Extensibility and Data Sharing in Evolving Multi-Tenant Databases

Stefan Aulbach#, Michael Seibold#, Dean Jacobs*, Alfons Kemper#

#*Technische Universität München*
Boltzmannstr. 3, 85748 Garching, Germany
firstname.lastname@in.tum.de

**SAP AG*
Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany
dean.jacobs@sap.com

***Abstract*—Software-as-a-Service applications commonly consolidate multiple businesses into the same database to reduce costs. This practice 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. The second is application modification and extension, which applies both to the database schema and master data it contains. The third is evolution of the schema and master data, which occurs as the application and its extensions are upgraded. 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. 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 have to be re-implemented. Instead, a next-generation multi-tenant DBMS should provide explicit support for Extensibility, Data Sharing and Evolution. As these three features are strongly related, they cannot be implemented independently from each other. Therefore, we propose FLEXSCHEME which captures all three aspects in one integrated model. In this paper, we focus on efficient storage mechanisms for this model and present a novel versioning mechanism, called *XOR Delta*, which is based on XOR encoding and is optimized for main-memory DBMSs.**

I. INTRODUCTION

In the Software-as-a-Service (SaaS) model, a Service Provider owns and operates an application that is accessed over the Internet by many users of different businesses. Customers of the Service Provider are typically charged a monthly service fee on a per-user basis. By careful engineering, it is possible to leverage economy of scale to reduce Total Cost of Ownership (TCO) relative to on-premise solutions. To this end, it is common practice to consolidate multiple businesses into the same database to reduce operational expenditures, since there are fewer processes to manage, as well as capital expenditures, since resource utilization is increased.

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 purposes 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 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 for 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 if ISVs and tenants can delay performing 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.

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* that combine schema and shared data. Data Sharing decreases the space requirements and allows for easier upgrades, and thus enables economy of scale. From a tenant's perspective, modifying shared data must not interfere with other tenants. Therefore, write access to shared data has to be redirected to a private, tenant-specific storage segment, thus transforming the updated shared data to private data 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

978-1-4244-8960-2/11/$26.00 © 2011 IEEE

schema and the shared data. Base and Extension Instances may evolve independently from each other, therefore the versioning has to be employed on the level of individual Instances. This mechanism also allows tenants to stay with certain versions of Instances.

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. As an example, force.com does its own mapping from logical tenant schemas to one universal physical database schema [1]. 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.

In this paper, we describe FlexScheme, an integrated model for multi-tenant databases that support Extensibility, Evolution and Data Sharing. We have developed a prototype implementation of FlexScheme that uses specialized query plan operators to implement these features. A novel aspect of this work is an efficient implementation for versioning of shared data based on XOR Deltas. This solution is optimized for the typical SaaS environment where the latest available version of the application is the most popular and a limited number of tenants lag one or more versions behind.

Next-generation DBMSs have to be optimized for emerging server hardware which provides big main-memory capacity and many processing cores [2]. Besides offering superior performance, recent work [3] shows that main-memory DBMSs consume less energy than disk-based systems. However, main-memory is an expensive resource which 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.

The remainder of this paper is organized as follows: Section II gives an overview on the features required by Multi-Tenant DBMSs. Section III introduces FlexScheme, our integrated data management model for Multi-Tenant DBMSs. Section IV discusses the application of FlexScheme in a Multi-Tenant DBMS. We compare several storage mechanisms for shared data versioning in Section V, including our novel XOR delta approach, and present evaluation results in Section VI. Finally, Section VII concludes the paper.

II. Required Features for Multi-Tenant DBMSs

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. Moreover, tenants may subscribe to specific extensions offered by Independent Software Vendors (ISVs).

