.2 shows how the above layout evolves to the Extension Table Layout by splitting off the extensions into separate tables. Because multiple tenants may use the same extensions, the extension tables as well as the base tables should be given a `Tenant` column. A `Row` column must also be added so the logical source tables can be reconstructed. The two gray columns in Figure 3.2 represent the overhead for meta-data in the data itself, the stored meta-data.

At run-time, reconstructing the logical source tables carries the overhead of additional joins as well as additional I/O if the row fragments are not clustered together. On the other

Account_Ext			
Tenant	Row	Aid	Name
17	0	1	Acme
17	1	2	Gump
35	0	1	Ball
42	0	1	Big

Healthcare_Account			
Tenant	Row	Hospital	Beds
17	0	St. Mary	135
17	1	State	1042

Automotive_Account		
Tenant	Row	Dealers
42	0	65

Figure 3.2.: Extension Table Layout

Universal							
Tenant	Table	Col1	Col2	Col3	Col4	Col5	Col6
17	0	1	Acme	St. Mary	135	–	–
17	0	2	Gump	State	1042	–	–
35	1	1	Ball	–	–	–	–
42	2	1	Big	65	–	–	–

Figure 3.3.: Universal Table Layout

hand, if a query does not reference one of the tables, then there is no need to read it in, which can improve performance. This approach provides better consolidation than the Private Table Layout, however the number of tables will still grow in proportion to the number of tenants since more tenants will have a wider variety of basic requirements.

This approach has its origins in the Decomposed Storage Model, proposed by Copeland and Khoshafian (1985), where an n-column table is broken up into n 2-column tables that are joined through surrogate values. This model has then been adopted by column-oriented databases, for example MonetDB (Boncz, 2002), which leverage the ability to selectively read in columns to improve the performance of analytics (Stonebraker et al., 2005) and RDF data (Abadi et al., 2007). The Extension Table Layout does not partition tables all the way down to individual columns, but rather leaves them in naturally-occurring groups. This approach has been used to map object-oriented schemas with inheritance into the relational model (Elmasri and Navathe, 2006).

3.1.2. Generic Structures

In contrast to the previous category, the following schema mapping techniques do not store fully-structured data. Instead, the logical single-tenant schema gets shredded into generic structures which allow the creation of an arbitrary number of tables with arbitrary shapes. The schema information that is necessary to transform the fully structured logical scheme to the generic structure is available in the persistence layer of the SaaS application only and cannot be used for query optimization by the DBMS, though.

Universal Table Layout

A Universal Table, as depicted in Figure 3.3, is a generic structure with a Tenant column, a Table column, and a large number of generic data columns. The data columns have a flexible type, such as VARCHAR, into which other types can be converted. The n-th column of each logical source table for each tenant is mapped into the n-th data column of the Universal Table. As a result, different tenants can extend the same table in different ways. By keeping all of the values for a row together, this approach obviates the need to reconstruct the logical source tables. However it has the obvious disadvantage that the rows need to be very wide, even for narrow source tables, and the database has to handle many *NULL* values. While DBMSs handle *NULL* values fairly efficiently, they nevertheless occupy some additional memory. Perhaps more significantly, fine-grained support for indexing is not possible: either all tenants get an index on a column or none of them do. Furthermore, indexes have to cope with various data types in the Universal Table. Consider the Col3 attribute in the Universal Table depicted in Figure 3.3. As the application's persistence layer maps the String attribute Hospital of the healthcare extension to the same physical attribute as the Integer attribute Dealers of the automotive extension, the indexing mechanism must be enhanced to support such a setup. As a result of these issues, additional structures must be added to this approach to make it feasible.

This approach has its origins in the Universal Relation presented by Maier and Ullman (1983), which holds the data for all tables and has every column of every table. The Universal Relation was proposed as a conceptual tool for developing queries and was not intended to be directly implemented. The Universal Table described here is narrower, and thus feasible to implement, because it circumvents typing and uses each physical column to represent multiple logical columns.

There have been extensive studies of the use of generic structures to represent semi-structured data. Florescu and Kossmann (1999) describe a variety of relational representations for XML data including Universal and Pivot Tables. Our work uses generic structures to represent irregularities between pieces of schema rather than pieces of data.

Pivot Table Layout

A Pivot Table is a generic structure in which each field of each row in a logical source table is given its own row. Figure 3.4 shows that, in addition to Tenant, Table, and Row columns as described above, a Pivot Table has a Col attribute that specifies which source field a row represents and a single data-bearing column for the value of that field. The data column can be given a flexible type, such as VARCHAR, into which other types are converted. In such a case the Pivot Table becomes a Universal Table for the Decomposed Storage Model. A better approach however, in that it does not circumvent typing, is to have multiple Pivot Tables with different types for the data column. To efficiently support indexing, two Pivot Tables can be created for each type: one with indexes and one without. Each value is placed in exactly one of these tables depending on whether it needs to be indexed.

This approach eliminates the need to handle many *NULL* values. However it has more columns of meta-data than actual data and reconstructing an n-column logical source ta-

Pivot$_{int}$

Tenant	Table	Col	Row	Int
17	0	0	0	1
17	0	3	0	135
17	0	0	1	2
17	0	3	1	1042
35	1	0	0	1
42	2	0	0	1
42	2	2	0	65

Pivot$_{str}$

Tenant	Table	Col	Row	Str
17	0	1	0	Acme
17	0	2	0	St. Mary
17	0	1	1	Gump
17	0	2	1	State
35	1	1	0	Ball
42	2	1	0	Big

Figure 3.4.: Pivot Table Layout

ble requires $(n-1)$ aligning joins along the Row column. This leads to a much higher runtime overhead for interpreting the meta-data than the relatively small number of joins needed in the Extension Table Layout. Of course, like the Decomposed Storage Model, the performance can benefit from selectively reading in a small number of columns.

Grust et al. (2004) use Pivot-like Tables in their Pathfinder query compiler to map XML into relations. Closer to our work is the research on sparse relational data sets, which have thousands of attributes, only a few of which are used by any object. Agrawal et al. (2001) compare the performance of Pivot Tables (called vertical tables) and conventional horizontal tables in this context and conclude that the former perform better because they allow columns to be selectively read in. Our use case differs in that the data is partitioned by tenant into well-known dense subsets, which provides both a more challenging baseline for comparison as well as more opportunities for optimization. Beckmann et al. (2006) also present a technique for handling sparse data sets using a Pivot Table Layout. In comparison to our explicit storage of meta-data columns, they chose an "intrusive" approach which manages the additional runtime operations in the database kernel. Cunningham et al. (2004) present an "intrusive" technique for supporting general-purpose pivot and unpivot operations.

Chunk Table Layout

We propose a third generic structure, called Chunk Table, that is particularly effective when the base data can be partitioned into well-known dense subsets. A Chunk Table, as shown in Figure 3.5, is like a Pivot Table except that it has a set of data columns of various types, with and without indexes, and the Col column is replaced by a Chunk column. A logical source table is partitioned into groups of columns, each of which is assigned a chunk ID and mapped into an appropriate Chunk Table. In comparison to Pivot Tables, this approach reduces the ratio of stored meta-data to actual data as well as the overhead for reconstruct-

$Chunk_{int|str}$

Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	1	Acme
17	0	1	0	135	St. Mary
17	0	0	1	2	Gump
17	0	1	1	1042	State
35	1	0	0	1	Ball
42	2	0	0	1	Big
42	2	1	0	65	–

Figure 3.5.: Chunk Table Layout

$Account_{Row}$

Tenant	Row	Aid	Name
17	0	1	Acme
17	1	2	Gump
35	0	1	Ball
42	0	1	Big

$Chunk_{Row}$

Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	135	St. Mary
17	0	0	1	1042	State
42	2	0	0	65	–

Figure 3.6.: Chunk Folding Layout

ing the logical source tables. In comparison to Universal Tables, this approach provides a well-defined way of adding indexes, breaking up overly-wide columns, and supporting typing. By varying the width of the Chunk Tables, it is possible to find a middle ground between these extremes. On the other hand, this flexibility comes at the price of a more complex query transformation layer.

Chunk Folding

We propose a technique called Chunk Folding where the logical source tables are vertically partitioned into chunks that are folded together into different physical multi-tenant tables and joined as needed. The database's "meta-data budget" is divided between application-specific conventional tables and a large fixed set of Chunk Tables. For example, Figure 3.6 illustrates a case where base attributes of the Account table are stored in a conventional table and all its extension attributes are placed in a single Chunk Table. In contrast to generic structures that use only a small, fixed number of tables, Chunk Folding attempts to exploit the database's entire meta-data budget in as effective a way as possible. Good performance is obtained by mapping the most heavily-utilized parts of the logical schemas into the conventional tables and the remaining parts into Chunk Tables that match their structure as closely as possible.

Account						
Tenant	Aid	Name	SPARSE			
17	1	Acme	Hospital	St. Mary	Bed	135
17	2	Gump	Hospital	State	Bed	1042
35	1	Ball				
42	1	Big	Dealer	65		

Figure 3.7.: Sparse Columns Layout

3.1.3. Semistructured Approaches

Although each individual logical single-tenant schema is fully-structured, the aggregate schema of all tenants is semi-structured due to the heterogeneity of extensions the tenants have subscribed. The following two approaches exploit this property by storing a fully-structured version of the common parts and a semi-structured version of each tenants' extensions.

Sparse Columns

Sparse Columns were originally developed to manage data such as parts catalogs where each item has only a few out of thousands of possible attributes. Storing such data in conventional tables with *NULL* values can decrease performance even with advanced optimizations for *NULL* handling. To implement Sparse Columns, SQL Server 2008 uses a variant of the Interpreted Storage Format (Beckmann et al., 2006; Chu et al., 2007), where a value is stored in the row together with an identifier for its column.

In our mapping for SaaS, the base tables are shared by all tenants and every extension field of every tenant is added to the corresponding base table as a Sparse Column, as illustrated in Figure 3.7. Sparse columns must be explicitly defined by a `CREATE/ALTER TABLE` statement in the DDL and, in this sense, are owned by the database. Nevertheless, the application must maintain its own description of the extensions, since the column names cannot be statically embedded in the code. The implementation of Sparse Columns in SQL Server 2008 requires special handling when accessing the data. For writes, the application must ensure that each tenant uses only those columns that they have declared, since the name-space is global to all tenants. For reads, the application must do an explicit projection on the columns of interest, rather than doing a `SELECT *`, to ensure that *NULL* values are treated correctly. In all other cases, the Sparse Columns approach is fully transparent to the SaaS application; no special care has to be taken when generating the SQL queries.

Sparse Columns requires only a small, fixed number of tables, which gives it a performance advantage over Private Tables; the experiment in the previous chapter shows that having many tables negatively impacts performance. On the other hand, there is some overhead for managing Sparse Columns. As an example, the SQL Server 2008 documentation recommends using a Sparse Column for an INT field only if at least 64% of the values are *NULL* (Microsoft Corporation, 2008).

Account			
Tenant	Aid	Name	Ext_XML
17	1	Acme	<ext> ↩ <hospital>St. Mary</hospital> ↩ <beds>135</beds> ↩ </ext>
17	2	Gump	<ext> ↩ <hospital>State</hospital> ↩ <beds>1042</beds> ↩ </ext>
35	1	Ball	
42	1	Big	<ext> ↩ <dealers>65</dealers> ↩ </ext>

Figure 3.8.: pureXML Columns Layout

IBM pureXML

According to Saracco et al. (2006), IBM pureXML was designed to allow processing of semi-structured data alongside of structured relational data. Although the extension schema is fully-structured for each individual tenant, the global schema across all tenants is semi-structured. Therefore, Taylor and Guo (2007) give recommendations on how to use pureXML for extensible SaaS applications. Traditionally, XML documents have either been *shredded* into multiple relations or have been stored as `CLOB` attributes. When the XML document has been stored as `CLOB`, it has to be fully retrieved from the DBMS as a single object and then processed in the application, without any ability to fetch only the substantial parts of the document. IBM pureXML provides server-side XQuery mechanisms which allows for extracting the relevant part of an XML document. Furthermore, indexes on XML documents are not possible. This limitation is exploited by introducing XML shredding, however, a shredded XML document cannot be stored alongside the base tables and thus make querying the data more error-prone.

With pureXML, the XML documents are stored in a single attribute of any table, thus the base tables are shared by all tenants and each base table is augmented by a column (Ext_XML) that stores all extension fields for a tenant in a flat XML document, as illustrated in Figure 3.8. Since these documents necessarily vary by tenant, they are untyped. This representation keeps the documents as small as possible, which is an important consideration for performance (Nicola, 2008).

HBase

HBase (2010), which is an open source version of Google BigTable (Chang et al., 2008), was originally designed to support the exploration of massive web data sets. These systems are increasingly being used to support enterprise applications in a SaaS setting (Bernstein, 2008).

In an HBase table, columns are grouped into *column families*. Column families must be explicitly defined in advance in the HBase "DDL"; for this reason they are owned by the database. There should not be more than tens of column families in a table and they should

Row Key	Account	Contact
17Act1	[name:Acme, hospital:St. Mary, beds:135]	
17Act2	[name:Gump, hospital:State, beds:1042]	
17Ctc1		[...]
17Ctc2		[...]
35Act1	[name:Ball]	
35Ctc1		[...]
42Act1	[name:Big, dealers:65]	

Figure 3.9.: HBase Pivot Columns Layout

rarely be changed while the system is in operation. Columns within a column family may be created on-the-fly, hence they are owned by the application. Different rows in a table may use the same column family in different ways. All values in a column are stored as Strings. There may be an unbounded number of columns within a column family.

Data in a column family is stored together on disk and in memory. Thus, a column family is essentially a Pivot Table; each value is stored along with an identifier for its column in a tall narrow table (Agrawal et al., 2001).

HBase was designed to scale out across a large farm of servers. Rows are range-partitioned across the servers by key. Applications define the key structure, therefore implicitly control the distribution of data. Rows with the same key prefix will be adjacent but, in general, may end up on different servers. The rows on each server are physically broken up into their column families.

The mapping we use for SaaS is illustrated in Figure 3.9. In keeping with best practices for HBase, this mapping ensures that data that is likely to be accessed within one query is clustered together. A single HBase table is used to store all tables for all tenants. The physical row key in HBase consists of the concatenation of the tenant ID, the name of the logical table, and the key of the row in the logical table. Each logical table is packed into its own column family, thus each row has values in only one column family. Within a column family, each column in the logical table is mapped into its own physical HBase column. Thus, since columns are dynamic, tenants may individually extend the base tables.

3.2. Case Study: The force.com Multi-Tenant Application Platform

The application development platform *force.com* uses some of the above schema mapping techniques in their product. Rather than being a SaaS application, force.com offers *Platform as a Service* (PaaS) to ISVs for developing and hosting multi-tenant SaaS applications. The force.com platform emerged from the CRM application *salesforce.com* which offers interfaces to ISVs for extending the CRM application. The extensions are available via *AppExchange* where individual tenants can subscribe to the extensions. In order to develop other applications than CRM extensions, ISVs can use the PaaS force.com.

Weissman and Bobrowski (2009) describe the full architecture of force.com, including application development and query optimization. In this section, we present a summary of their work, focussed on meta-data, schema mapping, and data persistence. It serves as an example on how to apply schema mapping techniques to a real world application.

3.2.1. Meta-Data-Driven Applications

The force.com platform pursues a meta-data-driven approach for application development. Thus, each component, like forms, reports, work flows, privileges, customizations, business rules, data tables, and indexes, is meta-data. The platform itself separates these meta-data describing either the base functionality of the ISV's application or the tenant's data and customizations, from the compiled runtime environment (kernel) and the application data. This separation allows for updating the system kernel, the base application, and the tenants' customizations independently.

All objects that are known to the force.com platform only exist virtually in the runtime environment. Virtual data tables containing tenants' data and indexes are mapped to the underlying DBMS' physical tables by a sophisticated schema mapping which combines multiple of the above discussed techniques. The meta-data is not available to the DBMS, so rather than managing the data, the DBMS simply stores the data, degenerating to a 'dumb data repository'. To reduce the I/O load, the meta-data is cached in main memory.

For each access to a virtual table, the runtime environment has to retrieve the appropriate meta-data and then transform the physical representation to a virtual table. This transformation is done by generating a query which accesses the underlying DBMS and casts the physical representation into the logical representation. The runtime environment has an internal multi-tenant-aware query optimizer that leverages the knowledge of the meta-data for generating optimal queries to the DBMS. Statistics for this cost-based optimizer are collected internally. This way, the optimizer and the statistic component of the DBMS is no longer used.

3.2.2. Data Persistence

For data persistence, the force.com platform uses several schema mapping techniques in combination. There are two kinds of physical tables. First there are data tables which serve as "data heap" for storing tenants' data, either structured data or CLOBs. Second, there are index tables for storing data which have a tenant-defined index on it. Beside these two kinds, there are specialized tables for storing the meta-data of the application. Although these meta-data is stored inside the database, the DBMS is not able to make use of it since they are application-specific and not database-specific. Changes on the tenants' virtual schema, such as creating new tables or changing attributes, modifies the meta-data only. The physical representation in the DBMS is not touched when performing such operations.

The data table is a Universal Table which stores data accessible by the application. There is only one big data table storing all entities of various types. The force.com variant of the Universal Table differs from the one in Section 3.1.2 in the following way. The objects are identified by a Global Unique Identifier (GUID) which serves as primary key. A unique compound key containing the tenant ID, the object ID and the virtual table's name serves as secondary key. The compound key contains the natural name of the entity, like "Account". Actual data is stored in variable length columns with generic names Val0,...,Val500. Mapping between the logical name and the ValN column is done within the meta-data. The ValN columns are of type VARCHAR, so arbitrary types have to be cast to store/retrieve data

in/from VARCHAR. A single ValN column can manage information from multiple fields, as long as the field stems from a different object. Customizations only change meta-data, that maps the virtual attribute to physical ValN columns. There is no online DDL when a customization is performed. Everything is handled within the meta-data.

Besides the data table, there are specialized pivot tables for storing unstructured data as CLOBs. In contrast to the Chunk Folding approach, the alignment is done using the GUID. Additional pivot tables simulate indexes, unique constraints and foreign keys. Native indexes for ValN columns are not practical since they may contain data of various virtual tables in various data types. force.com manages indexes by synchronously copying field data, that have been selected for indexing within the meta-data, to a pivot table. Thus, the index pivot table contains strongly typed, natively indexed columns. It contains one column per supported native type. When a virtual table is accessed, the internal query generator first queries the pivot table for each indexed attribute, and then aligns the result with the appropriate tuples from the data table. As an optimization, the force.com platform employs native DBMS partitioning mechanisms to physically partition the data and pivot tables by tenant.

3.3. Request Processing

The employment of schema mapping techniques leads to more complex request processing as data is now represented using two different schemas. Inside the SaaS application, tenants' data is represented using the logical, tenant-specific schema, while inside the DBMS back-end, the physical schema is used. Thus, the query generation inside the SaaS persistence layer must generate queries against the physical layer of the DBMS which reassembles the tuple in the logical schema. This section presents how data can be accessed when certain schema mappings are used.

3.3.1. Chunk Tables

Instead of presenting the request processing steps for all generic approaches from Section 3.1.2, we mainly consider Chunk Tables in this section. Depending on the width w of a Chunk Table, where w denotes the number of attributes, it may mimic a Pivot Table ($w = 1$) or a Universal Table ($w \to \infty$). Thus, the usage of the presented processing steps together with the Pivot Table and the Universal Table approaches is straight-forward.

Depending on the implementation, data casts may be necessary—especially when using the Universal Table layout. However, the data query process, as described in the following, is independent of using data casts.

Chunk Folding mixes Extension and Chunk Tables. The inclusion of Extension Tables does not affect the query part of the following section at all. The reason is that the only interface between the different tables is the meta-column Row, which is also available in the Extension Table Layout.

Transforming Queries

Consider the following query from Tenant 17 over the Private Table Layout as shown in Figure 3.1:

```
SELECT Beds
  FROM Account17                                        (Q1)
 WHERE Hospital = 'State'
```

The most generic approach to formulating this query over the Pivot Tables Pivot_{int} and Pivot_{str} is to reconstruct the original Account_{17} table in the FROM clause and then patch it into the selection. Such table reconstruction queries generally consists of multiple equi-joins on the column Row. In the case of Account_{17}, three aligning self-joins on the Pivot table are needed to construct the four-column wide relation. However in Query Q1, the columns Aid and Name do not appear and evaluation of two of the three mapping joins would be wasted effort.

We therefore devise the following systematic compilation scheme that proceeds in four steps.

1. Collect all table names and their corresponding columns in the logical source query.

2. For each table name, obtain the Chunk Tables and the meta-data identifiers that represent the used columns[1].

3. For each table, generate a query that filters the correct columns (based on the meta-data identifiers from the previous step) and aligns the different chunk relations on their Row columns. The resulting queries are all flat and consist of conjunctive predicates only.

4. Extend each table reference in the logical source query by its generated *table definition* query.

Query Q1 uses the columns Hospital and Beds of table Account_{17}. The two columns can be found in relations Pivot_{str} and Pivot_{int}, respectively. For both columns, we know the values of the Tenant, Table, and Col columns. The query to reconstruct Account_{17} checks all these constraints and aligns the two columns on their rows:

```
(SELECT s.Str as Hospital, i.Int as Beds
   FROM Pivotstr s, Pivotint i
  WHERE s.Tenant = 17
    AND i.Tenant = 17                                   (Q1Account17(P))
    AND s.Table = 0 AND s.Col = 2
    AND i.Table = 0 AND i.Col = 3
    AND s.Row = i.Row)
```

[1] To make Chunk Folding work, also the meta-data look-up for Extension Tables has to be performed.

When using Chunk Tables instead of Pivot Tables, the reconstruction of the logical Account_{17} table is nearly identical to the Pivot Table case. In our example, the resulting reconstruction query is particularly simple because both requested columns reside in the same chunk (`Chunk = 1`):

```
(SELECT Str1 as Hospital,
        Int1 as Beds
   FROM Chunk_int|str
  WHERE Tenant = 17
    AND Table = 0
    AND Chunk = 1)
```
$(\text{Q1}_{\text{Account}_{17}}(\text{C}))$

To complete the transformation, Query $\text{Q1}_{\text{Account}_{17}}(\text{C})$ is then patched into the `FROM` clause of Query Q1 as a nested sub-query:

```
SELECT Beds
  FROM (SELECT Str1 as Hospital,
               Int1 as Beds
          FROM Chunk_int|str
         WHERE Tenant = 17
           AND Table = 0
           AND Chunk = 1) AS Account_17
 WHERE Hospital = 'State'
```
$(\text{Q1}_{\text{Chunk}})$

The structural changes to the original query can be summarized as follows.

- An additional nesting due to the expansion of the table definitions is introduced.
- All table references are expanded into join chains on the base tables to construct the references.
- All base table accesses refer to the columns Tenant, Table, Chunk, and in case of aligning joins, to column Row.

We argue in the following that these changes do not necessarily need to affect the query response time.

Additional Nesting Fegaras and Maier (2000, Rule N8) proved that the nesting we introduced in the `FROM` clause—queries with only conjunctive predicates—can always be flattened by a query optimizer. If a query optimizer does not implement such a rewrite it will first generate the full relation before applying any filtering predicates—a clear performance penalty. For such databases, we must directly generate the flattened queries. For more complex queries (with e.g., `GROUP BY` clauses) the transformation is however not as clean as the technique described above.

Join Chains Replacing the table references by chains of joins on base tables may be beneficial as long as the costs for loading the chunks and applying the index-supported (see below) join are cheaper than reading the wider conventional relations. The observation that different chunks are often stored in the same relation (as in Figure 3.5) makes this scenario even more likely as the joins would then turn into self-joins and we may benefit from a higher buffer pool hit ratio.

Base Table Access As all table accesses refer to the meta-data columns Tenant, Table, Chunk, and Row we should construct indexes on these columns. This turns every data access into an index-supported one. Note that a B-Tree index look-up in a (Tenant, Table, Chunk, Row) index is basically a partitioned B-Tree look-up (Graefe, 2003). The leading B-Tree columns (here Tenant and Table) are highly redundant and only partition the B-Tree into separate smaller B-Trees (partitions). Prefix compression makes sure that these indexes stay small despite the redundant values.

Transforming Statements

The presented compilation scheme only covers the generation of queries over Chunk Tables. Thus, this section briefly describes how to cope with `UPDATE`, `DELETE`, and `INSERT` statements.

In SQL, data manipulation operations are restricted to single tables or update-able selections/views which the SQL query compiler can break into separate DML statements. For update and delete statements, predicates can filter the tuples affected by the manipulation. As insert and delete operations also require the modification of stored meta-data values, we devised a consistent DML query transformation logic based on single table manipulations.

Since multiple chunked tables are required for a single source table, a single source DML statement generally has to be mapped into multiple statements over Chunk Tables. Following common practice (Hamilton, 2007), we transform delete operations into updates that mark the tuples as invisible instead of physically deleting them, in order to provide a soft-delete mechanism. Such an update naturally has to mark all Chunk Tables as deleted in comparison to normal updates that only have to manipulate the chunks where at least one cell is affected.

Our DML transformation logic for updates (and thus also for deletes) divides the manipulation into two phases: (a) a query phase that collects all physical rows that are affected by an update and (b) an update phase that applies the update for each affected chunk with *local* conditions on the meta-data columns and especially column Row only. Phase (a) transforms the incoming query with the presented query transformation scheme into a query that collects the set of affected Row values. One possibility to implement the updates in Phase (b) is to nest the transformed query from Phase (a) into a nested sub-query using an `IN` predicate on column Row. This approach lets the database execute all the work. For updates with multiple affected chunks (e.g., deletes) the database however has to evaluate the query from Phase (a) for each chunk relation. An alternative approach would be to first evaluate the transformed predicate query, let the application then buffer the result and issue an atomic update for each resulted row value and every affected Chunk Table.

We now consider insert statements. For any insert, the application logic has to look up all related chunks, collect the meta-data for tables and chunks, and assign each inserted new row a unique row identifier. With the complete set of meta-data in hand, an insert statement for any chunk can be issued.

Other operations like DROP or ALTER statements can be evaluated on-line as well. They however require no access to the database. Instead only the application logic has to do the respective bookkeeping.

3.3.2. IBM pureXML

IBM pureXML offers SQL/XML, a hybrid query language that provides native access to both the structured and semi-structured representations. As the SaaS application manipulates data in the structured format, accessing extension data requires a correlated subquery to manage the XML. This sub-query extracts the relevant extension fields using the XMLTABLE function which converts an XML document into a tabular format using XQuery. The query with the XMLTABLE function has to be generated client- and query-specific to access clients' extension fields relevant in the particular query.

Since predicate handling in SQL/XML is different for predicates on relational attributes and XML attributes, we defer the transformation of Query Q1 into SQL/XML. Instead, we introduce Query Q2 which retrieves a particular entry from Tenant 42's Account_{42} table.

```
SELECT Aid, Name, Dealers
  FROM Account42                                  (Q2)
 WHERE Aid = 1
```

This query has the selection predicate Aid = 1 which can be evaluated using the relational base attribute Aid of the Account table's pureXML representation (cf. Figure 3.8). The transformed query $\text{Q2}_{\text{pureXML}}$ is as follows:

```
SELECT Aid, Name, Dealers
  FROM Account,
       XMLTABLE('$i/ext'
                PASSING Ext_XML AS "i"
                COLUMNS                           (Q2pureXML)
                  Dealers INTEGER PATH 'dealers'
       )
 WHERE Tenant = 42 AND Aid = 1
```

For each tuple from the Account table matching the relational predicate Tenant = 42 AND Aid = 1, the Ext_XML attribute is passed to XMLTABLE table function. Inside this function, the XML document is bound to the variable $i. The COLUMNS stanza then maps relative XPath expressions (e.g., PATH 'dealers') to typed attributes (e.g., Dealers INTEGER). Finally, the XMLTABLE function returns a relational view on the XML document, projected to the attributes in the COLUMNS stanza.

