Proxy Service for Multi-tenant Database Access

Haitham Yaish, Madhu Goyal, George Feuerlicht

To cite this version:

Haitham Yaish, Madhu Goyal, George Feuerlicht. Proxy Service for Multi-tenant Database Access. 1st Cross-Domain Conference and Workshop on Availability, Reliability, and Security in Information Systems (CD-ARES), Sep 2013, Regensburg, Germany. pp.100-117. hal-01506783

HAL Id: hal-01506783
https://inria.hal.science/hal-01506783

Submitted on 12 Apr 2017

Proxy Service for Multi-tenant Database Access

Haitham Yaish[1,2], Madhu Goyal[1,2], and George Feuerlicht[2,3]

[1] Centre for Quantum Computation & Intelligent Systems
[2] Faculty of Engineering and Information Technology
University of Technology, Sydney
P.O. Box 123, Broadway NSW 2007, Australia
[3] Faculty of Information Technology,
University of Economics, Prague, Czech Republic

{ haitham.yaish@student.uts.edu.au, madhu@it.uts.edu.au, george.feuerlicht@uts.edu.au}

Abstract. The database of multi-tenant Software as a Service (SaaS) applications has challenges in designing and developing a relational database for multi-tenant applications. In addition, combining relational tables and virtual relational tables to make them work together and act as one database for each single tenant is a hard and complex problem to solve. Based on our multi-tenant Elastic Extension Tables (EET), we are proposing in this paper a multi-tenant database proxy service to combine multi-tenant relational tables and virtual relational tables, to make them act as one database for each single tenant. This combined database is suitable to run with multi-tenant SaaS single instance applications, which allow tenants designing their database and automatically configuring its behavior during application runtime execution. In addition, these applications allow retrieving tenants data by simply calling functions from this service which spare tenants from spending money and efforts on writing SQL queries and backend data management codes, and instead allowing them to focus on their business and to create their web, mobile, and desktop applications. The feasibility and effectiveness of the proposed service are verified by using experimental data on some of this service functions.

Keywords: Software as a Service, SaaS, Multi-tenancy, Multi-tenant Database, Relational Tables, Virtual Relational Tables, Elastic Extension Tables.

1 Introduction

Configuration is the main characteristic of multi-tenant applications that allow SaaS vendors running a single instance application, which provides a means of configuration for multi-tenant applications. This characteristic requires a multi-tenant aware design with a single codebase and metadata capability. Multi-tenant aware application allows each tenant to design different parts of the application, and automatically adjust and configure its behavior during runtime execution without redeploy the application [3]. Multi-tenant data has two types: shared data, and tenant's isolated data. By combining these data together tenants can have a complete data which suits their business needs [5][11].

There are various models of multi-tenant database schema designs and techniques which have been studied and implemented to overcome multi-tenant database challenges [14]. Nevertheless, these techniques are still not overcoming multi-tenant database challenges [1]. NoSQL stands for Not Only Structured Query Language, is a non-relational database management system. This technique avoids join operations, filtering on multiple properties, and filtering of data based on the results of a subqueris. Therefore, the efficiency of NoSQL simple query is very high, but this is not the case for complex queries [4][10][6]. Salesforce.com [13], the pioneer of SaaS Customer Relationship Management (CRM) applications has developed a storage model to manage its virtual database structure by using a set of metadata, universal data table, and pivot tables. Also, it provides a special object-oriented procedural programming language called Apex, and two special query languages: Sforce Object Query Language (SOQL) and Sforce Object Search Language (SOSL) to configure, control, and query the data from Salesforce.com storage model [9].

We have proposed a novel multi-tenant database schema design to create and configure multi-tenant applications, by introducing an Elastic Extension Tables (EET) which consists of Common Tenant Tables (CTT) and Virtual Extension Tables (VET) . The database design of EET technique is shown in the Appendix. This technique enables tenants creating and configuring their own virtual database schema including: the required number of tables and columns, the virtual database relationships for any of CTTs or VETs, and the suitable data types and constraints for a table columns during multi-tenant application run-time execution [14]. In this paper, we are proposing a multi-tenant database proxy service called Elastic Extension Tables Proxy Service (EETPS) to combine, generate, and execute tenants' queries by using a codebase solution that converts multi-tenant queries into a normal database queries.

