Autom Softw Eng
DOI 10.1007/s10515-015-0178-2

Automatic query rewriting schemes for multitenant SaaS applications

Chun-Feng Liao · Kung Chen · Deik Hoong Tan · Jiu-Jye Chen

Received: 7 October 2013 / Accepted: 1 March 2015

Abstract In Software as a Service (SaaS) environments, designing and realizing multitenant schema-mapping that supports a shared database with custom extensions is a non-trivial task. Universal Table is one promising schema-mapping technique that is commonly used. However, there has been little research devoted to the design and realization of a query rewriting scheme for Universal Table. In this paper, we present a collection of general query rewriting schemes for Universal Table that can transparently transform tenant-specific logical queries into corresponding physical queries. Based on the design, we have developed a Java-based schema-mapping and query rewriting middleware for Universal Table and a sample online shopping SaaS application to verify its feasibility. Additionally, analytical results that can be used to predict the overhead of our schemes are also reported. Finally, we conduct a series of experiments and find that the results not only agree well with our analytical predictions but

C. -F. Liao (✉)
CS Department and Program in Digital Content and Technologies, National Chengchi University,
No. 64, Sec. 2, Zhi-Nan Rd., Wenshan District, Taipei 116, Taiwan
e-mail: cfliao@nccu.edu.tw

K. Chen · J. -J. Chen
CS Department, National Chengchi University,
No. 64, Sec. 2, Zhi-Nan Rd., Wenshan District, Taipei 116, Taiwan
e-mail: chenk@nccu.edu.tw

J. -J. Chen
e-mail: 100971009@nccu.edu.tw

D. H. Tan
IECS Department, Feng Chia University, No. 100, Wenhwa Rd., Seatwen District,
Taichung 407, Taiwan
e-mail: deikhoong@gmail.com

also show that our schemes are scalable to the number of tenants and the number of concurrent database connections.

Keywords Schema-mapping · SaaS · Query rewriting · Multitenant

1 Introduction

Software as a service (SaaS) is an emerging service model of cloud computing. Its main characteristic is the ability for customers to use a software application on a pay-as-you-go subscription basis. To be competitive, SaaS application providers typically offer a price that is much lower than running the application in an unshared software instance on dedicated hardware. As a result, SaaS applications must leverage resource sharing to a greater extent to reduce prime costs by accommodating different users of the application while making it appear to tenants that they have the application all to themselves. In this way, SaaS providers are able to achieve economic scalability and can therefore offer services at a much lower price than traditional vendors. This makes the service more affordable for a greater number of small to medium-sized companies, which gives SaaS providers exclusive access to an entirely new market. Chong and Carraro (2006) called this market the "long tail". Behind the scenes, the core technology which enables the sharing of a SaaS application instance is called multitenancy.

Several attempts have been made to identify the multitenancy concerns of SaaS applications, such as affnity (how tasks are transparently distributed), persistence, performance isolation, QoS differentiation, and customization (Krebs et al. 2012). Koziolek (2011) obtained similar results based on the software architecture point of view. Among various multitenancy design concerns, it is generally agreed that multitenant data architecture is one of the most important concerns when creating a SaaS application since data are the most important asset of any enterprise (Chong et al. 2006). However, there is little investigation on modularizing data layer concerns in a multitenant application.

In the design space of the data layout and management strategy for multi-tenant applications, various alternative approaches form a continuum between an isolated data style and a shared data style (Chong et al. 2006). Here we focus on one end of the continuum, namely shared data. Common practice is to pool all of the tenants' data together in the same set of tables in a shared database, and to equip all tables with a tenant ID column that associates every data record with the appropriate tenant. However, as pointed out by Aulbach et al. (2008), this kind of shared architecture, while providing very good consolidation, lacks the schema extensibility that is so essential for many SaaS applications. Indeed, it is highly desirable to have a multitenant data architecture for SaaS applications that can support a shared database with custom extensions. A systematic approach for addressing this requirement is to employ some schema mapping techniques that support the separation of logical schemas, the ones used in the application, from physical schemas, the ones implemented in the database (Aulbach et al. 2008).

According to the evaluations conducted by Li et al. (2012), when compared to alternative schema-mapping techniques, Universal Table is the most promising solution since it is very flexible and, if carefully designed, can achieve a reasonable level of performance. Originating from Universal Relation (Maier and Ullman 1983), Universal Table schema-mapping consists of a single large, generally structured table, namely the universal table, and a set of metadata tables that keeps track of meta-information storage in the universal table. In this way, different tenants can store their data in the universal table in different ways. Conceptually, the Universal Table approach "virtualizes" the logical tables so that the data can be more physically consolidated and consistently managed. The benefits are very similar to those of consolidating virtual machines in a single physical machine in an IaaS environment. In addition to the cost of hardware, the virtualized and consolidated entities can be managed at a lower cost with higher flexibility. In fact, it is the approach adopted by Force.com, which is a successful SaaS vendor best known for its CRM service that supports more than 55,000 tenants. Hence we think it is worthy of further investigation. Weissman and Bobrowski (2009) present a high-level overview of the data architecture of the Force.com platform. However, it is not clear how the Force.com SaaS applications leased by tenants transparently transform logical schema query statements into physical schema ones. Moreover, this also makes it hard to evaluate the approaches used by Force.com, and thus equally difficult to investigate possible improvements.

Although it has been pointed out that a set of query rewriting schemes is essential for realizing a multitenant schema-mapping, little research has been done in designing and analyzing query rewriting issues (Pereira and Chiueh 2007; Aulbach et al. 2008). Therefore, the objective of this work is to fill in the gaps by systematically investigating the design of generic tenant-aware query rewriting schemes, analyzing performance overhead incurred by Universal Table schema-mapping, and conducting experiments against the proposed schemes. The results of this research will relieve SaaS application developers from the burden of rewriting SQL statements manually, which is tedious and error prone. These rewriting schemes are designed based on relational algebra so that it can be easily analyzed and formally verified. Besides, we have implemented a Java-based prototype based on DataNucleus and JDO (Russell 2010) as well as a simple SaaS application prototype, Shopping-Force.com, to demonstrate the feasibility of our approach. Finally, we also present analytical performance results of the rewriting schemes that can be used to predict the I/O overhead and the experimental results that verify the scalability with respect to the number of tenants as well as concurrency level.

2 Related work

Without loss of generality, we classify multitenant data architectures into one of four styles by the desired levels of customization and consolidation: (1) Shared Table, Private Schema, (2) Shared Table, Shared Schema, (3) Private Table, Shared Schema, and (4) Private Table, Private Schema (see Fig. 1). Shared Table refers to a system in which all tenants' data reside in shared storage (e.g., a group of tables in a relational database), whereas Shared Schema means that the tenants' schematic definitions of data are identical. Higher levels of customization allow a tenant to customize the original data schema provided by the SaaS vendor to more closely fit tenant-specific business needs. However, higher levels of customization also introduce more complexity to the design of the data architecture. Meanwhile, the total maintenance cost can be greatly

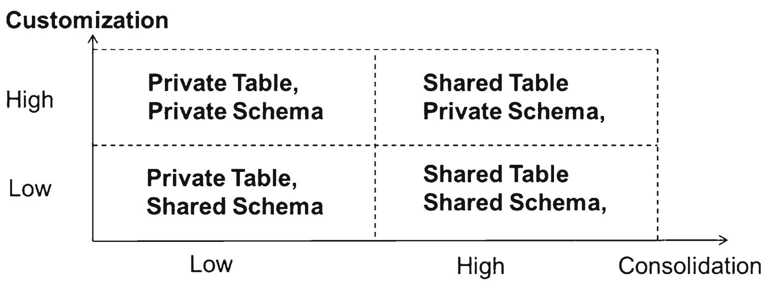

Customization
High
Low
Private Table, Private Schema
Shared Table Private Schema,
Private Table, Shared Schema
Shared Table Shared Schema,
Low
High
Consolidation

Fig. 1 A taxonomy of multitenant data architecture

reduced if a tenant surrenders some level of data isolation by consolidating data from different tenants.

Among these architectural styles, the “Shared Table, Private Schema” style, the focus of this work, is more attractive to small and medium-sized businesses since it is capable of achieving greater flexibility at a lower cost (Aulbach et al. 2008). Although Shared Table leads to lower data isolation, there are some data for most businesses that are not highly sensitive, such as those businesses publicly available information, and thus they can be put together to reduce costs. In other words, if a SaaS vendor is able to provide part of its data service in “Shared Table, Private Schema” style, then it is expected to increase revenue from a new market in the “long tail” (Chong and Carraro 2006).

Several approaches have been proposed to realize the “Shared Table, Private Schema” architectural style. Aulbach et al. (2008) conducted a comprehensive study on representative “Shared Table, Private Schema” schema-mapping techniques such as Extension Table, Universal Table, Pivot Table, Chunk Table, and Chunk Folding. Extension Table originates from the Decomposed Storage Model (Copeland and Khoshafian 1985), which is widely used in realizing object-relational mapping. By In Extension Table schema-mapping, common data are stored in a super table, whereas the data with different schema are placed in extension tables. The main weakness of Extension extension Table tables lies in the explosion explosive growth of in the number of tables: . although Although some columns are stored in parent tables, it still needs to create one an additional table is needed for each domain object that requires customization, causing the number of tables to grow in proportion to the number of tenants (Aulbach et al. 2008). A common way to alleviate this problem is to store the extended data in a general-purpose column using XML so that no additional table is needed (Du et al. 2010). However, it is reported that when the number of tenants increases, XML-based approaches lead to performance problems (Li et al. 2012). Recently, Yaish et al. (2011) have proposed a novel schema-mapping technique called EET (Elastic Extension Tables) which stores the meta meta-information of the extended data in a set of pre-defined tables. In this way, the tenant-specific extension tables are virtualized, and can be composed on the fly based on the meta tables on-the-y and thus they are “virtualized”. Nevertheless, it is hard to justify the performance of EET in a multitenant environment since neither theoretical nor experimental analysis nor experiments is has been provided.

The Pivot Table is a flexible schema-mapping technique in which the values of different data types are stored in different tables. The schema of a pivot table in a multitenant environment is typically: *(tenant_id, table_id, column_id, row_id, type)*, where the *type* column is used to store the business data and other columns are used to store the metadata used to index data values. As a result, a 4 out of 5, or 80 %, space overhead has been incurred. Hence, the Pivot Table architecture is space-ineffcient. Moreover, logical tables are recovered based on the query by dynamically joining these tables. As a result, the Pivot Table architecture is also time-ineffcient. A logical table with n data types require $n - 1$ joins for each query. Aulbach et al. (2008) propose two variations of the Pivot Table, Chunk Table and Chunk Folding, to improve time- and space-effciency. Based on the evaluations reported by Li et al. (2012), in a 100-tenant environment, Chunk Table and Chunk Folding are able to reduce response time by up to 2000 milliseconds in a query with 1000 entities.

Originating from Universal Relation (Maier and Ullman 1983), Universal Table is designed for generic data storage, usually consisting of a Global Unique Identifer (GUID), a tenant ID, and a fixed number of generic data columns (e.g., Force.com uses 500 generic data columns). The type of generic columns is usually defined as STRING or VARCHAR so that the values of these columns can be easily converted to their original type in the application. Based on the performance evaluation conducted by Li et al. (2012), the response time of Universal Table architecture is much shorter than Pivot Table, Chunk Table, Chunk Folding, and Extension Table architectures in all experiments. The schema-mapping technique used by Force.com (Weissman and Bobrowski 2009) falls into the category of Universal Table, which is the foundation of this research.

