Native Support of Multi-tenancy in RDBMS for Software as a Service

Oliver Schiller Benjamin Schiller Andreas Brodt Bernhard Mitschang

Applications of Parallel and Distributed Systems
IPVS, Universität Stuttgart
Universitätsstr. 38, 70569 Stuttgart
{schillor, schillbe, brodtas, mitsch}@ipvs.uni-stuttgart.de

ABSTRACT

Software as a Service (SaaS) facilitates acquiring a huge number of small tenants by providing low service fees. To achieve low service fees, it is essential to reduce costs per tenant. For this, consolidating multiple tenants onto a single relational schema instance turned out beneficial because of low overheads per tenant and scalable manageability. This approach implements data isolation between tenants, per-tenant schema extension and further tenant-centric data management features in application logic. This is complex, disables some optimization opportunities in the RDBMS and represents a conceptual misstep with Separation of Concerns in mind.

Therefore, we contribute first features of a RDBMS to support tenant-aware data management natively. We introduce tenants as first-class database objects and propose the concept of a tenant context to isolate a tenant from other tenants. We present a schema inheritance concept that allows sharing a core application schema among tenants while enabling schema extensions per tenant. Finally, we evaluate a preliminary implementation of our approach.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*data models, schema and subschema*; H.2.4 [**Database Management**]: Systems—*relational databases*

General Terms

Design, Management

Keywords

Multi-tenancy, Software as a Service, Relational Database Management Systems, Meta Data Management, Logical Data Model

EDBT 2011, March 22–24, 2011, Uppsala, Sweden.

1. INTRODUCTION

Software as a Service (SaaS) constitutes a fast-growing business model for the sales of software that bases upon the principle of outsourcing. At SaaS, a service provider hosts an application on its infrastructure and delivers it as a service to several organizations. An organization, called a tenant, subscribes for the service and accesses it across the Internet through standard web technology.

The choice of existing SaaS offerings already spans a wide range of business applications such as e-mail [25], e-commerce [10], and customer relationship management [21] applications. In the future, the market and the relevance of SaaS applications will further increase. The International Data Corporation (IDC) forecasts that the revenue of the SaaS market will grow at a compound annual growth rate of 25,3 % through 2014 [17].

From the provider's perspective, SaaS opens the possibility to acquire a large number of small companies, the so-called long tail [2], as new customers. These companies often cannot afford or dread the high costs when purchasing and operating business applications according to the traditional business model. Thus, a SaaS application may attract these companies if the service fee is much lower compared to the costs of a traditional solution. A service provider may accomplish offering its service at a low fee by leveraging economies of a scale. Thus, it is an integral challenge to reduce the overhead per tenant.

A key opportunity to reduce the overhead per tenant constitutes the concept of multi-tenancy that describes consolidating multiple tenants onto one resource [7], e. g. one software instance. Multi-tenancy accounts for a higher resource utilization and lower operational costs due to less resources that the provider must manage and maintain. Several approaches to enable multi-tenancy have been discussed throughout the whole application stack [13, 6].

1.1 Multi-tenancy at the Data Tier

A particularly important challenge in a SaaS application is concerned with enabling multi-tenancy at the data tier [7, 9]. Put simply, the challenge is to consolidate multiple tenants onto one data tier resource, e. g. one database server, while at the same time isolating them among one another, as if they were running on physically segregated resources. Note that the concept of Database as a Service (DaaS) has to cope with the same challenge, but has other requirements. DaaS offerings, e. g. Amazon SimpleDB [1], focus on providing a database management system (DBMS) that allows a tenant to run its application against. Hence, a broad range of different applications that have a broad range of characteristics access the system. Contrarily, systems at the data tier of a SaaS application are accessed by the same application for each tenant. This entails a high potential for reducing the overhead per tenant because data access patterns and data structures are very similar between tenants.

The common adoption of relational database management systems (RDBMS) at the data tier demands to keep in view how to implement multi-tenancy with these systems. Until now, three main approaches have been proposed and evaluated [16]. They vary in the degree of consolidation:

1. **Shared Machine:** The tenants share a single machine, but each tenant obtains a private database instance.
2. **Shared Process:** The tenants share a single database instance, but each tenant obtains a private set of tables.
3. **Shared Table:** The tenants share a single database instance and a single set of tables.

Jacobs and Aulbach [16] pointed out that the Shared Machine approach allows consolidating only a few tenants onto one machine due to the large main memory footprint of a database instance. The Shared Process approach consumes less main memory per tenant, yet main memory consumption increases quite fast with the number of tenants, as each tenant obtains a dedicated schema instance. In contrast, the main memory consumption of the Shared Table approach remains constant if the number of tenants increases. Furthermore, recent performance tests show that the Shared Table approach may yield a better buffer pool utilization than the Shared Process approach for a large number of small tenants [3, 23].

To conclude, the Shared Table approach seems promising for a provider that targets the long tail because it offers the lowest overhead per tenant and, thus, is suitable for a large number of small tenants, e. g. 1,000 tenants each having less than 50 MB of data and at most 5 concurrent users. A popular example that facilitates this estimation constitutes the customer relationship management application salesforce.com [21, 24]. They adopt the Shared Table approach to consolidate up to 17,000 tenants onto one database [18].

1.2 Shared Table Approach

The shared table approach shares a single set of tables between tenants. Each table possesses a column that stores the tenant a table row belongs to. A selection on this column allows to retrieve the row set of a certain tenant.

If the tenants share one set of tables, all tenants have the same table schema. However, even for simple applications, a tenant may want to extend its logical schema according to its needs. By way of example, assume that an e-commerce service, which offers creating a shop, stores the products to sell in a table *Item* that has the attributes *name* and *price*. All tenants share these attributes, but each tenant may want to add different additional attributes. For example, a tenant that sells books may want to add an attribute *pages*, whereas a tenant that sells shoes may want to add an attribute *color*. Moreover, the schema modifications of one tenant must not affect the logical schemas of other tenants. Besides schema extension per tenant, other requirements such as statistics per tenant, backup and recovery per tenant, migration of a tenant from one machine to another and so forth exist.

Current RDBMS implementations do not provide native functionality that fulfills the requirements of a tenant-aware data management based upon the Shared Table approach. Moreover, as pointed out by previous research, useful features may yield a high performance penalty or their scalability is limited. For example, row-based access control mechanisms may increase query latency by 15% [23] or Microsoft SQL Server permits only up to 30,000 Sparse Columns per table which reduces the scalability of per tenant schema extensions [4].

Therefore, existing solutions that adopt the Shared Table approach implement required tenant-aware data management features within the application. For example, schema mapping techniques within the application map the tenants' logical schemas onto the physical schema in the database [3, 4]. Thereby, the application itself manages all meta data, e. g. the data type of a column. Nevertheless, handling the meta data in the application does not solve by implication the redundancy of the core application schema and its management. In addition, the RDBMS loses knowledge about the data and the relations and thus optimization opportunities. For instance, generic access operations replace more efficient operations specific to a certain data type. Developing required tenant-aware data management functionality within the application is complex, error-prone and expensive. Furthermore, the implemented solution may lack in data security between tenants due to bugs in the application, which are likely given the complexity of development.