Our EETPS provides the following new advancements:

- Allowing tenants to choose from three database models. First, multi-tenant relational database. Second, combined multi-tenant relational database and virtual relational database. Third, virtual relational database.
- Avoiding tenants from spending money and efforts on writing SQL queries, learning special programing languages, and writing backend data management codes by simply calling functions from our service which retrieves simple and complex queries including join operations, filtering on multiple properties, and filtering of data based on subqueries results.

In our paper, we explored two sample algorithms for two functions of our service, and we carried out four types of experiments to verify the practicability of our service.

The rest of the paper is organized as follows: section 2 reviews related work. Section 3 describes Elastic Extension Tables Proxy Service, section 4 describes two sample algorithms of the Elastic Extension Tables Proxy Service, section 5 gives our experimental results and section 6 concludes this paper and descries the future work.

2 Related Work

There are various models of multi-tenant database schema designs and techniques which have been studied and implemented to overcome multi-tenant database challenges like Private Tables, Extension Tables, Universal Table, Pivot Tables, Chunk Table, Chunk Folding, and XML [1][2][7][8][14]. Nevertheless, these techniques are still not overcoming multi-tenant database challenges [1]. Salesforce.com, the pioneer of SaaS CRM applications has designed and developed a storage model to manage its virtual database structure by using a set of metadata, universal data table, and pivot tables which get converted to objects that the Universal Data Dictionary (UDD) keeps track of them, their fields and relationships, and other object definition characteristics. Also, it provides a special object-oriented procedural programming language called Apex which does the following. First, declare program variables, constants, and execute traditional flow control statements. Second, declare data manipulation operations. Third, declare the transaction control operations. Then Salesforce.com compiles Apex code and stores it as metadata in the UDD [13]. In addition, it has its own Query Languages, first, Sforce Object Query Language (SOQL), which retrieve data from one object at a time. Second, Sforce Object Search Language (SOSL), which retrieve data from multiple objects simultaneously [9] [13]. NOSQL is a non-relational database management system which designed to handle storing and retrieving large quantities of data without defining relationships. It has been used by cloud services like MongoDB, Cassandra, CouchDB, Google App Engine Datastore, and others. This technique avoids join operations, filtering on multiple properties, and filtering of data based on subqueries results. Therefore the efficiency of its simple query is very high, but this is not the case for complex queries. Moreover, unless configuring NoSQL consistency models in protective modes of operation, NoSQL will not assure the data consistency and it might sacrifice data performance and scalability [4][10]. Indrawan-Santiago [13] states that NoSQL should be seen as a complimentary solution to relational databases in providing enhanced data management capability, not as a replacement to them.

3 Elastic Extension Tables Proxy Service

In this paper, we are proposing a multi-tenant database proxy service to combine, generate, and execute tenants' queries by using a codebase solution which converts multi-tenant queries into normal database queries. This service has two objectives, first, to enable tenants' applications retrieve tuples from CTTs, retrieve combined tuples from two or more tables of CTTs and VETs, or retrieve tuples from VETs. Second, to spare tenants from spending money and efforts on writing SQL queries and backend data management codes by simply calling functions from this service, which retrieves simple and complex queries including join operations, union operations, filtering on multiple properties, and filtering of data based on subqueries results.

This service gives tenants the opportunity of satisfying their different business needs and requirements by choosing from any of the following three database models which are also shown in Fig.1.

- Multi-tenant relational database: This database model eligible tenants using a ready relational database structure for a particular business domain database without any need of extending on the existing database structure, and this business domain database can be shared between multiple tenants and differentiate between them by using a Tenant ID. This model can be applied to any business domain database like: CRM, Accounting, Human Resource (HR), or any other business domains.
- Combined multi-tenant relational database and virtual relational database: This database model eligible tenants using a ready relational database structure of a particular business domain with the ability of extending on this relational database by adding more virtual database tables, and combine these tables with the existing database structure by creating virtual relationships between them.
- Multi-tenant virtual relational database: This database model eligible tenants using their own configurable database through creating their virtual database structures from the scratch, by creating virtual database tables, virtual database relationships between the virtual tables, and other database constraints to satisfy their special business requirements for their business domain applications.