Figure 3.10 shows the associated query plan. Since rows are mostly accessed through relational base fields, there is no need to use the special XML indexes offered by pureXML as described by Saracco et al. (2006).

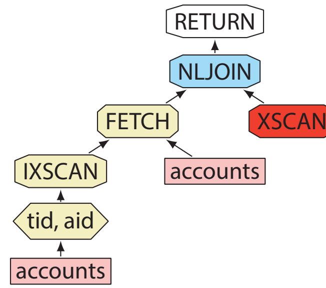


Figure 3.10.: Query Execution Plan for Query $Q2_{pureXML}$

However, the filter predicate of Query Q1 is comprised of an attribute stored in the XML document associated with each tuple, rather than a relational attribute. For the sake of convenience, Query Q1 is repeated below.

```
SELECT Beds
  FROM Account17
 WHERE Hospital = 'State'
```
(Q1)

Although an XML filter attribute can be evaluated by either specifying an appropriate XQuery expression within the XMLTABLE function[2] or by adding a predicate on the XML view columns[3], Nicola (2008) recommends to use the XMLEXISTS predicate function instead for performance reasons. The XMLEXISTS predicate function returns true, if an XQuery expression returns a sequence of one or more items; otherwise, false is returned. Thus, the transformed Query $Q1_{pureXML}$ is as follows:

```
SELECT Beds
  FROM Account,
       XMLTABLE('$i/ext'
                 PASSING Ext_XML AS "i"
                 COLUMNS
                   Beds INTEGER PATH 'beds'
       )
 WHERE Tenant = 17
   AND XMLEXISTS('$i/ext[hospital = "State"]'
                  PASSING Ext_XML AS "i"
       )
```
($Q1_{pureXML}$)

Hence, for transforming arbitrary queries to SQL/XML queries, the following steps have to be performed:

1. Collect all table names and their corresponding columns, i.e. relational attributes as well as attributes in the XML document, in the logical source query.

[2] For example, `$i/ext[Hospital = "State"]`
[3] For example, `... AND Hospital = 'State'`

Parent

Id	Col1	Col2	…	Col90

Child

Id	Parent	Col1	Col2	…	Col90

Figure 3.11.: Test Layout: Conventional

2. For each table, generate an XMLTABLE call which binds the Ext_XML attribute and extracts the XML attributes out of it.

3. If a filter predicate contains XML columns, transform the predicate to an XML predicate using the XMLEXISTS predicate function.

To insert a new tuple with extension data, the application has to generate the appropriate XML document. Updates to extension fields are implemented using XQuery 2.0 features to modify documents in place.

3.4. Chunk Table Performance

To assess the query performance of standard databases on queries over Chunk Tables, we devise a simple experiment that compares a conventional layout with equivalent Chunk Table layouts of various widths. The first part of this section will describe the schema and the query we use, the second part outlines the individual experiments.

3.4.1. Test Schema

The schema for the conventional layout consists of two tables Parent and Child, as shown in Figure 3.11. Both tables have an Id column and 90 data columns that are evenly distributed between the types INTEGER, DATE, and VARCHAR(100). In addition, table Child has a foreign key reference to Parent in column Parent.

The Chunk Table layouts each have two chunk tables: $Chunk_{Data}$ storing the grouped data columns and $Chunk_{Index}$ storing the key Id and foreign key Parent columns of the conventional tables. In the different Chunk Table layouts, the $Chunk_{Data}$ table varied in width from 3 data columns (resulting in 30 groups) to 90 data columns (resulting in a single group) in 3 column increments. Each set of three columns has types INTEGER, DATE, and VARCHAR(100) allowing groups from the conventional table to be tightly packed into the Chunk Table. In general, the packing may not be this tight and a Chunk Table may have *NULL* values, although not as many as a Universal Table. The $Chunk_{Index}$ table always had a single INTEGER column.

As an example, Figure 3.12 shows a Chunk Table instance of width 6 where each row of a conventional table is split into 15 rows in $Chunk_{Data}$ and 1 (for parents) or 2 (for children) rows in $Chunk_{Index}$.

Chunk$_{Data}$

Table	Chunk	Row	Int1	Int2	Date1	Date2	Str1	Str2

Chunk$_{Index}$

Table	Chunk	Row	Int1

Figure 3.12.: Test Layout: Chunk$_6$

For the conventional tables, we create indexes on the primary keys (Id) and the foreign key (Parent, Id) in the Child table. For the chunked tables, we create (Table, Chunk, Row) indexes on all tables[4] as well as an (Int1, Table, Chunk, Row) index on Chunk$_{Index}$ to mimic the foreign key index on the Child table.

The tests use synthetically generated data for the individual schema layouts. For the conventional layout, the Parent table is loaded with 10,000 tuples and the Child table is loaded with 100 tuples per parent (1,000,000 tuples in total). The Chunk Table Layouts are loaded with equivalent data in fragmented form.

3.4.2. Test Query

Our experiments use the following simple selection query.

```
SELECT p.Id, ...
  FROM Parent p, Child c
 WHERE p.Id = c.Parent
   AND p.Id = ?
```
(Q3)

Query Q3 has two parameters: (a) the ellipsis (...) representing a choice of data columns and (b) the question mark (?) representing a random parent id. Parameter (b) ensures that a test run touches different parts of the data. Parameter (a)—the *Q3 scale factor*—specifies the width of the result. The number of the ColN attributes chosen from the parent table is equal to the number of the attributes chosen from the child table. As an example, Query $Q3_3$ is Query Q3 with a scale factor of 3: the ellipsis is replaced by 3 data columns each for parent and child.

```
SELECT p.Id, p.Col1, p.Col2, p.Col3,
       c.Col1, c.Col2, c.Col3
  FROM Parent p, Child c
 WHERE p.Id = c.Parent
   AND p.Id = ?
```
($Q3_3$)

Higher Q3 scale factors (ranging up to 90) are more challenging for the chunked representation because they require more aligning joins.

[4] cf. *Base Table Access* on page 38

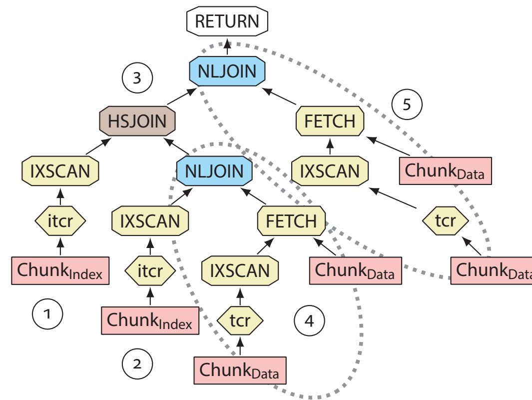


Figure 3.13.: Query Execution Plan for Simple Fragment Query

3.4.3. Transformation and Nesting

In our first test, we transformed Query Q3 using the methods described in Section 3.3.1 and fed the resulting queries into the open-source database MySQL (mysql.com, 2010) and the commercial database IBM DB2. We then used the database debug/explain facilities to look at the compiled query plans. The MySQL optimizer was unable to flatten the nesting introduced by our query transformation. DB2 on the other hand presented a totally unnested plan where the selection predicate on `p.Id` was even pushed into the chunk representing the foreign key of the child relation. DB2's evaluation plan is discussed in more detail in the next test.

We then flattened the queries in advance and studied whether the predicate order on the SQL level would influence the query evaluation time. We produced an ordering where all predicates on the meta-data columns preceded the predicates of the original query and compared it with the ordering that mimics DB2's evaluation plan. For MySQL, the latter ordering outperformed the former ordering by a factor of 5.

After adjusting the query transformation to produce flattened queries with predicates in the optimal order, we reran the experiment on both database systems. DB2 produced the same execution plan as before and MySQL was able to produce a plan that started with the most selective predicate (`p.Id = ?`). As one would expect, the query evaluation times for MySQL showed an improvement.

3.4.4. Transformation and Scaling

To understand how queries on Chunk Tables behave with an increasing number of columns (output columns as well as columns used in predicates) we analyzed the plans for a number of queries. The pattern is similar for most queries and we will discuss the characteristics based on Query $Q3_3$, which was designed for the Chunk Table Layout in Figure 3.12.

The query plan is shown in Figure 3.13[5]. The leaf operators all access base tables. If the base tables are accessed via an index, an IXSCAN operator sits on top of the base table

[5] We chose to present IBM DB2's plan. MySQL's query plan has the same characteristics

with a node in between that refers to the used index. Here, the meta-data index is called tcr (abbreviating the columns Table, Chunk, and Row) and the value index is called itcr. If a base table access cannot be completely answered by an index (e.g., if data columns are accessed) an additional FETCH operator (with a link to the base table) is added to the plan. Figure 3.13 contains two different join operators: a hash join (HSJOIN) and an index nested-loop join (NLJOIN).

The plan in Figure 3.13 can be grouped into 5 regions. In region ①, the foreign key for the child relation is looked up. The index access furthermore applies the aforementioned selection on the ? parameter. In region ②, the Id column of the parent relation is accessed and the same selection as for the foreign key is applied. The hash join in region ③ implements the foreign key join `p.Id = c.Parent`. But before this value-based join is applied in region ④, all data columns for the parent table are looked up. Note that region ④ expands to a chain of aligning joins where the join column Row is looked up using the meta-data index tcr if parent columns in different chunks are accessed in the query. A similar join chain is built for the columns of the child table in region ⑤.

3.4.5. Response Times with Warm Cache

Figure 3.14 shows the average execution times *with warm caches* on DB2 V9.1 on a 2.8 GHz Intel Xeon processor with 1 GB RAM. We conducted 10 runs; for all of them, we used the same values for parameter ? so the data was in memory. In this setting, the overhead compared to conventional tables is entirely due to computing the aligning joins. The queries based on narrow chunks have to perform up to 60 more joins (layout Chunk_3 with Q3 scale factor 90) than the queries on the conventional tables which results in a 35 ms slower response time. Another important observation is that already for 15-column wide chunks, the response time is cut in half in comparison to 3-column wide chunks and is at most 10 ms slower than conventional tables.

3.4.6. Logical Page Reads

Figure 3.15 shows the number of logical data and index page reads requested when executing Query Q3. For all chunked representations, 74% to 80% of the reads were issued by index accesses. Figure 3.15 clearly shows that every join with an additional base table increases the number of logical page reads. Thus this graph shows the trade-off between conventional tables, where most meta-data is interpreted at compile time, and Chunk Tables, where the meta-data must be interpreted at runtime.

3.4.7. Response Times with Cold Cache

Figure 3.16 shows the average execution times *with cold caches*. For this test, the database buffer pool and the disk cache were flushed between every run. For wider Chunk Tables, i.e. 15 to 90 columns, the response times look similar to the page read graph (Figure 3.15). For narrower Chunk Tables, cache locality starts to have an effect. For example, a single physical page access reads in 2 90 column-wide tuples and 26 6 column-wide tuples. Thus

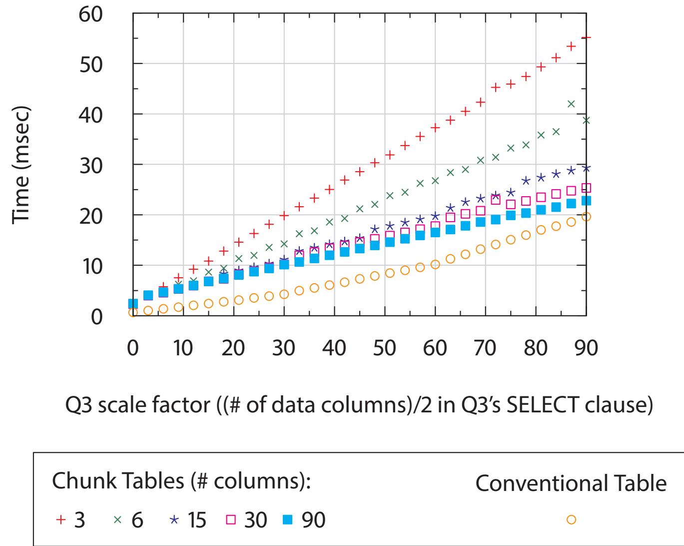


Figure 3.14.: Response Times with Warm Cache

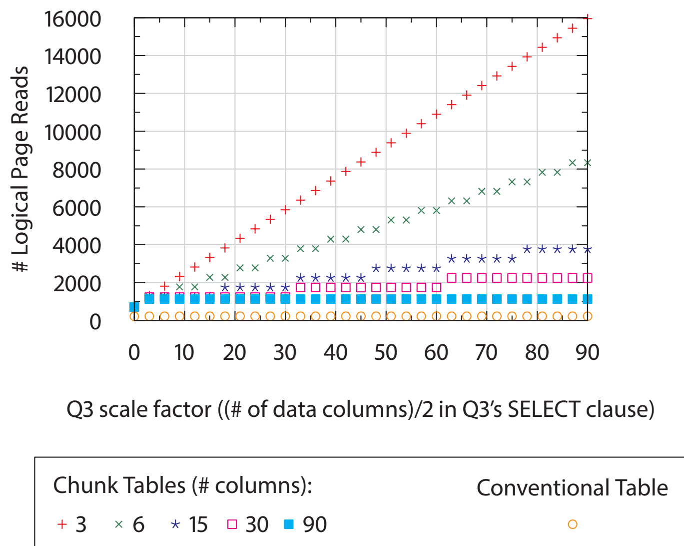


Figure 3.15.: Number of Logical Page Reads.

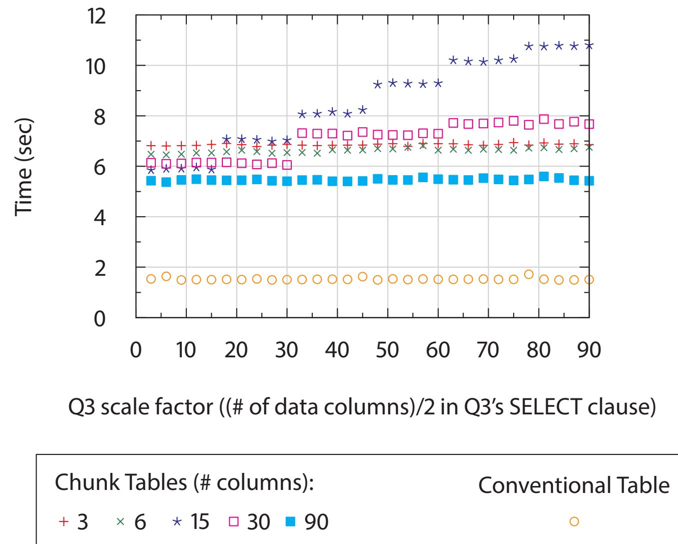


Figure 3.16.: Response Times with Cold Cache

the response times for the narrower Chunk Tables are lower than for some of the wider Chunk Tables. For a realistic application, the response times would fall between the cold cache case and the warm cache case.

3.4.8. Cache Locality Benefits

The effects of cache locality are further clarified in Figure 3.17, which shows the relative difference in response times between Chunk Folding and more conventional vertical partitioning. In the latter case, the source tables are partitioned as before, but the chunks are kept in separate tables rather than being folded into the same tables. For configurations with 3 and 6 columns, Chunk Folding exhibits a response time improvement of more than 50 percent. In the configuration with 90 columns, Chunk Folding and vertical partitioning have nearly identical physical layouts. The only difference is that Chunk Folding has an additional column Chunk to identify the chunk for realigning the rows, whereas in the vertical partitioning case, this identification is done via the physical table name. Since the Chunk column is part of the primary index in the Chunk Folding case, there is overhead for fetching this column into the index buffer pools. This overhead produces up to 25% more physical data reads and a response time degradation of 10%.

3.4.9. Additional Tests

We also ran some initial experiments on more complex queries (such as grouping queries). In this case, queries on the narrowest chunks could be as much as an order of magnitude slower than queries on the conventional tables, with queries on the wider chunks filling the range in between.

The overall result of these experiments is that very narrow Chunk Tables, such as Pivot Tables, carry considerable overhead for reconstruction of rows. As Chunk Tables get wider

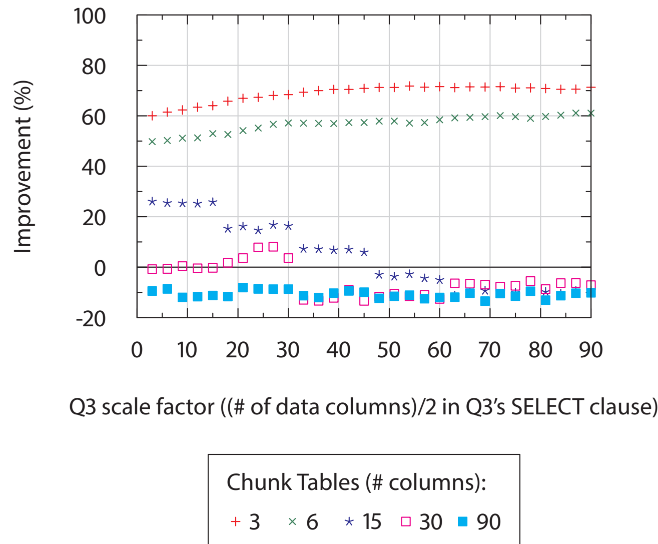


Figure 3.17.: Response Time Improvements for Chunk Tables Compared to Vertical Partitioning

however, the performance improves considerably and becomes competitive with conventional tables well before the width of the Universal Table is reached.

3.5. Schema Evolution on Semistructured Schema Mappings

In a second set of experiments, we evaluate the performance of the semi-structured approaches, especially in combination with schema evolution. The experiments have been run against our Multi-Tenancy CRM testbed. To better study schema evolution, we issued a series of schema alteration statements during a run of the testbed and measured the drop in throughput. As a second enhancement, we introduced differently sized tenants, where tenants with more data have more extension fields, ranging from 0 to 100. The characteristics of the dataset are modeled on *salesforce.com*'s published statistics (McKinnon, 2007).

For getting finer-grained results, we split the existing request classes, thus resulting in nine request classes as shown in Figure 3.18.

The experiments were run on Microsoft SQL Server 2008 and IBM DB2 V.9.5 on Windows 2008. The database host was a Virtual Machine on VMWare ESXi with 4 3.16 GHz vCPUs and 8 GB of RAM.

3.5.1. Microsoft SQL Server

Figure 3.19a shows the results of running our testbed on Microsoft SQL Server using three different mappings: Private Tables, Extension Tables, and Sparse Columns. The horizontal axis shows the different request classes from Figure 3.18, and the vertical axis shows the response time in milliseconds on a log scale.

In comparison to Private Tables, Extension Tables clearly exhibit the effects of vertical partitioning: wide reads (Sel 1, Sel 50, Sel 1000) are slower because an additional join is

Select 1: Select all attributes of a single entity as if it was being displayed in a detail page in the browser.

Select 50: Select all attributes of 50 entities as if they were being displayed in a list in the browser.

Select 1000: Select all attributes of the first 1000 entities as if they were being exported through a Web Services interface.

Reporting: Run one of five reporting queries that perform aggregation and/or parent-child-roll-ups.

Insert 1: Insert one new entity instance as if it was being manually entered into the browser.

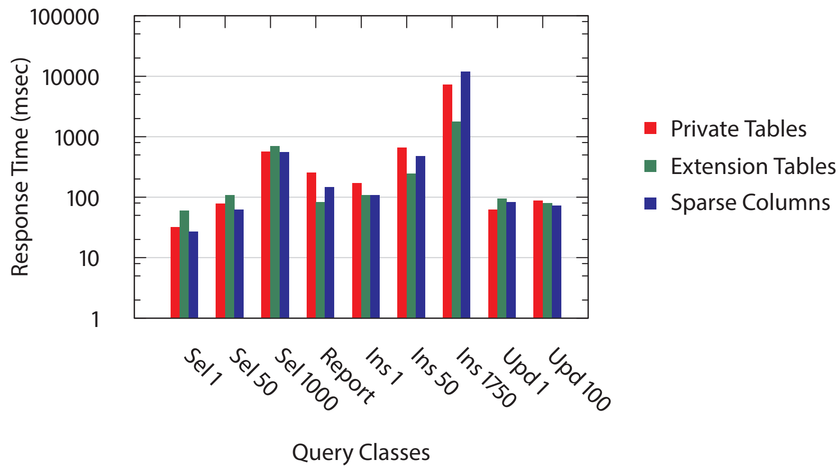
Insert 50: Insert 50 new entity instances as if data were being synchronized through a Web Services interface.

Insert 1750: Insert 1750 new entity instances as if data were being imported through a Web Services interface.

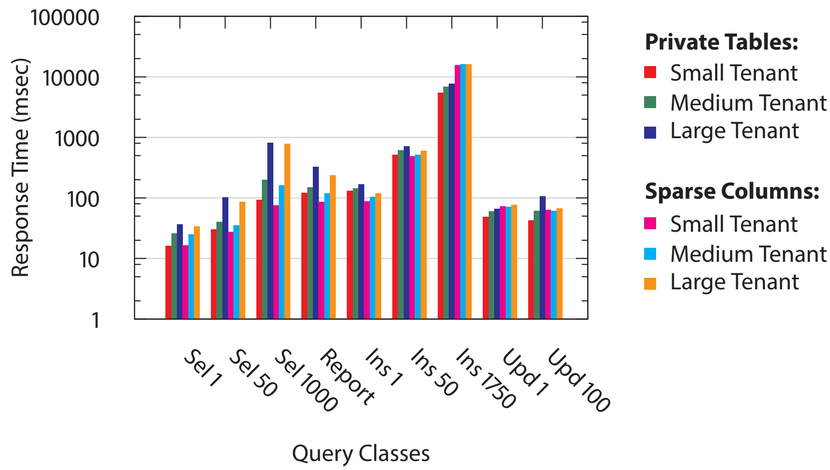
Update 1: Update a single entity as if it was being modified in an edit page in the browser.

Update 100: Update 100 entity instances as if data were being synchronized through a Web Services interface.

Figure 3.18.: Refined Worker Action Classes



(a) Overall



(b) By Tenant Size

Figure 3.19.: SQL Server Performance

required, while narrow reads (Report) are faster because some unnecessary loading of data is avoided. Updates (Upd 1, Upd 100) perform similarly to wide reads because our tests modify both base and extension fields. Extension Tables are faster for inserts because tables are shared among tenants so there is a greater likelihood of finding a page in the buffer pool with free space.

Sparse Columns perform as well or better than Private Tables in most cases. The additional overhead for managing the Interpreted Storage Format appears to be offset by the fact that there are fewer tables. Sparse Columns perform worse for large inserts (Ins 1750), presumably because the implementation of the Interpreted Storage Format is tuned to favor reads over writes.

Figure 3.19b shows a break down of the Private Table and Sparse Column results by tenant size. Recall that larger tenants have more extension fields, ranging from 0 to 100. The results show that the performance of both mappings decreases to some degree as the number of extension fields goes up.

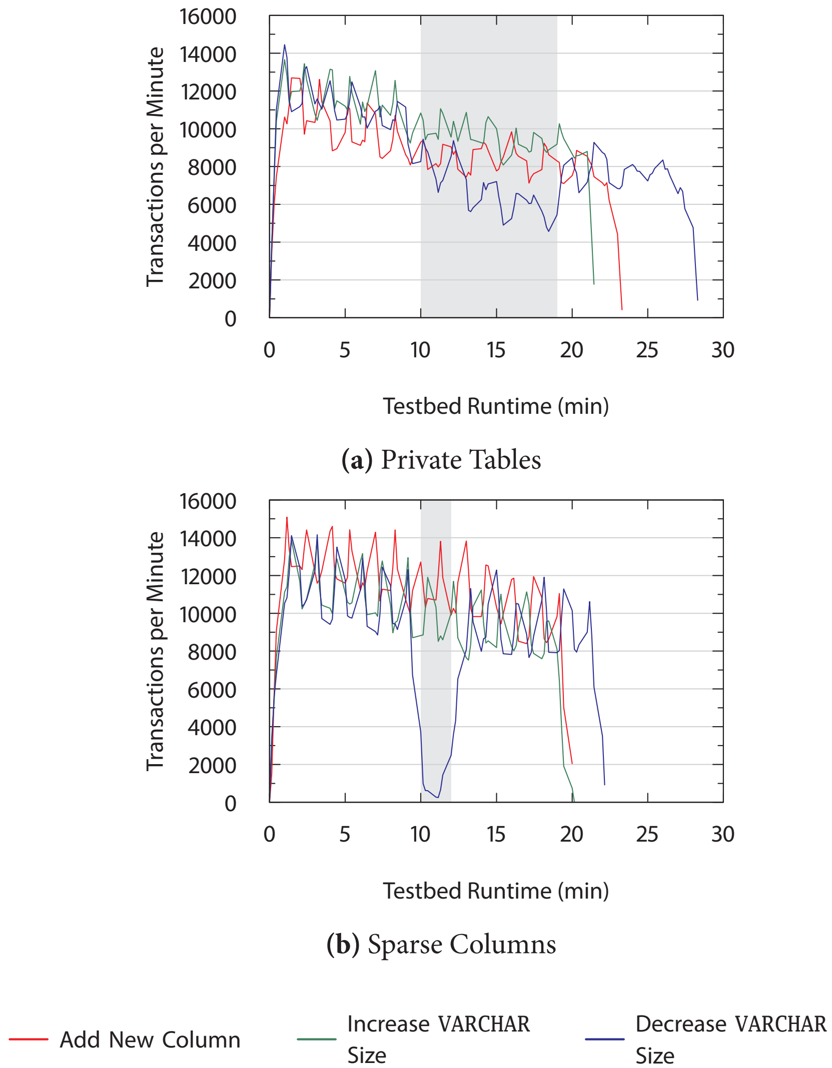
SQL Server permits up to 30,000 Sparse Columns per table. Our standard configuration of the testbed has 195 tenants, which requires about 12,000 columns per table. We also tried a configuration with 390 tenants and about 24,000 columns per table and there was little performance degradation. The number of extension fields per tenant in our testbed is drawn from actual usage, so SQL Server is unlikely to be able to scale much beyond 400 tenants. As a point of comparison, *salesforce.com* maintains about 17,000 tenants in one (very large) database (McKinnon, 2007).

Figure 3.20 shows the impact of schema evolution on throughput in SQL Server. In these graphs, the horizontal axis is time in minutes and the vertical axis is transactions per minute. The overall trend of the lines is downward because data is inserted but not deleted during a run. Part way through each run, the structure of five base tables are changed. For Private Tables, this results in 975 `ALTER TABLE` statements, as each tenant has its own set of base tables. However, with Sparse Columns, the base tables are shared, thus only 5 `ALTER TABLE` statements must be performed.

The first two lines in each graph show schema-only DDL statements: add a new column and increase the size of a `VARCHAR` column. The third line in each graph shows a DDL statement that affects existing data: decrease the size of a `VARCHAR` column. To implement this statement, SQL Server scans through the table and ensures that all values fit in the reduced size. A more realistic alteration would perform more work than this, so the results indicate a lower bound on the impact of evolution. The gray area on each graph indicates the period during which this third operation took place.

In the Private Tables case (Figure 3.20a), 975 `ALTER TABLE` statements were submitted, 5 for each of the 195 tenants. Individual schema-only alterations completed very rapidly, but nevertheless had an impact on throughput because there were so many of them. Adding a new column took about 1 minute to complete while increasing the size of a `VARCHAR` column took about 3 minutes. Decreasing the size of a `VARCHAR` column took about 9 minutes and produced a significant decrease in throughput. The overall loss of throughput in each case is indicated by the amount of time it took to complete the run.