In summary, we consider implementing tenant-aware data management features in the application a conceptual misstep, as a lot of the RDBMS efforts are re-implemented with multi-tenancy support. In addition, the principle Separation of Concerns is ignored. We argue that the native support of tenant-aware data management features in RDBMSs for SaaS is crucial. Such a system takes the burden of sharing data resources from the application and enhances the security and isolation between tenants. Furthermore, it is able to optimize data access and to provide tenant-centric system-level operations.

1.3 Contributions

Previous research also motivates the requirement of a RDBMS that provides native multi-tenancy support [3, 23, 14], but it does not propose concrete solutions. The development of such a system that optimally supports multi-tenancy bears a comprehensive challenge. As a first step to meet this challenge, we contribute a tenant-aware data dictionary that allows for tenant-aware metadata management. We consider this a significant groundwork for further tenant-aware features. Additionally, we show an integration of a storage model in our tenant-aware data dictionary. Our approach does not impose a restriction on the adopted storage model, though. In detail, our contributions are as follows:

1. We introduce a tenant as a first-class database object and the concept of a tenant context that determines the tenant's view of the database.
2. We present a tenant-aware schema inheritance concept in order to maintain the sharing of the application's core schema that is invariant between tenants while allowing per-tenant schema extensions. Moreover, our concept inevitably segregates tenant-specific objects between tenants and seamlessly integrates into our previously introduced tenant context.
3. We present a SQL language extension that allows managing tenants and building schema hierarchies according to our schema inheritance concept.
4. We illustrate a preliminary implementation of these concepts that, with respect to the physical storage layout, builds upon the Shared Table approach. Finally, we evaluate our tenant-aware data dictionary implementation regarding performance and usability.

The paper is organized as follows. After a detailed consideration of the Shared Table approach and related research work, we present in Sec. 3 our tenant-aware schema inheritance concept. Section 4 describes a prototypical implementation of the previously presented concept. In the subsequent section, we evaluate this prototype. Finally, Section 6 concludes our work.

2. SHARED TABLE APPROACH

Previous research [3, 23] and existing solutions [21, 24] characterize the Shared Table approach as well-suited for serving a large number of small tenants. This section reviews the Shared Table approach and its variants. We embark upon the basic principle of the Shared Table approach. Moreover, we describe variants to per-tenant schema extension as particularly important requirement. Finally, we give main drawbacks.

2.1 Basic Principle

The Shared Table approach shares one database instance and also one set of tables between tenants. That is, it stores tuples of one tenant intermingled with tuples of other tenants. In order to distinguish the tuples of different tenants, each table obtains an attribute *tenant* that stores the respective tenant of each tuple. Figure 1 shows an example of a Shared Table *Item* that stores tuples for the tenants *Gonzo Books* and *Kermit Shoes*.

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

```
SELECT * FROM Item
```

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

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

The Shared Table approach allows consolidating a large number of tenants onto one database instance. The number of tenants is not limited by the available main memory because the size of the data dictionary remains constant if a new tenant is created. The size remains constant because a tenant does not own dedicated database objects such as tables and indexes. Contrarily, the Shared Process approach assigns dedicated database objects to a tenant. Thus, each tenant requires a considerable amount of catalog information that reduces scalability and decreases the amount of main memory that remains for query processing.

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

Item

Tenant	Name	Price
Gonzo Books	1984	9,90
Kermit Shoes	Nike Free 5.0	100,00
Kermit Shoes	Brooks Glycerin 8	140,00
Gonzo Books	PostgreSQL	47,99

Figure 1: Example of the Shared Table approach.

2.2 Per-Tenant Schema Extension

In Section 1, we pointed out that tenants require extending the core application schema according to their individual needs. As a result of such extensions, the logical structure of a table may differ from tenant to tenant. Consolidating the differently structured logical tables into one Shared Table represents a difficult and eminently important challenge.

An apparent approach to fulfill this challenge is to add tenant-specific attributes to the schema of the Shared Table. Hence, the schema of the Shared Table represents the union of the tenants' logical table schemas. This approach scales poorly because it yields a wide sparse table. For instance, if each of 10,000 tenants defines five additional attributes, a tuple stores useful values only in a small fraction of the total number of attributes (at least 50,000) while filling the remaining attributes with NULL values. Popular RDBMSs do not support such a high number of attributes per table, e. g. IBM DB2 V9.7 LUW supports a maximum of 1012 [15] and Oracle 11g a maximum of 1000 attributes per table [20]. Moreover, storing many NULL values causes overhead that entails considerable performance degradation. The Interpreted Attribute Storage Format that is implemented in Microsoft SQL Server as Sparse Columns feature deals with this issue [5]. However, it prevents random attribute access optimizations and permits only up to 30,000 Sparse Columns per table.

To prevent such scalability issues, other approaches that rely on mapping the tenants' logical schemas onto a fixed generic schema in the RDBMS have been adopted. Subsequently, we outline two common techniques [7, 3]:

Pivot Table. This technique maps each cell value of the tenants' logical tables to one row of a single Pivot Table. To identify the cell of the logical table, it additionally stores the tenant, the row number and attribute number. That is, the Pivot Table describes the function: $(tenant \times rowno \times attributeno) \rightarrow value$.

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

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

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

There exist, of course, variants and hybrids of the two described approaches. The Extension Table technique stores the attributes of the core application schema in a Shared Table and the tenant-specific attributes in a so-called Extension Table. A common surrogate ties the parts of the tuple across the tables. A tenant may possess an own Extension Table or it may share one Extension Table with others if their tenant-specific attributes are exactly identical. This approach has the same scalability issues as described for the Shared Process approach due to the potentially high number of tables. To prevent this, all tenants may alternatively share one Extension Table that in turn adopts an approach such as Pivot Table. Aulbach et al have presented a technique called Chunk Folding [3]. This technique stores the tenant-specific attributes in a fixed set of so-called Chunk Tables. A Chunk Table is similar to a Pivot Table, but stores a set of columns instead of only one. The logical tables

are divided into suitable chunks and distributed across the fixed set of Chunk Tables. Several other related mapping techniques have been proposed and evaluated in the context of storing XML [8].

The described techniques are very flexible and scalable. They allow mapping an unlimited number of arbitrary logical tables onto a fixed generic schema. However, they possess several drawbacks in the context of multi-tenancy which we subsequently describe.

2.3 Drawbacks

The Shared Table approach facilitates to reduce the costs per tenant. Yet, it implements most per-tenant operations such as schema extension, backup and recovery, statistics gathering and so forth in the application. Consecutively, we discuss the main drawbacks of this approach focussing on the data dictionary as an important cross-cutting facility, on which our work concentrates.

The presented techniques to per-tenant schema extension assume that a query rewriter in the application conducts a major part of data dictionary management, e. g. storing the data type of a tenant-specific attribute. For this purpose, the query rewriter must store and manage the data dictionary information which may, in practice, limit the scalability of the presented techniques and does not avoid redundancy per se. Moreover, the loss of data dictionary responsibility prevents the RDBMS from using optimized data access operators, e. g. an integer comparator instead of a string comparator. In addition, the query rewriter enforces isolation of meta data and application data between tenants. Therefore, the rewriter is a complex and critical component that requires a clean design and sophisticated testing. Such a critical component should not be implemented by each and every application anew.