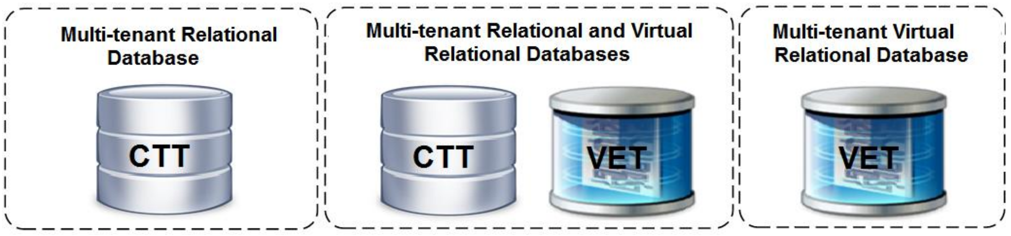

Multi-tenant Relational Database
CTT
Multi-tenant Relational and Virtual Relational Databases
CTT
VET
Multi-tenant Virtual Relational Database
VET

Fig. 1. EETPS database models.

The EETPS provides functions which allow tenants building their web, mobile, and desktop applications without the need of writing SQL queries and backend data management codes. Instead, retrieving their data by simply calling these functions, which return a two dimensional array (Object [α] [β]), where α is the number of array rows that represents a number of retrieved tuples, and β is the number of array columns that represents a number of retrieved columns for a particular virtual table. These functions were designed and built to retrieve tenants' data from the following tables:

- One table either a CTT or a VET.

- Two tables which have one-to-one, one-to-many, many-to-many, or self-referencing relationships. These relationships can be between two VETs, two CTTs, or one VET and one CTT.
- Two tables which may have or not have relationships between them, by using different types of joins including: Left Join, Right Join, Inner Join, Outer Join, Left Excluding Join, Right Excluding Join, and Outer Excluding Join. The Join operation can be used between two VETs, two CTTs, or one VET and one CTT.
- Two tables or more which may have or not have relationships between them, by using the union operator that combines the result-set of these tables whether they are CTTs or VETs.
- Two or more tables which have relationships between them, by using filters on multiple tables, or filtering data based on the results of subqueries.

Moreover, the EETPS functions have the capabilities of retrieving data from CTTs or VETs by using the following query options: Logical Operators, Arithmetic operators, Aggregate Functions, Mathematical functions, Using Single or Composite Primary Keys, Specifying Query SELECT clauses, Specifying Query WHERE Clause, Specifying Query Limit, and Retrieving BLOB and CLOB Values.

4 Sample Algorithms of the Elastic Extension Tables Proxy Service

In this section, two sample algorithms will be explored, Single Table Algorithm, and Union Tables Algorithm.

4.1 Single Table Algorithm

This algorithm retrieves tuples from a CTT or a VET. There are three different cases in this algorithm, first, retrieving tuples from a VET by specifying certain primary keys. Second, retrieving tuples from a VET by specifying certain table row IDs which are stored in 'table_row' extension table. Third, retrieving all tuples of a CTT or a VET. In this section we will explore the main algorithm and some of the subsidiary algorithms of the Single Table Algorithm including: the algorithm of the second case that mentioned in this paragraph, and Store Tuples in Array Algorithm. In addition, we will explore an example for each of these algorithms.

Single Table Main Algorithm. This main algorithm is outlined in Program Listing 1. The algorithm determines which of the three cases mentioned above will be applied by checking the passed parameters, and based on these parameters one of a three different query statement will be constructed, and then this query statement will be passed to 'getQuery' algorithm which will return SQL query results from 'table_row', 'table_row_blob', and 'table_row_clob' extension tables and store these results in a set. Then, this set will be passed to Store Tuples in Array Algorithm which will store the results in a two dimensional array, where the number of array rows represents a number of retrieved tuples, and the number of array columns represents a number of retrieved columns for a particular table.