Judging from the above, most of the current works have been devoted to the design of new flexible multitenant schema-mapping techniques. There has been little research devoted to the design and analysis of query rewriting schemes (Pereira and Chiueh 2007; Aulbach et al. 2008). The purpose of this work is therefore to systematically explore the empirical design of generic tenant-aware query rewriting schemes and to analyze the performance overhead incurred by Universal Table schema-mapping. Some preliminary results have been reported in Liao et al. (2012). The major additions to the previous work include: (1) A rigorous reformulation of the theoretical statements and analysis. (2) Identification and correction of theoretical errors with the query rewriting schemes. (3) Several enhancements to the performance of the transformed physical statements. (4) An in-depth investigation of the analyzed results (see Sect.5.2.5). (5) Empirical experiments that verify performance in a real-world environment and consistency with analytical results (see Sect.5.3).

3 Data definition and storage model

This section describes the underlying data definition and storage model used in our work. As mentioned, the underlying data definition and storage model of our approach is designed based on Universal Table schema mapping and is similar to the one used by the Salesforce.com platform (Weissman and Bobrowski 2009). As indicated in Fig. 2, the *Data* table is the Universal Table that stores all tenants' data. To facilitate schema

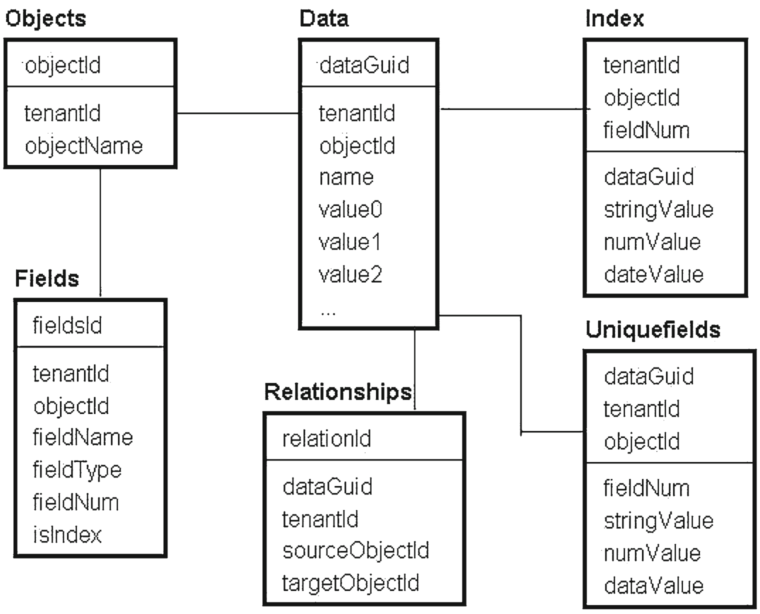


Fig. 2 Data definition and storage model used this work

mapping, we have additional tables such as *Object*, *Fields*, and *Relationships* to keep track of the meta-information of logical tables, logical columns, and relationships, respectively. This is a metadata-driven approach. Note that we also follow the convention used in Weissman and Bobrowski (2009) which uses the terms "objects" and "tables" as well as "fields" and "columns" interchangeably. For example, the *objectId* field, or column, refers to the unique number associated with a specific table, or object, in the logical schema, and a similar convention holds true for *eldId*. It is important to point out that schema mapping involves overhead of additional database I/O access since all of the meta-information for logical-physical mapping has to be stored in physical storage. As a result, additional index tables such as *Index* and *Uniqueelds* are provided to enhance query performance.

Figure 3 presents a multitenant architecture that is constructed based on the model depicted in Fig. 2. The overall architecture consists of four layers. The bottom layer is physical data and metadata used by all tenants based on Universal Table schema-mapping. Above this layer is the middleware that realizes schema-mapping and tenant-aware query rewriting. A multi-tenant SaaS application, which is able to host several tenant-specific virtual applications, can be built on top of the middleware layer.

As an example of how schema-mapping and query rewriting middleware works, consider a hypothetical SaaS application, ShoppingForce.com, that enables its tenants to sell products and process orders online. Since dierent tenants have their own unique needs in describing their products, ShoppingForce.com allows its tenants to create their own customized schemas. Figure 4 illustrates this scenario. Here we have two product tables(i.e. $Product_{tenant=667}$ and $Product_{tenant=604}$, where $tenant$ denotes the tenants identifier). The data in the two logical tables will be stored together in

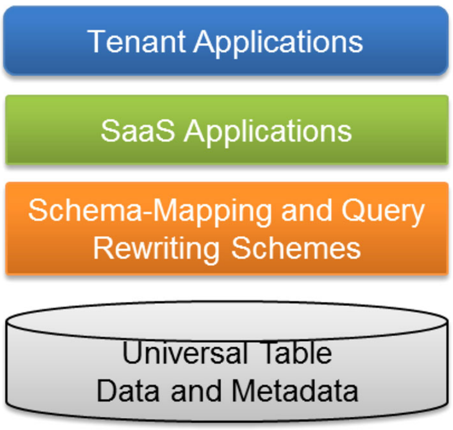
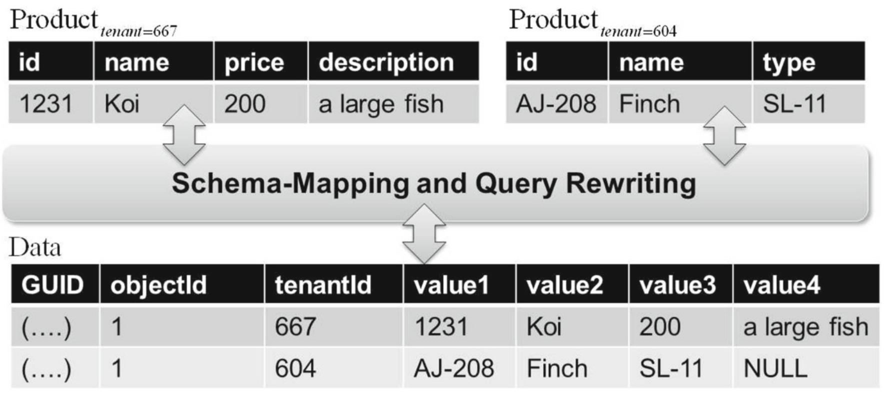

Tenant Applications
SaaS Applications
Schema-Mapping and Query Rewriting Schemes
Universal Table Data and Metadata

Fig. 3 Multitenant architecture based on the model depicted in Fig. 2



id	name	price	description
1231	Koi	200	a large fish

id	name	type
AJ-208	Finch	SL-11

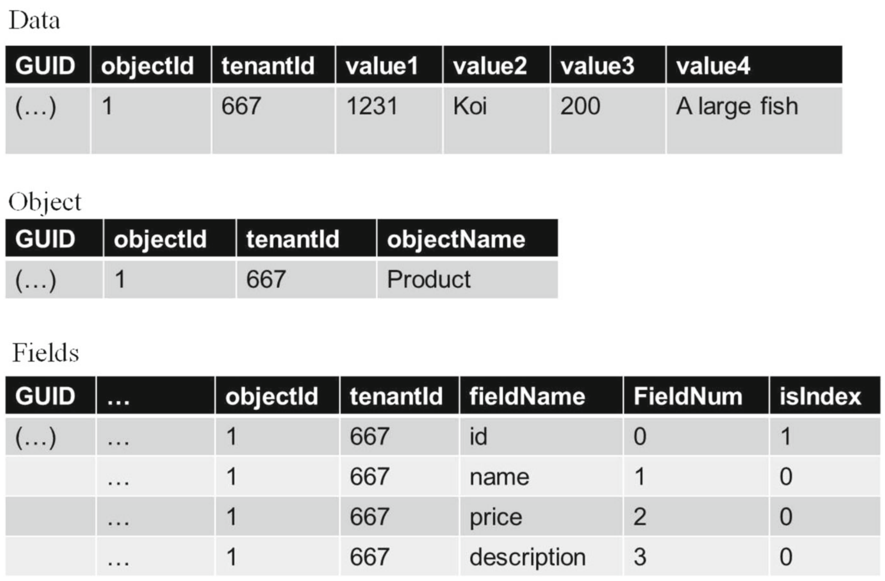
GUID	objectId	tenantId	value1	value2	value3	value4
(….)	1	667	1231	Koi	200	a large fish
(….)	1	604	AJ-208	Finch	SL-11	NULL

Fig. 4 The ShoppingForce.com example

the *Data* table via a schema mapping and query rewriting scheme. Figure 5 shows how an instance of a logical "Product" object belonging to tenant 667 and its related meta-information is stored physically using Universal Table schema mapping. Here the tenant identifier ("667") , object identifier ("1"), and the object name ("Product") are stored in the *Object* table whereas the field names of this object are stored in the *Field* table indexed by *objectId* and *tenantId* (i.e. *id*, *name*, *price*, and *description*). Thus, a full definition of the "Product" object can be obtained by joining the *Data*, *Object*, and *Fields* tables on *objectId* and *tenantId*.

4 Tenant-aware query rewriting

We are now ready to introduce designs for query rewriting schemes for Universal Table schema-mapping. Figure 6 depicts the overall query processing procedure and data flow of the proposed approach, in which the solid rectangular blocks refer to processing modules and the rounded rectangular blocks refer to input, output, or intermediate



Data

GUID	objectId	tenantId	value1	value2	value3	value4
(…)	1	667	1231	Koi	200	A large fish

Object

GUID	objectId	tenantId	objectName
(…)	1	667	Product

Fields

GUID	…	objectId	tenantId	fieldName	FieldNum	isIndex
(…)	…	1	667	id	0	1
	…	1	667	name	1	0
	…	1	667	price	2	0
	…	1	667	description	3	0

Fig. 5 Mapping an instance of "Product"

data. The inputs are *Tenant Profile* and the *Query for Logical Schema* (i.e., SQL statements) whereas the outputs are the rewritten query statements for the physical schema. Similar to typical DBMS (Database Management Systems), the overall query processing procedure used in this work is initiated by scanning, parsing, and validating the input query statements, and then breaking down the statements into basic units called query blocks (Elmasri and Navathe 2011). A query block consists of an atomic query statement such as projection, selection, or join. Next, a query tree, which is composed of query blocks, is built. The rewriting module then transforms the logical query statements hosted in the query block into new ones that fit the physical schema. After that, the query processing module generates a new query tree based on the generated statements. Finally, query optimizations can be performed before generating physical query statements.

Before presenting the details of the rewriting engine, it is helpful to explain a few notations, axioms and auxiliary functions that will be used repeatedly in further discussions. In this paper, we follow the conventions of Relational Algebra and use π, σ, $\bowtie$, ρ to denote projection, selection, join, and rename operations, respectively. Also, we use a "dot" notation to distinguish logical tables (objects) and columns (fields) from physical ones. More concretely, the "dot" in $\dot{X}$ means the entity X is "logical" and needs further transformations.

A projection, denoted $\pi_{<field\_names>}(object\_name)$, is an atomic query operation that (1) selects certain columns (fields), and (2) discards others from a set of tuples in a table (object). For example, the query "SELECT $orderDate$, $orderAmount$ FROM $Order$" denotes a projection $\pi_{orderDate,orderAmount}Order$. Likewise, a selection is an atomic query operation that selects rows from a set of tuples fulfilling the constraints specified by a boolean expression (or called condition), which can be specified

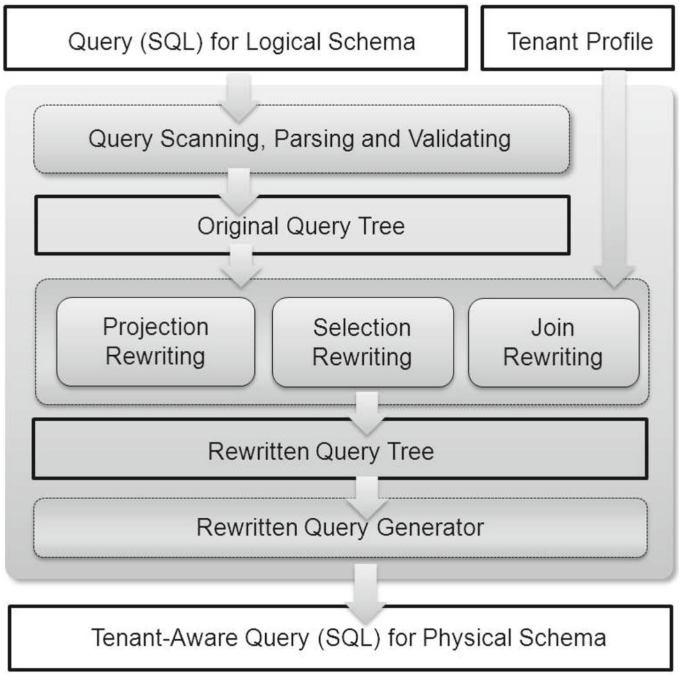

Query (SQL) for Logical Schema
Tenant Profile
Query Scanning, Parsing and Validating
Original Query Tree
Projection Rewriting
Selection Rewriting
Join Rewriting
Rewritten Query Tree
Rewritten Query Generator
Tenant-Aware Query (SQL) for Physical Schema

Fig. 6 Data flow of the proposed approach

by $\sigma_{<condition>}(object\_name)$. For example, the query "SELECT * FROM $Order$ WHERE $orderDate > 1998\text{-}7\text{-}31$" is a selection statement. Finally, a join operation, denoted by $O_1 \bowtie_{<condition>} O_2$, is used to combine sets of tuples coming from different tables (O_1 and O_2) and is then filtered by the given condition.