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

The presented techniques to per-tenant schema extension target high flexibility at the schema level. To obtain this flexibility, they accept a performance loss due to the implied storage layout. They typically have to struggle with tuple reconstruction or null compression overhead. In the scope of multi-tenancy, lower flexibility suffices, as a tenant's logical table structure is well-known and the corresponding data set is dense. A tenant-aware data dictionary can help to exploit this fact for a more efficient storage layout.

3. NATIVE MULTI-TENANCY SUPPORT

Our approach gets rid of the drawbacks discussed in the previous section. It allows maintaining an application core schema while enabling per-tenant schema extensions. At this, the RDBMS maintains the data dictionary, such as data types of tenant-specific attributes. This enables some optimizations. Moreover, it avoids redundancy to improve scalability and to reduce the per-tenant costs. Our approach also facilitates direct access to the RDBMS system without the need of a complex query rewriter. Thus, a system-level mechanism enforces isolation and security between tenants. The achievement of these objectives relies on a deeply integrated, native support of multi-tenancy in RDBMSs.

3.1 Tenants as First-Class Database Object

The primary goal for tenant-aware data management is to provide high degrees of sharing and suitable management features. For this purpose, tenants must use resources in common, but logically segregated by means of virtualization. From a conceptual view, each tenant requires a virtual database that logically contains the objects that relate to the tenant and logically isolates it from other tenants. To create such a virtual database, we introduce the idea of a *tenant context*. A tenant context is associated with a specific tenant and keeps all information that allows determining the tenant's virtual database. Hence, if the database management system uses a tenant context to carry out operations on a virtual database, we can easily switch the context and therefore the virtual database that is processed. This concept allows executing operations aligned to tenant boundaries, e. g. backup and recovery, as well as system-level operations that concern different tenants, e. g. reorganization of database files. Furthermore, different contexts may reference the same objects. This enables flexible sharing patterns. To handle, maintain and identify tenant contexts, a RDBMS must know about tenants in the first place. Therefore, we introduce tenants as first-class database objects that allow identifying a certain context. Further, we assume that the database management system knows the tenant that performs an operation, e. g. a query. With this information, the system can carry out the operation within the corresponding context and it can ensure that the operation is restricted to this context. Thus, the tenant context determines a logically closed container with inviolable boundaries that establishes system-level security and isolation between tenants.

3.2 Schema Inheritance Concept

RDBMSs provide a data dictionary that stores information about data and allows tailoring a database to the needs of a specific application. The data dictionary roughly stores two different kinds of meta data: physical and logical. The physical meta data describes how data is stored on disk and what the available access paths are. The logical meta data expresses the application-specific data structure by means of the relational model, i. e. by tables, attributes and constraints. We argue that the contents of the data dictionary look closely alike between tenants, especially with respect to the logical meta data. The reason is that, in our scenario, the application that accesses the system is the same across all tenants. Thus, there exists one core application schema that slightly differs between tenants by reason of discussed per-tenant extensions, e. g. books and shoes. Exploiting this fact to prevent storing and managing redundant meta data improves scalability. Moreover, manageability improves if the core application schema can be maintained centrally. Our approach accounts for these aspects. It allows centrally maintaining a core application schema that may be extended according to the needs of a tenant without affecting other tenants.

For this purpose, our approach picks up the idea of SQL schemas as proposed by the SQL standard. SQL schemas represent namespaces that allow to group database objects logically. For example, schemas allow segregating different applications or independent parts of an application that use the same database. Our concept adopts SQL schemas with a slightly different intention. We use them to group objects that are redundant among tenants and to segregate objects of different tenants. For this, we associate a tenant's context with a schema and introduce a schema inheritance concept. Schema inheritance allows deriving a schema from another schema. Thereby, a derived schema inherits the objects that are defined in the parent schema. Note that a derived schema must possess all objects defined in the parent schema. Our concept prohibits mod-

ifying or removing inherited objects. Yet, it allows extending and creating objects according to a defined set of rules. Therefore, it defines three different schema types: shared schema, virtual schema and tenant schema. Subsequently, we describe each schema type in detail. For our further discussion, it is mandatory to understand the difference of a table definition and a table instance. A table definition expresses the structure of the table whereas a table instance describes the actual data that is stored in rows.

3.2.1 Shared Schema

Multi-tenant applications may require tables to store data that is specific to the application and invariant between tenants. For example, an application may store country codes and names or the top 1000 securities of the stock market in a table within the database. In such a case, the tenants only read the table while the provider or an appropriate application maintains its contents. To support this pattern, we introduce the concept of a shared schema. A shared schema is shared among all tenants. That is, all tenant contexts include this schema. Currently, we assume that tenants access shared schemas read-only. We claim that this access pattern usually suffices, as tenants constitute closed organizational units. Yet, assuming a tenant-aware security model, existing access control concepts, e. g. row-based access control, could be applied as well.

Shared schemas are final. That is, another schema cannot inherit from a shared schema by what shared schemas do not support per-tenant extensions.

3.2.2 Virtual Schema

This type intends to define the core application schema that a tenant may extend according to its individual needs. Hence, a virtual schema describes the schema parts that are invariant between tenants. As opposed to the shared schema and the tenant schema, a virtual schema is without table instances. Consequently, it is impossible to store data using a virtual schema. We consider the defined tables in a virtual schema as pure definitions that may be extended and instantiated in derived schemas.

Virtual schemas are arranged hierarchically. The hierarchy describes an inheritance relation. That is, a virtual schema can inherit from another virtual schema. Thereby, the schema inherits all the database objects contained in the parent schema. Thus, an inheritance relation between two schemas also imposes an inheritance relation on the contained objects. We use the same nomenclature to describe the relations between these objects.

We allow extending inherited table definitions by new attributes, new constraints and indexes. Note that we forbid to modify inherited table definitions. For example, renaming or reordering of attributes defined in the parent table definition is disallowed. As derived table definitions cannot omit or modify inherited attributes, it is guaranteed that index and constraint definitions remain valid for derived tables. Newly created attributes are always placed after the attributes that already exist. Hence, we avoid intermingling attributes of different schemas. We claim that this helps to stay consistent with application code because the application must access the attributes either by name or by the position. A modern application should pursue a clean separation between data and its presentation. Hence, the ordering of the attributes in the data layer should not influence the ordering of the attributes in the presentation. For this purpose, this restriction facilitates a clean application design. Note that we disallow extending the primary key definition, existing referential integrity constraints and existing indexes.

Furthermore, our concept allows defining new tables in a derived virtual schema. Nevertheless, our concept mandates that the names are unique across the whole inheritance path.

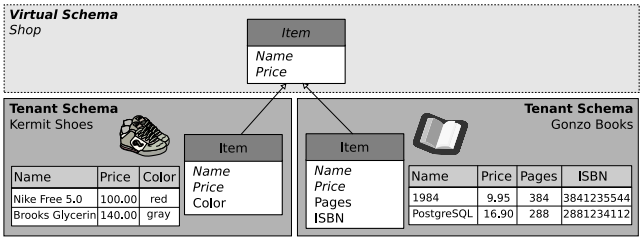


Figure 2: Illustration of our schema inheritance concept using the introduced e-commerce scenario.