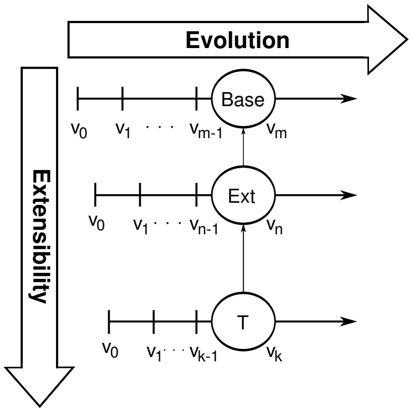

Evolution
Extensibility
Base
v_0 v_1 · · · v_{m-1} v_m
Ext
v_0 v_1 · · · v_{n-1} v_n
T
v_0 v_1 · · · v_{k-1} v_k

Fig. 1: Extensibility and Evolution from a Single Tenant's Perspective

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 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. 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. *Figure 1 shows the 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 by 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, a common practice of SaaS applications is to 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 be willing

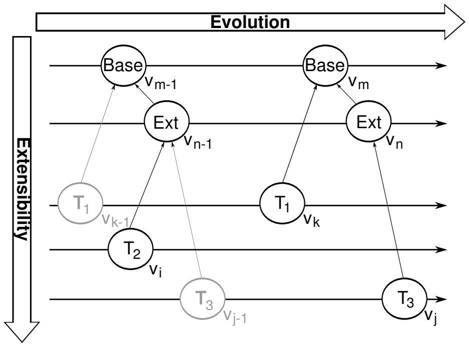


Fig. 2: Extensibility and Evolution for Multiple Tenants

to pay a higher service fee to avoid upgrading right away, allowing them to stay on a version with which all their extensions work. Managing each tenant's application instance separately is not feasible for 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. *Figure 2 depicts a SaaS application used by three tenants. It shows how the SaaS application and its extensions develop along 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, significant 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. *In Figure 2, 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 the 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.*

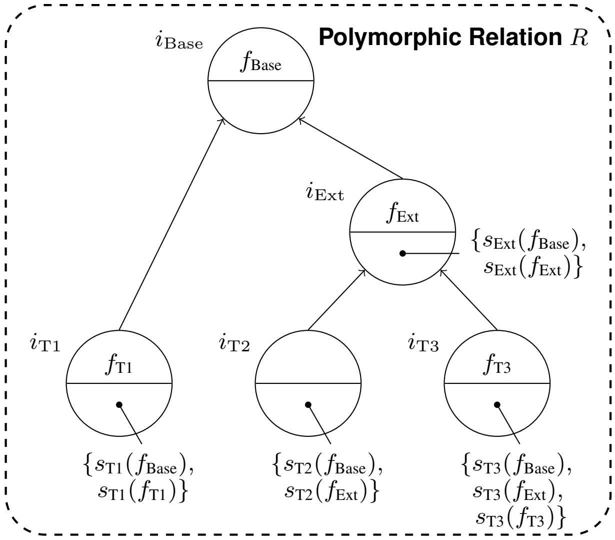

Polymorphic Relation R
i_{Base}
f_{Base}
i_{Ext}
f_{Ext}
$\{s_{\text{Ext}}(f_{\text{Base}}),$
$s_{\text{Ext}}(f_{\text{Ext}})\}$
i_{T1}
f_{T1}
i_{T2}
i_{T3}
f_{T3}
$\{s_{\text{T1}}(f_{\text{Base}}),$
$s_{\text{T1}}(f_{\text{T1}})\}$
$\{s_{\text{T2}}(f_{\text{Base}}),$
$s_{\text{T2}}(f_{\text{Ext}})\}$
$\{s_{\text{T3}}(f_{\text{Base}}),$
$s_{\text{T3}}(f_{\text{Ext}}),$
$s_{\text{T3}}(f_{\text{T3}})\}$

Fig. 3: Example of a Polymorphic Relation R

Shared data, e.g. supplier performance data, needs to be updateable, 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 a particular 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 should be 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 [4], and there are models for capturing the Evolution of an application [5], but there is currently no integrated model that captures both dimensions and Data Sharing together. We therefore propose such a model, called FLEXSCHEME.

III. FLEXSCHEME – A DATA MANAGEMENT MODEL FOR MULTI-TENANT DBMSS

FLEXSCHEME is a data management model for Multi-Tenant DBMSs. The model captures the interaction of a common base application and its extensions.

A. Data Structures

First, we discuss the needed data structures of the model. Figure 3 shows the global view with three Tenants T_1, T_2, and T_3, while Figure 4 shows the local view of Tenant T_3.