The following rules are presented in Elmasri and Navathe (2011) and will be used as axioms in the sequel without further proofs:

– *Conjunctive* σ *conditions* A selection (σ) with a set of conjunctive conditions can be rewritten to a sequence of simple selection operations:

$$\sigma_{c1 \wedge c2 \wedge \ldots c_n}(O) = \sigma_{c_1}\left(\sigma_{c_2}(\ldots(\sigma c_n(O))\ldots)\right), \tag{1}$$

where c_k are selection conditions and O is a table (object).

– *Commutativity of* σ Selection operations are commutative:

$$\sigma_{c_1}\left(\sigma_{c_2}(O)\right) = \sigma_{c_2}\left(\sigma_{c_1}(O)\right). \tag{2}$$

– *Reduction of* π All but the leftmost projection can be ignored:

$$\pi_{List_1}\left(\pi_{List_2}(\ldots(\pi_{List_n}(O))\ldots)\right) = \pi_{List_1}. \tag{3}$$

– *Commutativity between* σ *and* π Selection (σ) and Projection (π) are commutative if $A^{\pi} \subseteq A^{\sigma}$, where A^{π} and A^{σ} are the sets of columns involved in $List$ and C, respectively:

$$\pi_{List}\left(\sigma_C(O)\right) = \sigma_C\left(\pi_{List}(O)\right), \text{ where } A^{\pi} \subseteq A^{\sigma}. \tag{4}$$

Having introduced the basic transformation rules, we are now able to define and explain important concepts that are useful in deriving rewriting schemes.

Definition 1 (*Object Name Transformation*) The object name transformation function, ξ^{object}, that transforms a logical object name into a physical object name, is defined as follows:

$$\begin{aligned} &\xi^{object} : LObjName \times TenantId \rightarrow PObjName \\ &\xi^{object}(n, t) \equiv \pi_{<objectId>}\sigma_{objName=n}\sigma_{tenantId=t}(Objects), \end{aligned} \tag{5}$$

where n and t respectively denote input logical object name and tenant id.

In (5), $LObjtName$ is the set of logical object names and $PObjName$ is the set of physical object names; *objectId*, *objName*, and *tenantId* are column names in the *Objects* table.

To show why the transformation of $\xi^{object}(n, t)$ is valid, let us start by examining the schema definition of *Objects* depicted in Fig. 2):

$$Objects\big(objectId, tenantId, objectName\big).$$

The schema definition of *Objects* indicates that the system can lookup a physical object name (*objectId*) by specifying a logical object name (*objectName*) and a tenant ID (*tenantId*). The lookup process can be written in algebraic form:

$$\pi_{<objectId>}\big(\sigma_{objName=n \wedge tenantId=t}(Objects)\big).$$

Next, using (1), the conjunctive conditions can be broken into a sequence of selections:

$$\pi_{<objectId>}\sigma_{objName=n}\sigma_{tenantId=t}\big(Objects\big). \tag{6}$$

By Eq. (2), σ is commutative. Hence, the statement can be optimized by arranging the σ operations in the proper order. Typically, selection operations with more restrictive selection conditions should be evaluated first so that the size of intermediate data can be minimized. Here the "restrictiveness" of a selection operation is estimated by the concept of selectivity, which is defined below.

Definition 2 (*Selectivity of σ*) The selectivity of a selection operation σ_C on table O, denoted $\theta[\sigma_C, O]$, is defined as the ratio of the number of records that satisfy the selection condition(s) in C to the total number of records in the table O.

Table 1 Notations and preset values used in performance analysis

Notation	Description	Preset value
m	Number of tenants	1000
$\bar{O}$	Average number of logical objects per tenant	200
$\bar{f}$	Average number of logical fields per logical object	15
$\bar{r}$	Average number of instances(records) per logical object	5000
$\bar{l}$	Average number of relationships defined by one tenants	100
$\bar{\theta}$	Average selectivity	
b	Blocking factor (the number of columns that can fit into a disk block)	137 †

† block size = 8192 bytes and column size = 60 bytes; $\lceil 8192/60 \rceil = 137$

Intuitively, the lower selectivity of a selection operation, the more restrictive it is, since more records are filtered out by this operation. As a result, the selection operation with the smallest θ should be evaluated first. Table 1 summarizes the notations and chosen default values used in selectivity estimation and performance analysis, where the number of tenants, the average number of logical objects held by each tenant, the average number of logical fields per object, and the average number of records per logical table are denoted as m, $\bar{O}$, $\bar{f}$, and $\bar{r}$, respectively.

To estimate $\theta[\sigma_{tenantId=t}, Objects]$, the number of records in $Objects$ can be obtained by multiplying the average number of logical objects per tenant by the number of tenants, namely $m \cdot \bar{O}$; the number of records that satisfy $tenantId = t$ depend on how many distinct objects a tenant has, which can be estimated by $\bar{O}$. Therefore, the average selectivity, denoted as $\bar{\theta}$, of $\sigma_{tenantId=t}$ is:

$$\bar{\theta}\big[\sigma_{tenantId=t}, Objects\big] = \frac{\bar{O}}{m \cdot \bar{O}} = \frac{1}{m}.$$

Estimating $\theta[\sigma_{objName=n}, Objects]$ is more tricky. One extreme case is that every tenant has a logical object named n. In this case, there are m records that satisfy the condition $objName = n$, so that we have $m/(m \cdot \bar{O})$. The other extreme case is that if for all tenants there is only one logical object having the name n, then there is only 1 record that satisfies $objName = n$, thus the selectivity becoming $1/(m \cdot \bar{O})$.

A general form can be obtained by combining the two extreme cases mentioned above:

$$\frac{\kappa}{m \cdot \bar{O}}, \quad \text{where } 1 \leq \kappa \leq m.$$

Here we choose $\kappa = m/2$ as an average case. Hence, the average selectivity of $\sigma_{objName=n}$ is:

$$\bar{\theta}\big[\sigma_{objName=n}, Objects\big] = \frac{m/2}{m \cdot \bar{O}} = \frac{1}{2\bar{O}}.$$

In practice, $m > \bar{O}$, which means that the number of tenants is usually larger than the number of objects per tenant so that $\sigma_{tenantId=t}$ should be evaluated first.

Consequently, the order of selection operations in (6) is appropriate and does not need further modification. If m happened to be smaller than $\bar{O}$, then the order of $\sigma_{objName=n}$ and $\sigma_{tenantId=t}$ have to be exchanged to improve performance. Similar selectivity estimation techniques will be used to guide the derivation of relational algebra expressions presented in the following discussions.

Similar to the object name transformation function (Definition 1), the second transformation function which is used to lookup the fields of an object is defined below.

Definition 3 (*Field Name Transformation*) The field name transformation function ξ^{field} that returns the physical field name based on an input logical object name, a logical field name, and a tenant ID is defined as follows:

$$\begin{aligned}&\xi^{field} : LObjName \times LFieldName \times TenantID \rightarrow PFieldName\\ &\xi^{field}(n_o, n_f, t) \equiv\\ &\pi_{fieldNum}\big(\sigma_{fieldName=n_f}\sigma_{tenantId=t}\sigma_{objectId=\xi^{object}(n_o,t)}(Fields)\big),\end{aligned} \tag{7}$$

where n_o, n_f and t denote input logical object name, input logical field name and tenant id, respectively. Note that $LFieldName$ is the set of logical field names and $PFieldName$ is the set of physical field names; objectId, fieldName, and tenantId are column names.

Similar to object name transformation, we start by examining the schema definition of *Fields* (see Fig. 2), and then derive an initial algebraic expression for looking up a physical field name (the *fieldId* column) based on a physical object name (*objectId*), a logical field name (the *fieldName* column) and a tenant ID (*tenantId*). As a result, the physical object name can be obtained with the following expression:

$$\pi_{fieldNum}\big(\sigma_{objectId=\xi^{object}(n_o,t)\wedge fieldName=n_f\wedge tenantId=t}(Fields)\big).$$

Using (1), the above equation can be split into a sequence of simple selections:

$$\pi_{fieldNum}\big(\sigma_{objectId=\xi^{object}(n_o,t)}\sigma_{fieldName=n_f}\sigma_{tenantId=t}(Fields)\big). \tag{8}$$

The selection operations have to be arranged properly so that operations with smaller selectivity are evaluated earlier. In (8), there are three selection operations: $\sigma_{tenantId=t}$, $\sigma_{fieldName=n_f}$, and $\sigma_{objectId=\xi^{object}(n_o,t)}$. Calculating the selectivity of $\sigma_{tenantId=t}$ is straightforward: the number of records in *Fields* table is $m \cdot \bar{O} \cdot \bar{f}$ and the average number of records stored in *Fields* for each tenant is $\bar{O} \cdot \bar{f}$. Thus, we have:

$$\theta\big[\sigma_{tenantId=t}, Fields\big] = \frac{\bar{O} \cdot \bar{f}}{m \cdot \bar{O} \cdot \bar{f}} = \frac{1}{m}. \tag{9}$$

The general form of the selectivity of $\sigma_{fieldName=n_f}$ can be derived with a similar technique to the one used in Definition 1

$$\theta\big[\sigma_{fieldName=n_f}, Fields\big] = \frac{\kappa}{m \cdot \bar{O} \cdot \bar{f}}, \quad \text{where } 1 \leq \kappa \leq m \cdot \bar{O}.$$

The average selectivity can then be calculated by letting $\kappa = m \cdot \bar{O}/2$:

$$\bar{\theta}\left[\sigma_{fieldName=n_f}, Fields\right] = \frac{m \cdot \bar{O}/2}{m \cdot \bar{O} \cdot \bar{f}} = \frac{1}{2 \cdot \bar{f}}. \tag{10}$$

The selectivity of $\sigma_{objectId=\xi^{object}(n_o,t)}$ is fixed but implementation dependent. As the physical object name (*objectId*) is typically generated and assigned by the system. For example, the system can use a GUID (Globally Unique Identifier) generator to assign physical object names. In this case each value of *objectId* is unique in the *Fields* table. Therefore,

$$\theta\left[\sigma_{objectId=\xi^{object}(n_o,t)}, Fields\right] = \frac{1}{m \cdot \bar{O} \cdot \bar{f}}. \tag{11}$$

Alternatively, the system can use a sequence number as a physical object name. In this case, each tenant has the same set of physical object names, namely, 1, 2, ...n. Hence,

$$\theta\left[\sigma_{objectId=\xi^{object}(n_o,t)}, Fields\right] = \frac{m}{m \cdot \bar{O} \cdot \bar{f}} = \frac{1}{\bar{O} \cdot \bar{f}}. \tag{12}$$

Based on the results obtained from (9) to (12), we suggest that the following expression be used in cases where the number of tenants is not very large, namely, $m > \bar{O} \cdot \bar{f}$:

$$\pi_{fieldNum}\left(\sigma_{fieldName=n_f}\sigma_{tenantId=t}\sigma_{objectId=\xi^{object}(n_o,t)}(Fields)\right).$$

For example, if the system allows each tenant to hold up to 100 distinct objects (tables), and each object has up to 50 fields, then the term $\sigma_{tenantId=t}$ in (7) can be moved to the leftmost position to improve performance when there are more than 5000 tenants.