3.2.3 Tenant Schema

As opposed to shared schemas and virtual schemas, a tenant schema relates to a specific tenant. Each tenant possesses an associated tenant schema that represents a part of its context. A tenant schema must inherit from a virtual schema. The same rules as described for a virtual schema with respect to extending and creating objects apply. From this perspective, a tenant schema behaves similar to a virtual schema, but there exists an important difference. In contrast to a virtual schema, a tenant schema includes table instances and a tenant schema is final with respect to inheritance. That is, another schema cannot inherit from a tenant schema. We propose that creating a tenant automatically creates an associated tenant schema as well as instances of table definitions that are defined in the virtual schema from which the tenant schema inherits.

Figure 2 illustrates a simple example of the presented concept. The upper part of the figure models the virtual schema *Shop* that defines a table *Item*, as introduced in our e-commerce scenario. The table *Item* has two attributes *Name* and *Price*. The lower part of the figure illustrates two derived tenant schemas. The tenant schema *Kermit Shoes* in the left and the schema *Gonzo Books* in the right. The schema *Kermit Shoes* extends the table *Item* by an attribute *color*, whereas the schema *Gonzo Books* extends it by an attribute *pages* and another attribute *ISBN*. In addition, the tenant schemas contain the respective instances of the table *Item*.

3.3 Language Extension

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

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

The statement `CREATE SHARED SCHEMA` creates a new shared schema with the given name. In order to create a virtual schema, the statement `CREATE VIRTUAL SCHEMA` is available. The optional clause `INHERITS FROM` specifies the virtual schema from which the newly created schema inherits. Both statements create appropriate entries in the data dictionary (see Sec. 4.2).

To manage tenants, we propose the following statements:

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

The `CREATE TENANT` statement creates a new tenant with the given name. In addition to creating a tenant, the statement creates an associated tenant schema and table instances for the inherited

table definitions. For this purpose, the clause `SCHEMA INHERITS FROM` specifies a virtual schema from which the tenant's schema inherits. The `DROP TENANT` statement drops the given tenant as well as dependent objects, most notably the associated tenant schema.

In order to carry out operations within a tenant's context, the RDBMS requires to know the tenant that performs an operation. For this purpose, the `SET TENANT` statement explicitly causes the system to switch to the given tenant. Instead of explicitly setting the tenant, the authentication process may set the appropriate tenant by mapping users to tenants during connection establishment. Nevertheless, applications typically exploit a connection pool to avoid the overhead of re-establishing a connection. Thus, switching a tenant on an established connection is necessary. Our mechanism supports this requirement. To increase security, we envision an authentication mechanism that extends our approach, similar to proxy authentication in Oracle Database [19].

4. IMPLEMENTATION

We created a prototypical implementation of our presented approach to native multi-tenancy support in a RDBMS. We used PostgreSQL 8.4 as starting point because it is available as open source, it features a clean design and its source code is well-documented.

4.1 Architectural Overview

To illustrate our implementation, we first outline the architecture of PostgreSQL. PostgreSQL follows a client/server-model. A multiplexing process, called *PostMaster*, waits for incoming connections. For each incoming connection, it starts a new process, called *Backend*. The Backend executes submitted queries and returns the results.

To implement our approach, we only had to enhance the Backend as the query processing engine. Figure 3 depicts the core components of the Backend and our respective enhancements.

The Parser builds the syntax tree of a given statement. We enhanced the parser to support statements to manage schema hierarchies and tenants.

The Planner uses a cost-based approach to construct the presumably cheapest plan for executing the statement. We did not enhance the planner.

The Executor executes the constructed plan and builds the reply. We enhanced it to process only data of the tenant that issued the statement.

The Data Dictionary stores the information about the data. We enhanced it to maintain schema hierarchies and tenants.

The Heap provides arrays on disk to store tuples. As we intermingle tuples of different tenants in one array, we add a system attribute to a tuple that stores to which tenant it relates.

The Index enables the efficient search of tuples. To exploit this, we add the tenant key of a tuple to its index key.

The main architecture of current RDBMS implementations is quite similar. The implementation of our concept mainly affects the parser, the data dictionary and the executor which exist in most implementations. Thus, although our implementation considerations are specific to PostgreSQL, the main ideas are transferable to other systems as well.

The remainder of this section describes the main implementation considerations and decisions with which we dealt while applying the described enhancements.

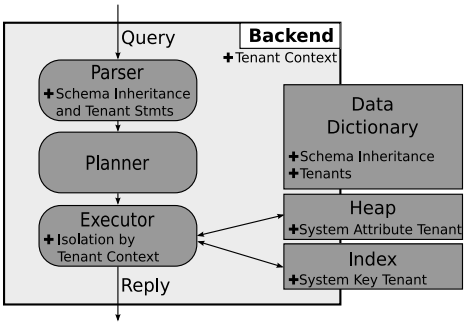


Figure 3: Architectural overview of a PostgreSQL Backend including our respective enhancements.

4.2 Data Dictionary

We modified the data dictionary so that it maintains information about tenants and schema hierarchies. We present our logical model as well as our main memory representation of the data dictionary. Thereafter, we discuss the maintenance and the access of the data dictionary.

4.2.1 Data Model

Figure 4 depicts a simple logical model of relevant data dictionary tables. Note that, for the sake of readability, we use simpler names than the actual names in the implementation, e. g. *Attribute* instead of *pg_attribute*. We have added the tables *Tenant* and *TableInstance* to the original data dictionary tables of PostgreSQL. The other tables represent original data dictionary tables that we have modified. For this discussion, we refer to tables of the data dictionary as *system tables* in order to distinguish them from application-specific tables. We exemplify our data model through a data dictionary excerpt of our e-commerce scenario that Fig. 5 shows.

The system table *Tenant* stores available tenants. They are identified by unique names and system-generated integer identifiers. According to our e-commerce scenario, Fig. 5 shows two entries for the tenants *Gonzo Books* and *Kermit Shoes*. For the sake of readability, we use names to indicate references to tuples. Actually, the system uses integer identifiers to store references.

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

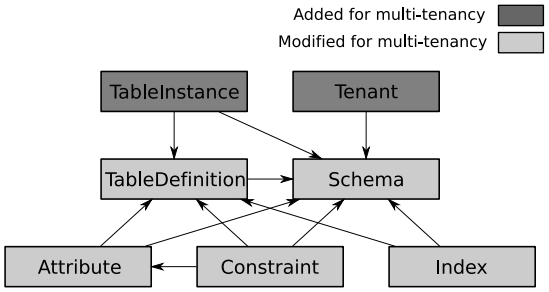


Figure 4: Data dictionary model.

Tenant

Name
Gonzo Books
Kermit Shoes

Schema

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

Attribute

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

TableDefinition

Schema	Name	NumRows
Shop	Item	342
Globals	Country	170

TableInstance

Schema	Name	NumRows
Gonzo Books	Item	42
Kermit Shoes	Item	300

Figure 5: Example of our data dictionary for the introduced e-commerce scenario.

objects visible in the schema (see Sec. 4.2.3). Note that shared schemas always have a path length of one because they cannot inherit from other schemas. In accordance with the tenants of our example, Fig. 5 shows two entries for the tenant schemas *Gonzo Books* and *Kermit Shoes*. The stored root path reflects that both schemas inherit from the virtual schema *Shop* which has also a related entry in *Schema*. Moreover, *Schema* has an entry for the shared schema *Globals*.