Definition 1. A *Fragment* $f^{(v)}$ is a set of n typed attributes defining the schema of a table chunk. Some of the attributes

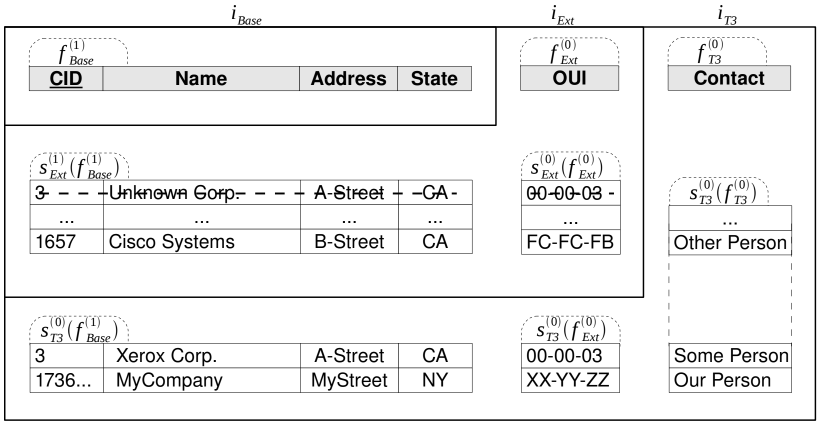


Fig. 4: Virtual Private Table for Tenant T_3

form the primary key attributes of the Fragment. v denotes a version number.

During application lifetime, the definition of a Fragment may change over time, 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 2. A *Fragment Sequence* $f = < f^{(0)}, \ldots, f^{(z)} >$ is a sequence of Fragments $f^{(v)}$. Within one Fragment Sequence f, the primary key attributes of each Fragment are identical: $pk(f) := pk(f^{(0)}) = \ldots = pk(f^{(z)})$.

Example. *Figure 3 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 shows the definition of $f^{(1)}_{Base}$.*

Data is stored within *Segments*. The schema of a single Segment is determined by a Fragment.

Definition 3. A *Segment* $s^{(v)}(f^{(w)})$ contains tuples that have the schema $f^{(w)}$. Again, v and w denote version numbers.

As discussed above, 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. A *Segment Sequence* $s(f)$ is a sequence of Segments $s^{(0)}(f^{(i)}), \ldots, s^{(z)}(f^{(j)})$. The Fragments $f^{(i)}, \ldots, f^{(j)}$ form part of the same Fragment Sequence f.

Example. *Figure 4 shows $s^{(1)}_{Ext}(f^{(1)}_{Base})$ which is part of the Segment Sequence $s_{Ext}(f_{Base})$. 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 5. 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, the application instances at the same level—as shown in Figure 2—get condensed to Instances, which contain Sequences of Fragments and Segments, representing schema and data evolution. The dependencies between Instance Versions are managed separately on a per-tenant basis.

Example. *The Instance i_{Ext} in Figure 3 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 6. 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 7. A *Polymorphic Database* D is a set of Polymorphic Relations.

B. Deriving Virtual Private Tables

The Polymorphic Relation R contains the versioning information for one particular relation across all tenants. To reconstruct the shape of the relation for a given tenant, the information from R has to be combined with the tenant-specific version information. This additional dependency is required to derive the tenant-specific shape of R. We call such a derived relation a *Virtual Private Table* (VPT), since this relation is specific to a particular tenant. The process of deriving a VPT for a particular tenant is comprised of several steps:

(1) First, the Instances on the path from the tenant's leaf node to the root node of the Polymorphic Relation have 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. *In Figure 3 the Instances i_{T3}, 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 $f^{(0)}_{T3}$, as seen in Figure 4.*

(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 overlayed. 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, where $t \triangleright s$ denotes the anti-join of t and s:

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

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. *In Figure 3 for Tenant T_3 there are two Segment Sequences $s_{Ext}(f_{Base})$ and $s_{T3}(f_{Base})$ available for the Fragment Sequence f_{Base}. The previous step selected the Fragment $f^{(1)}_{Base}$. As seen in Figure 4 the selected Segments are $s^{(1)}_{Ext}(f^{(1)}_{Base})$ and $s^{(0)}_{T3}(f^{(1)}_{Base})$. Since $s^{(0)}_{T3}(f^{(1)}_{Base})$ takes precedence over $s^{(1)}_{Ext}(f^{(1)}_{Base})$, the tuple with CID=3 from $s^{(1)}_{Ext}(f^{(1)}_{Base})$ is overwritten by the tuple from $s^{(0)}_{T3}(f^{(1)}_{Base})$ with the same CID.*

(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. *Figure 4 shows the full VPT of Tenant T_3.*

C. 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 [4], [6]. The object-oriented programming paradigm states that a program is a collection 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 [7].