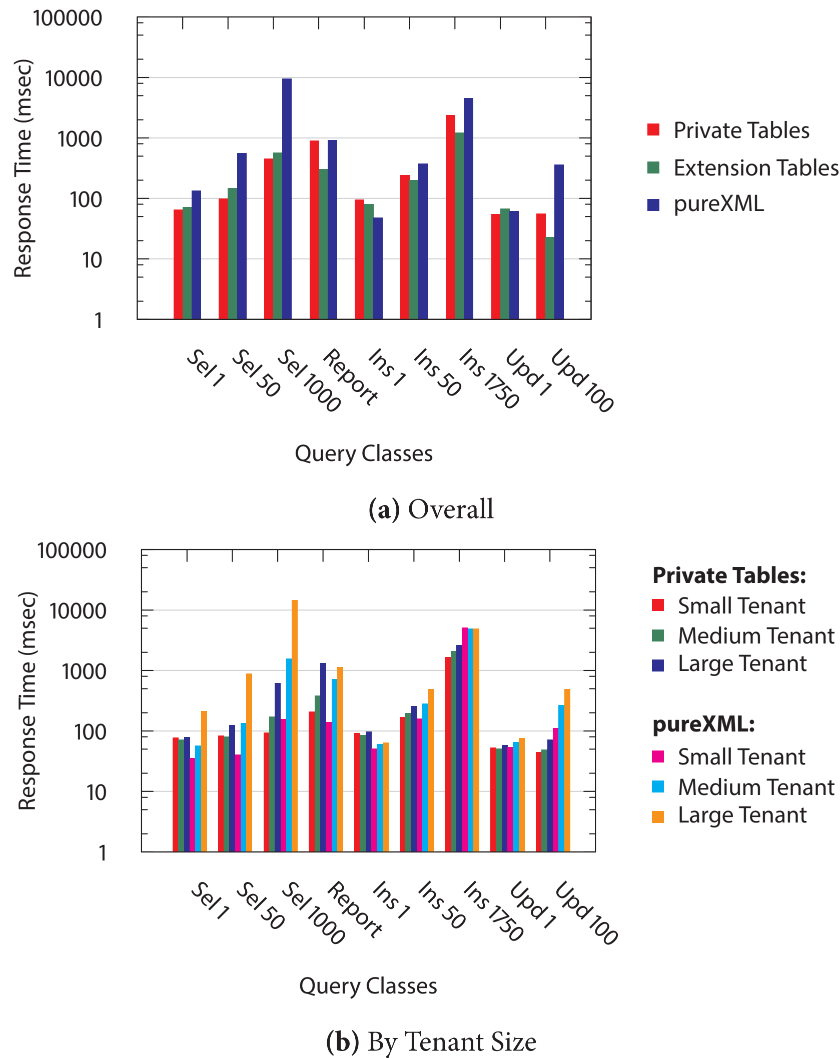
In the Sparse Columns case (Figure 3.20b), the tables are shared and 5 `ALTER TABLE` statements were submitted. The schema-only changes completed almost immediately and



(a) Private Tables

(b) Sparse Columns

Figure 3.20.: SQL Server Throughput



(a) Overall

(b) By Tenant Size

Figure 3.21.: DB2 Performance

had no impact on throughput. Decreasing the size of a VARCHAR column took about 2 minutes, during which throughput dropped almost to zero. The overall loss of throughput was greater for Private Tables, as indicated by the amount of time it took to complete the runs. However, the behavior of Private Tables is probably preferable in the SaaS setting because the throughput drop is never as deep, thus the servers don't need to be over-provisioned as much. In any case, neither of these mappings is ideal in that the application should have more control over when such resource-intensive operations occur.

3.5.2. IBM DB2

Figure 3.21a shows the results of running our testbed on IBM DB2 using three different mappings: Private Tables, Extension Tables, and XML using pureXML. The axes are the same as in Figure 3.19.

In comparison to Private Tables, Extension Tables exhibits the same performance variations as in SQL Server. However XML produces a decrease in performance in most cases. The decrease is particularly severe for reads, which require executing a correlated subquery containing an XQuery statement embedded in a call to the `XMLTABLE` table function, as described in Section 3.3.2. Figure 3.21b shows a break down of the Private Table and XML results by tenant size. Recall that larger tenants have more extension fields, ranging from 0 to 100. The results show that for reads, the performance decrease of XML is proportional to the number of extension fields. Note that in the Insert 1750 case, the results do not include the time to construct the XML document[6] and there is no variation based on tenant size.

XML gives the application complete control over schema evolution. In this setting, the application is responsible for performing any bulk transformations associated with schema alterations that impact existing data. To study the efficiency of such transformations, we ran our schema evolution throughput experiment on DB2 using pureXML. To simulate the decrease-VARCHAR case, we submitted a query for each of the five base tables that selects one field from all documents. These queries were run on the database server so no data transfer costs were incurred. The results were almost identical to the SQL Server Sparse Columns case shown in Figure 3.20b. The five queries took about 2 minutes to complete, during which time throughput dropped to a very low level. Of course, the advantage of the XML mapping is that the application need not perform such transformations all at once.

[6]The XML document keeping the extension data has been generated outside. Thus, inserting a tuple with an XML document attached is fairly cheap as we configured the DBMS to not validate the XML document against a XML schema

4

Next Generation Multi-Tenant DBMS

The discussion in the previous chapters shows that traditional DBMSs are not well-suited for massive multi-tenancy. The simpler approaches "Shared Machine" and "Shared Process" do not offer any support for schema flexibility. Simultaneously, they cope with poor buffer pool utilization and a high memory overhead due to inefficient meta-data handling. Furthermore, multi-tenancy requires features like schema flexibility which are typically not supported by currently available DBMSs.

Due to high level of consolidation, there is a high probability that one of the co-located tenants performs schema modifications. However, the evaluation in the previous chapter shows that, for traditional DBMSs, allowing on-line schema changes may heavily impact the overall system response time and thus must be prevented in a multi-tenant scenario.

Many database vendors circumvent these issues by recommending proprietary techniques for extensible schemas. However, as soon as schemas must be extensible, a schema mapping must be employed to ensure a consistent transformation of the logical tenant-specific schema to the physical schema inside the DBMS. In such a setup, the DBMS no longer maintains the meta-data of the tenants' schemas, rather the application is now responsible for dealing with the tenant-specific meta-data. The DBMS is then degraded to a 'dumb data repository' which only maintains the physical layout of the schema mapping strategy. Thus, more and more core functionalities of a DBMS, such as query optimization, statistics, and data management, have to be rebuilt inside the application, making further application development more costly.

Most SaaS applications combine the shared schema approach with schema mappings to circumvent the issues with bad resource utilization. However, support for schema flexibility is still lacking and the shared schema approach weakens the tenant isolation, as all tenants' data is stored inter-weaved with each other. A physically separated schema for each tenant would therefore offer a better data isolation and facilitate tenant migration. Moreover, with separated schemas, tenants can be distributed across a shared-nothing system by simply copying the data from one machine to another.

In this chapter, we describe our approach of addressing this issue with a specialized multi-tenant DBMS with native support for schema flexibility. Our approach provides the same level of tenant isolation as the shared process approach while simultaneously allowing schema flexibility with resource utilization similar to the shared table approach. We describe the key concepts and the architecture of the DBMS and present the underlying data model *FlexScheme* which serves as an integrated model for schema flexibility.

4.1. Estimated Workload

Before presenting the conceptual changes for a multi-tenant DBMS compared to traditional DBMSs, we estimate the typical DBMS workload of a SaaS application. Our estimations are based on the work of Harizopoulos et al. (2008) who present an in-depth analysis of OLTP applications.

SaaS applications tend to be light to medium complexity applications, as already discussed in Chapter 1. High complexity applications, such as Enterprise Resource Planning (ERP), are not suitable for SaaS, because such applications need heavy customizations to adapt the tenants' requirements and may have long running transactions.

Furthermore, the workload of SaaS applications is mostly OLTP-style and optimized for high throughput, which means that most of the requests are look-ups by primary key or other short running queries. When performing scans across a set of tuples, only small scans are allowed, e.g., for paginating lists; large scans should be avoided at all. Application developers have to ensure this workload during the design of the SaaS application's data model, for example, by only allowing parent-child relationships. Hamilton (2007) presents other recommendations for an optimized workload for highly-scalable services.

For enabling features like a "Dashboard", there may be the need for *Ad-Hoc Reports*. Those reports do lightweight analytical processing by running pre-defined reports, either synchronously or asynchronously as batch jobs. In this context, "Ad-Hoc Reports" are run on the transactional data, rather than on a dedicated data ware-house.

Typically, SaaS applications are accessed via Internet Browser. This circumstance can be exploited when choosing the DBMS's consistency level. As HTTP requests are stateless and thus do not allow for "real" cross-request sessions, the consistency level can be lowered. This way, the number of concurrent locks inside the DBMS can be reduced and thus throughput increases. However, due to the nature of HTTP requests, mechanisms like *Optimistic Locking* must be implemented in the application server to avoid update anomalies.

Besides the data-centric workload, the SaaS context also has a meta data-centric workload. The DBMS has to handle `ALTER TABLE` statements quite often, not for individual tenants, but due to the high number of co-located tenants.

A SaaS application is updated on a regular basis by the service provider and hence may be available in multiple versions. For our workload, we assume that the majority of tenants access the most recent version of the application. As a single tenant may have extensions, either developed by himself or by an ISV, updates to the base application may break the application for that tenant until a new version of these extensions is released. To guarantee application availability, the tenant may defer these update to the new version of the base application and the extensions, respectively. Therefore it might be possible, that a few tenants lag marginally behind the most recent version of the application.

4.2. Multi-Tenant DBMS Concepts

In the following, we illustrate the concepts of our multi-tenant DBMS prototype which leverage the previously introduced workload characteristics.

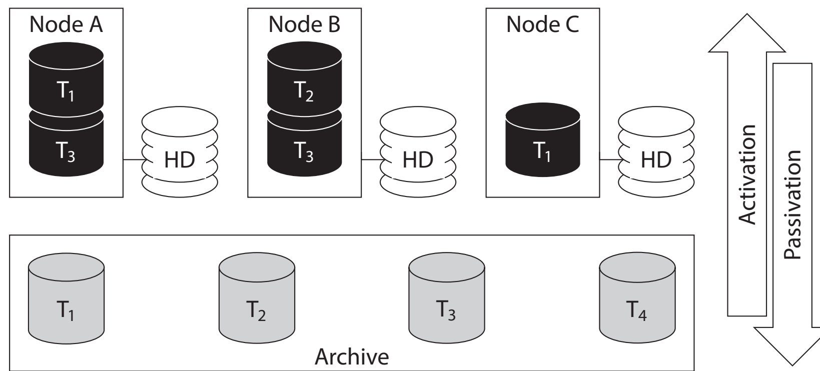


Figure 4.1.: Multi-Tenant DBMS Cluster Architecture

4.2.1. Native Schema Flexibility

The predominant requirement for our prototype is to flexibly handle a large number of tenants while allowing for schema flexibility. Thus, one of our key concepts is *Integrated Schema Flexibility* which is reflected in both the meta-data as well as the data handling. In the course of this chapter, we propose FlexScheme, an integrated data management model which captures the aspects of schema extensibility and schema evolution. Furthermore, it enables data sharing capabilities for cross-tenant data and meta-data, and is tightly integrated into the DBMS kernel. We postpone the discussion of native schema flexibility to Section 4.3.

4.2.2. Tenant Virtualization

Another concept of our multi-tenant DBMS is *Tenant Virtualization*. From a high level perspective, we treat each individual tenant as a "Virtual Machine" which is running on a hypervisor, the multi-tenant DBMS. This approach pushes the classical virtualization idea one step further: instead of virtualization at operating system level, we do virtualization at tenant level inside the DBMS.

For better scalability, multiple multi-tenant database systems are combined to a shared-nothing cluster for serving the individual tenants. As with traditional virtualization techniques, the multi-tenant infrastructure must support migrating tenants between cluster nodes and suspending off-line tenants to release cluster resources. In addition to the nodes, the cluster needs a storage back-end for storing currently inactive tenants.

Figure 4.1 shows the overall architecture of our approach. The shared-nothing nodes have access to a common storage back-end which solely stores inactive tenants' data. Active tenants are mounted on nodes of the cluster and do not use the common storage back-end. Hence, for this storage back-end a high-performant, and thus expensive Storage Area Network (SAN) is not necessary; cheap near-line storage or cloud-based storage like Amazon S3 (2011) is sufficient.

If a tenant becomes active, a target node for this tenant is selected and the tenant's data is transferred to this data node. Once this transformation is completed, the tenant is on-line. As soon as the tenant does not perform any operations, the tenant can be suspended by

transferring the active data of the tenant back to the storage system, and thus the tenant becomes off-line. This way, inactive tenants do not occupy expensive cluster capacities, like main-memory and CPU cycles.

4.2.3. Grid-like Technologies

The virtualization of tenants furthermore allows the usage of grid technologies, like distribution and replication, especially on low-complexity hardware. Projects like *Google BigTable* (Chang et al., 2008) use similar techniques for the same reason. BigTable serves as highly scalable data storage which is distributed and replicated on a cluster of commodity hardware, but relinquishes the relational model. BigTable stores each datum in a specific cell which can be addressed by the dimensions row, column, and time stamp. Cells of semantically adjacent data based on the dimensions are physically co-located. *Amazon Dynamo* (DeCandia et al., 2007) and *SimpleDB* (Amazon SimpleDB, 2011) pursue similar approaches, but with weakened guarantees on data consistency. In contrast to these simple key-value stores, *PNUTS* (Cooper et al., 2008) offers a relational data model. Key requirements of PNUTS are scalability, distribution, and high availability, but with restricted access possibilities and weakened consistency compared to traditional database systems.

These projects heavily rely on distribution and replication, and thus align their data models. Depending on the requirements on replication, e.g., how often data is replicated, and on data consistency the data model and the consistency guarantees are chosen, either unrestricted as in traditional DBMSs or restricted to satisfy special application needs.

For better availability, a master/slave replication at tenant level can be used. However, two tenants whose master instances are co-located need not necessarily have co-located slave instances. During tenant activation, multiple cluster nodes are selected to host one master instance and one or more slave instances of the same tenant.

As long as a tenant is suspended, any node from the cluster can be selected to host the tenant's replicas. This way a static load balancing can be achieved. For active tenants, a dynamic load balancing strategy might be to switch the master role of a particular tenant from an overloaded host to a not so heavy loaded slave replica. However, this strategy restricts the target nodes for the master role to those nodes already hosting the affected tenant's replica. An enhanced strategy would be to create a new slave replica at one particular node, either by loading the tenant's data from the archive and rolling forward the transaction logs, or by snapshotting and cloning an existing slave, and as soon as the new slave is available, switch the master role to this host. Afterwards, the number of replicas can be decreased again—if desired. This enhancement not only allows for dynamic load balancing, but also for migrating tenants seamlessly from one node to another in the cluster. Furthermore, the cluster can grow by simply adding more nodes which can be used immediately.

4.2.4. Main Memory DBMS

Our prototype is heavily influenced by *main-memory database systems* like H-Store (Kallman et al., 2008) or HyPer (Kemper and Neumann, 2010). H-Store and HyPer take advan-

tage of keeping all data in main-memory. We follow the approach proposed by Stonebraker et al. (2007) for the H-Store system and use a single-threaded execution model. Thereby locking and deadlock detection overhead is completely eliminated. This approach seems very suitable for processing many short-running queries in main-memory, where performance is limited by main-memory bandwidth only. I/O operations to disk-based storage systems are not required, as durability can be achieved without disk-based storage systems by replication to physically distant server nodes. Alternatively, log records can be written to disk to ensure durability.

Main-memory DBMS technology is very suitable for multi-tenant SaaS. Jim Gray's dictum *Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King* (Gray, 2006) anticipates the trend towards main-memory based DBMS. Plattner (2009) shows that a single off-the-shelf server that is available today provides enough main-memory to accommodate the entire data volume of even large-sized businesses. For even larger deployments, new techniques like RAMCloud (Ousterhout et al., 2009) may be used. RAMCloud provides a general-purpose storage system that aggregates the main-memories of lots of commodity servers and keeps all information in main-memory. Besides the superior performance for OLTP workloads, Poess and Nambiar (2010) show that there are other interesting factors, such as a low energy consumption of main-memory compared to disks. This further reduces the TCO.

Our prototype does not interfere with special optimizations for main-memory DBMS, such as second-level cache optimizations. We propose special query plan operators for enabling schema flexibility. These operators rely on standard access operators, so techniques like Cache-Conscious B-Trees (Rao and Ross, 2000) and cache-optimized compact data representations, e.g. PAX (Ailamaki et al., 2001), are fully transparent to our approach. Moreover, data compression to further reduce the space requirements can be applied to our approach as well.

4.2.5. Execution Model

Figure 4.2 shows the architecture of one single multi-tenant DBMS instance. One query processing thread per tenant takes requests from the tenant-specific input queue and processes these requests one by one. For each individual tenant, the H-Store-like execution model is applied, thus deadlocks and locking overhead are completely eliminated. Since there are no cross-tenant transactions, there is no need for cross-thread synchronization. Furthermore, due to this execution model, the tenant isolation is increased.

Requests by tenants can span across their private data and the common data, but tenant-specific data is completely separated from common data. However, there is still no need for synchronization, as tenant-specific updates to common data are redirected to the private data segment. Common data is read-only, so locks on common data are not necessary. However, the shared data can be updated by the service provider or ISVs, for example, in the context of application upgrades. We assume that the frequency of such updates is much lower than the frequency of tenant-specific updates.

During a tenant's thread lifetime, it has to perform various I/O operations. At initialization, the thread must retrieve the tenant's data from the archive store, and at the end it

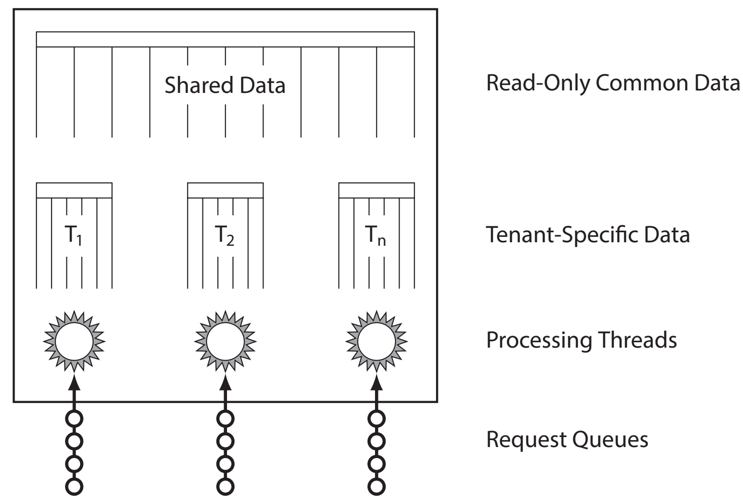


Figure 4.2.: Thread Model of the Multi-Tenant DBMS Instance

must write them back. In between, the thread has to perform either logging or replication to guarantee durability. The I/O induced by logging can be significantly reduced, if logging is implemented as described by Kemper and Neumann (2010) for the HyPer project.

4.3. Native Support for Schema Flexibility

Instead of using schema mappings from the outside, a multi-tenant DBMS for SaaS should offer inherent support for schemas that are flexible in the following two respects. First, it should be possible to *extend* the base schema to support multiple specialized variants of the application, e.g., for particular vertical industries or geographic regions. An extension may be private to an individual tenant or shared between multiple tenants. Second, it should be possible to *evolve* the base schema and its extensions while the database is on-line, including changing existing attributes. Evolution of a tenant-owned extension should be totally "self-service". The service provider should not be involved, otherwise operational costs would be too high. Furthermore, it may not be acceptable for some tenants to instantly upgrade their extensions whenever the service provider updates the base schema.

Extensions form a hierarchy, which can be used to share data among tenants, such as master data and default configuration data. This hierarchy is made up of *instances* which combine schema and shared data. *Data sharing* allows to co-locate even more tenants, and thus improves resource utilization and enables economy of scale. From a tenant's perspective, the shared data has to be update-able, but updates from one tenant might interfere with other tenants. Therefore, write access to shared data has to be redirected to a private storage segment of the updating tenant. If data sharing is employed, evolution affects not only the schema but also the shared data. This can be addressed by *versioning* the schema and the shared data. Base and extension instances may evolve independently from each other, therefore versioning has to be employed on the level of individual instances. This mechanism also allows tenants to stay with certain versions of instances.

Since versioning is handled at the level of individual instances, the target version number is not sufficient to reconstruct a relation from the instance hierarchy. Therefore, additional information about dependencies between instance versions is needed to derive a certain relation for a particular tenant. We call such a derived relation a *Virtual Private Table*, since this relation is specific to a particular tenant.

4.4. FlexScheme – Data Management Model for Flexibility

Traditional DBMSs do not offer mechanisms for flexible schemas and data sharing. Thus, we propose FlexScheme, a model for handling meta-data and shared data in multi-tenant DBMS, which captures extensibility, evolution and data sharing in one integrated model.

Our prototype keeps the tenants' data in main-memory, which is an expensive resource that has to be used efficiently: data that is not needed anymore has to be removed from main-memory as soon as possible. This is a big challenge, especially in the multi-tenancy context, when data is shared between tenants. FlexScheme allows to identify which versions of the instances are currently used by a given set of tenants, and which can be removed from main-memory.

4.4.1. Data Structures

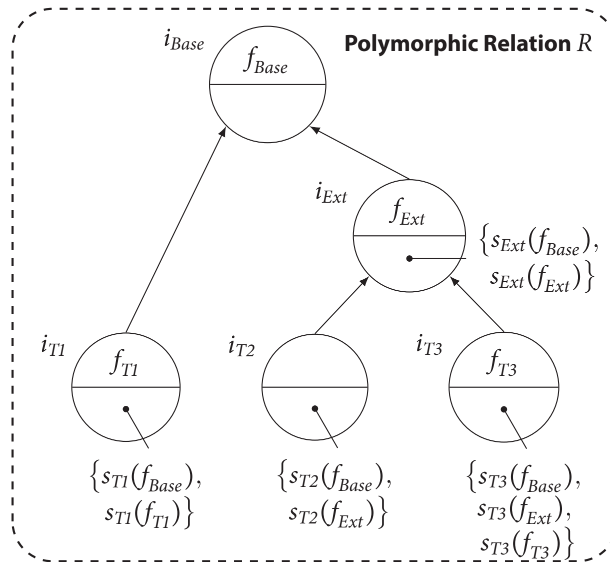
This section introduces the data structures of our data management model based on the example in Figure 4.3. The upper part of the figure (Fig. 4.3a) shows the global view of a particular relation, from which the local views for the individual tenants are derived; the lower part of the figure (Fig. 4.3b) shows such a local view for the Tenant T_3.

FlexScheme models schema information and data in separate data structures to allow for schema flexibility. *Fragments* store schema information, and *segments* store actual data. In the following, we introduce the data structures in detail.

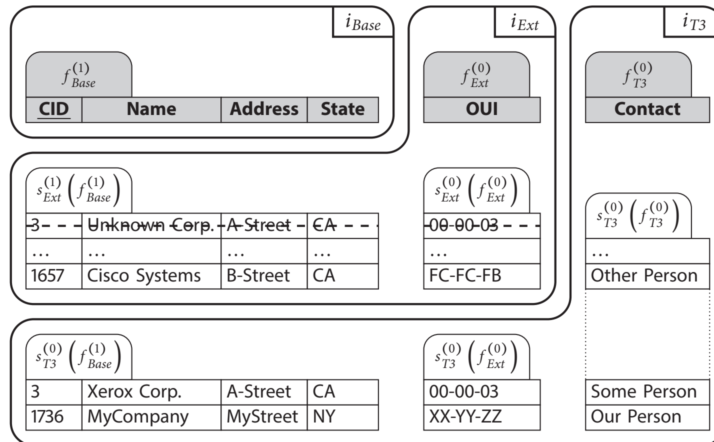
Definition 4.1 (Fragment):
A fragment $f^{(v)}$ is a set of n typed attributes, forming a schema definition. Some of the attributes form the primary key attributes of the fragment. v denotes a version number.

During application lifetime, the definition of a fragment may change, so we combine multiple fragments to a *fragment sequence*. For guaranteeing proper primary key access, the definition of the primary key attributes must not change across versions.

Definition 4.2 (Fragment Sequence):
A fragment sequence $f = \langle f^{(0)}, \ldots, f^{(z)} \rangle$ is a sequence of fragments $f^{(v)}$. Within one fragment sequence f, the primary key attributes of each fragment are identical: $pk(f) := pk(f^{(0)}) = \ldots = pk(f^{(z)})$.



(a) Global Perspective



(b) Derived Virtual Private Table for Tenant T_3

Figure 4.3.: Example of Polymorphic Relation R

Consider the example of the polymorphic relation in Figure 4.3a. For a first understanding, a polymorphic relation is the global, hierarchical view from which tenant-specific views are derived. We introduce the concept of polymorphic relations more formally in the course of this section.

Example 4.1: *Figure 4.3a shows the fragment sequence f_{Base} at the top of the polymorphic relation R. Suppose f_{Base} has two fragments $f^{(0)}_{Base}$ and $f^{(1)}_{Base}$. Figure 4.3b shows the definition of $f^{(1)}_{Base}$.*

Definition 4.3 (Segment):
A segment $s^{(v)}\left(f^{(w)}\right)$ contains tuples that have the schema $f^{(w)}$. Again, v and w denote version numbers.

Segments may be required in various versions, therefore we combine multiple segments in a *segment sequence*. Entries in the segment sequence may be independent from each other, however, the schemas of all segments within a segment sequence have to be defined by fragments of the same fragment sequence.

Definition 4.4 (Segment Sequence):
A segment sequence $s(f)$ is a sequence of segments $s^{(0)}\left(f^{(i)}\right), \ldots, s^{(z)}\left(f^{(j)}\right)$. The fragments $f^{(i)}, \ldots, f^{(j)}$ form part of the same fragment sequence f.

Example 4.2: *Figure 4.3b shows $s^{(1)}_{Ext}\left(f^{(1)}_{Base}\right)$ which is part of the segment sequence $s_{Ext}\left(f_{Base}\right)$. The schema of tuples in this segment is defined by the fragment $f^{(1)}_{Base}$.*

As discussed before, modern SaaS applications need extensibility. The schema definitions of the base application can be extended by ISVs or the tenants themselves, thus inheriting the definitions from the base application. In this context, we have two types of inheritance: *schema inheritance*, where fragment definitions are inherited, and *data inheritance*, where segments are inherited.

Definition 4.5 (Instance):
An instance i combines fragment sequences and segment sequences. Each instance contains zero or one fragment sequences, as well as a set of segment sequences. Each segment has either a local fragment as schema or inherits the schema definition from other instances.

Descriptively, instances contain sequences of fragments and segments, for representing schema and data evolution. The dependencies between instance versions are managed separately on a per-tenant basis.

Example 4.3: *The instance i_{Ext} in Figure 4.3a defines a local fragment sequence f_{Ext} and inherits f_{Base}. Furthermore, the instance holds two segment sequences, $s_{Ext}(f_{Base})$ and $s_{Ext}(f_{Ext})$, which store the tuples. $s_{Ext}(f_{Base})$ corresponds to the inherited fragment sequence f_{Base}, while $s_{Ext}(f_{Ext})$ corresponds to the locally defined fragment sequence.*

The individual instances together with the inheritance relationships are forming a *polymorphic relation*. Multiple polymorphic relations form a *polymorphic database*.

Definition 4.6 (Polymorphic Relation):
A polymorphic relation $R = (I, E, r)$ can be represented as a rooted tree. Its vertices are the instances I, and its edges E the inheritance relationships. Furthermore, there is one distinguished instance $r \in I$, that forms the root of the polymorphic relation.

Definition 4.7 (Polymorphic Database):
A polymorphic database D is a set of polymorphic relations.