To know which database objects relate to which schema, affected data dictionary tables establish a schema member relationship. For example, the system table *Attribute* has an attribute *Schema* that references the schema to which an attribute relates.

The system tables *TableDefinition* and *TableInstance* provide the basis to represent the extension of tables according to our schema inheritance concept. *TableDefinition* stores table definitions and *TableInstance* stores table instances. A table definition defines a table in terms of attributes, constraints and indexes, whereas a table instance represents a concrete occurrence of a table definition to store data. A common identifier ties a table definition and related table instances together. As each virtual schema or tenant schema may extend the lists of attributes, constraints and indexes of an inherited table definition, the system assembles the complete table definition by traversing the related schema inheritance path. In our example, the virtual schema *Shop* defines a table *Item* that has two attributes: *Name* and *Price*. The schemas *Gonzo Books* and *Kermit Shoes* respectively extend *Item* by additional attributes. Thus, *Attribute* contains corresponding entries: one entry for the attribute *Color* of *Kermit Shoes* and two entries for the attributes *Pages* and *ISBN* of *Gonzo Books*. Furthermore, each tenant schema contains an instance of the table *Item* which is reflected in *TableInstance*. Our current implementation only supports defining tenant schemas that inherit from a virtual schema.

The structure of *TableDefinition* and *TableInstance* is similar. They have the same attributes except some static attributes. Static attributes store information that is equal across all instances, e. g. the name. An instance can overwrite the common attributes. That is, if the value of an instance attribute is not NULL, this value is used. Otherwise, if the value is NULL, the system consults the table definition to obtain the required value.

According to our concept, a table definition and an instance together are required to store data. Yet, we support a special case that implicitly considers a table definition instantiated, i. e. *TableInstance* is without a related entry. We assume this if the schema that contains the table definition is final. In this case, only one instance of the table definition can exist. Thus, we can store instance-specific information directly in the table definition. From a conceptual view, this slightly violates our approach, but it is in line with the original way PostgreSQL handles table definitions and instances and, thus, helps to stay compatible to existing code. Moreover, this is more efficient because we can save a lookup in *TableInstance* in this case.

4.2.2 Main Memory Structure

The data dictionary is heavily accessed during query processing. Therefore, the database management system transfers the external format of the data dictionary into a quickly accessible network of main memory objects. It is important to keep this network small so that the system can easily keep the data dictionary in main memory as much as possible. Therefore, our main memory layout avoids redundancy by sharing common parts between tenants.

Figure 6 depicts a simplified example of the memory structures that hold the description for the table

The large boxes hold general information about a table. We call them *table descriptors*. The descriptor *Shop.Item* holds the information of the table definition that is invariant between tenants. Contrarily, the descriptors *Gonzo Books.Item* and *Kermit Shoes.Item* hold information that is specific to the respective tenant schema. They reference the descriptor *Shop.Item* to access the common definition parts easily. Each table descriptor references another structure that holds information about related attributes. We call them *attribute descriptor lists* and refer to them by the label of the referencing table descriptor. The lower part of Fig. 6 shows the related attribute descriptor lists. The attribute descriptor list *Shop.Item* references two attribute descriptors that describe the structure of the two attributes *Name* and *Price*. The attribute descriptor list *Gonzo Books.Item* additionally references an attribute descriptor for the attribute *Pages* and *ISBN*. Analogously, the attribute descriptor list *Kermit Shoes.Item* references an attribute descriptor for the attribute *Color*.

To search objects of the data dictionary, PostgreSQL adopts corresponding hash tables. We adapted the existing hash tables to account for schema identifiers. For example, a hash table to look up attributes by name features a hash key that consists of the schema identifier, the table identifier and the name of the attribute. In addition, we created additional hash tables to search tenants and table instances efficiently. *Item* of our e-commerce scenario.

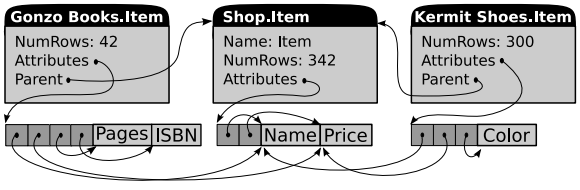


Figure 6: Example for the main memory layout of table and attribute descriptors.

4.2.3 Maintaining and Accessing the Data Dictionary

To maintain the presented data dictionary, we had to modify existing data definition operations so that they consider the schema hierarchy and the context they are executed within. For example, the statement *ALTER TABLE ADD ATTRIBUTE* must ensure that the name of the new attribute is unique across the inheritance path.

Accessing a table entails constructing a table descriptor in main memory. If the accessed table is inherited from a virtual schema, the system assembles relevant parts by traversing the related schema inheritance path. Listing 1 shows the construction of a table descriptor in pseudo code. The algorithm recursively steps through the schema inheritance path until it finds a descriptor of the desired table. After finding a descriptor, it goes back through the schema inheritance path and creates an appropriate chain of descriptors, as explained in Sec. 4.2.2. The shown algorithm may require more data dictionary lookups than the original algorithm. Yet, the penalty of the shown algorithm is moderate if the data dictionary is in main memory. Our system addresses OLTP-like workloads which require that the data dictionary is in main memory to obtain high transaction rates and low latencies. This is because a query in an OLTP workload is short running and typically accesses a small amount of data. Thus, accessing the data dictionary must be very fast to keep the relative overhead introduced by accessing it small. To achieve this, data dictionary accesses should prevent disk accesses, i. e. the data dictionary mostly resides in main memory.

4.2.4 Synchronizing the Data Dictionary

To ensure that tables are not dropped or modified in incompatible ways while executing a statement that references them, each statement acquires appropriate table-level locks. Our schema inheritance concept requires a locking approach that accounts for the relationship of tables according to the given schema inheritance. Modifying a table definition in a virtual schema requires to lock dependent table definitions in derived schemas. The same goes the other way, if a table definition is modified, the related table definitions on the root path of the schema must be locked. As modifying a tenant's schema should not affect other tenants, locking a table does not entail locking tables that have the same parent definition.

To accomplish the described behavior, we extended the original lock manager to support a multi-granularity scheme for table-level locks [12]. At this, the granularity of the table-level locks follows the inclusion relation of tables that is imposed by the inclusion relation of the particular schema hierarchy. If a statement wants to acquire a table-level lock, the lock manager records an absolute lock on the corresponding table and intentional locks on the related ancestors. The statement only obtains the requested lock if it does not conflict with existing locks on the table or on ancestor tables. Hence, a lock of a table in a virtual schema implicitly locks all descendents. In general, our use case entails a flat schema hierarchy. Thus, locking a table requires recording only few additional locks.