Definition 1 (Single Table Main Algorithm). T denotes a tenant ID, B denotes a table name, λ denotes a set of table row IDs, Ω denotes a set of primary keys, S denotes a string of the SELECT clause parameters, W denotes a string of the WHERE clause, F denotes a first result number of a query limit, M denotes the maximum amount of a query limit which will be retrieved, Q denotes the table type (CTT or VET), I denotes a set of VET indexes, C denotes a set of retrieved tuples from a CTT, V denotes a set of retrieved tuples from a VET, and Φ denotes a two dimensional array that stores the retrieved tuples.

Input.T, B, λ, Ω, S, W, F, M and Q.
Output. Φ.

```
1. if Q = CTT then
2.    C ← retrieve tuples from a CTT by using T, Ω, S,
      W, F, and M to filter the query results
3. else
4.   if Ω ≠ null then
5.      V ← retrieve tuples from a VET by using T, Ω, S,
        W, F,   and M to filter the query results
6.   else if λ ≠ null then
7.  /* This statement calls Table Row Query Algorithm */
8.      V ← retrieve tuples from a VET by using T, λ, S,
        W, F, and M to filter the query results
9.   else
10.     I ← retrieve the indexes of B by using ta-
        ble_index extension table
11.  end if
12.  if B has I then
13.      V ← retrieve tuples from a VET by using T, I,
         S, W, F, and M to filter the query results
14.   else
15.      V ← retrieve tuples from a VET by using T, S,
         W, F, and M to filter the query results
16.   end if
17. end if
18. /* This statement calls Store Tuples in Array Algo-
rithm */
19. store C or V in Φ
20. Return Φ
```

[1] The program listings of Single Table Algorithm.

Table Row Query Algorithm. This subsidiary query algorithm is used to retrieve tuples for a tenant from a VET. The database query which is used in this algorithm uses UNION operator keyword to combine the result-set of three SELECT statements for three tables: table_row, table_row_blob, and table_row_clob if the VET only contains BLOB and/or CLOB, however if the VET does not contain BLOB and CLOB then the UNION operator will not be used in the query.

Definition 2 (Table Row Query Algorithm). T denotes a tenant ID, B denotes a table name, λ denotes a set of table row IDs, S denotes a string of the SELECT clause parameters, W denotes a string of the WHERE clause, F denotes a first result number of the query limit, M denotes the maximum amount of the query limit which will be retrieved, Q denotes the table type (CTT or VET), and θ denotes a string of the select statement.

Input. T, B, λ, Ω, S, W, F, M and Q.
Output. θ.

```
1. θ = SELECT tr.table_column_name, tr.value,
tr.table_row_id, tr.serial_id FROM table_row tr WHERE
tr.tenant_id = T AND tr.db_table_id = B AND
tr.table_row_id IN (λ) AND table_column_id in (S) AND W
UNION
SELECT trb.table_column_name, trb.value,
trb.table_row_blob_id,trb.serial_id FROM table_row_blob
trb WHERE trb.tenant_id = T AND trb.db_table_id = B AND
trb.table_row_blob_id IN (λ)
UNION
```

```
SELECT  trc.table_column_name, trc.value,
trc.table_row_clob_id, trc.serial_id FROM table_row_clob
trc WHERE trc.tenant_id = T AND trc.db_table_id = B AND
trc.table_row_clob_id IN (λ)
ORDER BY 3, 4 LIMIT M OFFSET F
2. Return θ
```

[2] The program listings of Table Row Query Algorithm.

Store Tuples in Array Algorithm. This subsidiary algorithm is used to store the retrieved data from a CTT or a VET into a two dimensional array, the number of array rows represents a number of retrieved tuples, and the number of array columns represents a number of retrieved columns for a table. The column names get stored in the first element of this two dimensional array, and the data in these columns get stored in the rest elements of the array.