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 subtyping, 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 (OODBMS) 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 DBMSs. For example, SQL:1999 differentiates between type and relation [8]. 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 [9], as well as in the object-oriented database world [10], [11]. Recent work [5], [12], [13], [14] 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 OODBMS in the form of class hierarchies [10], [11].

IV. Applying FlexScheme to Multi-Tenant DBMSs

Traditional DBMSs do not offer mechanisms to support Extensibility, Evolution and Data Sharing from within, so the SaaS application itself has to provide such mechanisms. To gain this flexibility, the persistence layer of the application typically makes use of schema mapping techniques [15], [16] which transfer the internal data representation to generic structures like Pivot Tables. These mechanisms help to overcome the limitation of current DBMSs: on-line schema changes may have a very high impact on the DBMS and only a very basic form of resource sharing and extensibility is supported. However, such techniques let the DBMS degenerate to a 'dumb data repository', since characteristic DBMS features like query optimization and indexing have to be reimplemented inside the SaaS application. 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.

When FLEXSCHEME is applied to a Multi-Tenant DBMS, the DBMS has explicit knowledge of the tenants' *Virtual Private Tables* that are dynamically derived from the extension hierarchy. 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* [12], [14].

We implemented a Multi-Tenant Main-Memory DBMS prototype which leverages all features of FLEXSCHEME. Support for Schema Evolution is based on Schema Modification Operators [12]. 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. Extensibility is realized by decomposition of relations and data overlay. Moreover, our prototype has specialized query plan operators for data overlay which are optimized for different access patterns on tenant-specific and shared data. Furthermore, 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. One of the remaining challenges is an efficient storage mechanism for Segment Sequences when data is shared between tenants and evolves over time.

In the following, we focus on the physical data organization of Segment Sequences which can exploit the characteristics of a typical SaaS application's workload: 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 only a few tenants lag marginally behind the most recent version of the application. We assume that a majority of tenants are using the most recent version of the application.

FLEXSCHEME keeps track of which Fragments and Segments are in use by tenants. Fragments and Segments that are no longer accessed by tenants can be pruned. The pruning of unused components improves space efficiency; this is very important for main-memory DBMSs. Pruning should be done as soon as possible and as often as possible. Therefore Pruning must not directly affect the overall system's performance.

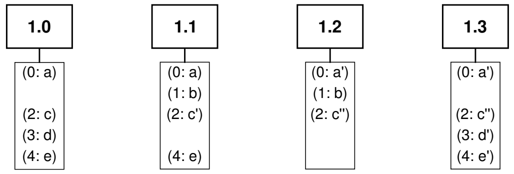


Fig. 5: Snapshot Approach

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 must be a light-weight operation to be able to perform this operation whenever needed without affecting co-located tenants.

V. PHYSICAL DATA ORGANIZATION OF SEGMENT SEQUENCES

In this chapter, we compare different physical data organization schemes. We discuss the data structures, the access behavior of pointwise lookups and scans, maintenance operations on Segment Sequences, as well as space efficiency of these approaches.

The maintenance operations on Segment Sequences such as creating or deleting Segments should be light-weight 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, but this is out of the scope of this paper.

A. 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 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.

1) 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.

2) Pointwise Access: Data tuples of a materialized Segment can be accessed pointwise by looking up the tuple identifier in the index of the materialized Segment.

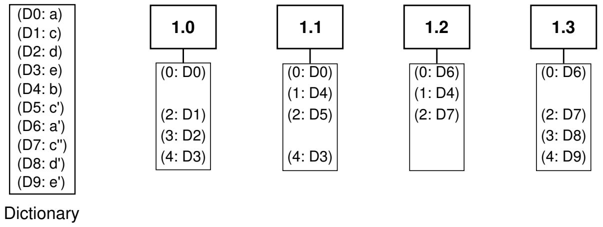


Fig. 6: Dictionary Approach

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

4) 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.