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

```
function GetTableDescriptor(tableId, schemaPath)
begin
 tabDescr = QueryTabDescrCache(tableId, schemaPath.first)
 if tabDescr not found then
  tabInfo  = BuildTabDescrFromDisk(tableId,
                                   schemaPath.first)
  if tabInfo not found and schemaPath.next exists then
   parent = GetTableDescriptor(tableId, schemaPath.next)
   tabInfo = CreateTabDescrForThisSchema(parent,
                                         schemaPath.first)
   PutTableDescrInCache(tabDescr)
   return tabDescr
  end if
 else
  return tabDescr
 end if
 return not found
end GetTableDescriptor
```

Listing 1: Table Descriptor Construction Algorithm

4.3 Storage Model

The storage model of the Universal Table approach has been characterized as well-suited for serving a large number of small tenants with regard to performance. As our work so far focused on the tenant-aware meta data management, we simply adopted this storage model and integrated it into our approach. To completely describe our implementation, we briefly present the integration of this storage model. Note that our so far presented approach does not rely on a specific storage model.

Our physical storage model to materialize tables depends on the type of the associated schema. Tables in shared schemas do not require to distinguish between tenants. Thus, the original storage model of PostgreSQL is adequate. Contrarily, the storage model for tables that are defined in a virtual schema and extended and instantiated in tenant schemas builds on the model of the Universal Table approach. That is, we store the tuples of different tenants intermingled in a single heap array. Each tuple stores the tenant to which it relates as system attribute in the tuple header. We refer to this attribute as *tenant attribute*. In contrast to the Universal Table approach, a clean separation of system-relevant data and user data exists. Moreover, this enables minor optimizations while accessing a tuple.

All insertion operators set the tenant attribute to the tenant of the tenant context in which they are executed. Furthermore, the operators that carry out heap scans only return tuples that relate to the tenant of the current tenant context. Nevertheless, maintenance operations that are triggered by the administrators may scan all tuples, e. g. to build an index or cluster a heap file. The system interprets the tuple according to the schema of the related tenant. This avoids storing NULL values to enable attribute extensions as opposed to the Universal Table approach.

As discussed, an index definition in a virtual schema is effective for all derived schemas. Thus, we index tuples of all tenants whose schema is derived from the virtual schema of the index definition by a single physical index. To distinguish between different tenants, the index automatically includes the tenant attribute in the index key definition. Thereby, the tenant attribute precedes the original index keys. Prepending the tenant attribute partitions the index by tenants and effectively leads to a Partitioned B-tree [11]. This is efficient for index scans because the index tuples of a tenant are stored consecutively. The insert and scan operations on the index take into account the tenant of the context in which they are executed, just as the analogous operations on the heap.

Our approach allows to define new tables and indexes in a tenant schema. Currently, we assign dedicated files to such tables and indexes. This approach may degrade the buffer pool hit ratio, similar to the Shared Process approach. Thus, in future work we will develop a storage model tailored towards multi-tenancy.

5. EVALUATION

Our approach mainly targets tenant-aware meta data management. Thus, our main enhancements affect the data dictionary. To evaluate data dictionary performance of our approach compared to other approaches, we embark upon an experiment to measure memory consumption and lookup time. At that, we draw a comparison between the Shared Process, the Shared Table and our approach. Using the results of this experiment, we discuss the usability of our approach as well as future work.

5.1 Experimental Results

With respect to performance, our approach targets saving main memory to increase scalability and maximize space available for query processing. We save main memory by sharing the application core schema among tenants. To retrieve insight about the effectiveness of this idea, we developed a test that extends an application core schema by a given number of custom attributes per tenant. Thereafter, the test issues queries for each table and each tenant and measures lookup time as well as main memory consumption of the data dictionary. PostgreSQL does not provide a mechanism to retrieve data dictionary lookup times. Therefore, we measured the total execution time of a query. To minimize effects that are not related to the data dictionary, our test prepends the `EXPLAIN` keyword to each query. This causes PostgreSQL to executes all query processing steps except for the execution of the generated plan. For simple queries, accessing the data dictionary takes considerable time during parsing and planning. Thus, the time for executing a query with `EXPLAIN` gives a rough estimation of lookup times for the data dictionary. In the remainder, we refer to a query that includes the `EXPLAIN` keyword as *explain query*.

5.1.1 Data Dictionary Test Environment

As application core schema, we used the TPC-C schema [22] depicted in Fig. 7. The schema comprises 9 tables, 12 indexes and 86 attributes. For the Shared Process approach, our test creates for each tenant a dedicated schema instance by mapping it onto a SQL schema whose name includes the related tenant, e. g. `shop_tenant23`. Thereafter, the test defines custom attributes in each schema. The Shared Table approach requires only one schema. Our test defines this schema with custom attributes and adds an attribute to store the tenant. For our approach, the test creates the schema depicted in Fig. 7 as virtual schema *Shop*. Next, it creates the given number of tenants and, thus, tenant schemas. A tenant's schema inherits from *Shop*. Finally, the test defines custom attributes in each tenant schema. For each approach, the test distributes the number of custom attributes per tenant evenly over the tables *Item*, *Customer*, *District*, and *Warehouse*. For example, 32 custom attributes per tenant entails 8 custom attributes per table. The custom attributes are of type `varchar`.

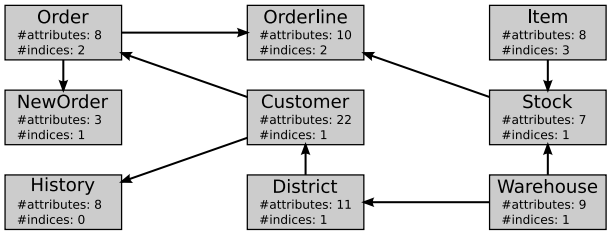


Figure 7: Schema that our test uses as application core schema. The test extends this schema per tenant by a given number of custom attributes.

After generating the schemas, our test creates a random sequence of explain queries that we call *explain sequence*. The sequence contains exactly one explain query for each table and each tenant. An explain query projects all attributes of the related table. Depending on the approach, the explain queries slightly differ. Explain queries for the Shared Process approach qualify a tenant's tables by placing the name of the associated schema in front of table names. Explain queries for the Shared Table approach obtain a selection predicate according to the related tenant. For our approach, the test places a suitable `SET TENANT` statement in front of the explain query in order to set the related tenant.

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

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

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

5.1.2 Measurements

Table 1 lists the main memory consumption of our test cases. We ran each test case three times. For each test case, we report the highest measured memory consumption of the three runs after executing the generated explain sequence the second time. The measured values after the first execution of the query sequence are nearly identical, though. That is because the required meta data was completely loaded during the first execution. Thus, the second execution just queries the data dictionary cache, but it does not build additional structures.

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

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

Table 1: PostgreSQL Backend's main memory consumption in MB with different numbers of custom attributes per tenant and different tenant cardinalities.

Considering the absolute memory consumption, our approach requires considerably less memory than the Shared Process approach. For 10000 tenants, the difference approximately amounts to 1900 MB, independent of the number of custom attributes. This result approves the effectiveness of sharing the application core schema.

Our approach consumes little main memory for 10000 tenants without customization. Yet, as already pointed out, our approach naturally consumes considerable more main memory by higher degrees of customization per tenant. Consequently, the scalability of our approach mainly depends on the customization requirements. Note that this does not apply to the Shared Process approach, as an additional tenant requires a new instance of the application core schema. Thus, each new tenant takes a serious amount of main memory. Regarding the high main memory consumption of the Shared Process approach for a large number of tenants, our measurements confirm results of previous research [16].