Definitions 3 (Store Tuples in Array Algorithm). T denotes a tenant ID, B denotes a table name, μ denotes a set of retrieved tuples from a CTT or a VET where each of these tuples is presented as τ and each column of τ is presented as χ , which means τ is a set of χ where
τ = { χ1, χ2, …, χn}. δ denotes a set of column names of a CTT or a VET, Φ denotes a two dimensional array to store the retrieved tuples, and τ n (χ m) denotes a value stored in χ m of τ n.
Input. T, B, and μ.
Output. Φ.

```
1. δ ← retrieve the column names of B from table_column
    extension table by using T to filter the query re-
    sults
2. Initialize Φ [size of μ] [size of δ]
3. i ← 0
4. For all column names ∈ δ Do
5.    Φ [0][i] = δi
6.    i ← i + 1
7. end for
8. n ← 0
9. for all τ ∈ μ Do
10.  m ← 0
11.  For all column names ∈ δ Do
12.    Φ [n+1][m] = τ n(χ m)
13.    m ← m + 1
14.  end for
15.  n ← n + 1
16.end for
17.Return Φ
```

[3] The program listings of Store Tuples in Array Algorithm.

Example. This example explores how the Single Table Algorithm retrieves virtual tuples from one VET. There are three cases that this algorithm is handling which mentioned above in this section. In this example we will explore the case where we pass a certain table Row ID to the algorithm. In this example we will pass the following five input parameters:

1. A tenant ID value, which equals 100.
2. A table ID value, which equals 7.
3. A table row ID value, which equals 2.
4. The SELECT clause parameter (S) is empty, this means that the query will retrieve all the columns of the 'store' VET.

5. The WHERE clause (W) is empty, this means that the query is not filtered by the WHERE clause.

Fig. 2 (a) shows the 'store' VET which we will retrieve tuples from. The query in Program Listing 4 is generated by using the Single Table Algorithm to retrieve a virtual tuple from the 'store' VET based on the passed parameters. Fig. 2 (b) shows the result of the virtual tuples that retrieved from table_row extension table by using this query listed in Program Listing 4. This virtual tuple is divided into three physical tuples, each of these physical tuples stores a column name and its value, and all of these tuples are sharing one 'table_row_id' which equals 2. The query in Program Listing 4 does not contain the UNION part of the query to retrieve BLOB and CLOB values because the 'store' VET structure does not contain any of them.

The two dimensional array that is shown in Fig. 2 (c) illustrates how the previous result which is shown in Fig. 2 (b) is stored in a well structured two dimensional array. The column names are stored in the first row elements, and the first tuple is stored in the second row elements of the array. Compared with the previous results of the tuples that is shown in Fig. 2 (b), this two dimensional array stores the virtual tuple in a structure which is very similar to any physical tuple that is structured in any physical database table, which in return will facilitate accessing virtual tuples from anyVET.

```
SELECT  tr.table_column_name, tr.value,
tr.table_row_id, tr.serial_id FROM  table_row tr WHERE
tr.tenant_id  =  100  AND  tr.db_table_id  =  7   AND
trb.table_row_id IN (2)
```

[4] The Program Listing of the query generated by using the Single Table Algorithm.

(a)

store_id	name	phone

(b)

table_column_name	value	table_row_id	serial_id
store_id	2	2	1
name	George Street Store	2	2
phone	+61294455331	2	3

(c)

	0	**1**	**2**
0	name	phone	store_id
1	George Street Store	+61294455331	2

Fig. 2. The 'store' VET and some tuples retrieved from it and stored in an array.

4.2 Union Tables Algorithm

In this section, we will explore the union function, which retrieves a combined result-set of two or more tables whether they are CTTs or VETs, and stores the result-set in an array. In addition, we will explore an example of this algorithm. The input parameters of this algorithm will determine a tenant, a set of CTTs and/or VETs that the union function needs to retrieve data from, SELECT clauses, and WHERE clauses which are required for each table. Program Listing 5 is showing the detailed algorithm. This algorithm will store the retrieved tuples in an array by using the subsidiary algorithm that mentioned in the Program Listings 3.