5) 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 light-weight operation in the Snapshot data organization scheme.

6) 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 other. Orthogonally, value-based compression can be applied to single Segments to further reduce space requirements.

B. 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 6 shows the Dictionary data organization scheme.

1) 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.

2) Pointwise Access: The data tuples of a Segment can be accessed pointwise by looking up the tuple identifier in the index and then retrieving the value from the dictionary.

3) 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.

4) 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.

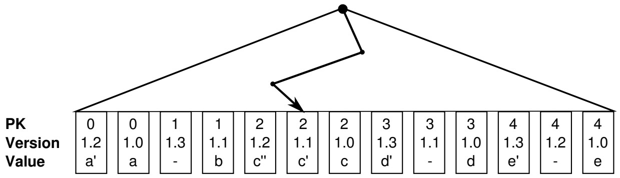


Fig. 7: Temporal Approach

5) 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 heavy-weight.

6) 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.

C. Temporal

Temporal databases [17], [18], [19] 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 Sequences 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 7 shows that approach.

1) 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 7 shows a simplified Temporal Index, since 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.

2) Pointwise Access: The tuples of a Segment can be accessed pointwise by performing a lower bound lookup of the key. The version with the highest version number smaller or equal to the desired version is returned.

3) Scan Access: Temporal databases support scanning all tuples of a version sorted by tuple identifier. As each Segment

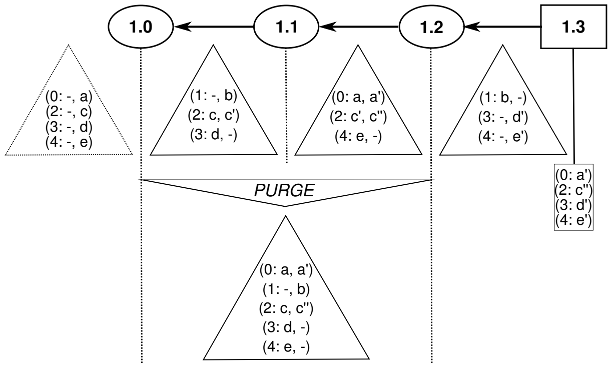


Fig. 8: Differential Delta Data Approach

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.

4) 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.

5) 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 heavy-weight.

6) 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.

D. 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 overlayed 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 8 shows this approach.

1) 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.

2) Pointwise Access: Pointwise 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 lookups 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.

3) Scan Access: Scans can be performed by overlaying the Deltas with the materialization of the last Segment. This overlay operation can be implemented efficiently with an N-ary merge join.

4) 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 overlayed with the currently available materialization to form a new materialization. The old materialization is then removed. This removal is light-weight, since the Deltas do not depend on a particular materialization.

5) 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 8. Since this can be performed without affecting shared data structures, we consider this a light-weight operation.

6) Space Efficiency: The Differential Delta data organization approach eliminates some redundancy compared to the Snapshot approach, as only the differences between subse-

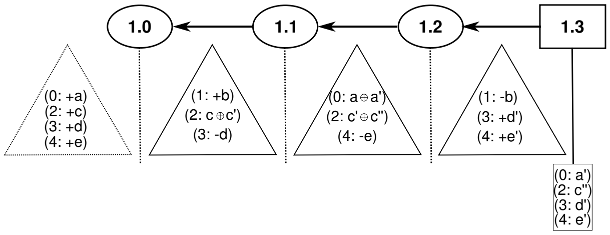


Fig. 9: XOR Delta Approach

quent 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 more 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 heavy-weight operations, especially the Purge operation. Since this is not desirable, we do not eliminate this redundancy.

E. XOR Delta Approach

The *XOR Delta* approach is based on the previously discussed Differential Delta data organization scheme. The previous approach enables light-weight 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 [20] the Diff Entries of the Differential Delta approach. We refer to this optimization as the *XOR Delta* approach. Figure 9 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.

1) 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.

2) Pointwise Access: Algorithm 1 describes the pointwise access by primary key, which is closely related to the pointwise 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.