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

Our execution time measurements demonstrate how larger main memory structures of the data dictionary degrade its lookup time. To exemplify the characteristics of the different approaches, we report the measured times for 10000 tenants and 64 custom attributes in Fig. 8. We only report one test case as others have shown identical characteristics. To smoothen the graph and reduce the number of points for readability, we accumulated the measured times for every 1000 explain queries. Note that we ran each test case three times. The results of the runs where consistent for which we only report the values of the first run.

The Shared Table approach provides constant lookup times over the whole run. Only at the beginning, the lookup time is higher as the system table pages are read from disk. The same behavior applies to the Shared Process and our approach (referred to as Native MT Support in the figure). Yet, it takes longer until the buffer pool can satisfy all page requests because the data dictionary is larger and, thus, occupies more pages.

The graph clearly indicates the finish of the first execution and the start of the second execution of the explain sequence (90000). As the second execution uses a warm data dictionary cache, all approaches provide constant times. The lookup times of the Shared

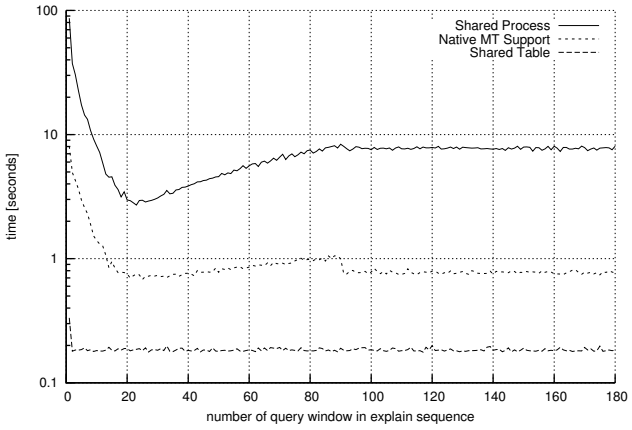


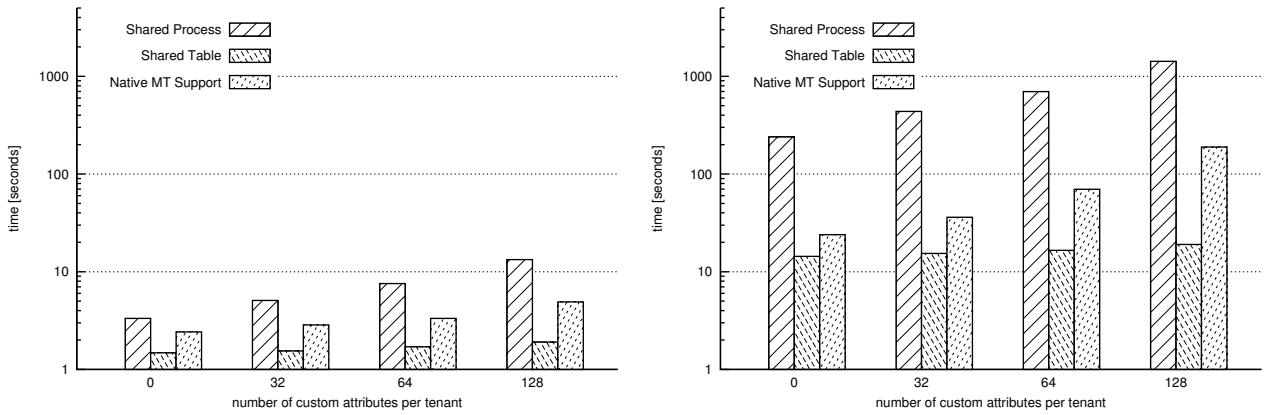
Figure 8: Data dictionary lookup times for 10000 tenants and 64 custom attributes per tenants. We ran the explain sequence in query windows of 1000 explain queries. We refer to our approach as *Native MT Support*.

Process approach increases until the first execution of the generated query sequence finishes. The increasing lookup times have their seeds in the considerable amount of used main memory and more collisions in the hash tables of the data dictionary cache. Our approach degrades for the same reasons during the first execution of the sequence, but lesser. The saltus in lookup times of our approach, after the first execution of the query sequence, hints at the efforts of recursively stepping through the schema inheritance hierarchy. This effort falls away for the second execution of the explain sequence.

Finally, Fig. 9 reports the accumulated lookup times for the second execution of the explain sequence, i. e. only explain queries against a warm data dictionary cache. Note that 9a) demonstrates accumulated times of 9000 explain queries, whereas 9b) demonstrates the accumulated times of 90000 explain queries. The times of the Shared Process approach increase quite linearly with the number of tenants. Contrarily, our approach provides almost identical times for 1000 and 10000 tenants without custom attributes. The times of the Shared Process and of our approach increase more than linearly with the number of custom attributes. As our approach maintains a smaller data dictionary, it does not degrade as much as in the Shared Process approach. Hence, for 10000 tenants and 64 custom attributes per tenant, our approach takes about 0,7 milliseconds per explain query as opposed to the Shared Process approach that takes about 7 milliseconds per explain query.

5.2 Discussion and Future Work

The Shared Table approach outperforms our approach in the presented measurements. However, regarding functionality, the naive comparison of the Shared Table approach to our approach compares apples and oranges. As opposed to our approach, the Shared Table approach requires to build tenant-aware meta data management functionality in the application on top of the RDBMS. This causes extra runtime overhead, which needs to be added to the |performance figures of the Shared Table approach for a fair comparison. Furthermore, this means that each and every application that adopts the Shared Table approach has to implement its own solution, which is complex, error-prone and expensive. This is naturally worse than implementing and testing only one solution, which is feasible as tenant-aware meta data management is typically not



(a) 1000 Tenants (b) 10000 Tenants

Figure 9: Accumulated data dictionary lookup times of querying each table (see Fig. 7) for each tenant on a warm data dictionary cache. We refer to our approach as *Native MT Support*.

specific to a single application. It might be objected that appropriate middleware solutions may compensate for this issue. Yet, building a tenant-aware meta data management solution on top of the RDBMS requires re-implementing competencies a RDBMS already offers. Solutions on top of the RDBMS must implement a data dictionary similar to the data dictionary of the RDBMS, although multi-tenancy does not change the data model, i. e. the relational model remains. Moreover, they must parse, interpret and rewrite queries; RDBMSs are specialized at this and perform these steps anyway. Thus, solutions on top of the RDBMS cause doubled efforts for implementation as well as for runtime. Furthermore, if the RDBMS loses control and responsibility for the information about data, it also loses some optimization opportunities, e. g. specialized index structures for certain data types. Finally, a solution on top of the RDBMS entails another piece of software or system which has to be maintained. On the basis of the points mentioned, we argue that native support of multi-tenancy in RDBMSs for SaaS is mandatory. This is also in line with Separation of Concerns.

As opposed to the Shared Table approach, our approach provides native features to tenant-aware data management in the RDBMS. So far, we focused on tenant-aware meta data management to ease the development of multi-tenant SaaS applications that rely on a RDBMS. Of course, further tenant-aware data management features are required, e. g. backup and recovery per tenant or migration of a tenant to another system. Nevertheless, we are confident that our approach builds a perfect infrastructure to create such operations as it provides the concept of a tenant context and introduces tenants as first-class database objects.