4.4.2. Deriving Virtual Private Tables

The process of deriving a Virtual Private Table (VPT) for a particular tenant is comprised of several steps:

Step 1
At first, the instances on the path from the tenant's leaf node to the root node of the polymorphic relation has to be determined. For each instance on this path, a fragment has to be selected from the fragment sequence of each instance. The selection is based on the dependency information of the tenant. The selected fragments are then concatenated. The concatenated fragments form the virtual schema of that tenant's VPT.

Example 4.4: *In Figure 4.3a the Instances i_{T_3}, i_{Ext}, and i_{Base} are on the path of Tenant T_3. The selected Fragments to be concatenated are $f^{(1)}_{Base}$, $f^{(0)}_{Ext}$, and $f^{(0)}_{T_3}$, as seen in Figure 4.3b.*

Step 2
In a second step, for each fragment from the previous step, a virtual segment containing the data has to be built. For each instance on the tenant's path to the root node, one segment

has to be selected from the segment sequences of each instance. Again, the selection is based on the dependency information of the tenant. For one particular fragment there may be multiple segments available which have to be overlaid. Since our model allows write access to shared data by redirecting the access to a tenant-specific segment, a data overlay precedence has to be defined: the lower the segment is defined in the polymorphic relation, the higher is the precedence over other segments on the path. We refer to this concept as *data overriding*.

The *overlay* of two segments s and t, where s has precedence over t, can be defined as follows. The term $t \triangleright s$ denotes the anti-join of t and s.

$$
\begin{aligned}
overlay(s,t) &= s \cup \overbrace{\left(t \setminus \left(t \ltimes_{pk(t)=pk(s)} s\right)\right)}^{t \triangleright_{pk(t)=pk(s)} s} \\
&= s \cup \left(t \triangleright_{pk(t)=pk(s)} s\right)
\end{aligned}
$$

This function is then applied iteratively from the leaf node to the root node to collapse all selected segments per fragment into one virtual segment.

Example 4.5: *In Figure 4.3a for Tenant T_3 there are two segment sequences $s_{Ext}(f_{Base})$ and $s_{T_3}(f_{Base})$ available for the fragment sequence f_{Base}. The previous step selected the fragment $f_{Base}^{(1)}$. As seen in Figure 4.3b the selected segments are $s_{Ext}^{(1)}\left(f_{Base}^{(1)}\right)$ and $s_{T_3}^{(0)}\left(f_{Base}^{(1)}\right)$. Since $s_{T_3}^{(0)}\left(f_{Base}^{(1)}\right)$ takes precedence over $s_{Ext}^{(1)}\left(f_{Base}^{(1)}\right)$, the tuple with CID=3 from $s_{Ext}^{(1)}\left(f_{Base}^{(1)}\right)$ is overwritten by the tuple from $s_{T_3}^{(0)}\left(f_{Base}^{(1)}\right)$ with the same CID.*

Step 3

Finally, the virtual segments from the previous step have to be aligned. This is similar to vertical partitioning, therefore well known defragmentation techniques can be applied. For proper alignment, each tuple in any segment has to replicate the primary key value. The result of this last step is the VPT which contains the tenant-specific data as well as the shared data with respect to data overriding.

Example 4.6: *Figure 4.3b shows the full VPT of Tenant T_3.*

4.4.3. Comparison to Other Models

FlexScheme extends the traditional relational data model with concepts known from the object-oriented world. Object-oriented systems are characterized by support for abstract data types, object identity, and inheritance as well as polymorphism and method overloading (Khoshafian and Abnous, 1990; Nierstrasz, 1989). The object-oriented programming paradigm states that a program is a set of interacting objects telling each other what to do

by sending messages. Each object has a type and all objects of a particular type can receive the same messages (Eckel, 2000).

In contrast to the object-oriented model, FlexScheme does not have facilities for methods. Thus, an instance can be seen as an abstract data type, which only has attributes, as well as a set of objects. The instance hierarchy is similar to a class hierarchy, but the semantics are different: in the object-oriented model, the inclusion polymorphism is specified by sub-typing, i.e. each sub-type can be treated as its super-type. However, in FlexScheme this is not the case: the inclusion polymorphism goes from the leaf node to the root node. To enable data sharing in the multi-tenancy scenario, the ability to override common data with tenant-specific data is needed. We call this concept *data overriding*. There is some similarity to the object-oriented concept of method overloading, but currently neither object-oriented DBMSs (OODBMSs) nor object-oriented programming languages support the functionality of data overriding. The reason may be, that data overriding conflicts with object identity, as a tuple of a tenant overrides a tuple of the base relation or an extension by using the same primary key.

The object-relational model transfers the object-oriented model to relational DBMS. For example, SQL:1999 differentiates between type and relation (Melton, 2002). This is similar to the differentiation between fragments and segments in FlexScheme. Theoretically, this mechanism could be used to implement data sharing, but SQL:1999 does not allow this: if two sub-tables reference a common super-table, changes to common attributes within the first sub-table would be reflected in the second sub-table as well.

The fundamental difference to the object-oriented and the object-relational model is that FlexScheme provides extensibility, evolution, and data sharing with data overriding in one integrated model. Schema evolution has been well studied in the relational database community (Roddick, 1995), as well as in the object-oriented database world (Moerkotte and Zachmann, 1993; Kemper and Moerkotte, 1994). Recent work (Curino et al., 2008; Moon et al., 2008; Curino et al., 2009; Moon et al., 2010) shows that schema evolution is still an important topic, especially in archiving scenarios or in combination with automatic information system upgrades. Schema extensibility has been studied separately in the context of OODBMSs in the form of class hierarchies (Moerkotte and Zachmann, 1993; Kemper and Moerkotte, 1994).

4.5. Applying FlexScheme to Multi-Tenant DBMS

Our objective is to give the knowledge on how to manage the SaaS application's data back to the database, rather than letting the application manage the data. Thus, when FlexScheme is applied to a multi-tenant DBMS, the DBMS has explicit knowledge of the tenants' *Virtual Private Tables*, which are dynamically derived from the polymorphic relation. FlexScheme integrates extensibility on both schema and data.

A multi-tenant DBMS based on FlexScheme enables sharing of meta-data as well as data between tenants by introducing fragments and segments. FlexScheme introduces versioning by the notion of fragment sequences and segment sequences. This is required because both, shared schema and data, can be part of an evolution process. The concepts for schema

versioning have been extensively studied in the context of schema evolution (Curino et al., 2008; Moon et al., 2010).

We implemented a multi-tenant Main-Memory DBMS prototype which leverages all features of FlexScheme. The details of this implementation are discussed in the following chapters. Extensibility is realized by decomposition of relations and data overlay. For this, our prototype has specialized query plan operators for data overlay which are optimized for different access patterns on tenant-specific and shared data. Moreover, it supports updating shared data by individual tenants; write access to shared data is redirected to a private data segment per tenant, thus resulting in a "Copy-on-Write" shadow copy. The implementation of data sharing is further discussed in Chapter 5, including the physical data representation and the data overlay mechanism for retrieving common data. Support for schema evolution is based on *Schema Modification Operators* (Curino et al., 2008). We implemented query plan operators that allow for graceful on-line schema evolution by deferring the physical data reorganization until the first read access to a tuple. Chapter 6 outlines the details.

5

Data Sharing Across Tenants

For better TCO, the level of consolidation can be further increased by sharing data across tenants, like configuration variables or catalogs. However, for some of the shared data, it may be necessary to allow changes by individual tenants. In this chapter, we present the data sharing component of our Multi-Tenancy DBMS prototype. Based on the FlexScheme model, we allow write access to shared data by individual tenants. Our approach provides an *overlay* mechanism, such that changes of one particular tenant to the shared data do not interfere with changes of other co-located tenants.

5.1. Data Overlay

Generally, shared data is physically read-only in our prototype. However, we allow logical write access to the shared data employing the following mechanism: if a tenant issues a write operation on the shared data, the actual request is performed by copying the original data from the shared segment to the private segment of the tenant. The issued write operation is then performed on the tenant's private segment. As the shared data remains read-only at all the time, we avoid synchronization issues during access to the shared data. This mechanism is inspired by techniques used in modern file systems to provide snapshots (e.g., Hitz et al., 1994). Some database systems offer similar mechanisms to provide read-only access to historical data, e.g., Oracle Flashback Technology. Our approach provides *Copy-on-Write* (CoW) semantics, where shared data that has to be updated is copied to the *private segment* of the tenant. In contrast to previous mechanisms which work at the page level, our approach allows per tuple granularity which is enabled by our main-memory-centric architecture. We refer to this process as *Data Overlay* and use a special operator called *Overlay Operator* to perform this action.

The basic functionality of this operator is to weave two inputs, the shared data and the tenant-specific data. Both inputs have to be segments of the same fragment. Despite the support for multiple fragment versions, the logical identifier (primary key, PK) of a tuple is independent of the fragment version[1]. One of the following two cases applies.

1. If the PK is contained in the shared input, but not in the tenant-specific input, then the tuple from the shared input is returned.

[1] A in-depth discussion of the physical tuple layout of our prototype can be found in Section 6.2.3

```
(SELECT pk, ... FROM tenant-specific)
UNION
(SELECT pk, ... FROM shared
  WHERE pk NOT IN
   (SELECT pk FROM tenant-specific)
)
```

Figure 5.1.: Data Overlay in SQL

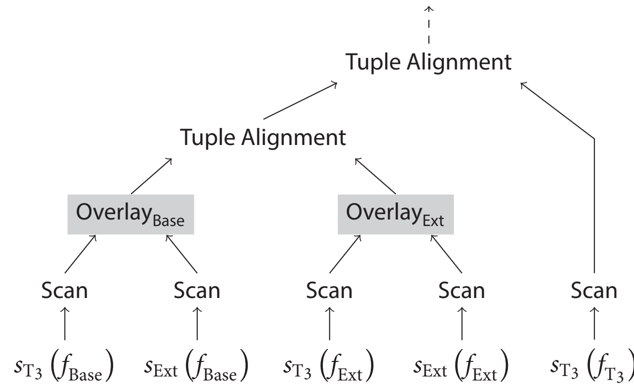


Figure 5.2.: Query Execution Plan Example

2. If the PK is contained in the tenant-specific input, always this tuple is returned, no matter if there is a corresponding tuple in the shared input or not.

Figure 5.1 shows how an overlay could be formulated in a conventional relational DBMS. This implementation has two major drawbacks. First, the overlay has to be handled explicitly, either with a view or within a middleware layer. However, this would interfere with our approach to give as much knowledge about the data as possible to the DBMS. Second, the database system has to process both selects and compute the union of the intermediate results. For the calculation of the union, the DBMS has to process both inputs entirely, which causes certain overhead. In the next section we propose our new data overlay operator that is able to process this query much more efficiently by leveraging the knowledge that FlexScheme provides.

5.2. Accessing Overridden Data

We implemented a family of pipelining overlay operators which can be placed in the query execution plan by the query optimizer. Depending on the implementation, they support efficient full table scans or point-wise look-ups by leveraging specialized indexes. A query execution plan may contain several overlay operators, as a table of a given tenant may consist of several fragments. These operators can be chained together or may be replaced by one N-ary overlay operator that can process multiple inputs at the same time. The correct alignment of overwritten tuples is guaranteed by the PKs that are stored inside each tuple.

A very basic strategy of request processing with common data is to substitute the logical table reference in the query by a sub-plan as shown in Figure 5.2[2] that overlays and aligns data. The query optimizer might then re-arrange operators to allow for early or late alignment.

We implemented two different types of overlay operators. The first type uses already existing indexing techniques—we refer to them as *basic implementations*—, while the second type uses specialized data overlay indexing mechanisms.

5.2.1. Basic Implementations

Basic implementations do not require special index support. For example a *Hash-Join-based* implementation only uses a transient index. In the build phase, the PKs for the tenant-specific segment are hashed into a temporary hash set. All tuples of this segment are passed through to the operator above the overlay. In the probe phase, the shared segment is opened. Only those tuples, which do not have an entry in the temporary hash table are passed to the operator above. This implementation needs enough main memory resources for the temporary hash set.

Another basic implementation without the need for a specialized index is *Merge-Join-based*. For this operator, the inputs have to be sorted on the primary keys. If there are tuples with identical PKs from both inputs, only the tuples from the overlay input are returned. An interesting property of this operator is that the result is implicitly sorted on the primary key.

5.2.2. Special Index-Supported Variants

Special index support can be used by overlay operators to look up tuples by primary key. We implemented two different variants, both supporting pipelined execution.

Separate Indexes for Shared and Private Data

The first approach uses two separate indexes—one for the common data and one for the tenant-specific data. The index for the common data can be shared across tenants. For this first approach, two different strategies may be employed: tenant first or common first access.

Tenant-First Access This strategy first accesses a tenant's data before the shared data. When accessing a tuple by its key, at first the tenant-specific index is queried. If an entry is found there, the tuple is retrieved immediately. If there is an index miss on the tenant-specific index, the shared index is queried for an index hit. Only if both indexes do not return a match, a tuple with the requested key does not exist.

[2] The shown sub-plan reconstructs the VPT of Tenant T_3 from the polymorphic relation in Figure 4.3.

> **Example 5.1:** *Figure 5.3a shows data segments and indexes for two tenants T_1 and T_2. Furthermore, a shared segment with an index exists. The queries Q_{T1} and Q_{T2} of tenant T_1 and tenant T_2, respectively, request a tuple with the same key. Tenant T_2 has overwritten the tuple, tenant T_1 not.*
>
> *Request Q_{T1} first accesses the index $IdxT_1$. As this access results in a miss (↯), the tuple has to be requested via the shared index. For request Q_{T2}, the index $IdxT_2$ immediately returns the requested entry.*

Common-First Access In contrast, the common first strategy always looks up the key in the shared index first and then does a subsequent look-up in the tenant-specific index, if necessary. For increasing look-up efficiency, we added an additional data structure per tenant to minimize the number of secondary look-ups: a bit-set stores the information whether the tuple has been overwritten by that tenant or not. Thus, the bit-set allows to skip the second index access if the tuple is not overwritten by the tenant. The shared index is enhanced to store a bit-set position together with the physical position, thus for each tenant, the bit-set has the same number of entries, as it is aligned with the shared index.

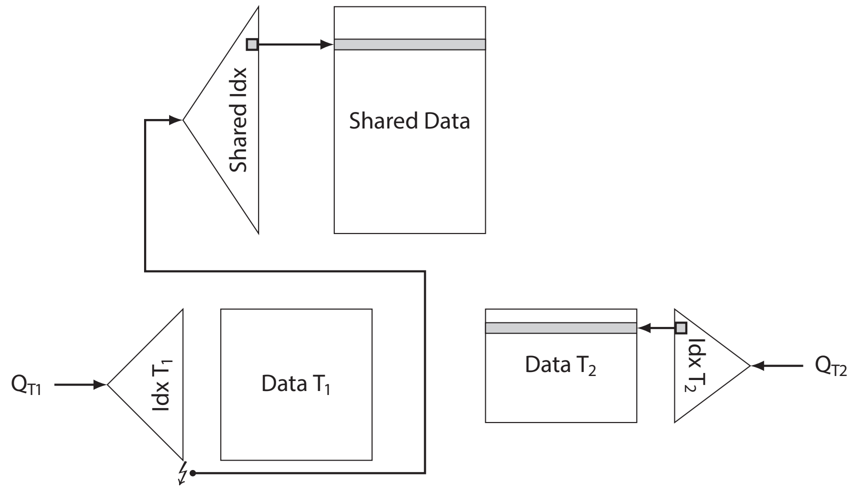
If a tuple with the requested key exists in the shared data, the look-up in the shared index returns its physical location in the shared data segment together with the bit-set position for the tenant index. Then the bit-set of the given tenant is queried for the position obtained from the shared index. If the bit-set has a zero at that position, the tenant has not overwritten the tuple. In this case, the tuple is looked up by the previously retrieved physical location within the shared segment. If the bit-set has a one at that position, the tuple has to be looked up using the tenant index. The look-up key for the tenant-specific index is the same as that for the shared one.

If the key is not found in the shared index, there has to be a second index look-up in the tenant-specific index. Only if both indexes do not contain an entry, the key is not present.

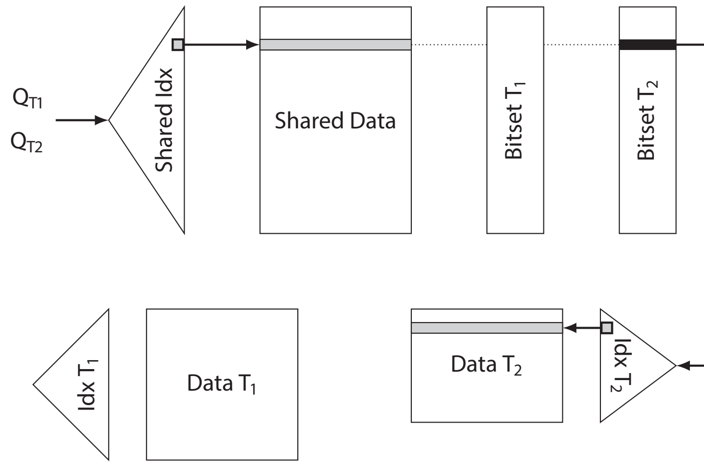
> **Example 5.2:** *Consider Figure 5.3b. The setup is identical to the previous example. Again, both tenants request the same key.*
>
> *Both requests first access the shared index. The shared index look-up retrieves the position of the tuple in the shared segment, as well as the position in the tenant-specific bit-set. The bit-set of tenant T_1 does not have an entry at that position, thus Q_{T1} retrieves the tuple stored from the shared data segment. As the physical position of the tuple is already known, no further index access is necessary.*
>
> *In contrast, tenant T_2 has an entry in its bit-set. Thus a subsequent look-up in its private index must be performed to retrieve the tuple from T_2's data segment.*

Single Combo Index

The second approach employs a tenant-specific combo index which actually indexes the result of the overlay operator, and therefore contains the keys for the shared data as well as



(a) Tenant First Strategy



(b) Common First Strategy

Figure 5.3.: Overlay with Separate Indexes for Shared and Private Data

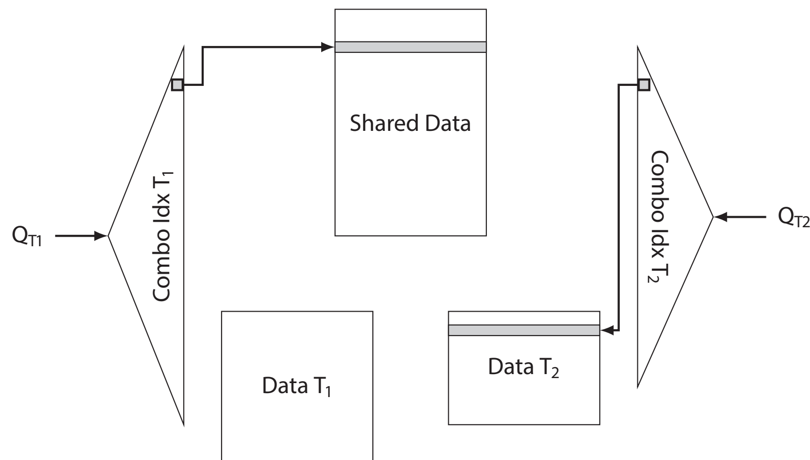


Figure 5.4.: Overlay with Single Combo Index per Tenant

the tenant-specific data. For each key, the reference to the segment (a single bit—shared or tenant-specific) is stored together with the physical tuple identifier. When looking up a key, the operator has to select either the shared or the tenant-specific segment, depending on the segment reference from the index look-up result. Since this type of index cannot be shared, it affects tenant packaging. In a polymorphic relation, combo indexes are only associated with instances at the leaf level.

Example 5.3: *Figure 5.4 shows the usage of combo indexes for the two tenants T_1 and T_2 already known from the previous examples. The request is directed against the tenant-specific combo index. If there is an index hit, the index entry contains a reference to the data segment where the tuple resides. Q_{T1} accesses a tuple that resides on the shared data segment, while Q_{T2} accesses a tuple from the T_2's private data segment.*

5.2.3. Data Maintenance

In our prototype, the segment containing the shared data is read-only and thus each tenant has to maintain an individual segment for storing the tenant-specific changes to the shared data. As the data is spanning across multiple segments, data maintenance operations are different for changing shared data or private data.

Tenant-specific Data Update

Besides maintaining the actual data segments, the additional structures like indexes and bit-sets have to be maintained. Before inserting a new tuple in the tenant-specific segment, a look-up in the shared data index must be performed to retrieve the bit-set position. If this look-up does not return a result, the new tuple does not overwrite a shared tuple and

thus the tenant-specific data segment and its index can be updated. Otherwise, besides updating the segment and its index, the appropriate position in the bit-set has to be set.

Updating an already existing tuple incurs a check if the tuple is still only in the shared data segment or if it has been updated some time before. If the tuple has been updated previously, a corresponding entry in the tenant-specific index and the bit-set is already present, so only the private data segment must be updated[3]. Otherwise, the tuple and its bit-set position has to be retrieved. The tuple is then modified and inserted in the tenant-specific data segment and its index. Finally, the bit-set position has to be set.

For deleting a tuple, a *compensation record* may be necessary: if a shared tuple should be deleted, the private segment must contain a record to mark such a tuple as deleted. The procedure is similar to the updating procedure. If the tuple is defined in the private segment without overwriting an existing record, it can be simply deleted from the tenant's segment and its index.

Globally Shared Data Update

In our prototype, the segment containing the shared data is read-only. However, there might be situations where the shared data has to be updated globally, although we assume that this is rarely the case.

Those global changes on shared data may affect the tenant-specific data structures. If a globally shared tuple gets updated, its physical location must not change. All traditional DBMSs can already handle this. If a global tuple gets deleted, it only disappears for those tenants who did not overwrite the tuple. Furthermore, if the bit-set position number is not recycled, the tenant-specific bit-sets are still valid. When inserting a tuple, it is only visible to those tenants that do not have a private tuple with the same primary key. What is more, the tenant-specific bit-sets become invalid, so they have to be recreated. For the combo index approach, all insert and delete operations on shared segments result in maintaining all existing combo indexes.

5.2.4. Secondary Indexes

For some setups, it might be interesting to have secondary indexes on shared data, e.g. for performance reasons or for ensuring constraints. For these secondary indexes, the same mechanisms as discussed above can be used. However, as our prototype is main-memory-centric, it may be sufficient for some workloads to do full scans of the table. Our evaluation in Chapter 5.5 shows that scanning data with the overlay operator only places a little higher load on the system as scanning conventional data segments.

5.2.5. Overlay Hierarchies

The previous sections only describe overlaying an existing shared data segment with a single tenant-specific data segment. However, there may be scenarios where this single-level

[3] We assume the classic slotted pages layout, where a tuple never changes its initially assigned physical tuple identifier.

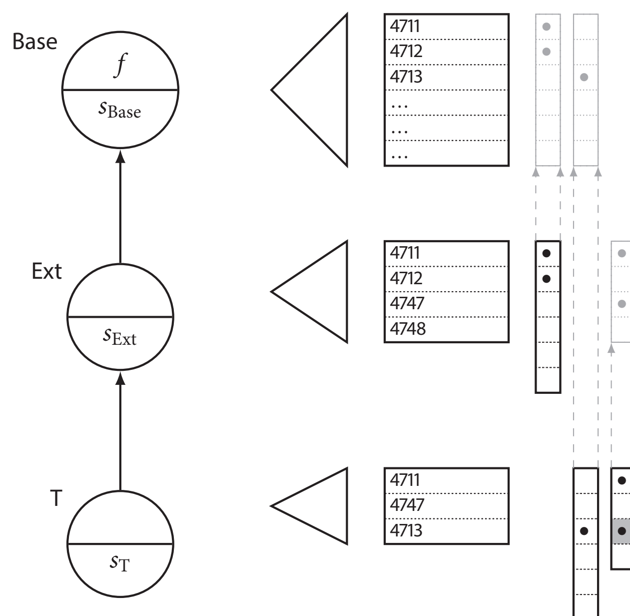


Figure 5.5.: Overlay Hierarchy

overlay is not sufficient, for example, if a tenant subscribed for an extension already overriding shared data. Thus, the overlay mechanism has to support *Overlay Hierarchies* by allowing multi-level overlays.

> **Example 5.4:** *Figure 5.5 shows an excerpt of a Polymorphic Relation on the left, where a tenant* T *overrides data from a extension* Ext *that in turn overrides shared data in* Base.
>
> *For each level of the hierarchy, our prototype maintains an index (the triangles in the figure), the segment, and bit-sets (densely printed) for invalidating tuples from upper segments, e.g., at level* Ext *one bit-set for the* Base *level is present.*
>
> *As the bit-sets affect above-laying segments, they have been duplicated (lightly printed) and raised in the figure for better readability. For example, the marked bit-set-entry at level* T *invalidates the tuple* 4747 *in level* Ext.

Within an hierarchy, the lowest level overrides the data from all upper levels, thus the overlay hierarchy has to be evaluated top-down. For example, let the hierarchy be *Base – Ext – T*, then building the hierarchy is evaluated as following:

$$\mathit{overlay}\left(\mathrm{T}, \mathit{overlay}\left(\mathrm{Ext}, \mathrm{Base}\right)\right)$$

The top-down evaluation of the overlay hierarchy may be applicable, if the overlay of *Ext* with *Base* has a lower cardinality than *T*, i.e. the local changes are dominant within the hierarchy. However, there may be situations where a bottom-up evaluation may be the better choice, e.g., if the globally shared data are predominant in the hierarchy, and thus the overlay of *T* with *Ext* has a lower cardinality than the global data.

As the overlay operator is associative[4], any overlay hierarchy can be either evaluated top-down or bottom-up. Thus the above hierarchy from the example can also be evaluated as follows:

$$\mathit{overlay}\left(\mathit{overlay}\left(\mathrm{T}, \mathrm{Ext}\right), \mathrm{Base}\right)$$

Impact of Overlay Hierarchies on Data Structures

The associativity of the logical overlay operator affects those physical implementations of the operator which maintain external structures for performance reasons. The changes affect the combo indexes and the bit-sets.

Combo Index The combo indexes are indexing the result of the overlay across the whole hierarchy. Thus, each index entry has to refer to the appropriate level within the hierarchy where the referenced tuple is actually stored. Write operations to the leaf associated with the combo index result in index maintenance operations; write operations to upper levels render all dependent combo indexes invalid. Either the index has to be rebuilt or the Multi-Tenant DBMS has to offer mechanisms to update multiple dependent combo indexes if shared data changes.

Bit-Sets As Figure 5.5 shows, each segment has one bit-set for all inherited segments. Thus, when writing tenant-specific data, the write mechanism has to determine if (1) a new tuple is inserted, or (2) an existing tuple is overwritten, and if so, at which level of the hierarchy the tuple has been defined before. Modifying the bit-set associated with the definition level of the tuple allows for bottom-up or top-down overlay, as described in the next section.

> **Example 5.5:** *Consider Figure 5.5. Tenant T wants to update the tuple with key* 4711. *Right before this update, a query for the key* 4711 *would return the value from the segment* s_{Ext}*, at whose level the tuple* 4711 *has been defined before. Thus, the update mechanism has to insert the new value with key* 4711 *at segment* s_T *and modify the appropriate bit in the tenants bit-set associated with the segment* s_{Ext}.

Lookup and Scan

When using overlay operators without index support, such as merge-based overlay, there is no specialization necessary. The placement of these operators in query execution plans is similar to join operators, such that they result in either deep trees or bushy trees. Furthermore, the associativity of the overlay operator can be exploited. As the schema of the individual inputs and interim results are identical, one possible optimization is the introduction of N-ary overlay operators, that allow multiple overlays for one base segment.

[4] The proof can be found in Section A.1

As the combo index acts as a tenant-specific index for the result of the overlay across the hierarchy, the only difference to two-level overlays is the fact that the combo index has to distinguish multiple overlay levels.