4.1 Overall approach

To sketch the overall idea of the proposed rewriting schemes, recall that in Universal Table schema-mapping, all records belonging to a tenant specific logical table are physically stored in the *Data* table which uses generic physical field names such as "value1, value2, ..." instead of logical field names (see Fig. 5). Thus, the general rule of thumb for transforming a tenant-aware logical statement into physical one is first to reconstruct the logical table using a physical SQL statement, including the recovery of physical filed names such as "value1, value2, ..." to logical field names, and then arbitrary query operations can be applied to the reconstructed table. In the context of multitenancy, we extend the original notation by specifying "tenant ID" in a square bracket so that if the tenant ID is t, then $[t](\dot{O})$ denotes the logical table held by tenant t.

To explain the approach more clearly, consider the mapping of the logical table *Product* shown in Fig. 5. Let us assume that the following logical SQL statement is submitted by tenant 667:

SELECT price, description **FROM** Product.

The algebraic from of the above SQL statement is

$$\pi_{<price,description>}[667]\big(Product\big). \tag{13}$$

As mentioned, we can find all records belonging to the logical table *Product* that also belong to tenant 667 from the physical table $Data$ by performing a physical selection statement filtered by $tenantId$ and $objectId$. The value of physical field $objectId$ can be obtained by the object name transformation function $\xi^{object}(Product, 667)$ which is assumed to be 1 in this example:

$$\sigma_{objectId=1 \wedge tenantId=667}\big(Data\big). \tag{14}$$

The next step is to obtain the logical field names using the field name transformation function $\xi^{field}(Product, n_f, 667)$, where the logical field names *id*, *name*, *price*, and *description* are obtained by substituting n_f with *value1*, *value2*, *value3*, and *value4*, respectively. As a result, the logical table Product can be reconstructed by appending a rename and a projection operation in front of (14):

$$[667]\big(Product\big) = \rho_{\big(id,name,price,description\big)}\ \pi_{<value1,value2,value3,value4>}\ \sigma_{objectId=1 \wedge tenantId=667}\big(Data\big). \tag{15}$$

Note that the projection operation $\pi_{<value1,value2,value3,value4>}$ is required since the Data table has additional fields to keep track of the metadata of a record such as the *GUID*, *objectId* and *tenantId* fields of the *Data* table in Fig. 5.

Now that we have reconstructed the logical table *Product* so that arbitrary query operations can be applied to it, with (13) and (15), we have:

$$\pi_{<price,description>}[667]\big(Product\big) = \pi_{<price,description>}\ \rho_{\big(id,name,price,description\big)}\ \pi_{<value1,value2,value3,value4>}\ \sigma_{objectId=1 \wedge tenantId=667}\big(Data\big). \tag{16}$$

Then, the physical form of the tenant-aware logical projection statement can be derived:

```
SELECT price, description FROM (
    SELECT value1 AS id, value2 AS name,
               value3 AS price, value4 AS description
    FROM Data
    WHERE objectId=1 AND tenantId=667
).
```

The above example brings us to the derivation of a general form of table reconstruction. Again, we start by finding all records belonging to the logical table $[t]\dot{O}$ from *Data* and then storing the outcomes in a temporary storage T:

$$T \leftarrow \sigma_{objectId=\xi^{object}\left(\dot{O},t\right)\wedge tenantId=t}\left(Data\right). \tag{17}$$

Next, let us denote the set of all logical fields of $[t]\dot{O}$ as $\dot{F}*$, then for each field $\dot{f} \in \dot{F}*$, the corresponding physical field in T is projected and renamed. In this way, $[t](\dot{O})$ is reconstructed.

$$[t]\left(\dot{O}\right) \leftarrow \rho_{\dot{F}*}\pi_{<\left\{\xi^{field}\left(\dot{O},\dot{f},t\right)|\dot{f}\in\dot{F}*\right\}>}(T). \tag{18}$$

Finally, we can merge (17) and (18) to obtain the general form:

$$\begin{aligned}[t]\left(\dot{O}\right) = {} & \rho_{\dot{F}*}\pi_{<\left\{\xi^{field}\left(\dot{O},\dot{f},t\right)|\dot{f}\in\dot{F}*\right\}>} \\ & \sigma_{objectId=\xi^{object}\left(\dot{O},t\right)\wedge tenantId=t}\left(Data\right).\end{aligned} \tag{19}$$

4.2 Projection rewriting

We are now ready to derive the rewriting schemes for projection. A projection $\pi_F(O)$ is essentially a column filter on table O that discards all columns except the fields where $a \in F$. In this way, the statement $\pi_{\dot{F}}[t](\dot{O})$ stands for a logical projection on the tenant-aware logical table $[t](\dot{O})$ which is formally defined as follows.

Definition 4 (*Tenant-Aware Logical Projection*) A tenant-aware logical projection is defined as $\pi_{\dot{F}}[t](\dot{O})$, where $\dot{F}$ is a tuple of projected logical fields, t is the tenant ID, and $\dot{O}$ is the logical object in the projection.

As mentioned, the overall approach is to reconstruct the logical table $[t](\dot{O}$ after which the logical projection $\pi_{\dot{F}}$ can be applied to the reconstructed table. Formally, using the general form of logical table reconstruction (19), we can substitute $[t](\dot{O})$ with a physical query statement:

$$\begin{aligned}\pi_{\dot{F}}[t]\left(\dot{O}\right) = {} & \pi_{\dot{F}}\rho_{\dot{F}*}\pi_{<\left\{\xi^{field}\left(\dot{O},\dot{f},t\right)|\dot{f}\in\dot{F}*\right\}>} \\ & \sigma_{objectId=\xi^{object}\left(\dot{O},t\right)\wedge tenantId=t}\left(Data\right).\end{aligned} \tag{20}$$

Unfortunately, the logical table reconstruction approach does not produce an optimized general form. The point to observe is that (17) is essentially for sieving out desired records from the logical table T. However, building a logical table for each query is inefficient both in time and in space. As a result, we can use the same technique as the one presented in Definitions 1 and 3, that is, arranging operations in proper order so that the most selective operations are evaluated earlier.

To move $\pi_{\dot{F}}$ to the right side of ρ, all logical fields $\dot{f} \in \dot{F}$ have to be renamed. Hence, $\pi_{\dot{F}}$ is changed to $\pi_{<\{\xi^{field}(\dot{O},\dot{f},t)|\dot{f}\in\dot{F}\}>}$ after $\pi_{\dot{F}}$ is moved to the right side of ρ. Using (3), the two projections $\pi_{<\{\xi^{field}(\dot{O},\dot{f},t)|\dot{f}\in\dot{F}\}>}$ and $\pi_{<\{\xi^{field}(\dot{O},\dot{f},t)|\dot{f}\in\dot{F}^*\}>}$ can be merged. Note that $\dot{F} \subseteq \dot{F}^*$, so that $\pi_{<\{\xi^{field}(\dot{O},\dot{f},t)|\dot{f}\in\dot{F}^*\}>}$ is dropped and $\rho_{\dot{F}^*}$ is replaced by $\rho_{\dot{F}}$:

$$\rho_{\dot{F}}\pi_{<\left\{\xi^{field}(\dot{O},\dot{f},t)|\dot{f}\in\dot{F}\right\}>}\sigma_{objectId=\xi^{object}\left(\dot{O},t\right)\wedge tenantId=t}\left(Data\right).$$

Then, using (1), the conjunctive conditions can be broken into two consecutive selections:

$$\rho_{\dot{F}}\pi_{<\left\{\xi^{field}(\dot{O},\dot{f},t)|\dot{f}\in\dot{F}\right\}>}\sigma_{objectId=\xi^{object}\left(\dot{O},t\right)}\sigma_{tenantId=t}\left(Data\right).$$

The calculation of $\theta[\sigma_{tenantId=t}, Data]$ and $\theta[\sigma_{objectId=\xi^{object}(\dot{O},t)}, Data]$ will not be elaborated on here, as the procedure and the results are similar to that of Def.3. The results indicate that, unless $m > \bar{O}$, the following statement should be used, where m is the number of tenants and $\bar{O}$ is the average number of logical objects per tenant:

$$\pi_{\dot{F}}[t]\left(\dot{O}\right) = \rho_{\dot{F}}\pi_{<\left\{\xi^{field}\left(\dot{O},\dot{f},t\right)|\dot{f}\in\dot{F}\right\}>}\sigma_{tenantId=t}\sigma_{objectId=\xi^{object}\left(\dot{O},t\right)}\left(Data\right). \tag{21}$$

In other words, if there are a lot of tenants and each one owns relatively few tables and records, then the positions of the two selection operations in (21) should be exchanged.

If one of the field names specified in $\dot{F}$ happened to be a primary key or an index in the logical table $\dot{O}$, then the query can be rewritten to take advantage of the index specified in $Index$ and $UniqueFields$. For example, by inspecting the $Fields$ table, it can be observed that $orderId$ in the query $\pi_{orderId}[1](Order)$ is a primary key in the logical table $Order$. As a result, we can search for the set of data GUIDs (G) by looking up $UniqueFields$, that is,

$$G \leftarrow \pi_{dataGuid}\sigma_{tenantId=t}\sigma_{objectId=\xi^{object}\left(\dot{O},t\right)}\left(UniqueFields\right).$$

The rewriting scheme then can be rewritten as:

$$\pi_{\dot{F}}[t]\left(\dot{O}\right) \equiv \rho_{\dot{F}}\pi_{<\left\{\xi^{field}\left(\dot{O},\dot{f},t\right)|\dot{f}\in\dot{F}\right\}>}\sigma_{dataGuid\in G}\left(Data\right). \tag{22}$$

The rewriting schemes for non-primary key indexes can be obtained in a similar way except that the table to look up is $Index$.

4.3 Selection rewriting

A selection, denoted $\sigma_C(O)$, picks out records that fulfill the constraints specified in a boolean expression C from a table O. We can extend this notation in the context of multitenancy by specifying the tenant ID t in a square bracket. In this way, the definition of tenant-aware logical selection is given below.

Definition 5 (*Tenant-Aware Logical Selection*) A tenant-aware logical selection is defined as $\sigma_{\dot{C}}[t](\dot{O})$, where $\dot{C}$ is a list of assertions on logical fields, t is the tenant ID, and $\dot{O}$ is the logical object name.