To improve scalability while providing tenant-aware meta data management in the RDBMS, our approach exploits characteristics specific to multi-tenant SaaS applications that rely on a RDBMS. Our idea is to share the application core schema among tenants. Our measurements show that adopting this idea may considerably decrease main memory consumption and lookup times of the data dictionary compared to a totally dedicated schema per tenant, as in the Shared Process approach. Of course, lookup times of the data dictionary only form a small fraction of total query time. For instance, taking the 7 milliseconds for 10000 tenants and 64 custom attributes of the Shared Table approach and assuming an average runtime of 280 milliseconds for a simple query, the data dictionary lookup times only form 2.5 %. Thus, we do not consider our results of data dictionary lookup times significant to the overall performance of query processing. However, the experiments show that our approach does not significantly degrade dictionary lookup times, although meta data management in our approach is more complex due to its schema inheritance hierarchy. The actual performance benefit of our approach is the moderate main memory consumption of the data dictionary while providing tenant-aware meta data management. This is an important point, as the data dictionary may seriously degrade main memory consumption, thus limiting available space for the buffer pool, which ultimately impacts overall query performance. For instance, assume that a machine with 32 GB of main memory runs our test case for 10000 tenants and 128 custom attributes. In this case, the Shared Process approach would leave 29 GB for query processing, whereas our approach would leave 31 GB, which is about 7 % more.

Note that the discussed issues also apply to pre-compiled queries. Pre-compiled queries avoid the overhead of parsing and planning during runtime and, thus, decrease accesses to the data dictionary. However, to execute pre-compiled queries, the RDBMS still requires to access meta data in order to interpret accessed data. Storing all required meta data in the plans of the pre-compiled queries yields similar scalability issues as in the Shared Process approach. Thus, pre-compiled queries also benefit from our approach, if they use the data dictionary cache during execution.

Besides meta data management, the storage model is crucial in RDBMS. For the present, we integrated a storage model that bases upon the Universal Table approach. This decision was driven by the proven and good performance of the Universal Table approach. As a benefit of our tenant-aware data dictionary, our approach does not require storing NULL values to enable per-tenant schema extension, as opposed to the Universal Table approach. Our own preliminary performance measurements show that this fact causes a performance improvement if the ratio between the maximum number and average number of custom attributes is high. In this case, the benefit of avoiding to store NULL values amortizes the penalty due to the higher memory consumption of our approach.

Despite the proven and good performance of the described storage model, it is unclear whether it really meets the requirements to build an efficient tenant-aware data management. Considerations so far concentrate on the performance of query processing. Yet, tenant-aware data management also requires appropriate tenant-aware administrative operations such as backup and recovery of one tenant, migration or replication of one tenant to another ma-

chine, removal of one tenant and so forth. The described storage model implies that these operations have to deal with single tuples. This is an apparent issue that requires further analysis.

On the basis of the points mentioned, we see two main directions to a completely tenant-aware RDBMS:

1. Creating tenant-aware administrative operations.

2. Developing a storage model that is tailored to the requirements of tenant-aware data management.

As our concrete next steps, we plan to identify relevant tenant-aware administrative operations. Thereafter, we plan to evaluate the identified operations on different storage models.

6. CONCLUSION

We motivated the requirement for appropriate functionality in RDBMSs to support tenant-aware data management natively and presented first features to fulfill this requirement. At this, we focused on tenant-aware meta data management.

We introduced the concept of a tenant context that logically assembles all information that describes the tenant's view of the database and cleanly isolates tenants among another. Furthermore, we presented a schema inheritance concept tailored to multi-tenancy. The concept offers different schema types for different challenges. Virtual schemas intend to describe application core schemas. Virtual schemas that inherit from other virtual schemas enable to specialize the application core schema for specific domains. Tenant schemas that inherit from virtual schemas enable schema isolation and per-tenant schema extension.

Our approach eases the development of multi-tenant SaaS applications that rely on a RDBMS, as it prevents the implementation of a complex query rewriter on top of the RDBMS. Moreover, it facilitates the central maintenance of the application core schema and the individual maintenance of tenants' schemas. As the RDBMS knows and controls the application core schemas as well as the tenants' schemas, the RDBMS can optimize data access and data processing. Moreover, our concept avoids redundancy by sharing the application core schema among tenants. Our measurements show that sharing the application core schema considerably decreases main memory consumption and lookup times of the data dictionary compared to totally dedicated schemas per tenant.

A lot of further work exists to obtain a RDBMS that facilitates complete and efficient support for multi-tenancy. Our future work concentrates on better understanding the tradeoffs between normal query processing and administrative operations of different storage models for multi-tenancy.

Acknowledgements

We would like to thank Nazario Cipriani and Alexander Moosbrugger for improving our work by their useful advice.

7. REFERENCES

[1] Amazon. Amazon SimpleDB. `http://aws.amazon.com/simpledb/`, 2010.

[2] C. Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, 2006.

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

[4] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold. A Comparison of Flexible Schemas for Software as a Service. In *Proc. of SIGMOD Conf.*, pages 881–888, 2009.

[5] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs to Support Sparse Datasets Using an Interpreted Attribute Storage Format. In *Proc. of ICDE*, pages 1–58, 2006.

[6] M. T. Carl Osipov, Germán Goldszmidt and I. Poddar. Develop and Deploy Multi-Tenant Web-delivered Solutions using IBM middleware: Part 2: Approaches for enabling multi-tenancy. IBM Corp. Website, 2009.

[7] F. Chong and G. Carraro. Architecture Strategies for Catching the Long Tail. Microsoft Corp. Website, 2006.

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

[9] G. C. Frederick Chong and R. Wolter. Multi-Tenant Data Architecture. Microsoft Corp. Website, 2006.

[10] GoECart. GoECart. http://www.goecart.com, 2010.

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

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

[13] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. A Framework for Native Multi-Tenancy Application Development and Management. In *Proc. of CEC/EEE*, pages 551–558, 2007.

[14] M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting Database Applications as a Service. In *Proc. of ICDE*, pages 832–843, 2009.

[15] IBM. DB2 9.7 LUW Infocenter – SQL and XML Limits, 2010.

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

[17] R. P. Mahowald. Worldwide Software as a Service 2010-2014 Forecast: Software Will Never Be the Same. In IDC Report, Number 223628, 2010.

[18] T. McKinnon. Plug Your Code in Here - An Internet Application Platform. `http://www.hpts.ws/papers/2007/hpts_conference_oct_2007.ppt`, 2007.

[19] Oracle. Oracle Database JDBC Developer's Guide and Reference 11g (11.1) – Proxy Authentication.

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

[21] Salesforce.com. Salesforce. http://www.salesforce.com, 2010.

[22] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification, Revision 5.10.1, 2009.

[23] Z. H. Wang, C. J. Guo, B. Gao, W. Sun, Z. Zhang, and W. H. An. A Study and Performance Evaluation of the Multi-Tenant Data Tier Design Patterns for Service Oriented Computing. In *Proc. of ICEBE*, pages 94–101, 2008.

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

[25] Zimbra. Zimbra. http://www.zimbra.com, 2010.