For example, to access the tuple with PK 2 in version 1.1, cf. Figure 9, 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 $c'' \oplus (c' \oplus c'') = (c'' \oplus c'') \oplus c' = 0 \oplus c' = c'$.

Algorithm 1 Lookup tuple with key k in version ver

```
 1: $V \leftarrow$ nearest materialized version to $ver$
 2: if $ver \leq V$ then
 3:     $data \leftarrow$ retrieve data for key $k$ from materialization $V$
        /* $data$ might be NUL */
 4:     for $i = V$ to $ver + 1$ do
 5:         $xor \leftarrow$ XOR entry for key $k$ from delta $(i - 1 \leftrightarrow i)$
 6:         if $xor$ is update then
 7:             $data \leftarrow data \oplus xor$
 8:         else if $xor$ is delete then
 9:             $data \leftarrow xor$
10:         else if $xor$ is insert then
11:             $data \leftarrow \mathrm{NUL}$
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
```

As in the Differential Delta approach, a materialization of the last Segment increases performance. Otherwise, the tuple value has to 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 pointwise lookups by choosing the closest materialization as starting point for the reconstruction of the tuple value.

3) 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.

4) Creation of New Segments: The creation of new Segments is identical to the Differential Delta approach.

5) 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 light-weight.

6) 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.

VI. Evaluation

In this section, we compare the different data versioning approaches presented in the previous section. The criteria are (1) execution times of pointwise lookups and full table scans, (2) space requirements, and (3) execution times of maintenance operations.

The various approaches have been implemented in Java 6. Indexes and Deltas make use of Oracle Berkeley DB Java Edition 4.0 that offers a B$^+$-Tree implementation. The fan-out of the B$^+$-Trees has been configured to mimic Cache Conscious B$^+$-Trees [21]. Data areas 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.

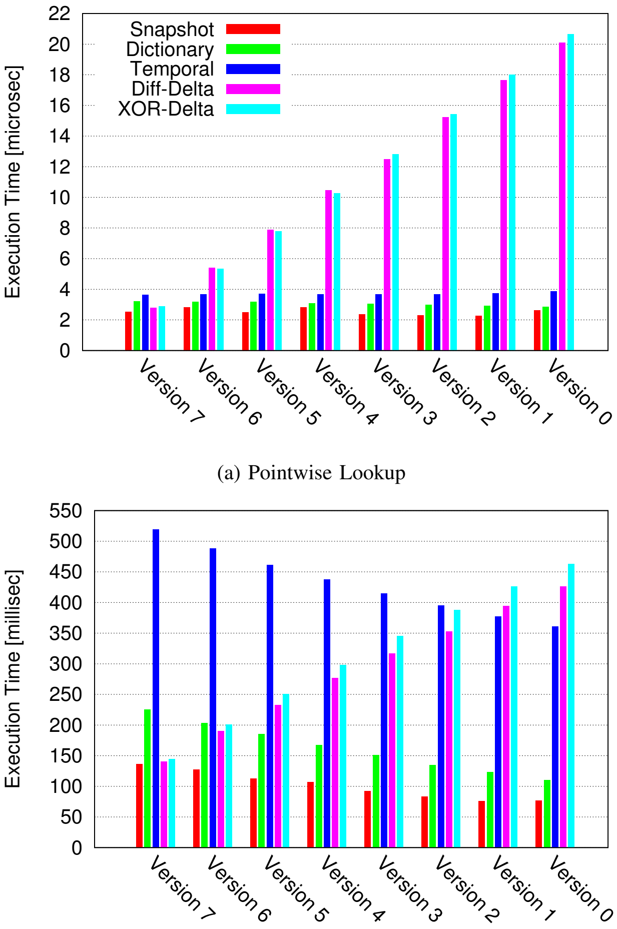
All experiments have been configured as follows. A single Segment Sequence consists of eight Segments, each representing a particular version of the shared data. Version 0 is the base version with 100,000 tuples. The average tuple size is 120 bytes, the primary key size is 16 bytes, corresponding to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

A. Data Access

Figure 10a shows the behavior of the five different implementations for pointwise accesses by primary key lookup. 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 lookups 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 lookup operation, as described in Section V.