Similarly to rewriting logical projection statements, the overall approach is to reconstruct the logical table $[t](\dot{O}$ so that logical selection $\sigma_{\dot{C}}$ can be applied to the logical table produced by rewritten statements. Formally, using (19), we can substitute $[t](\dot{O})$ with physical statements:

$$\sigma_{\dot{C}}[t](\dot{O}) = \sigma_{\dot{C}}\rho_{\dot{F}*}\pi_{<\{\xi^{field}(\dot{O},\dot{f},t)\mid \dot{f}\in\dot{F}*\}>} \sigma_{objectId=\xi^{object}(\dot{O},t)\wedge tenantId=t}(Data). \tag{23}$$

Like (21), the statement listed above can be further optimized by breaking apart the conjunctive conditions and then arranging the conditions in the proper order:

$$\sigma_{\dot{C}}[t](\dot{O}) \equiv \sigma_{\dot{C}}\rho_{\dot{F}*}\pi_{<\{\xi^{field}(\dot{O},\dot{f},t)\mid \dot{f}\in\dot{F}*\}>}\sigma_{tenantId=t}\sigma_{objectId=\xi^{object}(\dot{O},t)}(Data). \tag{24}$$

The calculation of $\theta[\sigma_{tenantId=t}, Data]$ and $\theta[\sigma_{objectId=\xi^{object}(\dot{O},t)}, Data]$ is similar to that of a tenant-aware logical projection and will not be re-iterated here. If one of the logical field names specified in $\dot{C}$ happened to be a primary key or an index in the logical table $\dot{O}$, then the query can be rewritten differently to take advantage of the index specified in $Index$ and $UniqueFields$. Hence, we can search for the set of data GUIDs (G) by looking up the $UniqueFields$, that is,

$$G \leftarrow \pi_{dataGuid}\sigma_{tenantId=t}\sigma_{objectId=\xi^{object}(\dot{O},t)}(UniqueFields).$$

The above statement then can be rewritten as the statement shown below:

$$\sigma_{\dot{C}}[t](\dot{O}) \equiv \sigma_{\dot{C}}\rho_{\dot{F}*}\sigma_{(dataGuid\in G)}(Data). \tag{25}$$

Taking the following query as an example:

SELECT $*$ **FROM** Product **WHERE** id=1,

In Fig. 5, the logical schema of *Product* is defined as follows:

$$Product(\underline{id}, name, price, description),$$

where the underline indicates that the field *id* is the primary key of the *Product* table. If the tenant ID is 667, then the algebraic form is $\sigma_{id=1}[667](Product)$. We can lookup G in $UniqueFields$ since $orderId$ is a primary key. Hence, we have:

$$G \leftarrow \pi_{dataGuid}\sigma_{tenantId=667}\sigma_{objectId=\xi^{object}\left(Product,667\right)}\left(UniqueFields\right).$$

The query can be rewritten as :

$$\sigma_{id=1}\rho_{\left(id,name,price,description\right)}\sigma_{dataGuid\in G}\left(Data\right).$$

As a result, the generated query statement is:

```
SELECT * FROM (
                SELECT value1 AS id, value2 AS name,
                             value3 AS price, value4 AS description
                FROM Data
                WHERE objectId=1 AND
                        tenantId=667 AND
                        dataGuid IN (
                            SELECT dataGuid
                            FROM UniqueFields
                            WHERE objectId=1 AND tenantId=667
                        )
) WHERE id=1.
```

4.4 Join rewriting

The join operation, denoted by $O_1 \bowtie_C O_2$, is used to combine sets of tuples coming from two tables. By convention, a join operation refers to an inner join which essentially obtains a Cartesian product of joining tables, and then performs a selection based on specific conditions, that is, $\sigma_C(O_1 \times O_2)$. In the context of multitenancy, the notation can be refined as follows:

Definition 6 (*Tenant-Aware Logical Join*) A tenant-aware logical join is defined as $\dot{O}_1 \bowtie_{\dot{C}} [t](\dot{O}_2)$, where $\dot{C}$ is a list of condition on logical fields, t is the tenant ID, and $\dot{O}$ is the logical object name.

Likewise, the overall strategy is to reconstruct logical tables $\dot{O}_1$ and $\dot{O}_2$ so the logical conditions $\dot{C}$ can be evaluated. The rewriting schemes for constructing logical tables (i.e., $\dot{O}_1$ and $\dot{O}_2$) are already derived in (19) and (23). Consequently, we have:

$$\begin{aligned} O_1 &\leftarrow \rho_{\dot{F}*}\sigma_{tenantId=t}\sigma_{objectId=\xi^{object}\left(\dot{O}_1,t\right)}\left(Data\right), \text{ and} \\ O_2 &\leftarrow \rho_{\dot{F}*}\sigma_{tenantId=t}\sigma_{objectId=\xi^{object}\left(\dot{O}_2,t\right)}\left(Data\right). \end{aligned} \tag{26}$$

Now that the physical field names are replaced by logical ones so that the join operation can be performed directly on $\dot{C}$. The rewriting scheme can be summarized below:

$$\dot{O}_1 \bowtie_{\dot{C}} [t](\dot{O}_2) \equiv O_1 \bowtie_{\dot{C}} O_2, \tag{27}$$

where O_1 and O_2 have been derived in (26).

Nevertheless, performing join operations are costly so that we can take advantage of the index table maintained in the physical schema. The idea is similar to the one used in Sect. 4.3 except that in a join operation, *relationships* is used as an index table. The first step is to search for all GUIDs of O_1 and O_2 respectively:

$$\begin{aligned} G_1 &\leftarrow \pi_{dataGuid}\sigma_{sourceObjectId=\xi^{object}(\dot{O}_1,t)} \\ &\quad \sigma_{targetObjectId=\xi^{object}\sigma_{tenantId=t}(\dot{O}_2,t)}(Relationships) \\ G_2 &\leftarrow \pi_{dataGuid}\sigma_{sourceObjectId=\xi^{object}(\dot{O}_2,t)} \\ &\quad \sigma_{targetObjectId=\xi^{object}(\dot{O}_1,t)}\sigma_{tenantId=t}(Relationships). \end{aligned} \tag{28}$$

In this way, (26) can be rewritten below:

$$\begin{aligned} O_1 &\leftarrow \rho_{\dot{F}}\sigma_{dataGuid\in G_1}(Data) \\ O_2 &\leftarrow \rho_{\dot{F}}\sigma_{dataGuid\in G_2}(Data). \end{aligned} \tag{29}$$

The θ values for the selection operations in (28) can be derived using the same techniques as (9), (11), and (12). The selectivity for selection operations on *Relationship* depends on the average number of relationships $\bar{l}$ among objects defined by the tenant. For example ,

$$\theta\left[\sigma_{tenantId=t}, Relationships\right] = \frac{\bar{l}}{\bar{l}\cdot m} = \frac{1}{m}.$$

Likewise,

$$\theta\left[\sigma_{sourceObjectId=\xi^{object}(\dot{O}_k,t)}, Relationships\right] = \frac{1}{m\cdot\bar{l}},$$

if *objectName* is assigned using GUID or

$$\theta\left[\sigma_{sourceObjectId=\xi^{object}(\dot{O}_k,t)}, Relationships\right] = \frac{1}{\bar{l}},$$

otherwise. In the second case,

$$\theta\left[\sigma_{tenantId=t}, Relationships\right] < \theta\left[\sigma_{tenantId=t}, Relationships\right]$$

holds when $m > \bar{l}$, so that we can consider exchanging the position of the terms $\sigma_{tenantId=t}$ and $\sigma_{sourceObjectId=\xi^{object}(\dot{O}_k,t)}$ in (28) if the number of tenants are larger than that of average relationships a tenant defines.

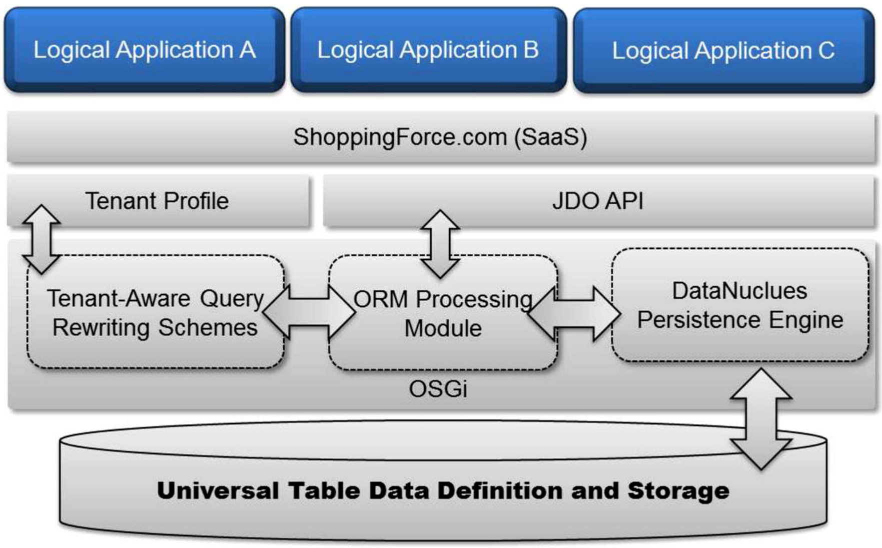


Fig. 7 The prototype implementation of the proposed approach

5 Evaluation

In this section, we verify the feasibility by implementing a transparent query rewriting engine prototype as well as a SaaS application, namely, the ShoppingForce.com. In addition, a performance analysis is also conducted to estimate the I/O access overhead.

5.1 Implementation

We study the feasibility of the proposed engine by developing a prototype based on an open source Java universal data management platform called DataNucleus. As shown in Fig. 7, this platform is constructed on top of a well-known service-oriented module management standard called OSGi (The OSGi Alliance 2012) so that DataNucleus is plug-in driven and highly extensible. The rewriting engine implements standard APIs required by DataNucleus and is developed as an OSGi bundle. Then, the bundle is registered as a service in the DataNucleus platform so that the rewriting engine is able to perform transparent query rewriting for the application.

To verify the prototype, we implemented a simple SaaS application called ShoppingForce.com on top of DataNucleus. ShoppingForce.com is basically a multitenant enhanced JPetStore (Clinton 2004), which is a sample full-fledged three-tier on-line shopping application widely used for educational and research purposes. The ShoppingForce.com can now being customized to sell different products. Figure 8 shows three different on-line shopping applications hosted on ShoppingForce.com. Technically speaking, the above mentioned applications are able to access the rewriting engine deployed on DataNucleus through JDO (Java Data Object) (Russell 2010),

Fig. 8 Example on-line shopping applications hosted by ShoppingForce.com

which is one of the official Java-based ORM (Object-Relational Mapping) specifications. To access the physical schema, the application uses JDOQL (JDO Database Query Language) (Russell 2010) and manipulates JDO API. Then, The JDOQL is translated internally to SQL statements and then used as the inputs of the proposed rewriting engine. To create a logical application, the tenant applies for an account on-line and a tenant profile is then generated accordingly. Sometimes a tenant needs to modify default logical schema such as adding tenant-specific columns. Currently, the schema customization functionality has not been implemented yet. However, after the CREATE statement is supported by the underlying rewriting engine, schema customization can be realized by a schema customization page in the account management page of ShoppingForce.com.

5.2 Analysis

In this research, we adopt query performance analysis techniques similar to those presented in Jarke and Koch (1984), where queries are decomposed into atomic steps and then the I/O access counts need to be carried over as each step is estimated. Note that data distribution issues such as communication complexity (Lynch 1996) are out of the scope of this research. The performance is estimated by the I/O access to the secondary storage, namely the disk I/O, required by the rewriting modules. CPU costs, memory accesses, and cache hits/misses are not taken into account, either, since total cost is dominated by access time of the secondary storage.

Recall that the number of tenants, the average number of logical objects held by each tenant, the average number of virtual fields per object, the average number of records per logical table, and the average number of relationships defined by a tenant are defined as m, $\bar{O}$, $\bar{f}$, $\bar{r}$, and $\bar{l}$, respectively. Then, the size of the tables (number of records in a table) can be calculated, as shown in the second column of Table 2. For instance, since there are m tenants, each of them has $\bar{O}$ tables that contain $\bar{r}$ records in average. Therefore, the size of *Data* table is $m \cdot \bar{O} \cdot \bar{r}$ records.