Due to multi-level overlays, the tenant-first and the common-first access strategies become the *bottom-up* and the *top-down* strategies, respectively. With the bottom-up strategy, point-wise look-ups do not need bit-set support, and thus the bit-sets are used for scans only. With the top-down strategy, the bit-sets are further necessary for point-wise look-ups.

For filtering out the overwritten tuples from upper levels in the hierarchy, all bit-sets across the hierarchy corresponding to the same segment, are bit-wise ORed to form one single bit-set. This single bit-set is then applied to the segment, thus filtering out tuples that are overwritten at lower levels in the hierarchy.

Example 5.6: *Again, consider Figure 5.5. For filtering overwritten tuples from segment s_{Base}, the appropriate bit-sets at Level s_{Ext} and s_T are bit-wise ORed. The resulting bit-set is then used to filter the tuples with keys* 4711, 4712, *and* 4713.

This mechanism collapses multiple bit-sets for one particular segment. These resulting bit-sets are then applied to the corresponding segments. Finally, the result of the overlay can be retrieved by combining the filtered segments across the hierarchy, either bottom-up or top-down.

5.2.6. Comparison to Other Delta Mechanisms

Very similar to the overlay mechanism as described above are those mechanisms that are used in read-optimized database systems. Those systems offer a read-optimized store in which the data is specially organized to exploit performance optimizations. A very popular example of such a specialized optimization are column-oriented databases where each column of a table is stored in its own container. Tuples are reconstructed by aligning the columns of one particular table which is done using offsets. As each column is sorted by the same sort key, a particular tuple can be identified by its offset. In such column databases, scans and aggregations across one single column can be processed very fast, thus exploiting the low amount of cache misses (Boncz, 2002). However, as updates and inserts have to be performed in compliance with the organization of the table, such operations become costly. Write operations are collected in a write-optimized store and are regularly propagated into the read-optimized store, thus transforming single write operations into one batch operation. Representative implementations of this mechanism are C-Store (Stonebraker et al., 2005) and its commercial offspring Vertica (Vertica Systems, 2011). As in these setups data is spanning across two different storage areas, a query has to access both of them and merge them on-the-fly to guarantee up-to-date results.

For managing the write-optimized store—the so-called *delta*—there are two strategies: the value-based delta and the positional delta. In both setups, the delta keeps track of which tuples were inserted, deleted, or modified.

The value-based delta uses a RAM-resident data structure whose sorting is based on the sort key of the underlying read-optimized store. Thus, when merging the delta with the read-optimized store, the tuples of the two storage areas are aligned by the common sort key attributes. Typical implementations for value-based deltas are RAM-friendly B-Trees. They can efficiently be merged into the read-optimized store by simply scanning the leaves of the B-Tree. Value-based deltas have a major drawback when used in read-mostly databases: for queries, the delta has to be transparently merged into the read-optimized store. However, if the sort key spans across multiple attributes, the scan has to include all of them, which may reduce scan performance, e.g. if only a single column is requested. Other techniques follow the rationales of *Log Structured Merge Trees* (O'Neil et al., 1996), which also transform random write operations into a batch operation.

Positional deltas align the tuples by the position of the tuple instead of the sort key. Héman et al. (2010) propose a variant called *Positional Delta Trees* (PDT). The PDT contains the same form of differences as in the value-based delta, but the data structure has been optimized for a fast merge operation by providing the actual tuple positions of the read-optimized store where the differences have to be applied. Thus, scans on the read-optimized store can count down to the next position where a differential update has to be handled. This way, the scan is independent of the sort key used for organizing the read-optimized store.

Not only column-oriented database systems make use of deltas; there are also scenarios outside the column-oriented world where deltas become reasonable: RDF-3X (Neumann and Weikum, 2010) is a specialized database system for RDF data. As it heavily relies on indexing, a single update would incur write access to 15 compressed indexes. This renders direct updates unattractive. Thus, RDF-3X uses a staging architecture, where write access to the RDF store are redirected to differential indexes, that only contain the write operations. These differential indexes form the delta of the RDF-3X system, and are regularly merged into the RDF store, thus collapsing singular write accesses into one big batch update operation. This procedure lowers the access costs, however, the differential deltas have to be merged on-the-fly by the query processor during queries. Therefore, the query processor integrates additional merge joins into the query execution plan.

All the described systems merge the contents of the delta into the main stores, either periodically or as soon as a certain threshold has been reached. This behavior is the main difference between their and our approach. As globally shared data (i.e. "the main store") and the tenants' data (i.e. "the delta") must be completely separated, there is no merging across the overlay hierarchy. Furthermore, all systems from above have been designed for scan-intensive OLAP applications and are not well-suited as transactional systems. To increase the overall performance in OLAP systems, it is evident that scan performance is the critical factor. However, this is not the case in our prototype, where OLTP-style look-ups and updates make up the majority of a typical SaaS workload. Thus, our approach makes use of value-based deltas, in combination with specialized indexes to support fast look-ups.

5.3. Shared Data Versioning

As our Multi-Tenant DBMS prototype allows for schema flexibility, the mechanisms of data sharing have to be highly integrated with the extensibility and evolution mechanisms. FlexScheme allows for schema evolution by keeping multiple schema definitions within one fragment sequence. Furthermore, within one segment sequence multiple versions of the data can exist to allow seamless application evolution.

These mechanisms are useful when upgrading the SaaS application to a new revision. Most service providers schedule a downtime on a regular basis where the application is upgraded. After that downtime, all tenants use the new application revision. However, this may not be desirable in every case: some tenants heavily rely on extensions developed by themselves which require a particular application revision. Those tenants may be willing to pay an increased subscription fee if they can stick with a certain application revision, or at least, if they can defer the upgrade.

As new application revisions may also lead to updates of the globally shared data, the data sharing mechanism must be able to handle not yet updated tenants as well. We thus extend the data sharing component of our Multi-Tenant DBMS prototype with a *shared data versioning component* which is designed to exploit a typical SaaS application's life-cycle: a SaaS application is updated on a regular basis by the service provider. As a single tenant may have extensions, either developed by himself or by an ISV, those updates may break the application for that tenant. To guarantee application availability, the tenant may defer the update to the new version of the base application and the extensions, respectively. Therefore it might be possible, that a few tenants lag marginally behind the most recent version of the application. However, we assume that a majority of tenants is using the most recent version of the application.

5.3.1. Pruning of Unused Versions

FlexScheme keeps track of fragment and segment usage. Fragments and segments that are no longer accessed by tenants can be pruned, thus improving space efficiency; this is very important, especially for main-memory DBMSs, so pruning should be done as soon as possible and as often as possible, but without affecting the tenants' performance.

The pruning of a segment from a segment sequence erases the segment. We refer to this operation as the *Purge* operation. A purge operation can occur at any time, as it may be a direct result of a particular tenant's actions, for example, if the tenant is the last remaining user of a specific segment and migrates to a subsequent version within the segment sequence. Thus, the purge operation has to be a light-weight operation, to be able to perform this operation whenever needed without affecting co-located tenants.

5.3.2. Tenant-specific Reconciliation

As tenants can freely decide whether they immediately follow application upgrades or postpone them, the application has to provide routines for individually pulling global changes into the tenant's workspace. This leads to three different scenarios that have to be handled:

1. A tenant wants to immediately update to the newest application revision, including all dependent shared data.

2. A tenant wants to defer the application upgrade, and temporarily stays with its current application revision, including its dependent shared data.

3. A tenant does not want to upgrade the application at all, and wants to stick at its current application revision, including the dependent shared data.

The scenarios (2) and (3) from above are very closely related, as tenants—even if the tenant decided to stick at some particular application revision—may want to roll forward to the most recent version after they kept their old revision for a very long time. As we allow these different scenarios, there is no need for special treatment of off-line tenants. However, depending on the above scenarios, there may be synchronization conflicts during application upgrade if a tenant made changes to global data.

Example 5.7: *Suppose the initial segment $s_{Base}^{(1)}(f_{Base})$ of the Base segment sequence contains two tuples with keys* 17 *and* 42. *Tenant T's overlay segment $s_T^{(1)}(f_{Base})$ contains two entries* 17 *and* 55, *thus overriding entry* 17 *and adding a new entry with key* 55.

Now, the global Base segment gets updated by the service provider, by adding a new segment $s_{Base}^{(2)}(f_{Base})$ which contains an additional tuple with key 55. *If tenant T moves from the initial segment $s_{Base}^{(1)}(f_{Base})$ to the new segment $s_{Base}^{(2)}(f_{Base})$, the tuple with key* 55 *has conflicting changes which have to be resolved.*

Conflict resolution is highly application dependent, thus the Multi-Tenant DBMS cannot reconcile autonomously, but has to provide mechanisms to facilitate reconciliation. Typically, the reconciliation is based on a three-way merge algorithm, as frequently used by revision control systems like Subversion (SVN). Such a merge algorithm takes the shared data segment $s_{Base}^{(N)}(f)$ in revision N, the tenant-specific changes $s_T^{(N)}(f)$, and the new shared segment $s_{Base}^{(N+1)}(f)$. The merge algorithm then creates a new tenant-specific overlay segment $s_T^{(N+1)}(f)$ which is based on the new shared data segment. A three-way merge can recognize conflicting changes. These detected conflicts are then used as input for application-specific conflict resolution mechanisms inside the SaaS application.

In the Mobile DBMS world, such reconciliation processes occur frequently. Saito and Shapiro (2005) state that optimistic replication allows for efficient data sharing in mobile environments and wide-area networks. In contrast to pessimistic algorithms that do synchronous updates, the optimistic algorithms let data be accessed without a priori synchronization.

Optimistic replication allows for sites and users to remain autonomous. However, as optimistic replication is based on asynchronous operations, consistency may suffer in the form of diverging replicas and conflicts between concurrent operations. One of the prominent usage scenarios for optimistic replication is Revision Control Management, like SVN. Optimistic replication can be transfered to our usage scenario. In the shared data setup,

the shared data segment forms the main site, whereas the tenants' segments form remote sites. When the main site gets updated, there may be conflicting concurrent operations.

For shipping data between two replicas, optimistic replication has the notion of *state transfers* and *operation transfers*. If two replicas are synchronized via state transfers, only the final values for particular keys are shipped and then compared. Thus, the reconciliation process can solely decide on the values of the entries which of the possible alternatives is selected as the new value.

Example 5.8: *Suppose there are two replicas D_1 and D_2 of a database. In the initial state, the value* 42 *has been stored for key k. The user of D_1 increased the value of key k to* 43*. During reconciliation, the state information $D_1 : k \leftarrow 43$ and $D_2 : k \leftarrow 42$ has to be merged.*

Operation transfers do not ship the final value for a particular key, but rather the operations that have led to the new value. This way, the reconciliation process can leverage the semantics to find a reconciled value.

Example 5.9: *Let the initial setup be as in the previous example, and let the user of D_1 perform the identical operation as before. During reconciliation, the operation information $D_1 : k \leftarrow k + 1$ and $D_2 : k \leftarrow k$ has to be merged.*

Our prototype makes use of both transfer methods. For fragment sequences, we exclusively use operation transfers to connect subsequent fragment versions. For the segment sequences, we offer different physical data representations, that offer either state or operation transfers. An in-depth discussion of the physical representations can be found in the following section.

5.4. Physical Data Organization

In this chapter, we compare different physical data organization schemes. We discuss the data structures, the access behavior of point-wise look-ups and scans, maintenance operations on FlexScheme's segment sequences as well as the space efficiency of these approaches.

The maintenance operations on segment sequences such as creating or deleting segments should be lightweight operations to avoid performance impacts. A new segment is based on the last segment of a segment sequence. Changes to the new segment become available only after the new segment is released. This way new segments can be prepared without affecting co-located tenants. Purging a segment is only possible, if no tenant depends on it any more.

This section covers versioning of data. Data versioning is only useful for shared data, i.e. at base or extension level. However, there may be circumstances where data versioning may be applicable for tenant-specific data as well. However, the approaches presented in this section are primarily optimized for shared data versioning.

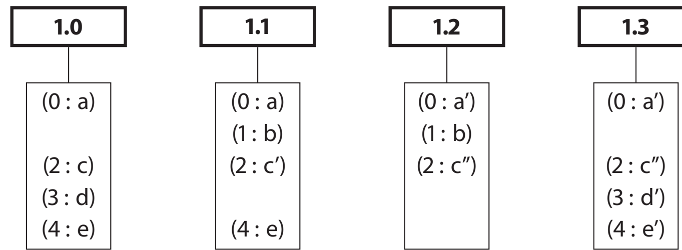


Figure 5.6.: Snapshot Approach

5.4.1. Snapshot Approach

The simplest data organization scheme for segment sequences is to materialize each segment. We refer to this approach as the *snapshot* data organization scheme. Figure 5.6 shows the snapshot data organization scheme for one segment sequence. In the example, there are four materialized segments which are all currently used by tenants.

Data Structures Each materialized segment is stored within a separate storage area which contains the data tuples and an index on the primary key of the corresponding fragment. In the following, we refer to this primary key as the tuple identifier.

Point-Wise Access Data tuples of a materialized segment can be accessed point-wise by looking up the tuple identifier in the index of the materialized segment.

Scan Access An index scan can be used to scan the data tuples of a materialized segment sorted by tuple identifier.

Creation of New Segments A new segment in a segment sequence is created by copying the data area and its index from the last segment in the segment sequence. The newly created segment is then available for modifications.

Purge Operation A materialized segment can be purged by deleting the corresponding storage area and its index. Other segments are not affected by this operation as those are stored in different storage areas. Therefore the purge operation is a lightweight operation in the snapshot data organization scheme.

Space Efficiency The big disadvantage of the snapshot approach is that redundancy between subsequent segments is not eliminated. There is a high potential for redundancy across segments within a segment sequence, if there are only few changes from one segment to the next. Orthogonally, value-based compression can be applied to single segments to further reduce space requirements.

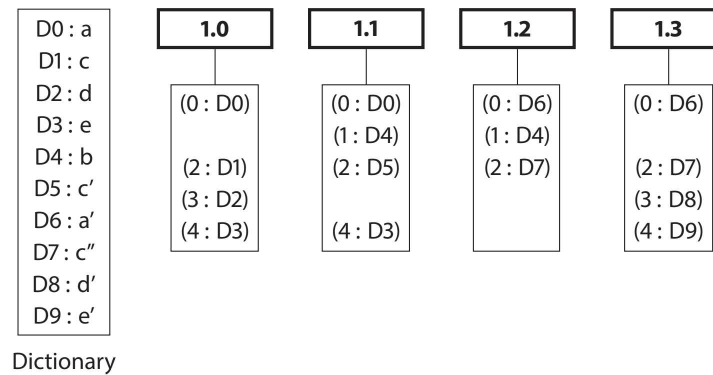


Figure 5.7.: Dictionary Approach

5.4.2. Dictionary Approach

The snapshot data organization scheme can be refined to eliminate redundancy across segments of the same segment sequence. The *dictionary* approach has one independent data area for each segment to store the segment index and a separate storage area for the dictionary of the whole segment sequence. Tuple values are stored in the dictionary; the segment indexes then only reference the tuple values of the particular segment. Figure 5.7 shows the dictionary data organization scheme.

Data Structures Besides the dictionary, each segment sequence has one index per segment. The indexes reference the values from the dictionary. A reference counter for each entry in the dictionary keeps track of how often an entry is referenced in segment indexes.

Point-Wise Access The data tuples of a segment can be accessed point-wise by looking up the tuple identifier in the index and then retrieving the value from the dictionary.

Scan Access For a scan across a segment, an index scan across the segment's index has to be performed. For each entry in the index, the corresponding value from the dictionary needs to be retrieved.

Creation of New Segments When creating a new segment, the index of the last segment has to be copied. Then, the new index has to be scanned, and, for all entries in the index, the reference counter in the dictionary has to be increased.

Purge Operation The purge operation has to delete the segment index and has to remove obsolete values from the dictionary. For all tuples referenced in the index to purge, the reference counter of that tuples must be decreased. Only then, the segment index can be deleted. If the reference counter reaches zero, the tuple can be completely removed from the dictionary. Since the purge operation modifies a shared data structure, which may affect performance, we consider this operation as heavyweight.

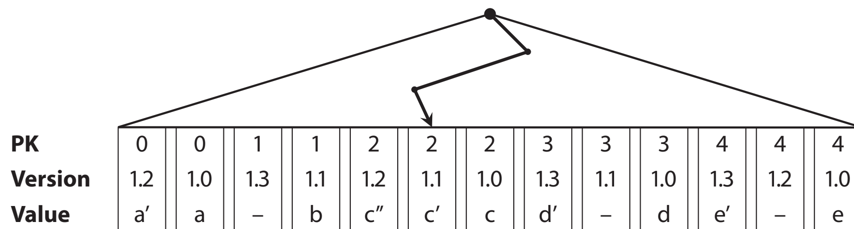


Figure 5.8.: Temporal Approach

Space Efficiency With this approach, the redundancy in the data areas of the snapshot approach is eliminated by introducing a shared dictionary. The size of the indexes remains unchanged since the indexes of the dictionary approach are organized similarly to the indexes of the snapshot approach. Further compression of the dictionary is orthogonal.

5.4.3. Temporal Approach

Temporal databases (Stonebraker, 1987; Jensen and Snodgrass, 1999, 2009) can manage data in many different versions, which makes them an interesting candidate for managing a segment sequence. The interesting feature of temporal databases is that only those versions of a tuple are stored in which the value of this tuple changes. However, the temporal database allows to query any version of a tuple. When a segment sequence is stored in a temporal database, a segment corresponds to a specific version number. We refer to this approach as the *temporal* data organization scheme. Figure 5.8 shows that approach.

Data Structures Temporal databases typically handle the versioning by a specialized index, the *temporal index*. The entries of such an index are augmented by versioning information which can be comprised of, for example, validity timestamps or version numbers. Our setup uses the latter. In comparison to the dictionary approach, there is only one common index, the temporal index, instead of one index per segment. Figure 5.8 shows a simplified temporal index, since the data tuples are stored in the leaf nodes of the temporal index, rather than in the storage area. In our implementation the version numbers are stored bit-wise inverted; this is useful if you want to query the temporal index for a particular key without specifying the version information. This way the index can perform a lower bound access to retrieve the most recent version of the key.

Point-Wise Access The tuples of a segment can be accessed point-wise by performing a lower bound look-up of the key. The version with the highest version number smaller or equal to the desired version is returned.

Scan Access Temporal databases support to scan all tuples of a version sorted by tuple identifier. As each segment corresponds to a specific version number, all tuples of a segment can be scanned. However, to perform this operation, temporal databases have to scan the whole temporal index which may be large, as it contains entries for all versions of all tuples in which the value of a tuple changes.

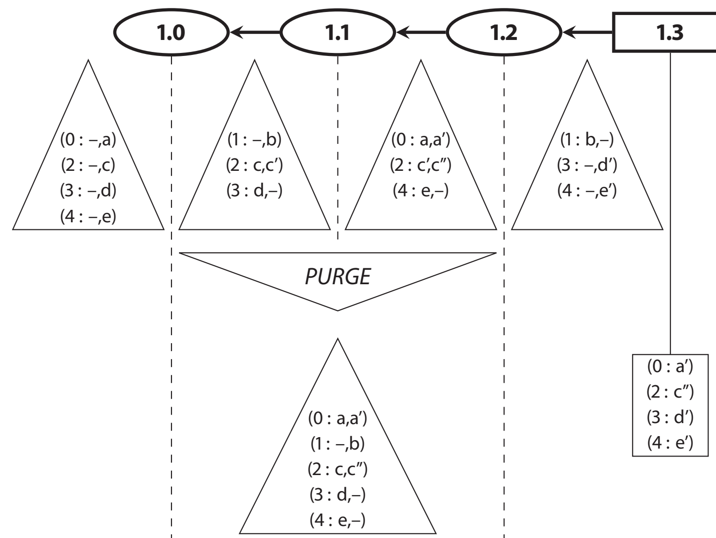


Figure 5.9.: Differential Delta Data Approach

Creation of New Segments For the creation of new segments, only the internal information referring to the most recent version number has to be updated.

Purge Operation In contrast to the dictionary approach, the data of all segments is stored within one common temporal database. The purge operation has to remove all tuples of a given version from the temporal database. For a given version of a tuple, the temporal database may not have an entry. This occurs if the tuple has not been changed in this version. In this case nothing has to be done to purge the tuple from the temporal database. However, if the temporal database contains an entry for the given version of a tuple, then this entry cannot be removed right away, because it may be required for other versions of the tuple. Then, the entry has to be replaced by a new entry for the next higher or next lower implicitly stored tuple. This process is performed on a common data structure, and thus makes the purge operation heavyweight.

Space Efficiency As only those versions of a tuple are stored in which the value of this tuple changes, the temporal database eliminates a lot of redundancy. For a further reduction of space requirements the data area containing the actual values can be compressed, which is orthogonal.

5.4.4. Differential Delta Approach

In the *differential delta* approach, the differences between two subsequent segments of a segment sequence are stored in one storage area. We refer to these differences between two segments as the *delta*. As the deltas of all preceding segments have to be overlaid to restore the tuples of a given version, it is inefficient to perform operations on arbitrary segments of a segment sequence only with deltas. As discussed above, we assume that the major part of the workload is performed on the last segment of a sequence, and hence we materialize the

last segment. We refer to this approach as the differential delta data organization scheme. Figure 5.9 shows this approach.

Data Structures The delta corresponding to a segment of a segment sequence is the set of differences between the given segment and the preceding segment. Each delta is stored as a B^+-Tree. Each entry has the primary key of the affected tuple as key and a *diff entry* as value. A diff entry stores two values of the affected tuple, the values before and after the change. Therefore, the differential delta approach has the interesting property that a delta can be used in both directions, forward and backward. We refer to this as *random direction property*. The last segment in the segment sequence is materialized. The data is stored in a separate data area and indexed by a separate index.

Point-Wise Access Point-wise access behavior depends on the queried version. If the materialized segment is queried, the differential delta approach behaves like the snapshot approach. If other segments are queried, there may be a series of look-ups in the materialized version as well as in deltas: in case the delta corresponding to the wanted segment contains a diff entry for the given key, the after value of that diff entry represents the value of the tuple. Otherwise, the preceding deltas must be iteratively queried to find the correct value. The iteration terminates as soon as a diff entry is found, or if the first segment of the segment sequence has been reached. If a diff entry is found, the after value contains the tuple data. Alternatively, the succeeding deltas can be iteratively queried. In this case, the iteration terminates at the end of the sequence or as soon a diff entry has been found. The tuple value is then retrieved from the before value of the found diff entry.

Scan Access Scans can be performed by overlaying the deltas and the materialization of the last segment. This overlay operation can be implemented efficiently with an N-ary merge join.

Creation of New Segments For creation of a new segment, a new B^+-Tree has to be generated. We use the random direction property of the differential delta approach to prepare a new delta in the newly allocated tree. As soon as the new segment is released, the new delta is overlaid with the currently available materialization to form a new materialization. The old materialization is then removed. This removal is lightweight, since the deltas do not depend on a particular materialization.

Purge Operation When purging a segment, it is not sufficient to simply delete the affected delta. Rather, this delta has to be merged into the delta of the succeeding segment in the segment sequence. The actual purge operation is based on a merge join. If only one of the deltas contains an entry with the given key, the diff entry is simply copied. Otherwise, if both deltas contain an entry with the given key, a new diff entry is created with the before value of the affected segment and the after value of the subsequent segment. An example of a purge operation is shown in Figure 5.9. Since this can be performed without affecting shared data structures, we consider this a lightweight operation.

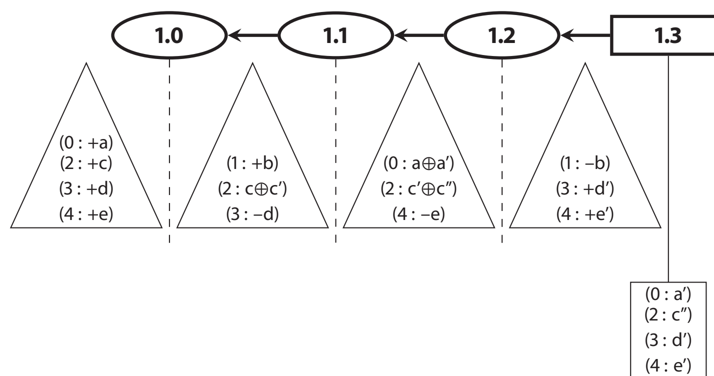


Figure 5.10.: XOR Delta Approach

Space Efficiency The differential delta data organization approach eliminates some redundancy compared to the snapshot approach, as only the differences between subsequent segments are stored. However, the diff entries may still contain redundancy if only a small part of the tuple's data changed. This can be addressed by employing an XOR encoding scheme (see next section). Compression of a particular delta to further reduce space requirements is orthogonal. Space consumption can be reduced even further by eliminating redundancy across deltas of the same segment sequence. To eliminate this redundancy some form of common data structure between segments would be required, e.g. a dictionary. With such a common data structure, the data that corresponds to one segment is spread all over the common data structure. This turns operations on all data of one segment into heavyweight operations, especially the purge operation. Since this is not desirable, we do not eliminate this redundancy.

5.4.5. XOR Delta Approach

The *XOR delta* approach is based on the previously discussed differential delta data organization scheme. The previous approach enables lightweight maintenance operations for segment sequences. However, within a diff entry there is still potential of eliminating redundancy. The XOR delta approach addresses this issue by XOR-encoding (Cha and Song, 2004) the diff entries of the differential delta approach. We refer to this optimization as the XOR delta approach. Figure 5.10 shows the XOR delta data organization scheme.

The random direction property of the differential delta approach is retained in the XOR delta approach. We make use of this property when creating, purging, or querying segments.

Data Structures The XOR delta data organization scheme is an enhancement of the differential delta data organization scheme. The only difference is the implementation of diff entries. Instead of storing the before and the after value of the tuples in the diff entry, the XOR-encoded value *before* ⊕ *after* is stored. Furthermore, each XOR entry denotes either an insert, an update, or a delete operation. Again, the last segment of a segment sequence is materialized.

Algorithm 1 Lookup tuple with key k in version *ver*

```
1:  V ← nearest materialized version to ver
2:  if ver ≤ V then
3:    data ← retrieve data for key k from materialization V {data might be NUL}
4:    for i = V to ver + 1 do
5:      xor ← XOR entry for key k from delta (i − 1 ↔ i)
6:      if xor is update then
7:        data ← data ⊕ xor
8:      else if xor is delete then
9:        data ← xor
10:     else if xor is insert then
11:       data ← NUL
12:     else
13:       continue {No XOR entry found, do nothing}
14:     end if
15:   end for
16:   return data
17: else
18:   [...] {Symmetric to lines 3 to 16}
19: end if
```

Point-Wise Access Algorithm 1 describes the point-wise access by primary key, which is closely related to the point-wise access in the differential delta approach. Since the last segment of a segment sequence is materialized, this segment is used as starting point. For each delta between the target version and the materialized version, the XOR entries are retrieved. The result is then calculated by XOR-ing the XOR entries with the materialized segment.

Example 5.10: *To access the tuple with PK 2 in version 1.1, cf. Figure 5.10, the value c'' of the tuple with PK 2 at the materialized version 1.3 has to be XOR-ed with the value $(c' \oplus c'')$ of XOR Entry 2 from Delta (1.2 → 1.1). Delta (1.3 → 1.2) has no matching entry for the PK 2, so it can be skipped. Since XOR is associative and commutative, the result of this operation is $c'' \oplus (c' \oplus c'') = (c'' \oplus c'') \oplus c' = 0 \oplus c' = c'$.*

As in the differential delta approach, a materialization of the last segment increases performance. Otherwise, the tuple value has to be reconstructed starting at the delta of the first segment in the segment sequence, which would be very inefficient for our SaaS workload.