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 10b 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.



(a) Pointwise Lookup

(b) Full Table Scan

Fig. 10: Query Execution Times

When scanning previous versions, the Snapshot, Dictionary, and Temporal approaches have the same performance as scanning the most recent version, except that the number of scanned tuples varies. As stated before, the Differential Delta and XOR-Delta performance decreases due to the lookups in the deltas.

B. Space Requirements

Figure 11 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

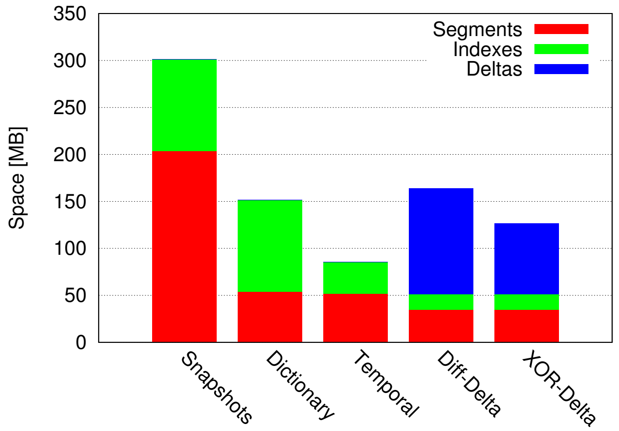
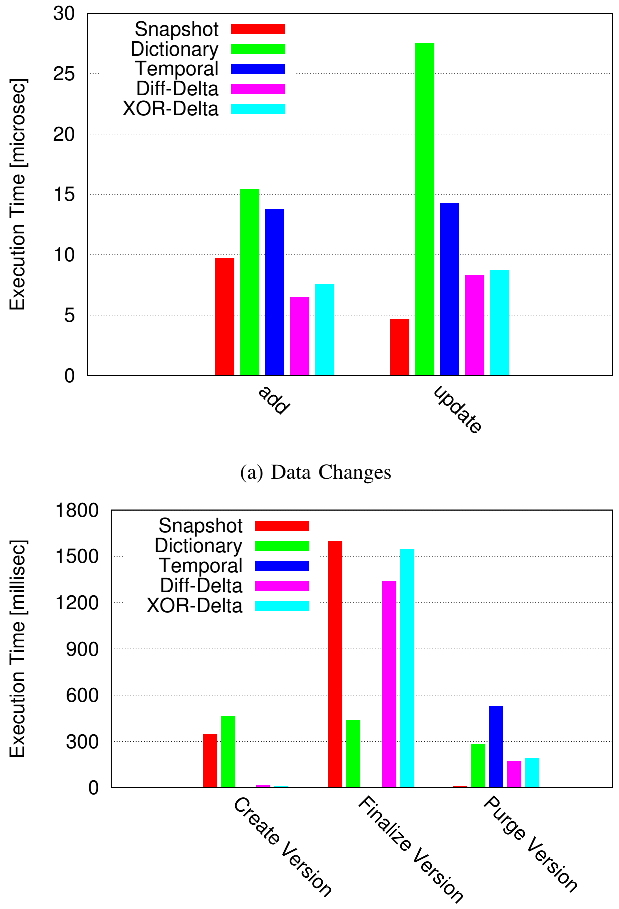


Fig. 11: Space Requirements



(a) Data Changes

(b) Versioning Operations

Fig. 12: Maintenance Execution Times

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 stores 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. Second, shared data is accessed by all co-located tenants concurrently. Therefore, no additional overhead should be put on data access to shared data.

C. Maintenance Operations

Figure 12a 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 lookup 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 data of 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 lookups 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 12b 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.

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.

D. Summary

The XOR Delta data organization scheme is a promising approach for implementing versioning of shared data in a Multi-Tenant Database. 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, since the XOR Deltas can be applied in a forward and a backward manner without reorganization.

VII. Conclusions and Future Work

In this paper we presented FlexScheme, a model for handling meta-data and shared data in a Multi-Tenant DBMS. FlexScheme captures Extensibility, Data Sharing and Evolution in one integrated model. Furthermore, we implemented FlexScheme as part of our Multi-Tenant DBMS and focused on efficient storage mechanisms. We presented a novel approach, called XOR Delta, which is based on XOR encoding and is optimized for a main-memory DBMS.