One important parameter for estimating the I/O count for a database operation is the blocking factor (Elmasri and Navathe 2011), which is traditionally defined to be the number of records that can fit into a disk block. This definition assumes that the size of records for all tables are equal. Unfortunately, this is not true in Universal Table schema-mapping, since the size of records in Universal Table (i.e., *Data*) is much greater than that of other tables. Consequently, we take a fine-grained approach, that

Table 2 I/O Count for Universal Table schema-mapping

Table name	Table size (records)	I/O capacity (records/block)	I/O count per table scan
Data	$m \cdot \bar{O} \cdot \bar{r}$	$b/504$	$\ulcorner m \cdot \bar{O} \cdot \bar{r} \cdot 504/b \urcorner$
Objects	$m \cdot \bar{O}$	$b/3$	$\ulcorner m \cdot \bar{O} \cdot 3/b \urcorner$
Fields	$m \cdot \bar{O} \cdot \bar{f}$	$b/7$	$\ulcorner m \cdot \bar{O} \cdot \bar{f} \cdot 7/b \urcorner$
UniqueFields	$m \cdot \bar{O}$	$b/7$	$\ulcorner m \cdot \bar{O} \cdot 7/b \urcorner$
Relationships	$m \cdot \bar{l}$	$b/5$	$\ulcorner m \cdot \bar{l} \cdot 5/b \urcorner$

is, we refine the original definition so that the blocking factor refers to the number of columns that can fit into a disk block:

$$b \text{ (blocking factor)} = \llcorner \frac{\text{Block size}}{\text{Average column size}} \lrcorner .$$

Depending on the blocking factor of the underlying storage, several records can be accessed in one I/O operation. Column 3 of Table 2 shows the I/O capacity for each table in context of Universal Table schema-mapping. For instance, consider the *Data* table, which contains 504 columns (500 for *Value1, Value2,... Value500* plus 4 additional columns, as shown Fig. 2), the number of records that can be accessed in one I/O operation, namely the I/O capacity of a table, is $b/504$ records per block. From this example, it can be observed that the I/O capacity of a table is mainly determined by the number of columns as defined by the table schema. As shown in Column 4 of Table 2, I/O count per table scan can then be derived from the size of the table (Table 2, Column 2) over the I/O capacity (Table 2, Column 3).

The actual I/O count for performing a query operation are implementation dependent. As a baseline analysis, we assume that no parallel or interleaved I/O operations are used and that one table scan is required for a sequence of selections on the same table where the order of selections is taken into account by the underlying implementation. In addition, we also assume that the size of main memory is sufficiently large enough that the system is able to reserve a sufficient buffer for storing records that fulfill the selection conditions.

5.2.1 Object and field name transformation

From Definition 1, ξ^{object} contains a sequence of selections so that it essentially performs a table scan on *Object* (i.e., read: $\ulcorner 3 \cdot m \cdot \bar{O}/b \urcorner$; write: 0, since the data are transient). Likewise, ξ^{field} defined in Definition 3 includes a sequence of selections on *Fields*, so the I/O count is equal to performing a table scan on *Fields*, namely,

$$\ulcorner m \cdot \bar{O} \cdot \bar{f} \cdot 7/b \urcorner .$$

The I/O count for writing is also zero since the results of ξ^{field} are transient.

It is important to point out that both the ξ^{object} and ξ^{field} perform selections on the *tenantId* field. Hence, we can configure the schema of *Objects* and *Fields* so that *tenantId* is an indexed field. Taking *Objects* as an example, one can perform an index-assisted binary search to obtain all records belonging to a tenant as an intermediate result with $\ulcorner log_2(m \cdot \bar{O}) \cdot 3/b \urcorner$ I/O accesses. The size of intermediate results is $\bar{O}$ on average so that we need to scan through these $\bar{O}$ records. In this way, the I/O count for ξ^{object} becomes

$$\ulcorner log_2(m \cdot \bar{O}) \cdot 3/b \urcorner.$$

Here the scanning of $\bar{O}$ records does not involve any I/O access since they are already loaded into memory. Note that $\ulcorner log_2(m \cdot \bar{O}) \cdot 3/b \urcorner \le \ulcorner m \cdot \bar{O} \cdot 3/b \urcorner$ as m and $\bar{O}$ are positive integers. Likewise, if the field *tenantId* is indexed in *Fields* table, then the I/O count for ξ^{field} becomes $\ulcorner log_2(m \cdot \bar{O} \cdot \bar{f}) \cdot 7/b \urcorner$ which is smaller or equal to $\ulcorner m \cdot \bar{O} \cdot \bar{f} \cdot 7/b \urcorner$ as m, $\bar{O}$, and $\bar{f}$ are positive integers. It follows that it is usually worthy to configure the *tenantId* as an index in *Objects* and *Fields* tables.

5.2.2 Tenant-aware logical projection

From (20), the rewriting involves a table scan on *Data*, a call to ξ^{object} and a call to ξ^{field}. Therefore, the read counts are the sum of of these operations, all of which can be obtained by looking up Table 2. Thus, the read counts for π are

$$m \cdot \bar{O} \cdot \bar{r} + \ulcorner m \cdot \bar{O} \cdot 3/b \urcorner + \ulcorner m \cdot \bar{O} \cdot \bar{f} \cdot 7/b \urcorner.$$

The cost for performing table scans on *Data* is very high since the table has low I/O capacity. Again, we can configure the schema of *Data* so that *tenantId* is indexed in *Objects* and *Fields* and *objectId* is indexed in *Data*. Thus, all records with a specific *objectId* can be loaded by

$$log_2\big(m \cdot \bar{O} \cdot \bar{r}\big) + \ulcorner log_2\big(m \cdot \bar{O}\big) \cdot 3/b \urcorner + \ulcorner log_2\big(m \cdot \bar{O} \cdot \bar{f}\big) \cdot 7/b \urcorner$$

I/O operations. As mentioned, if the projecting fields happen to be a primary key or an index in the logical table, then the query can be rewritten to take advantage of logical index tables such as *UniqueFields* or *Index*. In this case, the rewriting with indices contains a search of the *UniqueFields* table as well as a call to ξ^{object}. Assume that *objectId* is indexed in *Objects* and *UniqueFields*, then the read count for searching *UniqueFields* and ξ^{object} are respectively $\ulcorner log_2(m \cdot \bar{O}) \cdot 7/b \urcorner$ and $\ulcorner log_2(m \cdot \bar{O}) \cdot 3/b \urcorner$. Finally, $\ulcorner \bar{O} \cdot \bar{r} \cdot \bar{f}/b \urcorner$ is the cost of loading all $dataGuid \in G$ from *Data*. GUIDs are globally unique, so the *dataGuid* field can be implemented with a hash index. To sum up, the read count of the indexed approach is

$$\ulcorner log_2\big(m \cdot \bar{O}\big) \cdot 7/b \urcorner + \ulcorner log_2\big(m \cdot \bar{O}\big) \cdot 3/b \urcorner + \ulcorner \bar{O} \cdot \bar{r} \cdot \bar{f}/b \urcorner.$$

On the other hand, the write count depends on the size of the results. Given that $|\dot{F}|$ columns are selected, the write counts are $\ulcorner |\dot{F}| \cdot \bar{r}/b \urcorner$.

5.2.3 Tenant-aware logical selection

It follows from (23) that σ has the same read access counts as π since it also involves a table scan on *Data*, a call to ξ^{object}, and a call to ξ^{field}. Here $\sigma_{\dot{C}}$ does not cause additional read count since the set of all logical fields in $\dot{C}$, denoted $\dot{F}$, is a subset of $\dot{F}^*$, that is, $\dot{F} \in \dot{F}^*$, so that all of the required fields are loaded into memory before the evaluation of $\sigma_{\dot{C}}$. As a result, the read counts for σ is also

$$m \cdot \bar{O} \cdot \bar{r} + \ulcorner m \cdot \bar{O} \cdot 3/b \urcorner + \ulcorner m \cdot \bar{O} \cdot \bar{f} \cdot 7/b \urcorner.$$

Likewise, for the indexed approach, the read count is also the same as π, namely,

$$\ulcorner log_2\big(m \cdot \bar{O}\big) \cdot 7/b \urcorner + \ulcorner log_2\big(m \cdot \bar{O}\big) \cdot 3/b \urcorner + \ulcorner \bar{O} \cdot \bar{r} \cdot \bar{f}/b \urcorner.$$

However, this is not the case for write access since the size of selection results depends on selectivity $\theta[\sigma_{\dot{C}}, \dot{O}]$, where $\dot{O}$ is the logical table. For simplicity, let us assume that the average selectivity of logical selections is $\bar{\theta}$. Then, we can expect that each logical selection obtains $\bar{r} \cdot \bar{\theta}$ records on average. The write count is thus $\bar{r} \cdot \bar{\theta}/b$. The higher selectivity indicates more results will be produced and thus more write accesses.

5.2.4 Tenant-aware logical join

When analyzing tenant-aware logical join, we assume that the join is implemented by the Nested–Block Join method (Elmasri and Navathe 2011) as it is general-purpose and is the default algorithm for most database. Specifically, for each record o_1 in O_1, we retrieve every record o_2 in O_2 and test whether the two fetched records match the join condition, namely $\dot{C}$ in (27). In this way, (27) can be rewritten as (26) and (26) indicates a table scan on $Data$, a call to ξ^{object}, and a call to ξ^{field}, the read access counts of O_1 and O_2 are also identical to that of π and σ. As a result, the outcome is 2 times the read access counts of π or σ.

When taking advantage of the *Relationships* index table, the join operation involves searching the *Relationships* table as well as a call to ξ^{object}. Assume that *objectId* is indexed in *Objects* and *Relationships*, then the read count for searching *Relationships* and ξ^{object} are respectively $\ulcorner log_2(m \cdot \bar{l}) \cdot 5/b \urcorner$ and $\ulcorner log_2(m \cdot \bar{O}) \cdot 3/b \urcorner$. Lastly, since GUIDs are globally unique, $\ulcorner \bar{O} \cdot \bar{r} \cdot \bar{f}/b \urcorner$ is the cost of loading all $dataGuid \in G$ from *Data*, and the *dataGuid* field can be implemented with a hash index. As a result, the read count of tenant-aware logical join is

$$2 \cdot \big(\ulcorner log_2(m \cdot \bar{l}) \cdot 5/b \urcorner + \ulcorner log_2(m \cdot \bar{O}) \cdot 3/b \urcorner + \ulcorner \bar{O} \cdot \bar{r} \cdot \bar{f}/b \urcorner\big).$$

Similar to selection operations, the write count is $\bar{r} \cdot \bar{\theta}/b$. The higher selectivity indicates more results will be produced and thus more write accesses. Based on the above results, the analysis of I/O counts for ξ^{object}, ξ^{field}, π, σ, and $\bowtie$ are summarized in Table 3.

Table 3 I/O count for logical query operations

Item	Read	Write
ξ^{object}	$\lceil m \cdot \bar{O} \cdot 3/b \rceil$	0
$\xi^{object*}$	$\lceil log_2(m \cdot \bar{O}) \cdot 3/b \rceil$	0
ξ^{field}	$\lceil m \cdot \bar{O} \cdot \bar{f} \cdot 7/b \rceil$	0
ξ^{field*}	$\lceil log_2(m \cdot \bar{O} \cdot \bar{f}) \cdot 7/b \rceil$	0
$\pi_{\dot{F}}$	$m \cdot \bar{O} \cdot \bar{r} + \lceil m \cdot \bar{O} \cdot 3/b \rceil + \lceil m \cdot \bar{O} \cdot \bar{f} \cdot 7/b \rceil$	$\lceil \lvert \dot{F} \rvert \cdot \bar{r}/b \rceil$
$\pi_{\dot{F}}*$	$\lceil log_2(m \cdot \bar{O}) \cdot 7/b \rceil + \lceil log_2(m \cdot \bar{O}) \cdot 3/b \rceil + \lceil \bar{O} \cdot \bar{r} \cdot \bar{f}/b \rceil$	$\lceil \lvert \dot{F} \rvert \cdot \bar{r}/b \rceil$
$\sigma_{\dot{C}}$	$m \cdot \bar{O} \cdot \bar{r} + \lceil m \cdot \bar{O} \cdot 3/b \rceil + \lceil m \cdot \bar{O} \cdot \bar{f} \cdot 7/b \rceil$	$\lceil \bar{r} \cdot \theta/b \rceil$ †
$\sigma_{\dot{C}}*$	$\lceil log_2(m \cdot \bar{O}) \cdot 7/b \rceil + \lceil log_2(m \cdot \bar{O}) \cdot 3/b \rceil + \lceil \bar{O} \cdot \bar{r} \cdot \bar{f}/b \rceil$	$\lceil \bar{r} \cdot \bar{\theta}/b \rceil$ †
$\bowtie_{\dot{C}}$	$2 \cdot (m \cdot \bar{O} \cdot \bar{r} + \lceil m \cdot \bar{O} \cdot 3/b \rceil + \lceil m \cdot \bar{O} \cdot \bar{f} \cdot 7/b \rceil)$	$\lceil \bar{r} \cdot \bar{\theta}/b \rceil$ †
$\bowtie_{\dot{C}}$ *	$2 \cdot (\lceil log_2(m \cdot \bar{l}) \cdot 5/b \rceil + \lceil log_2(m \cdot \bar{O}) \cdot 3/b \rceil + \lceil \bar{O} \cdot \bar{r} \cdot \bar{f}/b \rceil)$	$\lceil \bar{r} \cdot \bar{\theta}/b \rceil$ †