For better performance, other segments than the last in the segment sequence can be materialized as well. Then the random direction property can be exploited for point-wise look-ups by choosing the closest materialization to the desired segment as starting point for the reconstruction of the tuple value. Algorithm 1 already resembles this optimization as it allows to start at the nearest materialized version.

Scan Access As in the differential delta data organization scheme, scans are performed by overlaying the deltas with the materialization using an N-ary merge join.

Creation of New Segments The creation of new segments is identical to the differential delta approach.

Purge Operation As in the differential delta data organization scheme, the purge operation involves the merge of two deltas corresponding to subsequent segments in the segment sequence.

Two XOR entries can be merged by XOR-ing the XOR values of the two entries. Suppose the first XOR entry stores $a \oplus b$ and the second XOR entry stores $b \oplus c$, where a is the value of the affected tuple before the first change, b the value after the first change and thus before the second change, and c the value after the second change. Then the merged XOR entry stores the XOR value $(a \oplus b) \oplus (b \oplus c)$. Since the XOR operation is associative and commutative, this results in $(a \oplus b) \oplus (b \oplus c) = a \oplus (c \oplus (b \oplus b)) = a \oplus (c \oplus 0) = a \oplus c$.

The XOR encoding of the XOR entries has no impact on the complexity of the purge operation. Therefore, the purge operation remains lightweight.

Space Efficiency The redundancy within the diff entries of the differential approach has been eliminated by introducing XOR encoding. If only small parts of a tuple have changed in subsequent segments, the XOR entry could be further compressed by run-length-encoding, since the unchanged bits are zero, but this is orthogonal to the XOR encoding.

5.5. Evaluation

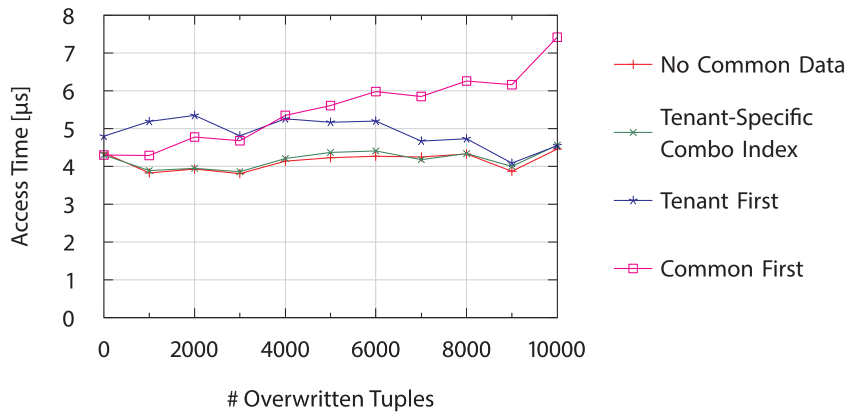
In this section, we evaluate the impact of the overlay operator on the access behavior of our Main-Memory DBMS prototype and compare the different data versioning approaches.

Our Main-Memory DBMS prototype has been implemented in Java 6. Indexes and deltas use Oracle Berkeley DB Java Edition 4.0, which offers a B+-Tree implementation. The fanout of the B+-Trees has been configured to mimic Cache Conscious B+-Trees (Rao and Ross, 2000). Segments are implemented as slotted pages of 4KB, and data is represented as byte arrays only.

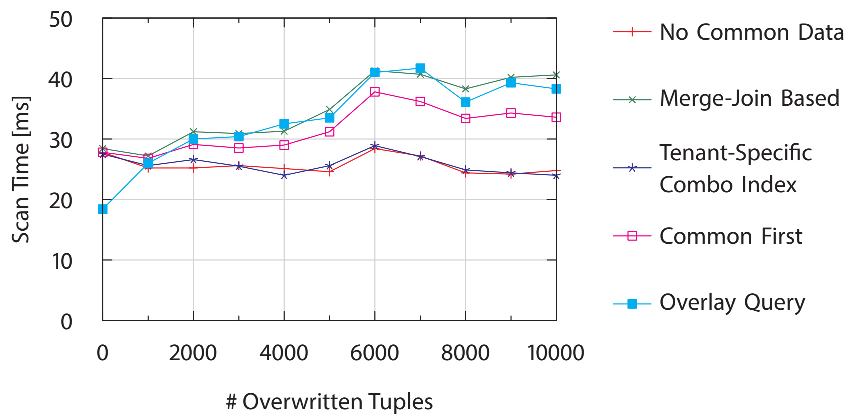
The runtime environment of the experiments is a Intel Xeon X5570 Quad Core-based system with two processors at 2.93 GHz and 64 GB of RAM, running a recent enterprise-grade Linux.

5.5.1. Overlay Operator

At first, we evaluate the performance impact of different overlay operator implementations. For our evaluation we use B+-Trees as index implementation. Hashing based indexes show similar relative behavior, but obviously with lower absolute access times.



(a) Point-Wise Lookup



(b) Full Table Scan

Figure 5.11.: Overlay Operator Response Times

Point-Wise Access

Developers tend to optimize an application's access behavior by using mostly point-wise operations to achieve better performance. Therefore it is crucial that the overlay operator does not place much overhead on the access time for this kind of query.

In Figure 5.11a, we compare the access times of our approach with access times of the traditional approach when no data is shared. The overlay operator is only required when common data is shared. We compare three different physical operators for the logical overlay operator: Tenant-First-Overlay using index sharing without bit-set support, Common-First-Overlay using index sharing with bit-set support, and Tenant-Specific-Combo-Index-Overlay where each tenant has an index also containing references to shared data.

In this experiment the shared data segment contains 10,000 randomly generated tuples. The number of overwritten tuples is increased in steps of 1,000 tuples. Each data point represents the average value across 200 runs. Each run does 10,000 random point-wise accesses. After each run the segment and the indexes are discarded and newly created. The reported value is the average time for one point-wise access.

As baseline we selected a layout where no data sharing occurs. The *No Common Data* result is an evaluation of a Private Table Layout with a single index on the table. Figure 5.11a shows that the *Tenant-Specific-Combo* configuration with its single index has the same performance characteristics as the baseline. This validates our assumption that the single bit, which is stored in the index to denote from which segment the tuple has to be fetched, does not impact performance. The Tenant-Specific-Combo approach does no index sharing.

In a next step, we compare the overhead of implementations with multiple indexes. The Common-First-Overlay operator has a higher hit ratio for lower percentages of overwritten keys, since it first queries the index on the shared data. With increasing percentage of overwritten keys, the seek time increases due to the fact that most shared data has been overwritten. For the Tenant-First-Overlay, the behavior is vice versa. The break-even point of the Tenant-First-Overlay and the Common-First-Overlay operator is at around 40% of overwritten data.

As a sanity check, we implemented point-wise overlay access in a classical DBMS. As a baseline we took point-wise access to the shared table so that no overlay occurs. When implementing overlay with a `UNION` operator as displayed in Figure 5.1, the response times increased up to 300%. In contrast, the performance impact in our setup is limited to 25% in the break-even point of the two multiple indexes implementation.

Scans

Second, we analyze the performance of the overlay operator for scan operations using the same test scenario. The scan returns 10,000 tuples. The results are depicted in Figure 5.11b; each data point is an average across 200 scans. Again, the baseline (*No Common Data*) is a scan against a Private Table Layout. Like in the point-wise access test, the *Tenant-Specific-Combo* implementation causes no additional overhead.

The *Overlay Query* result is an implementation based on the query plan of a SQL query like Figure 5.1; it performs similar to the pipelining *Merge-Join-Based* implementation.

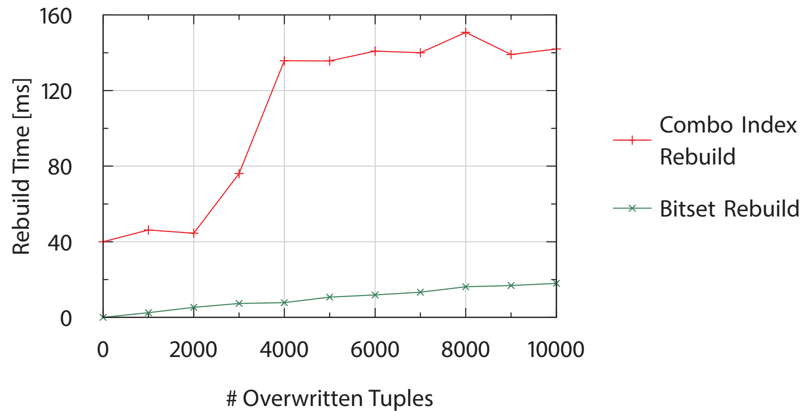


Figure 5.12.: Rebuild Times for Bit-Set and Combo-Index Rebuild

Furthermore, the execution plan of the Overlay Query implements a Tenant-First strategy, so we omitted reporting this result. As soon as 10% of the tuples are overwritten, our specialized *Common-First* overlay operator performs better than the standard implementation. If more than 60% of the tuples are overwritten, the performance gain exceeds 40%.

These results show that the overlay operator can be realized as a lightweight operation in a main-memory DBMS.

Index Rebuild

An interesting aspect of index maintenance in this scenario are the rebuild costs for combo indexes and bit-sets. As previously discussed, there are situations where these structures become invalid, i.e., if shared data has been changed. In this section, we present the time spent for *fully* rebuilding the structures.

Figure 5.12 shows the result of our experiment where we removed the existing tenant-specific structures and measured the raw time for rebuilding them. For recreating the combo index of a tenant, a full scan across the shared segment and then the tenant-specific segment is performed. If no tuples are overwritten, the index rebuild lasts 40 ms, which is the time to build the index across the shared data only. The more tuples are overwritten, the more influence is gained by the index updates during the scan of the tenant-specific segment. For each tuple in the tenant's segment, a look-up for the update position in the combo index has to be performed, followed by the actual index update. The setup of this experiment where we only overwrite existing tuples avoids rebalancing the tree, as there are no additional keys in the tenant's segment. As an alternative, it may be possible to deep clone an already existing index on the shared data which is then updated with a scan across the tenant-specific segment.

Much cheaper are the rebuilds of the bit-sets. For this test, we assume that the tenant-specific index already exists. Thus, if the shared data changes, the bit-set can be rebuilt by performing a merge-join like operation on the tenant-specific index and the shared data index. As this operation can exploit the sorting of the data, the rebuild time decreases.

```
SELECT   account.id, account.name,
         SUM(opportunity.probability
             * opportunity.amount)
           AS value
  FROM   accounts, opportunities
  WHERE  opportunities.account = account.id
GROUP BY account.id, account.name
ORDER BY value ASC
```

Figure 5.13.: Roll-Up Query

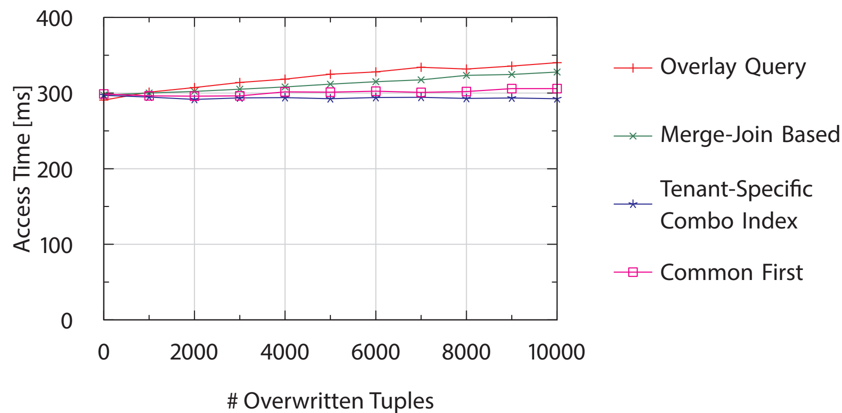


Figure 5.14.: Effects of Data Overlay on Queries

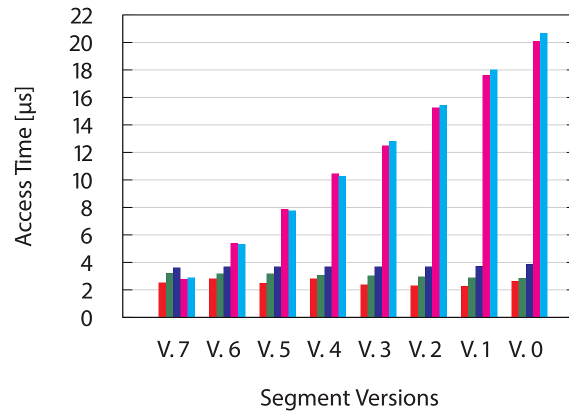
For supporting off-line tenants, this experiment is of special interest: As the rebuild times are fairly short, it might be interesting to drop all tenant-specific indexes and bit-sets, respectively, when a tenant is suspended, and recreate these structures during tenant activation.

Influence on Queries

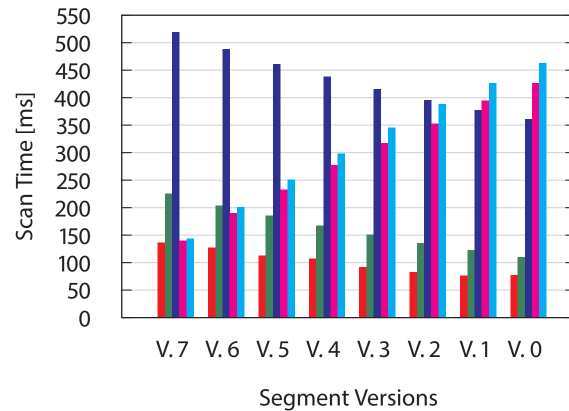
To analyze the effect of the overlay operator on on-going queries, we analyze a roll-up query (cf. Figure 5.13) from our multi-tenancy testbed. The query performs an aggregation and a sort.

The test data set consists of two segments belonging to the Account and the Opportunities polymorphic relations. The Account segment contains 10,000 tuples; 5 opportunities are generated for each account. Overlay is performed on the account fragment only.

Figure 5.14 shows the query results of the test for a variable number of overwritten tuples. The data points in the results are an average across 50 runs of the query. After each run, the data set and the indexes are discarded and newly created. The reported value is the execution time for the whole query in milliseconds. As expected, the customized implementations with overlay operator clearly out-perform the classical query-based approach (*Overlay Query*) for overlaying the tenant-specific and common data as long as there are overwritten tuples.



(a) Point-Wise Lookup



(b) Full Table Scan

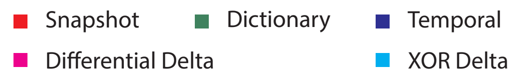


Figure 5.15.: Query Execution Times

5.5.2. Physical Data Organization

With our second set of experiments, we compare the different data versioning approaches presented in the previous section. The criteria are (1) execution times of point-wise lookups and full table scans, (2) space requirements, and (3) execution times of maintenance operations.

All experiments have been configured as follows. A single segment sequence consists of eight segments, each representing a particular version of the shared data. Version 0 is the base version with 100,000 tuples. The average tuple size is 120 bytes, the primary key size is 16 bytes, corresponding to the length of UUIDs. From one version to another, 10% new tuples are added, and 10% of the existing tuples are updated.

Data Access

Figure 5.15a shows the behavior of the five different implementations for point-wise accesses by primary key look-up. The snapshot implementation marks the baseline. For each version, an individual data segment and an individual index is used. The differential delta and the XOR delta have competitive results for look-ups to the most recent version 7. The dictionary and the temporal implementation have worse performance than the other approaches. The dictionary approach has to maintain a reference counter, while the temporal approach requires a more complex look-up operation, as described in Section 5.4.3.

When accessing previous versions, the execution time for the snapshot, the dictionary, and the temporal implementation do not differ from the access time to the most recent version. However, the execution time of the differential delta and the XOR delta approach depends on the number of deltas to process.

Figure 5.15b shows the scan behavior of the approaches. For each version a sorted scan has been performed. Since the number of tuples increases by 10% between two subsequent versions, the scan times increase from version 0 to version 7 for the snapshot, dictionary, and temporal approaches. For accessing the most recent version, differential delta and XOR delta approaches have the same scan times as the snapshot approach because of the materialization of the most recent version. The dictionary approach is a little bit slower because the reference counter has to be stripped off before returning the tuple. However, for the temporal approach the full temporal index has to be scanned. Since this index contains all available versions, the scan takes longer.

When scanning previous versions, the snapshot, dictionary, and temporal approaches have the same performance as scanning the most recent version, except from the fact that the number of scanned tuples varies. As stated before, the differential delta and XOR delta performance decreases due to the look-ups in the deltas.

Space Requirements

Figure 5.16 shows the space requirements of the approaches. The snapshot approach has the highest space requirements. Since the dictionary approach has a common data segment across all versions, but an individual index for each version, the space requirement of the indexes is identical to the snapshot variant, while the segment size is lower. A further reduction can be achieved by the temporal approach. The differential delta and the XOR delta approaches materialize and index the latest version and store the differences to previous versions in deltas. They require a lot less space than the snapshot approach and are competitive with the dictionary approach. From a space requirement perspective, the temporal approach seems very promising. The approaches dictionary, differential delta, and XOR delta have comparable sizes, while the snapshot needs twice the amount of space as the previous three approaches.

In our experimental evaluation, we do not consider intra-segment compression techniques for the following reasons. First, compression efficiency depends on the data distribution, so we assumed the worst case in which there is no intra-segment redundancy, i.e. tuples are pairwise totally distinct. The second—and more important—reason is that

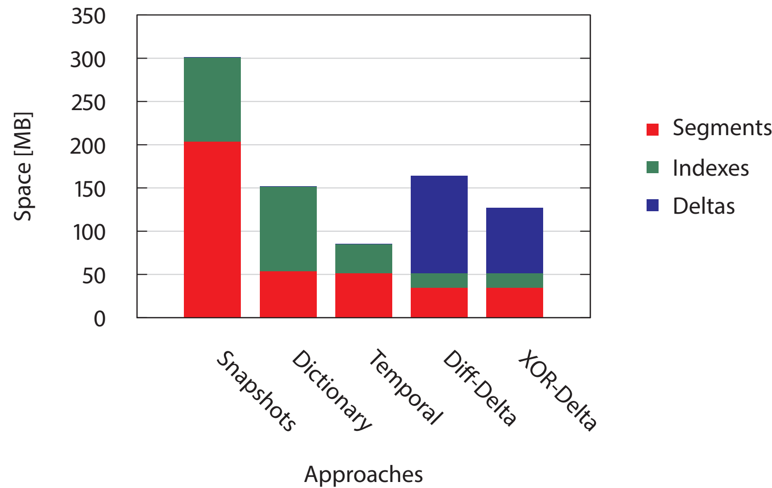


Figure 5.16.: Space Requirements

shared data is accessed by all co-located tenants concurrently. Therefore, we avoid the high decompression overhead caused by the frequent accesses to the shared data.

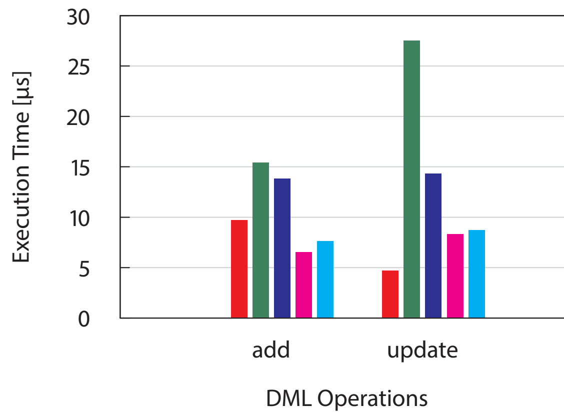
Maintenance Operations

Figure 5.17a shows the execution time for data maintenance operations. The results for delete operations have been omitted in the chart; they are nearly identical to the update operations.

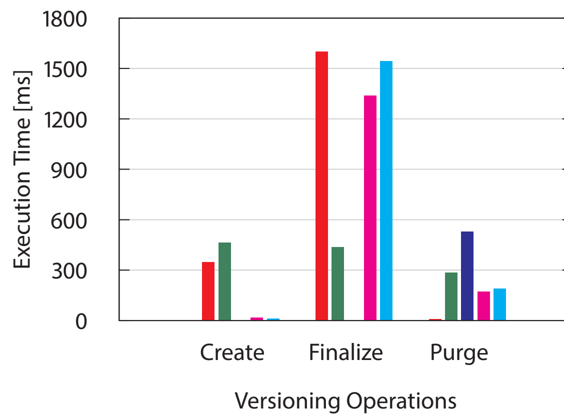
Adding tuples to the latest snapshot results in an index update and a segment update. The differential delta and the XOR delta only update the delta, so only one update operation is necessary. The dictionary approach has to maintain a reference counter, so there is slightly higher execution time. For the temporal approach, the version information has to be maintained, so there is a higher execution time as well.

Updating tuples in a snapshot results in an index look-up for the physical position of the tuple and an update-in-place operation in the data segment. The dictionary approach cannot do an update-in-place operation. Instead, the old tuple has to be retrieved to decrease the reference counter, then a new tuple has to be added to the data segment and the index has to be updated with the new physical position of the tuple. Updating a tuple when using the temporal approach is identical to adding a new tuple. In the differential delta and XOR delta approach, two look-ups are required to perform an update, as the before image has to be retrieved either from the materialized version or the non-finalized delta.

Figure 5.17b shows maintenance operations on versions. When creating a new version, the most recent snapshot together with its index has to be copied. For the dictionary approach, only the segment index has to be copied, but for all tuples in the index, the reference counter of that particular tuple, which is maintained in the data segment, has to be increased. The temporal approach only increases its internal version number which corresponds to the next segment in the segment sequence. For the differential delta and XOR delta approach, new empty B^+-Trees have to be created.



(a) DML Operations



(b) Versioning Operations

Figure 5.17.: Maintenance Execution Times

When finalizing a version, all data structures that are not shared across multiple versions are reorganized. For the dictionary approach, only the segment index can be reorganized, while for the snapshot approach also the data segment can be reorganized. In the temporal approach there is no reorganization possible without affecting the availability of the shared data. The differential delta and XOR delta approaches use a merge-join variant to create a new data segment with its index.

Purging a certain version which is no longer needed simply removes the data segment and its index in the snapshot case. The dictionary and the temporal approach have to perform a more complex purge operation which are discussed in the previous section. The differential delta and XOR delta approaches use a merge-join variant to create a new delta with the recalculated entries. Note that purging a version in the dictionary and temporal approaches affects shared data structures and thus affects performance.

Summary

The XOR delta data organization scheme is a promising approach for implementing versioning of shared data in a Multi-Tenant DBMS. Its access and scan behavior at the most recent version does not differ from the snapshot approach, which serves as baseline. However, access and scan performance on older versions is worse than in the snapshot approach, but this has no large impact for a typical SaaS workload, as only a few tenants lag marginally behind the latest version. Maintenance operations on versions can be done without affecting performance, since these operations do not affect shared data structures. Furthermore, the space requirement is competitive to other approaches like the dictionary approach. Although the temporal approach provides better space utilization, its drawbacks regarding data access and maintenance operations prevail.

As an optimization, more than one version in the XOR delta approach could be materialized, as XOR deltas can be applied in a forward and a backward manner without reorganization.

6

Graceful On-Line Schema Evolution

Allowing tenants to tailor the application to their needs increases the success of typical SaaS applications as a more flexible application can also target those customers who require special customizations. The individual schemas of the tenants inherit the majority of definitions from the base application and its extensions. However, even if tenants only redefine small bits of the base application, there is a constant change over time, when considering not only one tenant, but all co-located tenants: An on-premise single-tenant application typically evolves over time by changing the schema to adapt the application to new requirements or processes, or to introduce new functionality. In the SaaS context, the base application is evolved for the same reason. However, each tenant customizes the base application independently, and thus—figuratively spoken—creates its own branch of the application, which evolves over time as well. Furthermore, the ISVs' extensions also evolve over time.

These constant changes stress currently available DBMSs. As each schema change has to be performed with DDL statements, there is a lot of DDL as part of the workload. However, the majority of DDL statements involve heavyweight I/O operations as the affected data has to either be checked, e.g. for type compliance, or reorganized, e.g. when the default value has changed. These I/O operations typically consist of full table scans which exclusively lock the whole affected table, thus rendering it unaccessible for concurrently running operations.

DBMS vendors address this issues by providing specialized data migration tools like Oracle Online Table Redefinition, which lower the actual downtime by using a shadow copy for performing the actual redefinition.

In a SaaS environment, changing schema information must not be heavyweight to guarantee service availability. The high number of co-located tenants lead to continuous schema changes in the Multi-Tenant DBMS. Thus, for multi-tenancy, schema changes have to be lightweight in order to not affect system availability and performance.

We base our Multi-Tenant Schema Evolution mechanism on the following assumptions:

Extensibility Tenants make use of the base application. They customize their application with extensions, either developed by themselves or by ISVs.

Re-usability Between two subsequent changes, only small differences exist, so much schema information can be re-used across versions or extensions.

FlexScheme offers mechanisms to track these changes and allows for efficiently storing several versions of the same schema. Thus, if the history of schema changes is available, it can be used for a lightweight schema evolution mechanism which—instead of performing the necessary I/O operations immediately—defers these until the data needing checks or reorganization is actually accessed.

This *Lightweight Schema Evolution* consists of two components: (1) schema versioning and (2) lightweight physical data reorganization. The idea behind this approach is to separate logical redefinition as performed by DDL statements from the physical reorganization or checks of the actual data.

The schema versioning component is a fundamental part of the FlexScheme approach. It allows for keeping the history of the schema information for a particular table by introducing the notion of fragment sequences and fragment versions. The physical reorganization is performed by a special operator, the *evolution operator*, which is part of the query plan and actually performs the physical reorganization on-demand. Once a tuple is accessed, the evolution operator applies the scheduled schema changes to migrate the tuple to the most recent schema version.

However, there is a trade-off: The reorganization must take place sometime since data has to be consistent. Instead of performing one heavyweight reorganization process right after the schema change which negatively impacts the service availability, the costly reorganization process is spread across lots of small requests which form the usual workload of the DBMS.

6.1. Schema Versioning

To enable graceful schema evolution, the meta-data of the DBMS has to be aware of schema versions. According to Roddick (1995), schema versioning is accommodated when a DBMS allows access of all data, both retrospectively and prospectively, through user definable version interfaces. However, in the SaaS context, this paradigm can be restricted to a simpler variant of schema versioning. The following two restrictions are sufficient to make the meta-data aware of versioning by simultaneously allowing lightweight table redefinition.

1. Our prototype does not make the versioning interface publicly available, rather it hides this layer by introducing transparent schema migration. Only the most recent schema information is exposed to the application.

2. Queries are formulated against the most recent schema versions. These queries access all tuples, even if some of them are stored in older schema versions.

In our prototype, schema versioning is tightly integrated into the FlexScheme approach. In FlexScheme, a Virtual Private Table for a given tenant is decomposed into fragments. Each fragment stores the history of schema changes by providing *fragment versions* which provide a view of the fragment at a given version number. As discussed above, we assume only small changes between subsequent fragment versions, thus our prototype leverages the high degree of re-usability by sharing lots of information, such as data types and column attributes, across different fragment versions.

6.1.1. Versioning Concepts

As multiple fragment versions have to be stored for a single fragment, the history of the schema changes must be tracked. In the world of Object-Oriented DBMS (Roddick, 2009) and in the model-driven software development (Kögel et al., 2009), two well-known techniques are available for tracking history information: state-based and operation-based versioning. In this section, we discuss these two versioning concepts and show, how these are applied to FlexScheme. Our focus in this section is on fragment versioning; the discussion of segment versioning can be found in Section 5.3.2.

State-Based Versioning

When using state-based versioning, only the individual states of a model are stored. Each of these states consist of the materialization of the schema information, in our case the fragment versions, containing the attribute names, data types and type attributes at a given version.