Our evaluation shows that the XOR Delta data organization scheme is a promising approach for implementing versioning of shared data under a typical SaaS workload, where the majority of tenants uses the most recent version of shared data and only a few tenants lag marginally behind the latest version.

As future work, the model could be enhanced to support branching in the evolution dimension, as known from revision control systems. We assumed a typical SaaS workload, where the latest available version has to be materialized, but a next generation SaaS application could offer several branches, e.g. a current branch with the newest features and a stable branch with long term support. In this case, it would make sense to materialize more than one Segment of a Segment Sequence in the XOR Delta data organization scheme, as two versions of the application would be very popular.

With branching, the evolution dimension can be seen as a graph of Instance Versions. A differential data organization scheme would materialize some Instance Versions corresponding to nodes in the graph. Apart from that, only Deltas corresponding to edges in the graph would be stored. Any Instance Version in this graph could be materialized and it even would be possible to add new edges to the graph by computing the Deltas. This could be used to reduce the number of deltas on the shortest path from any materialized Instance to a target Instance. For a given workload, the graph could be augmented with node weights corresponding to the popularity of an application version. Based on such a representation, it may be possible to find the optimal combination of deltas and materializations for a given workload.

References

[1] C. D. Weissman and S. Bobrowski, "The Design of the force.com Multitenant Internet Application Development Platform," in *SIGMOD*, 2009.

[2] H. Plattner, "A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database," in *SIGMOD*, 2009.

[3] M. Poess and R. O. Nambiar, "Tuning Servers, Storage and Database for Energy Efficient Data Warehouses," in *ICDE*, 2010.

[4] S. Khoshafian and R. Abnous, *Object Orientation: Concepts, Languages, Databases, User Interfaces*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[5] C. Curino, H. J. Moon, and C. Zaniolo, "Automating Database Schema Evolution in Information System Upgrades," *HotSWUp*, 2009.

[6] O. Nierstrasz, "A Survey of Object-Oriented Concepts," in *Object-Oriented Concepts, Databases and Applications*. ACM Press and Addison Wesley, 1989.

[7] B. Eckel, *Thinking in Java (2nd ed.): The Definitive Introduction to Object-Oriented Programming in the Language of the World-Wide Web*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[8] J. Melton, *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features*. New York, NY, USA: Elsevier Science Inc., 2002.

[9] J. F. Roddick, "A Survey of Schema Versioning Issues for Database Systems," *Information and Software Technology*, vol. 37, no. 7, 1995.

[10] A. Kemper and G. Moerkotte, *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice-Hall, 1994.

[11] G. Moerkotte and A. Zachmann, "Towards More Flexible Schema Management in Object Bases," in *ICDE*. IEEE Computer Society, 1993.

[12] C. Curino, H. J. Moon, and C. Zaniolo, "Graceful Database Schema Evolution: the PRISM Workbench," *PVLDB*, vol. 1, no. 1, 2008.

[13] H. J. Moon, C. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo, "Managing and Querying Transaction-Time Databases Under Schema Evolution," *PVLDB*, vol. 1, no. 1, 2008.

[14] H. J. Moon, C. Curino, and C. Zaniolo, "Scalable Architecture and Query Optimization for Transaction-time DBs with Evolving Schemas," *SIGMOD*, 2010.

[15] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, "Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques," in *SIGMOD*, 2008.

[16] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold, "A Comparison of Flexible Schemas for Software as a Service," in *SIGMOD*, 2009.

[17] C. S. Jensen and R. T. Snodgrass, "Temporal Database," in *Encyclopedia of Database Systems*, 2009.

[18] ——, "Temporal Data Management," *IEEE Trans. Knowl. Data Eng.*, vol. 11, no. 1, 1999.

[19] M. Stonebraker, "The Design of the POSTGRES Storage System," in *VLDB*, 1987.

[20] S. K. Cha and C. Song, "P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload," in *VLDB*, 2004.

[21] J. Rao and K. A. Ross, "Making B$^+$-Trees Cache Conscious in Main Memory," in *SIGMOD*, 2000.