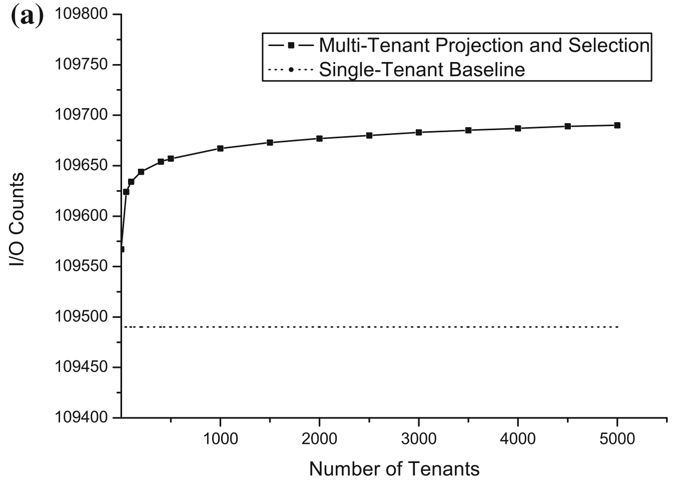
* specific fields are indexed; † $0 \leq \theta \leq 1$ is the selectivity the operation

5.2.5 Discussion

Having derived the general form of I/O counts, we are now able to discuss the impact on the overall performance of several important parameters such as number of tenants, block sizes, and average number of objects per tenant on the overall performance. The parameters of the derived rewriting schemes (see Table 3) are first set to reasonable default values (see Table 1). Then, we alternate one of the parameters and fix other values to observe how the overall I/O count is affected.

The first interesting result worth noting is tenant-size scalability. Figure 9 shows the relationship between the I/O count and the number of tenants using the indexed rewriting schemes (i.e. $\pi^*_{\dot{F}}$, $\sigma^*_{\dot{C}}$, and $\bowtie^*_{\dot{C}}$). The I/O count for single tenant query operations is depicted using the dotted line. The distance between the solid line and the dotted line is therefore the overhead imposed by multitenancy using our approach. Although the I/O count undergo a significant initial increase, after the number of tenants reaches 500, the I/O count increases very slowly. As a result, the proposed approach appears to be scalable to the number of tenants for both projection/selection and join operations. The reason is that for the indexed rewriting schemes, the I/O count for searching GUID is independent of tenant size (see Table 3).

Similar results can be obtained by alternating $\bar{O}$, $\bar{f}$, and $\bar{r}$. For instance, Fig. 10 shows the relationship between the I/O count and the average number of objects per tenant using indexed rewriting schemes. As indicated in this figure, the I/O overhead increases from 167 to 200 and from 258 to 310 using tenant-aware projection/selection and tenant-aware join, respectively, where the I/O overhead is calculated by finding the difference between tenant-aware operations and single-tenant ones. As a result, the proposed approach is also scalable to the number of average objects per tenant for both projection/selection and join operations.



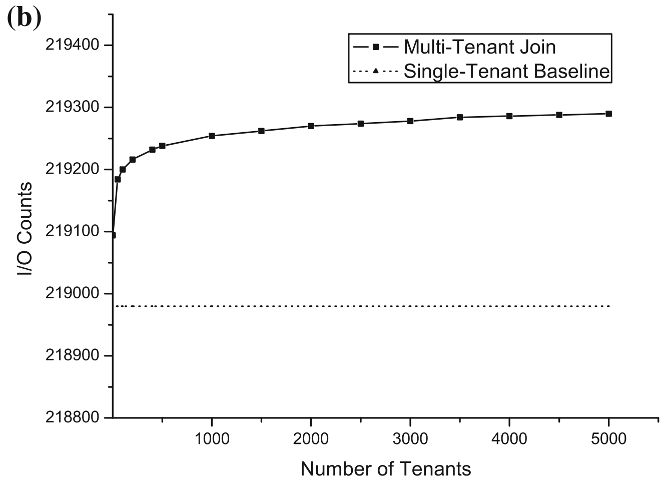


Fig. 9 Relationship between the I/O count and the number of tenants using the indexed rewriting schemes **a** Tenant-aware project and selection. **b** Tenant-aware join

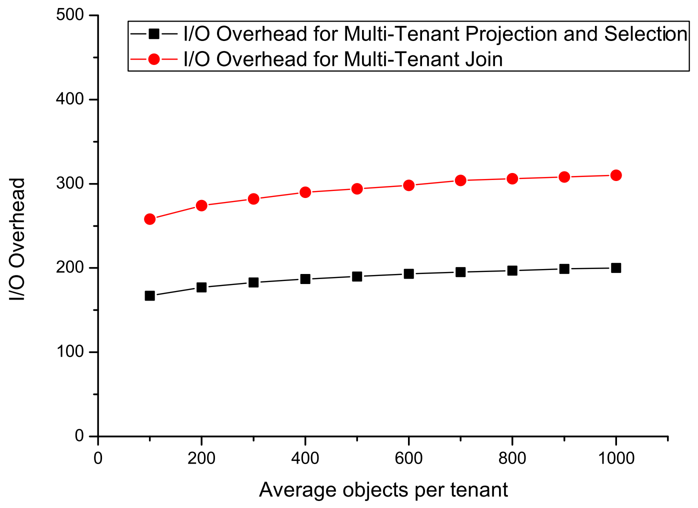


Fig. 10 I/O overhead in different average number of objects per tenant

5.3 Experiments

To study the consistency between analytical results and real-world experimental results, we conducted experiments in a switched network to further investigate the performance of the proposed rewriting schemes. The experiments consisted of two parts: turnaround time and concurrency level. Turnaround time is evaluated by the variation of average turnaround time as the number of tenants increases; concurrency level means the influence of the turnaround time as the number of concurrent clients increases. For comparison, Private Table schema layout, which implements the same sets of logical schema, was also implemented and tested. In the Private Table schema layout, each logical table is mapped to a physical table. The tenant id is added to the prefix of each table to avoid a name conflict (e.g., *T604_Product* versus *T667_Product*). As mentioned in Sect. 2, since logical queries can be performed directly on the physical tables with few transformations, namely adding the prefix to the table names, Private Table can achieve the highest performance but also has the highest costs. On the contrary, Universal Table has the lowest cost at the price of lower performance. Theoretically, the performance of Universal Table must be worse than that of Private Table since for the same logical SQL statement, the transformed physical SQL statement for Universal Table is much more complex than Private Table. In this sense, a rewriting mechanism for Universal Table is very effective if it causes the turnaround time for Universal Table to approach that of Private Table.

Initially, our program generates the physical schema that realizes the Private Table and Universal Table schema-mapping of the logical schema in JPetStore (Clinton 2004) automatically based on the scenario, and then for each tenant, each of its logical tables is filled with 10,000 randomly generated test records. Therefore, depending on different schema layout approaches, these records are distributed differently in the physical schema. For instance, in the Private Table schema layout, each tenant has the

Table 4 The turnaround time of queries on Private Table with caching and indexing are turned off (in seconds)

Num. of Tenants	Select	Project	Join
10	1.184	7.825	13.820
20	1.224	7.859	13.964
30	1.252	7.922	14.273
40	1.276	7.959	14.400
50	1.303	8.197	14.763

same set of physical tables so that the records are uniformly distributed. On the other hand, all logical records are physically placed in the *Data* table in Universal Table schema-mapping. In each experiment, the client issues query requests to the database server and then receives query results. Each request and the results are correlated so that the turnaround time of issuing the query can be measured.

After a few rounds of preliminary tests, we found that it is very hard to distinguish the performance overhead caused by the rewriting schemes from the overhead caused by other middleware components (e.g., DataNucleus or OSGi) if we perform experiments using the prototype constructed in Sect. 5.1. As a result, in the following experiments, overall evaluating process are driven directly by the Apache JMeter 2.9 (Halili 2008), a well-known open source and general-purpose performance measurement platform, which can be used to simulate arbitrary load types on the server or network to test overall performance under different load types. We modified the source code of JMeter JDBC (Java Database Connectivity) plug-in so that the rewriting module is able to intercept the SQL requests and performs transformations. MySQL Community Server 5.7 with InnoDB engine on Ubuntu Linux 12.04 is used as the database server. The client (JMeter and the rewriting module) and the server are deployed on two separate PCs with Intel Core i7 3.4-GHz processors with 4 GB of memory connected by a 100 Mbps switch. For each test, based on pre-configured scenario settings, several concurrent threads that issue SQL requests to the database server are initiated. Finally, the JMeter platform is responsible for gathering responses and reporting the average turnaround time.

5.3.1 Turnaround time

The objective of the first experiment is to measure the variation of average turnaround time as the number of tenants increases. In each test, 200 threads are initialized, and then each of them issue an SQL request concurrently. We performed experiments for the Selection, Projection, and Join statements of Private Table and Universal Table schema-mapping, respectively. Initially, we disabled the cache and index so that the experimental results were easier to verify and compare to the analytical results. Tables 4 and 5 display the plain turnaround time for the schema mappings of Private Table and Universal Table, respectively. To see the effects of caching and indexing, we also tested the performance of cached, indexed, and mixed Selection statements for the two schema-mappings mentioned above. As shown in Tables 6 and 7, the improvement

Table 5 The turnaround time of queries on Universal Table with caching and indexing are turned off (in seconds)

Num. of Tenants	Select	Project	Join
10	150.882	164.926	268.746
20	157.730	170.833	271.164
30	163.046	175.556	273.157
40	169.632	182.755	276.158
50	175.155	188.198	279.159

Table 6 The turnaround time of select operations on Private Table with caching and indexing are turned on (in seconds)

Num. of Tenants	Select	Select with cache	Indexed select	Indexed select with cache
10	1.184	1.123	.284	.229
20	1.224	1.162	.302	.287
30	1.252	1.170	.316	.300
40	1.276	1.182	.337	.310
50	1.303	1.209	.353	.327

Table 7 The turnaround time of select operations on Universal Table with caching and indexing are turned on (in seconds)

Num. of Tenants	Select	Select with cache	Indexed select	Indexed select with cache
10	150.882	4.275	1.542	.571
20	157.730	4.382	1.560	.585
30	163.046	4.491	1.590	.605
40	169.632	4.578	1.626	.638
50	175.155	4.611	1.664	.674

results for Private Table are little while those for Universal Table are significant. Besides, regardless of caching or not, the results show that the proposed rewriting scheme is scalable to the number of tenants.

By comparing the Select, Project and Join columns in Table 5, we can learn that Universal Table induces a greater amount of overhead if caching and indexing are disabled. As shown in Sect. 5.2, for Universal Table, a major portion of the overhead comes from processing complex statements and local disk I/O operations. However, as shown in Table 7, the performance of our approach is greatly improved when caching are turned on. Moreover, if the fields appear in a location where the clauses are indexed, than the turnaround time can be reduced further to be less than one second, which is comparable with Private Table and is reasonable in practice. It is also worthy to point out that the cache and indexing do not improve the performance of Private Table as much as Universal Table since the queries are relatively simple compared to Universal Table (see Tables 6 and 7).