However, for schema evolution the differences between subsequent fragment versions must be derived, which involves an expensive two-step computation: in the first step, the *matching* process identifies similarities between two subsequent versions; the second *comparison* step then identifies the changes.

Operation-Based Versioning

In contrast to the above approach, the operation-based versioning approach records the differences between two subsequent fragment versions. At least one materialization is necessary as starting point for iteratively applying the changes. Depending on the implementation of operation-based versioning, changes can be applied only in one direction (mostly forward) or in two directions. The latter allows for choosing any fragment version as materialization to serve as starting point.

Operation-based versioning simplifies schema evolution as differences can be applied to tuples without the need of an expensive computing step beforehand.

Each change is modeled as an atomic *modification operation* (MO) which describes a transformation process for one particular component of a fragment version. For complex changes, multiple atomic MOs are chained. The concept of MOs is discussed in more detail in the following sections.

FlexScheme-Based Versioning

Each of the previous two concepts has major drawbacks when used in a SaaS context: As different tenants may have different fragment versions active, there might be multiple fragment versions active in one DBMS instance. Thus, as the operation-based versioning only stores the differences between two versions, long chains of MOs must be processed for schema evolution, beginning at a common starting point for all tenants. This is not the case with state-based versioning, however the computational overhead might affect the overall system performance.

Hence, we propose a hybrid approach where we allow for multiple materializations in combination with operation-based versioning. Each fragment version which is in use by a particular tenant is materialized, as with the state-based versioning approach. Furthermore, we record the changes between two subsequent versions using atomic MOs as in the operation-based versioning approach.

As active fragment versions are materialized, this hybrid approach enables fast access to schema information as with the state-based approach, by simultaneously allowing for an easier schema evolution, as the MO chains are shorter due to multiple potential starting points.

However, this advantage is caused by the following trade-off: as two different versioning concepts are combined, there is an increased memory footprint for the schema information, as parts of them are kept redundantly. To lower the footprint, materializations of fragment versions must be purged after the last tenant using this fragment version switched to another one.

6.1.2. Atomic Modification Operations

As already mentioned in the previous sections, our approach of schema versioning is based on *atomic modification operations*, a well known approach for describing schema changes.

Shneiderman and Thomas (1982) proposed a set of atomic schema changes, including structural changes and changes on keys and dependencies. Bernstein et al. (2006, 2008) proposed schema modification operations for schema evolution using algebra-based constraints as primitives. Recent work of Curino et al. (2008) uses modification operations to describe differences between subsequent versions of schema definitions. Modification operations are atomic, that means, each MO transforms only one component of the schema. For complex changes, MOs must be chained. Moreover, Curino et al. (2008) introduce the notion of *compensation records* which allows for rolling back schema changes.

Our prototype takes up the set of schema modification operations as defined by Curino et al. (2008). In its current implementation, the prototype supports only the following structural MOs for tables: `ADD COLUMN`, `DROP COLUMN`, `RENAME COLUMN`, `CHANGE DATATYPE`, `SET DEFAULT VALUE`, and `SET NULLABLE`. However, it can be extended to support other MOs as well. Furthermore, it supports user-defined schema modification operations, for example complex string operations. This can be used for application specific schema modifications, such as splitting a NAME column into GIVENNAME and SURNAME using a given delimiter.

ADD COLUMN

This operation adds a column at the end of the already existing fragment version. The MO requires the specification of a name and type information consisting of the type name, the type length, the default value, and the information whether this field is null-able. During physical data redefinition, the tuple is extended to provide additional space for the new attribute.

DROP COLUMN

The complementary action to the above one removes a column from a fragment version. During physical redefinition, the `DROP COLUMN` operation should remove the data from the tuple. However, our prototype implements a *soft-delete* mechanism, where data is not instantly deleted. Thus, it simply sets a flag inside the tuple which hides the attribute. This way, backward compatibility to older fragment versions is maintained as the tuple can be converted to an older version.

RENAME COLUMN

Renaming a column simply changes the information in the fragment versions. No physical redefinition is necessary, as the physical access is based on offsets, not on mnemonics.

CHANGE DATATYPE

This operation changes the type of a given column to a new data type and/or type length. Currently, our prototype only allows for changing to compatible data types, where the domain is increased. During physical redefinition, the tuple format may have to be reorganized to fit the new data type.

Changes to the data type where the domain will be limited cannot be made using this MO, as this atomic MO cannot handle cases where the existing value conflicts with the new domain. For such changes, our prototype offers the possibility of custom MOs, where the user can specify a callback to handle conflicting changes automatically.

SET DEFAULT VALUE

Setting the default value of a column changes the data type attribute of the affected column. Physical redefinition has to respect the history of the affected tuple. If a `SET DEFAULT VALUE` MO has been issued before, the default value must not be changed: in conventional DBMSs changing the default value only affects newly created tuples.

SET NULLABLE

This operation changes the type definition of the column. During physical redefinition, the tuple layout has to be redefined to accept *`NULL`* values. Depending on the history of the tuple, two different redefinitions are necessary: (1) if a column was previously defined as `NOT NULL` and now can accept *`NULL`* values, there is no need to change anything during the reorganization phase, and (2) if a column description has changed to `NOT NULL` (which can only happen if a default value has been defined) space for the new default value must be allocated.

6.1.3. Applying Modification Operations

As the physical reorganization of the tuples is spread over time, there might be situations where tuples are accessed which have not been evolved to the most recent schema version.

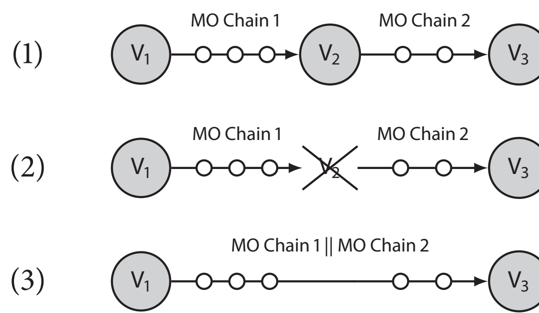


Figure 6.1.: Pruning of Unused Fragment Versions

Such tuples are evolved immediately at table access by applying all scheduled MOs that are necessary to move the tuple to the most recent schema version.

For a specific tuple, the fragment version of the current materialization is extracted from the physical representation. As FlexScheme stores the history of schema changes, for any given fragment version, the path to the most recent version can be determined. Furthermore, for each successor relation between two subsequent fragment versions, the schema MOs transforming the old version into the new version can be retrieved.

As each edge in the path is annotated with a chain of schema MOs, the tuple can be evolved by sequentially applying the chains, starting at the current fragment version of the current physical materialization. After the chain has been applied, the tuple conforms with the most recent schema version and can be processed by the application.

6.1.4. Version Pruning

To lower the overhead of administering a high number of fragment versions, FlexScheme allows to purge versions which are no longer used by any tuple.

The DBMS keeps statistics on how many tenants are still referencing a certain fragment version. As soon as a particular fragment version is no longer used, it can be pruned. Two different scenarios can occur:

1. The fragment version to be pruned does not have any predecessors. In this case, the materialized state and the chain of MOs pointing to other versions can easily be removed.

2. In any other case—as there might be tuples which rely on even older fragment versions—, only the materialized state can be pruned. As Figure 6.1 shows, the MO chain pointing to the pruned version and the MO chain pointing away from the pruned version have to be linked together.

After re-chaining in the second case, the MO chain may contain operations on the same target, for example, (1) `ADD COLUMN CHAR(10) 'colx'` and (2) `CHANGE DATATYPE CHAR(20) 'colx'`. Such operations may be collapsed into one single MO `ADD COLUMN CHAR(20) 'colx'` to reduce chain length.

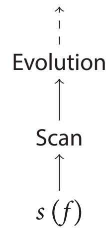


Figure 6.2.: Evolution Operator Placement within Query Execution Plan

As a further enhancement, rarely used versions can be pruned by the same mechanism, if the affected tuples are evolved to the most recent version. This process can be scheduled on a regular basis to further reduce the number of concurrently active fragment versions.

6.2. Lightweight Physical Data Reorganization

The second important component of the *Graceful Schema Evolution* mechanism is responsible for actually performing the schema evolution on the physical tuples while they are accessed. This on-the-fly reorganization allows for spreading the load across many small accesses to avoid service unavailability.

6.2.1. Objectives

We implemented a strategy-driven query plan operator which is placed in the access path to perform the schema evolution on demand, before the tuple is processed by the upper query plan operators. The strategy of the so-called *evolution operator* decides if the recently evolved tuple is immediately written back to the background storage or if the write process is deferred until the next user-requested update of the tuple.

When writing data, either by inserting new tuples or updating existing ones, the tuple conforms with the latest schema version. Thus, reorganization only occurs on reading tuples not conforming to the latest schema. Reorganization is only necessary if more than one schema version is active per tenant.

6.2.2. Evolution Operator

The *evolution operator* is placed in the access path of the query plan which retrieves the tuples from the back-end. Figure 6.2 shows a excerpt of a query execution plan. After the tuple has been accessed, the evolution operator transforms—if necessary—the physical representation of the tuples based on the schema changes recorded within FlexScheme. The transformation is done as follows.

After accessing the raw tuple, the current tuple version is determined. The evolution operator uses the information from a fragment to determine the appropriate fragment version. If the tuple conforms to the most recent fragment version, the tuple is pipelined to the above-laying operators of the query plan. If the tuple does not conform with the most

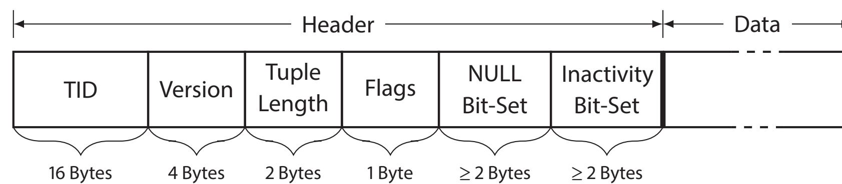


Figure 6.3.: Physical Tuple Layout

recent fragment version, the tuple is evolved to the most recent fragment version, by applying the chain of schema modification operations in sequence. After that, an *evolution strategy* decides whether to write back the data to the segment or not. Finally, the tuple is pipelined to upper operators. Possible evolution strategies are discussed in Section 6.3.

Although the granularity of the evolution is per-tuple, it might be possible to use the same technique with a per-page granularity. However, the coarser the granularity, the more heavyweight the physical reorganization gets. To mimic the behavior of traditional DBMSs, the physical reorganization must be performed at segment granularity.

6.2.3. Physical Tuple Format

As Gray and Reuter (1993) point out, traditional DBMSs are using the *N-ary Storage Model* (NSM) as the layout for physically storing tuples. The data segments containing the records are slotted into equally-sized pages. The pages then contain the actual records. Using slotted pages causes one more indirection during look-up, but simplifies free-space management. However, our setup has some additional requirements on the physical tuple format:

Version Number For identifying the version a particular tuple conforms to, the physical representation of the tuple has to be enhanced with an attribute for storing the version number. The version information is stored inside the tuple header.

Tuple Visibility Flag Tuple deletion is implemented as soft-deletes, thus a visibility flag is needed.

Attribute Visibility Bit-Set As the `DROP COLUMN` MO would cause data loss if physically dropping the column, we implemented soft-delete on attribute level. This way, we enable transforming the tuple to previous versions, although the evolution layer of our prototype does not support this at the moment. Soft-delete at attribute level requires a bit-set per tuple for storing the visibility of each individual attribute.

For our implementation, we enhanced NSM to satisfy the requirements from above. However, the enhancements can be implemented to be fully transparent to other record layouts, like PAX (Ailamaki et al., 2001), Interpreted Storage Layout (Beckmann et al., 2006), or columnar storage (Stonebraker et al., 2005). The resulting layout is depicted in Figure 6.3.

The header of an NSM-style tuple is extended by a fixed-length field storing the version information. The field stores the version information of the particular tuple. The value is accessed with every read operation, however, updates to this value only take place when writing the tuple.

We keep the position of the primary key (PK) fixed. The PK is stored outside the actual data field to allow for fast PK access without the need for schema evolution operator. However, this means that the type of the PK cannot be changed via graceful schema evolution. Thus, if the type changes, a complete conversion of the affected segment has to be performed.

Many databases, for example PostgreSQL (2011), have a bit-set in the tuple header for storing flags, such as if the tuple contains *NULL* values or has variable-width attributes. We add two more flags `TUPLE_INVALID` and `HAS_INACTIVE_ATTRIBUTES`. The first flag denotes tuples that have been marked as deleted by the soft-delete functionality. Tuples marked as invalid are removed by a periodically running clean-up job. Instead of such a flag, it may also be possible to store the time-stamp of the deletion operation. The second flag denotes whether an *inactivity bit-set* is present. Similar to the *NULL* bit-set of PostgreSQL, where attributes containing *NULL* values are marked, the inactivity bit-set marks attributes as inactive. Attributes become inactive by a `DROP COLUMN` MO. Inactive attributes physically contain a value, but do not show up during access. However, physically keeping the old value allows virtually rolling back tuples to older schema versions without information loss. This allows backward compatibility of queries[1].

6.2.4. Access Behavior

The physical transformation of tuples to conform with the newest schema version has to be applied right before other operators in the query plan access attributes of the tuple. Thus, the table access operators immediately feed their results into the evolution operator. As the tuple transformation is pipelined, the overall query runtime may depend on the time for transforming tuples.

For each incoming tuple, the evolution operator determines the source and the target version and then re-assembles the tuple. As a prerequisite for the transformation, the meta-data have to be queried to determine the chain of MOs transforming the tuple from the source to the target version. For fast access to the fragment versions, our prototype has hash-indexes on the versioning information. Furthermore, the time for re-assembling the tuple is dependent on the length of the MO chain. Thus, it is important that unnecessary versions are pruned from the meta-data.

Each segment has a *lightweight evolution strategy* assigned, which is consulted after tuple transformation to decide, whether the tuple is written back to the segment or not.

Figure 6.4 shows one exception, where the evolution operator is not necessarily the first operator above the table access operator. When data is overridden, the affected segments may be overlaid before the actual evolution is done. This is possible, as the overlay operator only relies on the PK, which is physically stored outside the data field.

[1] Our prototype currently does not support this type of queries, but the physical tuple representation has been designed to support this in the future.

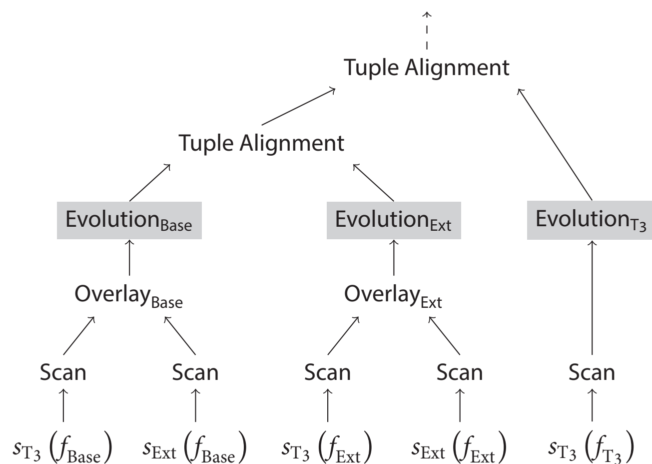


Figure 6.4.: Query Execution Plan with Evolution and Overlay Operators

6.3. Evolution Strategies

The evolution operator delegates the decision whether to write physically reorganized tuples back to the segment to an *evolution strategy*. This section discusses various evolution strategies and their influence on the schema evolution process.

6.3.1. Immediate Evolution

Typical DBMSs perform an immediate physical reorganization. After receiving the `ALTER TABLE` statement, the DBMS immediately performs the requested changes to the table. Depending on the request, the tuples in the table must either be restructured to conform with the new schema (for example, if fixed-length attributes are modified, `CHAR(20)` to `CHAR(25)`), or at least checked, e.g. if the type-domain has been narrowed from (`VARCHAR(25)` to `VARCHAR(20)`).

These changes incur a full table scan, thus touching all tuples of the affected table. As the full table scan needs exclusive access (X-Locks) on the table, it is not accessible for other queries during the time of the full table scan. As a consequence, the response time for the blocked queries increases. For some `ALTER TABLE` requests, there is no need to perform a full table scan, e.g. if the length of variable-length attributes is increased. Most DBMSs recognize such requests and omit the costly reorganization.

Some DBMS vendors offer solutions for keeping the relation accessible during the reorganization process, for example *Oracle Online Data Reorganization and Redefinition* (Oracle Corporation, 2005). As already discussed in Section 2.3.2, this mechanism keeps the affected table available during the redefinition phase. However, according to the feature's documentation, there may still be impacts on the overall system performance.

6.3.2. Lightweight Evolution

Our approach of lightweight evolution optimizes the schema evolution process. It is not desirable that database objects become unavailable during scheme evolution. Thus, compared to the previous approaches, the lightweight evolution is intended to allow for schema evolution without affecting service uptime *and* performance.

All the following strategies have in common that they are combined with our special schema evolution operator. The evolution operator involves one of the following strategies to decide whether the internally evolved tuple is written back to disk or not.

Eager Evolution

When using this strategy, the schema evolution operator immediately writes the tuple back to the storage segment after it has been evolved to the most recent schema version. Thus, if this strategy has been selected, a tuple is evolved at the first read access after the schema evolution. However, in the worst case, the eager strategy turns each read into a write, so this strategy might result in expensive write operations.

If a certain tuple is read and immediately updated, the tuple is written twice. Therefore, this strategy is suitable for workloads where a tuple is read several times before it gets updated. An example for the usage of such a strategy is a phone book record, which is displayed most often in a list (e.g. in a directory listing), and rarely updated.

Lazy Evolution

In contrast to the previous strategy, the evolution operator never writes back the internally evolved tuple when the lazy strategy is employed. The actual evolution is performed when the tuple gets written by an update request.

As each read request has to evolve the tuples, this strategy is suitable for data which is rarely read or if each read is immediately followed by an update request. However, the worst case for this strategy are data whose schema is updated more often than the actual data. In this case, the length of the MO chain increases over time. In the long tail, this may reduce the overall performance when accessing those tuples.

An example for a suitable usage of lazy evolution is an event ticket reservation system where usually a ticket is reserved once (data is always written in the latest version), the data is read only at the event once, changes occur rarely, for example change of the names of the attending guests. Historical data is read only for OLAP queries and analyses. Since most business intelligence applications nowadays apply an ETL process for the data preparation and work on the generated copy lazy evolution is most suitable for such examples.

Snoozing Evolution

To overcome the drawbacks of the two previous strategies, the snoozing strategy combines them by introducing a threshold for read operations. If the number of read access is below the threshold, the tuple is not yet evolved. As soon as the read access is above the threshold, the tuple is evolved and the read access counter is reset.

The current implementation maintains a data-structure inside the page for managing the number of read accesses to each individual tuple in the page. Furthermore, the threshold is fixed across all pages of a segment. As an future enhancement, the threshold could be adjusted adaptively depending on the access behavior of the workload.

As the snoozing strategy relies on an additional data structure for bookkeeping, applying this strategy to a segment may change the overall access pattern as each read access has to update the bookkeepings. To reduce the number of write accesses to the page for bookkeeping, the evolution might be performed on a per-page basis rather than on per-tuple basis. However, as the evaluation shows, maintaining a read access counter per tuple does not severely impact the performance of our main-memory-based prototype.

6.3.3. Strategy Selection

Choosing the right strategy for on-line schema evolution is vital for the overall system performance. As a rationale, unnecessary writes during read access must be avoided. Unnecessary in this sense are those writes which are performed during an evolution step, although the tuple would be updated in the next step by the application.

Besides manual selection of the strategy on a per-segment basis, our prototype implements a very simple form of an adaptive strategy selection based on the workload of a particular segment. We keep statistics about the number of reads and writes per page, and once an evolution step is necessary, the operator decides based on a certain threshold, which of the three strategies from above is selected.

6.4. Evaluation

For evaluating our approach, we used the same execution environment as already presented in Section 5.5. As baseline for our experiments we take the traditional behavior of DBMSs where no evolution takes place at runtime. For the baseline, we do not measure the time for actually performing the necessary physical redefinition after issuing a DDL statement, thus we assume that the physical redefinition has been performed during a service downtime window.

6.4.1. Point-Wise Access

To analyze the effect of evolution on point-wise accesses we use a single data segment, containing 10,000 randomly generated tuples. Before the run of the benchmark and after the initial data generation, schema modifications consisting of costly operations such as `ADD COLUMN`, `DROP COLUMN`, and `CHANGE DATATYPE` are performed, creating two new versions of the fragment. We omitted using lightweight modification operations like `RENAME COLUMN`.

Each test run consists of 50,000 point-wise queries which randomly select tuples with a 80/20 distribution. According to the Pareto Principle, 80% of the queries (40,000 queries) access 20% of the tuples (2,000 tuples). This means that each of these 2,000 tuples is accessed 20 times during a benchmark run. The remaining tuples (8,000) are accessed at least once. After each run the data set and the indexes are discarded and newly created.

We study two different scenarios: The first scenario consists of a read/write workload, where tuples may be written by the application after they have been read. The second scenario is comprised of a read-only workload.

For our baseline, we mimic the behavior of current DBMSs. If the required reorganization would be performed immediately after a schema modification, the fragment would be blocked for around 200 ms, as the reorganization involves a full table scan on 10,000 tuples with tuple transformation on each item and write-back. Subsequent reads (see Fig. 6.5a) cost around 12 microseconds. With our single-threaded execution model, queries would have to wait in queue during the immediate full scan and update. Therefore they might miss their response time goals. In order to overcome this problem, we defer the data conversion until the tuple is first read and use the described write-back strategies.

With the eager strategy, for each tuple one evolution and a subsequent write-back step is required. After that all subsequent reads of that tuple do not require conversion anymore. With the lazy strategy, 20 evolution operations are required, but no write-back is needed. With the snoozing strategy and a threshold of 5 reads, each tuple is converted 5 times before it is written back. The following 15 reads do not require conversion.

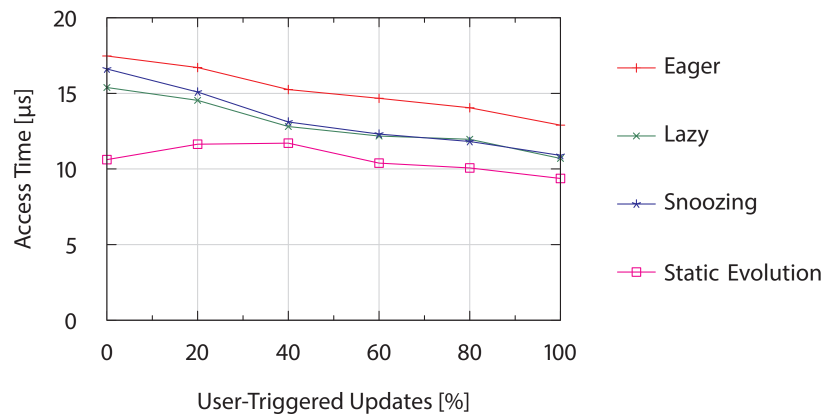
The experimental results for the read/write workload (Figure 6.5a) show that the eager evolution strategy is the most time consuming, which is caused by the immediate write on read. Lazy evolution is fastest because it only does the steps necessary for transforming the tuple to conform to the newest version of the fragment. These results underline that evolution costs less than a write back, which stresses out that the use of 'lights-out' schema evolution is a lightweight operation. Furthermore, we see that with the 80/20 workload the snoozing strategy's response times converge with those of lazy evolution in the long run. Therefore, the snoozing strategy can overcome the major drawback of the lazy strategy as described in the previous sections.

Figure 6.5b shows the results for the read-only workload. The X-axis shows the amount of tuples which conform with the most recent schema version. Like in the previous scenario, the costs for writing back evolved tuples heavily influences the eager and the snoozing strategy. However, as more tuples have been evolved, the different strategies converge. When comparing graceful schema evolution with traditional access to the data without evolution ("Static Evolution"), an overhead of about 20 % becomes visible. This overhead results from the meta-data look-up within FlexScheme as well as from reorganizing the tuple within main-memory.

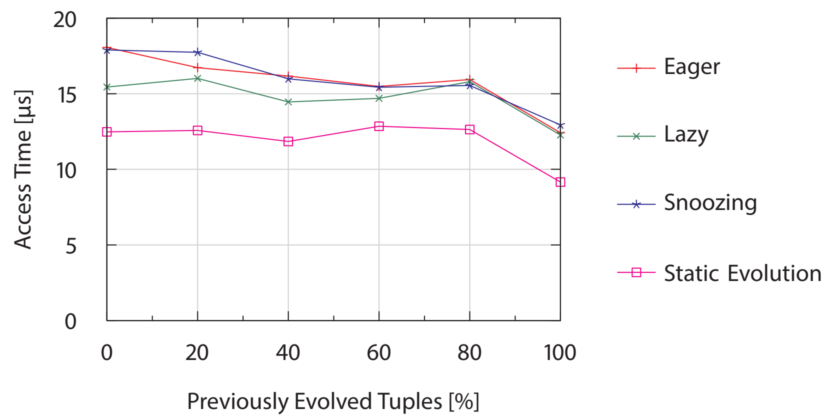
6.4.2. Queries

A second experiment evaluates the impact on report-style queries. The query is the same as in the evaluation of our data sharing component. As a reminder, the query (cf. Figure 5.13) performs a join of two relations with aggregation and grouping. The two relations are accessed via a full table scan. In this experiment, we compare the two graceful strategies "eager" and "lazy" with traditional query execution without schema evolution.

Figure 6.6 shows the result of the experiment. As the report query is read-only, we vary the amount of already evolved tuples as in the previous experiment. The experiment shows a very similar result as the previous one. If a high number of tuples have to be evolved, the



(a) Read/Write-Workload



(b) Read-Only-Workload

Figure 6.5.: Evolution Operator Performance for Point-Wise Accesses

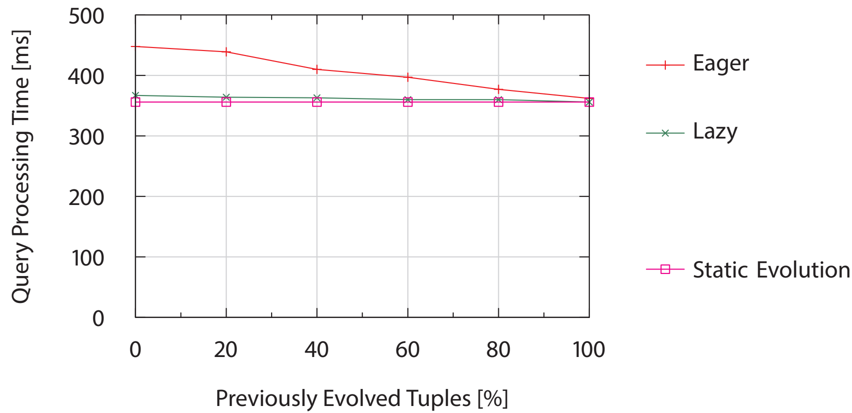


Figure 6.6.: Computational Overhead of Query Processing with Evolution

eager strategy is dominated by the costs of the immediate write-back. However, as more tuples are already evolved, the eager strategy converges with the lazy strategy. Comparing the lazy strategy to query processing with static evolution, only a very small overhead becomes visible. This small overhead is owed to the meta-data requests and the in-memory processing of the tuple.

6.4.3. Modification Operation Chain Length

In the previous experiments, the number of schema modification operations was fixed. To determine the trade-off point at which lazy and snoozing evolution are not efficient enough to perform better than the eager strategy, we vary the number of schema modification operations.

The setup remains identical to the previous experiments: we consider a single segment with 10,000 tuples which are accessed in a point-wise fashion. A single version consists of a chain of five atomic modification operations. During the experiment, we periodically "release" new schema versions. After a new version is available, a random set of tuples is accessed.

When using the eager strategy, any access of a tuple after the version release would automatically convert the tuple to the new version. This means that every new version requires five MOs to be applied and a single write-back per accessed tuple. Remember the results from the previous experiments, where the write-back costs dominate the reorganization costs. As the lazy strategy evolves the accessed tuple only in main-memory, the MO chain length to be processed increases with every released version. Although the tuples are accessed after each version release, the chain length increases as no implicit write-back is performed.

Figure 6.7 shows the results of the experiment. We see that the eager evolution result is nearly constant during the test run. This is caused by the fact, that tuples are accessed regularly during the test and thus the chain length does not grow much. After the release of the sixth version (or after 30 MO) the lazy approach becomes more costly than the eager approach. Beginning at that point, the costs for applying a long chain of MOs are increasing

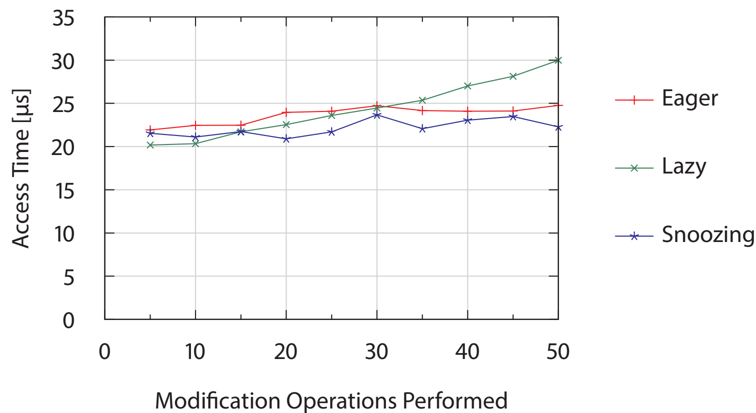


Figure 6.7.: Modification Operations Influence

compared to the costs for writing back the evolved tuple. The snoozing strategy overcomes this issue. We parameterized the threshold of the strategy, that after three accesses per tuple without write-back, the tuple is evolved and written back at the fourth access. As Figure 6.7 shows, the snoozing strategy can save the costs for the write-back by avoiding long MO chains.

6.4.4. Summary

In a nutshell, graceful on-line schema evolution places only a very light overhead on the query processing. For average MO chain lengths there is an overhead of about 20 %, which is tolerable. Performing lazy evolution can have advantages as long as the MO chain length does not explode, as the costs for transforming the tuples in the main memory are much lower than writing back the tuple. However, with increasing chain length, the tuple should be evolved as the costs of in-memory tuple transformation increase linearly with the MO chain length. To avoid a regularly running fix-up job, we proposed the snoozing strategy and our experiments show, that this strategy is able to provide a trade-off between the costs of processing long MO chains and writing back the tuple. The snoozing strategy has to be parameterized by a threshold, which denotes the number of evolution processes per tuple, before the tuple evolution is forced to the background storage. This threshold can be either set manually depending on the expected workload or by the query optimizer, whose statistics are enhanced to consider MO chain length as well.

6.5. Related Work

In the past, schema evolution has been extensively studied in various contexts. Two related contexts are relational DBMSs (Roddick, 1995), Object-Oriented DBMSs (Banerjee et al., 1987; Moerkotte and Zachmann, 1993; Kemper and Moerkotte, 1994; Claypool et al., 1998), and temporal DBMSs (Roddick and Snodgrass, 1995). Furthermore, schema evolution has gained significance in the research area of XML databases (Moro et al., 2007).

Recent work (Curino et al., 2008; Moon et al., 2008; Curino et al., 2009; Moon et al., 2010) shows that schema evolution is still an important topic, especially in scenarios where information systems must be upgraded with no or less human intervention.

7

Outlook and Future Challenges

In this thesis, we introduced a special-purpose DBMS which has been optimized for the requirements and specialities of a multi-tenant application. The typical use case for such an application is Software as a Service where multiple businesses are consolidated onto the same operational system and access the service via Internet browsers.

From a service provider's perspective, the database back-end should be totally self-service. This means, that tenants should be able to customize their application instance without any human intervention from the service provider's staff. From a tenant's perspective, its data should be well isolated from co-located tenants.

At first, we presented mechanisms on how to address these issues on conventional DBMSs. We discussed a set of schema mappings which transform the logical tenant-specific schema into a physical schema inside the database back-end. However, the evaluation of this approach shows that—even though the overall system performs well—not all of the requirements are met.

Thus, we introduced a next-generation DBMS which addresses these issues by providing schema flexibility in two dimensions: schema extensibility and schema evolution. Moreover, it allows sharing of common data to further reduce the TCO. Our prototype is based on FlexScheme, a hierarchical meta-data model supporting native schema flexibility in our prototype.

The components responsible for applying schema flexibility to the physical data representations have been implemented as operators that can be placed in the access path of a query execution plan. The overlay operator is responsible for allowing read/write access to shared data, where writes are redirected to the tenant's private segment and thus do not interfere with co-located tenants. The evolution operator allows for graceful on-line schema evolution without affecting the system availability. Rather than performing a costly data reorganization right after receiving a schema change request, it uses the schema versioning mechanism of FlexScheme to spread the actual physical reorganization across a lot of small requests. This way, the availability of the overall system can be guaranteed.

Our evaluation shows, that our modifications are lightweight in a sense that they do not severely impact the system performance. We conclude, that—given the specialities of a SaaS application's workload—the overall Quality of Service (QoS) goals can be met.

At the moment, our prototype only provides a basic key/value-based interface. As a future enhancement, SQL-based query mechanisms should be included. In this context, the query optimizer should be able to handle specialities of the Multi-Tenant DBMS. Espe-

cially, the selection of the best evolution strategy can be implemented as part of the optimizer: instead of relying on a threshold-based selection, the optimizer can use a feedback loop for observing the access behavior of the queries and then select the strategy adaptively. For example, IBM DB2 uses the feedback loop of LEO, the LEarning Optimizer, to adaptively adjust the access cardinalities in the database statistics (Stillger et al., 2001; Markl and Lohman, 2002).

When reconciling changes due to the migration from one version of the shared data to another version, there might be conflicts that have to be resolved. Our prototype currently has a very rigid behavior as it has a fixed precedence by having a bias towards the most local changes. However, this may not be sufficient in every case. Thus, the overlay mechanism has to be enhanced to reconcile such conflicting changes autonomously by using a application-specific conflict resolution strategy. The same mechanism can then be used to process conflicting changes due to schema evolution, which may occur when lowering attribute domains, e.g. `CHAR(20)` to `CHAR(10)`.

Furthermore, schema evolution can affect the usage of indexes on attributes other than the PK attributes. As those attributes are located in an area of the physical representation which is subjected to schema evolution, indexes on those attributes may become stale. This issue is not yet addressed in our prototype and may be subject of further research. Closely related to this issue is an even lazier schema evolution strategy. At the moment, the complete tuple is evolved by the schema evolution operator. However, it might be sufficient to evolve only a subset of the attributes, as the other attributes may not be considered.

To further lower the TCO, the whole Multi-Tenant DBMS cluster can be managed by an adaptive controller (Gmach et al., 2008). Each tenant is treated as a service that has to be placed optimally on a set of cluster nodes. Our support for off-line tenants and tenant replication allows for migrating tenants, even if they are on-line.

A

Appendix

A.1. Associativity of Data Overlay

In Section 4.4.2 we defined the semantics of the overlay operator as follows:

$$overlay(S, T) := S \cup \big(\overbrace{T \setminus \big(T \ltimes_{pk(T)=pk(S)} S\big)}^{T \triangleright_{pk(T)=pk(S)} S}\big) \tag{A.1}$$

$$= S \cup \big(T \triangleright_{pk(T)=pk(S)} S\big) \tag{A.2}$$

Theorem A.1:
The overlay operator as defined above is associative, thus

$$overlay(A, overlay(B, C)) = overlay(overlay(A, B), C) \quad \diamond$$

Proof by Contradiction

The join attributes of the anti-join $R \triangleright_{pk} S$ are the primary key attributes pk of the relations R and S. The schema of R and S are identical as required by the definition of the overlay operator.

Thus for any primary key value x, the transformation

$$x \in \pi_{pk}\big(R \triangleright_{pk} S\big) \Leftrightarrow \big(x \in \pi_{pk}(R)\big) \wedge \big(x \notin \pi_{pk}(S)\big) \tag{A.3}$$

and its negation

$$x \notin \pi_{pk}\big(R \triangleright_{pk} S\big) \Leftrightarrow \big(x \notin \pi_{pk}(R)\big) \vee \big(x \in \pi_{pk}(S)\big) \tag{A.4}$$

can be performed. Furthermore,

$$\begin{aligned} &\big(x \in \pi_{pk}(R)\big) \vee \big(x \in \pi_{pk}\big(S \triangleright_{pk} R\big)\big) \\ &\Leftrightarrow \big(x \in \pi_{pk}(R)\big) \vee \big(\big(x \in \pi_{pk}(S)\big) \wedge \big(x \notin \pi_{pk}(R)\big)\big) \\ &\Leftrightarrow \big(x \in \pi_{pk}(R)\big) \vee \big(x \in \pi_{pk}(S)\big) \end{aligned} \tag{A.5}$$

and its negation

$$\begin{aligned}&\left(x \notin \pi_{pk}(R)\right) \wedge \left(x \notin \pi_{pk}\left(S \triangleright_{pk} R\right)\right)\\ \Leftrightarrow &\left(x \notin \pi_{pk}(R)\right) \wedge \left(x \notin \pi_{pk}(S)\right)\end{aligned} \tag{A.6}$$

For the proof by contradiction, we assume

$$\mathit{overlay}(A, \mathit{overlay}(B, C)) \neq \mathit{overlay}(\mathit{overlay}(A, B), C) \tag{A.7}$$

Case 1

$$\begin{aligned}&\exists x : \left[x \in \pi_{pk}(\mathit{overlay}(A, \mathit{overlay}(B, C)))\right]\\ &\wedge \left[x \notin \pi_{pk}(\mathit{overlay}(\mathit{overlay}(A, B), C))\right]\end{aligned}$$

With (A.2):

$$\begin{aligned}\Rightarrow \exists x : &\left[\left(x \in \pi_{pk}(A)\right) \vee \left(x \in \pi_{pk}\left(\mathit{overlay}(B, C) \triangleright_{pk} A\right)\right)\right]\\ \wedge &\left[\left(x \notin \pi_{pk}(\mathit{overlay}(A, B))\right) \wedge \left(x \notin \pi_{pk}\left(C \triangleright_{pk} \mathit{overlay}(A, B)\right)\right)\right]\end{aligned}$$

With (A.5) and (A.6):

$$\begin{aligned}\Rightarrow \exists x : &\left[\left(x \in \pi_{pk}(A)\right) \vee \left(x \in \pi_{pk}(\mathit{overlay}(B, C))\right)\right]\\ \wedge &\left[\left(x \notin \pi_{pk}(\mathit{overlay}(A, B))\right) \wedge \left(x \notin \pi_{pk}(C)\right)\right]\end{aligned}$$

Again, with (A.2):

$$\begin{aligned}\Rightarrow \exists x : &\left[\left(x \in \pi_{pk}(A)\right) \vee \left(x \in \pi_{pk}(B)\right) \vee \left(x \in \pi_{pk}\left(C \triangleright_{pk} B\right)\right)\right]\\ \wedge &\left[\left(x \notin \pi_{pk}(A)\right) \wedge \left(x \notin \pi_{pk}\left(B \triangleright_{pk} A\right)\right) \wedge \left(x \notin \pi_{pk}(C)\right)\right]\end{aligned}$$

Finally:

$$\begin{aligned}\Rightarrow \exists x : &\left[\left(x \in \pi_{pk}(A)\right) \vee \left(x \in \pi_{pk}(B)\right) \vee \left(x \in \pi_{pk}(C)\right)\right]\\ \wedge &\left[\left(x \notin \pi_{pk}(A)\right) \wedge \left(x \notin \pi_{pk}(B)\right) \wedge \left(x \notin \pi_{pk}(C)\right)\right]\end{aligned}$$

This is a contradiction.

$$\Rightarrow \bot$$

Case 2

$$\begin{aligned}&\exists x : \left[x \notin \pi_{pk}(\mathit{overlay}(A, \mathit{overlay}(B, C)))\right]\\ &\wedge \left[x \in \pi_{pk}(\mathit{overlay}(\mathit{overlay}(A, B), C))\right]\end{aligned}$$

$$\Rightarrow \exists x : \left[\left(x \notin \pi_{pk}(A) \right) \wedge \left(x \notin \pi_{pk} \left(overlay(B, C) \triangleright_{pk} A \right) \right) \right]$$
$$\wedge \left[\left(x \in \pi_{pk}(overlay(A, B)) \right) \vee \left(x \in \pi_{pk} \left(C \triangleright_{pk} overlay(A, B) \right) \right) \right]$$

$$\Rightarrow \exists x : \left[\left(x \notin \pi_{pk}(A) \right) \wedge \left(x \notin \pi_{pk}(overlay(B, C)) \right) \right]$$
$$\wedge \left[\left(x \in \pi_{pk}(overlay(A, B)) \right) \vee \left(x \in \pi_{pk}(C) \right) \right]$$

$$\Rightarrow \exists x : \left[\left(x \notin \pi_{pk}(A) \right) \wedge \left(x \notin \pi_{pk}(B) \right) \wedge \left(x \notin \pi_{pk} \left(C \triangleright_{pk} B \right) \right) \right]$$
$$\wedge \left[\left(x \in \pi_{pk}(A) \right) \vee \left(x \in \pi_{pk} \left(B \triangleright_{pk} A \right) \right) \vee \left(x \in \pi_{pk}(C) \right) \right]$$

$$\Rightarrow \exists x : \left[\left(x \notin \pi_{pk}(A) \right) \wedge \left(x \notin \pi_{pk}(B) \right) \wedge \left(x \notin \pi_{pk}(C) \right) \right]$$
$$\wedge \left[\left(x \in \pi_{pk}(A) \right) \vee \left(x \in \pi_{pk}(B) \right) \vee \left(x \in \pi_{pk}(C) \right) \right]$$

$$\Rightarrow \bot$$

Conclusion

The assumption (A.7) is wrong. Thus, the overlay operator is associative. ■

Bibliography

Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In Koch et al. (2007), pages 411–422. ISBN 978-1-59593-649-3.

Rakesh Agrawal, Amit Somani, and Yirong Xu. Storage and Querying of E-Commerce Data. In Apers et al. (2001), pages 149–158. ISBN 1-55860-804-4.

Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In Apers et al. (2001), pages 169–180. ISBN 1-55860-804-4.

Amazon EC2. Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`, 2011.

Amazon S3. Amazon Simple Storage Service. `http://aws.amazon.com/s3/`, 2011.

Amazon SimpleDB. Amazon SimpleDB. `http://aws.amazon.com/simpledb/`, 2011.

Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors. *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, 2001. Morgan Kaufmann. ISBN 1-55860-804-4.

Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In Wang (2008), pages 1195–1206. ISBN 978-1-60558-102-6.

Stefan Aulbach, Dean Jacobs, Alfons Kemper, and Michael Seibold. A Comparison of Flexible Schemas for Software as a Service. In Çetintemel et al. (2009), pages 881–888. ISBN 978-1-60558-551-2.

Stefan Aulbach, Dean Jacobs, Jürgen Primsch, and Alfons Kemper. Anforderungen an Datenbanksysteme für Multi-Tenancy- und Software-as-a-Service-Applikationen. In Freytag et al. (2009), pages 544–555. ISBN 978-3-88579-238-3.

Stefan Aulbach, Michael Seibold, Dean Jacobs, and Alfons Kemper. Extensibility and Data Sharing in Evolving Multi-Tenant Databases. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, pages 99–110, 2011.

Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD Conference*, pages 311–322. ACM Press, 1987.

Jennifer L. Beckmann, Alan Halverson, Rajasekar Krishnamurthy, and Jeffrey F. Naughton. Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *ICDE*, page 58. IEEE Computer Society, 2006.

Phil Bernstein. Google MegaStore. `http://perspectives.mvdirona.com/2008/07/10/GoogleMegastore.aspx`, 2008.

Philip A. Bernstein, Todd J. Green, Sergey Melnik, and Alan Nash. Implementing Mapping Composition. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB*, pages 55–66. ACM, 2006. ISBN 1-59593-385-9.

Philip A. Bernstein, Todd J. Green, Sergey Melnik, and Alan Nash. Implementing Mapping Composition. *VLDB J.*, 17(2):333–353, 2008.

P. A. Boncz. *Monet: A Next-Generation Database Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, May 2002. URL `http://oai.cwi.nl/oai/asset/14832/14832A.pdf`.

Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, 2009. ACM. ISBN 978-1-60558-551-2.

Sang Kyun Cha and Changbin Song. P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. In Nascimento et al. (2004), pages 1033–1044. ISBN 0-12-088469-0.

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

Eric Chu, Jennifer L. Beckmann, and Jeffrey F. Naughton. The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 821–832. ACM, 2007. ISBN 978-1-59593-686-8.

Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner. SERF: Schema Evaluation through an Extensible, Re-usable and Flexible Framework. In Georges Gardarin, James C. French, Niki Pissinou, Kia Makki, and Luc Bouganim, editors, *CIKM*, pages 314–321. ACM, 1998. ISBN 1-58113-061-9.

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, 2008.

George P. Copeland and Setrag Khoshafian. A Decomposition Storage Model. In Shamkant B. Navathe, editor, *SIGMOD Conference*, pages 268–279. ACM Press, 1985.

Conor Cunningham, Goetz Graefe, and César A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In Nascimento et al. (2004), pages 998–1009. ISBN 0-12-088469-0.

Carlo Curino, Hyun Jin Moon, and Carlo Zaniolo. Graceful Database Schema Evolution: The PRISM Workbench. *PVLDB*, 1(1):761–772, 2008.

Carlo Curino, Hyun Jin Moon, and Carlo Zaniolo. Automating Database Schema Evolution in Information System Upgrades. In Tudor Dumitras, Iulian Neamtiu, and Eli Tilevich, editors, *HotSWUp*. ACM, 2009. ISBN 978-1-60558-723-3.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 205–220. ACM, 2007. ISBN 978-1-59593-591-5.

Bruce Eckel. *Thinking in Java (2nd ed.): The Definitive Introduction to Object-Oriented Programming in the Language of the World-Wide Web*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000. ISBN 0-13-027363-5.

Ahmed K. Elmagarmid and Divyakant Agrawal, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, 2010. ACM. ISBN 978-1-4503-0032-2.

Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison Wesley, March 2006. ISBN 0321369572.

Leonidas Fegaras and David Maier. Optimizing Object Queries Using an Effective Calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

Daniela Florescu and Donald Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Research Report RR-3680, INRIA, 1999. URL `http://hal.inria.fr/inria-00072991/PDF/RR-3680.pdf`. Project RODIN.

Johann Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors. *Datenbanksysteme in Business, Technologie und Web (BTW 2009), 13. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, 2.-6. März 2009, Münster, Germany*, volume 144 of *LNI*, 2009. GI. ISBN 978-3-88579-238-3.

Daniel Gmach, Stefan Krompass, Andreas Scholz, Martin Wimmer, and Alfons Kemper. Adaptive Quality of Service Management for Enterprise Services. *TWEB*, 2(1), 2008.

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

Jim Gray. Tape is dead, Disk is tape, Flash is disk, RAM locality is king. `http://research.microsoft.com/en-us/um/people/gray/talks/Flash_is_Good.ppt`, 2006.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN 1-55860-190-2.

Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29:91–131, 2004.

James R. Hamilton. On Designing and Deploying Internet-Scale Services. In *LISA*, pages 231–242. USENIX, 2007.

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through The Looking Glass, And What We Found There. In Wang (2008), pages 981–992. ISBN 978-1-60558-102-6.

HBase. HBase. `http://hadoop.apache.org/hbase/`, 2010.

Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter A. Boncz. Positional Update Handling in Column Stores. In Elmagarmid and Agrawal (2010), pages 543–554. ISBN 978-1-4503-0032-2.

Dave Hitz, James Lau, and Michael A. Malcolm. File System Design for an NFS File Server Appliance. In *USENIX Winter*, pages 235–246, 1994.

Dean Jacobs. Data Management in Application Servers. In *Readings in Database Systems, 4th edition*. The MIT Press, 2005. ISBN 0262693143.

Dean Jacobs and Stefan Aulbach. Ruminations on Multi-Tenant Databases. In Kemper et al. (2007), pages 514–521. ISBN 978-3-88579-197-3.

Christian S. Jensen and Richard T. Snodgrass. Temporal Data Management. *IEEE Trans. Knowl. Data Eng.*, 11(1):36–44, 1999.

Christian S. Jensen and Richard T. Snodgrass. Temporal Database. In Liu and Özsu (2009), pages 2957–2960. ISBN 978-0-387-35544-3, 978-0-387-39940-9.

Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2):1496–1499, 2008.

Alfons Kemper and Guido Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice-Hall, 1994. ISBN 0-13-629239-9.

Alfons Kemper and Thomas Neumann. HyPer: HYbrid OLTP & OLAP High PERformance Database System. Technical Report, Technische Universität München, 2010. URL `http://drehscheibe.in.tum.de/forschung/pub/reports/2010/TUM-I1010.pdf.gz`.

Alfons Kemper, Donald Kossmann, and Florian Matthes. SAP R/3: A Database Application System (Tutorial). In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD Conference*, page 499. ACM Press, 1998. ISBN 0-89791-995-5.

Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors. *Datenbanksysteme in Business, Technologie und Web (BTW 2007), 12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, 7.-9. März 2007, Aachen, Germany*, volume 103 of *LNI*, 2007. GI. ISBN 978-3-88579-197-3.

Setrag Khoshafian and Razmik Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces*. John Wiley & Sons, Inc., New York, NY, USA, 1990. ISBN 0-471-51802-6.

Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors. *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, 2007. ACM. ISBN 978-1-59593-649-3.

Maximilian Kögel, Markus Herrmannsdörfer, Jonas Helming, and Yang Li. State-based vs. Operation-based Change Tracking. In *MODELS '09 MoDSE-MCCM Workshop, Denver, USA, 2009*, 2009. URL `http://wwwbruegge.in.tum.de/static/publications/view.php?id=205`.

Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems*. Springer US, 2009. ISBN 978-0-387-35544-3, 978-0-387-39940-9.

David Maier and Jeffrey D. Ullman. Maximal Objects and the Semantics of Universal Relation Databases. *ACM Trans. Database Syst.*, 8(1):1–14, 1983.

Volker Markl and Guy M. Lohman. Learning Table Access Cardinalities with LEO. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, page 613. ACM, 2002. ISBN 1-58113-497-5.

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

MediaTemple. Anatomy of MySQL on the GRID. `http://blog.mediatemple.net/weblog/2007/01/19/anatomy-of-mysql-on-the-grid/`, 2007.

Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, Version 15 . `http://csrc.nist.gov/groups/SNS/cloud-computing/index.html`, 2009.

Jim Melton. *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features*. Elsevier Science Inc., New York, NY, USA, 2002. ISBN 1558606777.

Microsoft Corporation. Microsoft SQL Server 2008 Books Online – Using Sparse Columns. http://msdn.microsoft.com/en-us/library/cc280604.aspx, 2008.

Guido Moerkotte and Andreas Zachmann. Towards More Flexible Schema Management in Object Bases. In *ICDE*, pages 174–181. IEEE Computer Society, 1993. ISBN 0-8186-3570-3.

Hyun Jin Moon, Carlo Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and Querying Transaction-Time Databases Under Schema Evolution. *PVLDB*, 1(1): 882–895, 2008.

Hyun Jin Moon, Carlo Curino, and Carlo Zaniolo. Scalable Architecture and Query Optimization for Transaction-Time DBs with Evolving Schemas. In Elmagarmid and Agrawal (2010), pages 207–218. ISBN 978-1-4503-0032-2.

Mirella Moura Moro, Susan Malaika, and Lipyeow Lim. Preserving XML Queries During Schema Evolution. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *WWW*, pages 1341–1342. ACM, 2007. ISBN 978-1-59593-654-7.

mysql.com. mysql.com. http://www.mysql.com/, 2010.

Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors. *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, 2004. Morgan Kaufmann. ISBN 0-12-088469-0.

Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDB J.*, 19(1):91–113, 2010.

Matthias Nicola. 15 Best Practices for pureXML Performance in DB2. http://www.ibm.com/developerworks/data/library/techarticle/dm-0610nicola/, 2008.

Oscar Nierstrasz. A Survey of Object-Oriented Concepts. In *Object-Oriented Concepts, Databases and Applications*. ACM Press and Addison Wesley, 1989.

Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, 1996.

Oracle Corporation. Oracle Streams. http://www.oracle.com/technetwork/testcontent/streams-oracle-streams-twp-128298.pdf, 2002.

Oracle Corporation. Oracle Database 10g Release 2 Online Data Reorganization & Redefinition . http://www.oracle.com/technology/deploy/availability/pdf/ha_10gR2_online_reorg_twp.pdf, 2005.

John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *Operating Systems Review*, 43(4):92–105, 2009.

Hasso Plattner. A Common Database Approach for OLTP and OLAP using an In-Memory Column Database. In Çetintemel et al. (2009), pages 1–2. ISBN 978-1-60558-551-2.

Meikel Poess and Raghunath Othayoth Nambiar. Tuning Servers, Storage and Database for Energy Efficient Data Warehouses. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *ICDE*, pages 1006–1017. IEEE, 2010. ISBN 978-1-4244-5444-0.

PostgreSQL. The world's most advanced open source database. `http://www.postgresql.org`, 2011.

Jun Rao and Kenneth A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD Conference*, pages 475–486. ACM, 2000. ISBN 1-58113-218-2.

John F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7), 1995. ISSN 0950-5849.

John F. Roddick. Schema Evolution. In Liu and Özsu (2009), pages 2479–2481. ISBN 978-0-387-35544-3, 978-0-387-39940-9.

John F. Roddick and Richard T. Snodgrass. Schema Versioning. In *The TSQL2 Temporal Query Language*, pages 425–446. Kluwer, 1995.

Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

Salesforce AppExchange. Salesforce AppExchange. `http://sites.force.com/appexchange/home`, 2010.

Cynthia M. Saracco, Don Chamberlin, and Rav Ahuja. *DB2 9: pureXML – Overview and Fast Start*. IBM, `http://www.redbooks.ibm.com/abstracts/sg247298.html?Open`, 2006. ISBN 073849557.

Ben Shneiderman and Glenn Thomas. An Architecture for Automatic Relational Database System Conversion. *ACM Trans. Database Syst.*, 7(2):235–257, 1982.

Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In Apers et al. (2001), pages 19–28. ISBN 1-55860-804-4.

Michael Stonebraker. The Design of the POSTGRES Storage System. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB*, pages 289–300. Morgan Kaufmann, 1987. ISBN 0-934613-46-X.

Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A Column-oriented DBMS. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, pages 553–564. ACM, 2005. ISBN 1-59593-154-6, 1-59593-177-5.

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In Koch et al. (2007), pages 1150–1160. ISBN 978-1-59593-649-3.

Mary Taylor and Chang Jie Guo. Data Integration and Composite Business Services, Part 3: Build a Multi-Tenant Data Tier with Access Control and Security. http://www.ibm.com/developerworks/db2/library/techarticle/dm-0712taylor/, 2007.

Transaction Processing Performance Council. TPC-C V5.11 – On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/, 2010.

Vertica Systems. The Vertica Database. http://www.vertica.com/wp-content/uploads/2011/01/verticadatabasedatasheet1.pdf, 2011.

Jason Tsong-Li Wang, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, 2008. ACM. ISBN 978-1-60558-102-6.

Craig D. Weissman and Steve Bobrowski. The Design of the force.com Multitenant Internet Application Development Platform. In Çetintemel et al. (2009), pages 889–896. ISBN 978-1-60558-551-2.