The next interesting result to observe is that the ratios of turnaround time for Select, Project, and Join statements for Private Table and Universal Table are different. For

Private Table, the ratio is around 1:6:12 (see Table 4) whereas for Universal Table the ratio is about 1:1:2 (see Table 5). To explain the underlying reason for this difference, one has to note that the turnaround time consists of three parts: network delay, server computation time, and local disk I/O time. There is no need to rewrite the queries for Private Table, thus, unlike network delay, the statements cause very low server computation and local disk I/O. In other words, the turnaround time for issuing queries to Private Table is dominated by network delay, that is, the ratio reflects the data size for the queries and larger data size causes more network traffic. The network delay of Private Table for Select statements is lower since a lot of data are filtered out based on the where clause. The ratio is depending on the selectivity of the WHERE clause. On the other hand, the query processing is relatively more complex in Universal Table schema-mapping. Therefore, it takes a longer time to process the statements and to perform local disk I/O. The queries are performed by 200 threads concurrently so that some data can be transmitted over the network while other data are still being processed by the server. That is to say, the turnaround time in Universal Table is now dominated by server computation and local disk I/O time. As a result, selectivity has little impact in Universal Table. Since our analysis emphasizes local disk I/O overhead (see Sect. 5.2), the ratio of disk I/O overhead in the analytical results (Fig. 10), for example, roughly align with the experimental results for Universal Table, namely 1:1:2 for Select, Project and Join, respectively.

5.3.2 Concurrency level

The objective of the second set of tests is to study the influence of the turnaround time as the concurrency level (i.e., the number of concurrent JDBC connections) increases. Initially, a specific number of JDBC connections, ranging from 20 to 200, are initialized without using the JDBC connection pool, and then each of them issue an SQL request concurrently. We repeatedly perform experiments on Selection, Projection, and Join statements with Universal Table schema-mapping for 10, 30 and 50 tenants. The cache option is turned on and the SQL statements use the index in the experiments.

Figure 11 depicts the results of the experiments. Overall, the turnaround time increases gradually when the concurrency level increases. The results are consistent with our analysis, that is, the turnaround time is roughly the same for Select and Project statements while the turnaround time of Join is nearly 2 times that of Select and Project. Figure 11 also reveals tenant size scalability since the turnaround time only increases a bit when the tenant size increases. One more thing to note is that the gap between Select/Project and Join is smaller when the size of tenants is larger. We believe that this is the effect of turning on the cache.

5.3.3 Comparison to the analytical result

By comparing experimental results with the analytical results, we can find that, consistent with the analysis in Sect. 5.2.5, the turnaround time for indexed queries in Universal Table also reveal good tenant scalability. Also, as pointed out in Sect. 5.3.1, the ratio of local disk I/O overhead for Select, Project, and Join without caching and

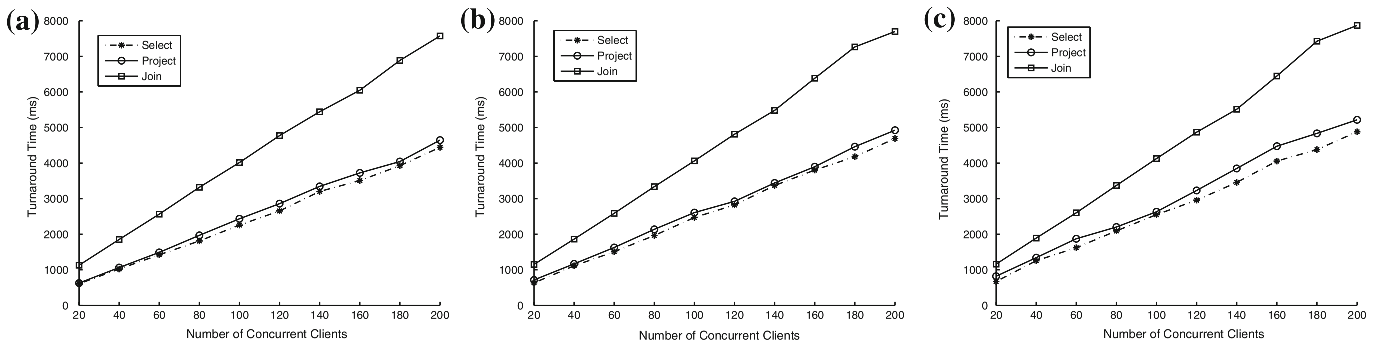


Fig. 11 Turnaround time of the proposed rewriting schemes for Universal Table as the number of concurrent clients (i.e. JDBC connections) increases **a** 10 tenants; **b** 30 tenants; **c** 50 tenants; (Cache is turned on and queries are indexed)

indexing in the analytical results is 1:1:2. This roughly aligns with the experimental results.

In addition, the results of the Concurrency Level experiments are also consistent with our analysis, that is, the turnaround time is roughly the same for Select and Project statements while the turnaround time of Join is nearly 2 times of that of Select and Project. Figure 11 also reveals tenant-size scalability since the turnaround time only increases a bit when the tenant size increases.

5.3.4 Summary

In this subsection, we report the results of experiments on turnaround time and concurrency level. The results not only agree well with the analytical predictions but also show that the schemes are scalable to the number of tenants and number of concurrent database connections. Overall, both analytical and experimental results indicate that the performance of Universal Table is able to approach that of Private Table when the index and cache are both available. This observation reveals that with appropriate configuration of the underlying database and careful design of the logical schema, the performance of Universal Table schema-mapping is reasonable in practice. That is to say, in the application layer, it is very important to guide SaaS application developers so that most of the logical SQL statements they use can be indexed. In addition, the built-in caching mechanism of a database can be unapplicable to our approach when the number of concurrent users is too large. In this case, high-performance caching middleware such as memcached can be used since many popular databases such as Oracle and MySQL have built-in supports for memcached. Nevertheless, it is important to point out that caching is beneficial but not essential for our approach. Table 7 shows that the approach is still practical when cache is absent, namely, if indexing is enabled and cache is absent, the performance can be also greatly improved. Finally, as the rationale of Universal Table is to achieve lower costs at the price of performance, it seems reasonable to say that the design of the proposed rewriting scheme is successful since it is able to realize Universal Table within a reasonable standard of performance.

6 Conclusion

Multitenancy is helpful to reduce the cost of hardware equipments and software licenses. With multitenancy, the virtualized and consolidated entities can be managed at a lower price yet with higher flexibility. Nevertheless, the benefits come at the price of performance. In this paper, we have investigated designs for query rewriting schemes that support multitenant SaaS applications using a Universal Table-based data architecture. Not only do we provide a theoretical design and analysis of these schemes, we also present a prototype implementation and a sample multitenant SaaS application based on the proposed schemes. Above all, we provide an empirical account of our query rewriting schemes which shows that the proposed schemes are scalable to the number of tenants and to the concurrency level. Through analytical and experimental results, we learn that appropriate tuning of the physical database and careful design of the rewriting mechanisms can greatly improve the performance of Universal Table, thus making it amenable to use in practical situations.

Currently, we focus mainly on the interactions between the middleware (i.e., rewriting schemes) and the database. In a real-world enterprise environment, performance issues can be much more complex. As we have observed in the experimental results, the turnaround time is greatly affected by database configuration, network latency, and middleware facilities such as connection pools. To obtain more sophisticated results, one can use advanced models such as the queuing model and discrete events simulation. However, as reported earlier, we still observe a strong consistency between analytical and experiment al results. The present work also focuses on supporting core operations such as σ, π, ρ, and $\bowtie$, where a $\bowtie$ implies $\times$ followed by a σ. Indeed, to construct a full-scale rewriting mechanism for SaaS applications, there is still much work to be done. However, as far as we know, there are relatively few works which focus on the design and analysis of such query rewriting mechanisms. Thus, we believe that this work lays down a solid foundation for future research. From a relational algebra perspective, it has been proved that $\{\sigma, \pi, \cup, \rho, -, \times\}$ is a complete set, that is, any of the other operations can be expressed as a sequence of operations from this set (Codd 1972). In the future, we shall enhance the current work by supporting the $\cup$ and $-$ operations, and aggregation functions such as SUM or COUNT.

Acknowledgments This work is partially sponsored by Ministry of Science and Technology, Taiwan, under Grant 103-2221-E-004-005, 103-2221-E-004-018, 103-2218-E-004-003 and 103-2218-E-004-001.

References

Aulbach, S., Grust, T., Jacobs, D., Kemper, A., Rittinger, J.: Multi-Tenant databases for software as a service: schema-mapping techniques. In: Proceedings of the ACM International Conference on Management of Data. ACM, New York (2008)

Chong, F., Carraro, G.: Architecture strategies for catching the long tail. http://msdn.microsoft.com/en-us/library/aa479069.aspx (2006). Accessed 26 May, July 2013

Chong, F., Carraro, G., Wolter, R.: Multi-tenant data architecture. http://msdn.microsoft.com/en-us/library/aa479086.aspx (2006). Accessed 26 May 2013

Clinton, B.: iBATIS JPetStore 4.0.5 (2004). http://sourceforge.net/projects/ibatisjpetstore/. Accessed 26 May 2013

Codd, E. F.: Relational completeness of database sublanguages. In: Database Systems. Prentice-Hall, Upper Saddle River (1972)

Copeland, G. P., Khoshafian, S. N.: A decomposition storage model. In: Proceedings of the ACM International Conference on Management of data (SIGMOD), pp. 268–279. ACM, New York (1985)

Du, J., Wen, H. Y., Yang, Z. J.: Research on data layer sturcture of multi-tenant E-commerce system. In: Proceedings of the IEEE 17th International Conference on Industrial Engineering and Engineering Management (IE&EM), pp. 362–365. IEEE (2010)

Elmasri, R., Navathe, S.B.: Database Systems: Models, Languages, Design, and Application Programming. Pearson Education, Upper Saddle River (2011)

Halili, E.H.: Apache Jmeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites. Packt Publishing, Birmingham (2008)

Jarke, M., Koch, J.: Query optimization in database systems. ACM Comput. Surv. **16**(4), 111–152 (1984)

Koziolek, H.: The SPOSAD architectural style for multi-tenant software applications. In: Proceedings of the 9th Working IEEE/IFIP Conferences on Software Architecture, pp. 320–327. IEEE (2011)

Krebs, R., Momm, C., Konev, S.: Architectural concerns in multi-tenant SaaS applications. In: Proceedings of the International Conference on Cloud Computing and Service Science (CLOSER12) (2012)

Li, C.: Transforming relational database into HBase: a case study. In: Proceedings of the IEEE International Conference on Software Engineering and Service Sciences (ICSESS). IEEE (2010)

Li, H., Yang, D., Zhang, X.H.: Survey on multi-tenant data architecture for SaaS. Int. J. Comput. Sci. Issues **9**(6–3), 198–204 (2012)

Liao, C. F., Chen, K., Chen, J. J.: Toward a tenant-aware query rewriting engine for universal table schema-mapping. In: Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom'12), Presented in 2012 International Workshop on SaaS (Software-as-a-Service) Architecture and Engineering, pp. 833–838. IEEE (2012)

Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, Burlington (1996)

Maier, D., Ullman, J.D.: Maximal objects and the semantics of universal relation databases. ACM Trans. Database Syst. **8**(11), 1–14 (1983)

OSGi Core Release 5, The OSGi Alliance (2012)

Pereira, J., Chiueh, T.C.: SQL Rewriting Engine and its Applications. Technical Report. Stony Brook University (2007)

Russell, C.: Java Data Objects 2.0. JSR 243 Specification (2010)

Weissman, C. D., Bobrowski, S.: The design of the Force.com multitenant internet application development platform. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM, New York (2009)

Yaish, H., Goyal, M., Feuerlicht, G.: An elastic multi-tenant database schema for software as a service. In: Proceedings of the IEEE International Conference on Dependable, Autonomic and Secure Computing, pp. 737–743. IEEE (2011)