Definition 4 (Union Tables Algorithm). T denotes a tenant ID, Π denotes a set of CTTs and VET names, where each of these tables has got one or more tuples (Π = { τ 1, τ 2, …, τ m}), each tuple is presented as τ and each column of τ is presented as χ, which means τ is a set of χ where τ = { χ1, χ2, …, χn}. υ denotes a set of table columns which are related to the set Π and the columns are ordered according to the table orders, W denotes a set of WHERE clauses which are related to the set Π and the columns are ordered according to the table orders of Π, F denotes a first result number of a query limit, M denotes a maximum amount of a query limit which will be retrieved, Q denotes the table type (CTT or VET), C denotes a set of retrieved tuples from CTT, V denotes a set of retrieved tuples from VET, Φ denotes a two dimensional array which stores the retrieved tuples, and τ n (χ m) denotes a value stored in χ m of τ n.

Input. T, Π, υ, W, F, and M.

Output. Φ.

```
i ← 0
For all tables ∈ Π Do
  if Q = CTT then
     C ← retrieve τ from a CTT by using υ, W, F, and M
     to  filter the query results
  else
     V ← retrieve tuples from a VET by using υ, W, F,
     and (M * size of υ) to filter the query results
  end if
  n ← 0
  for all τ ∈ Πi Do
    m ← 0
    For all column names ∈ τ Do
      Φ[n+1][m] = τ n(χ m)
       m ← m + 1
   end for
   n ← n + 1
  end for
  i ← i + 1
end for
Return Φ
```

5 The program listings of Union Tables Algorithm.

Example. This example explores how the Union Table Algorithm retrieves tuples from two tables, the first one CTT and the second one VET. In this example we will pass to the algorithm the following six input parameters:

1. A tenant ID value, which equals 1000.
2. A set of table IDs (Π) which equals {product, 17} where 'product' is a CTT that is shown in Fig. 3(a) and the ID 17 is the ID which represents the 'sales_fact' VET that is shown in Fig. 3 (b).
3. A set of table columns (υ), which equals {{shr_product_id, price}, {58,61}}, where this set contains two other sets, the first one contains the columns of 'product' CTT, and the second one contains the IDs of 'sales_fact' VET. ID 58 represents the virtual 'product_id' column and ID 61 represents the virtual 'unit_price' column.
4. The set of WHERE clauses of the tables (W) are empty, because this example has not got any WHERE clauses parameter passed to the function to filter the tables queries.
5. The first number of the query limits (F), which equals 0.
6. The maximum amount of the query limits (M), which equals 1.

After we passed the parameters to the function, the function iterated the set of tables (Π) , the first table in the set was 'product_id' CTT, the function executed the query which is shown in Program Listing 6 to retrieve the tuples of this table, and the results of this query are shown in Fig. 3 (c). The second table in the set was the ‘sales_fact’ VET with ID equals 17, the function executed the query in Program Listing 7 and 8. The query in Program Listing 7 was used to retrieve the indexes of the ‘sales_fact’ VET from ‘table_index’ extension table, and the query in Program Listing 8 was used to retrieve the virtual tuples from ‘sales_fact’ VET by using the passed parameters and the ‘table_row_id’ which were retrieved from the query that shown in Program Listing 7. The results of the two queries of Program Listing 7 and 8 are shown in Fig. 3 (d) and (e).

Finally, the output of the queries of the CTT and the VET that mentioned above are stored in two dimensional array as shown in Fig. 3 (f), the two elements [0] [0] and [0] [1] represent the column names, the Union functions shows generic names like column1, and column 2, however the other functions which our service provides show column names of CTT and VET. The two elements [1] [0] and [1] [1] represent the column’s values of the CTT, and the two elements [2] [0] and [2] [1] represent the column’s values of the VET.

```
SELECT product_id, price FROM product WHERE tenant_id =
1000 LIMIT 1;
```

[6] The program listing of the query which retrieved the tuples of the ‘product’ CTT.

```
SELECT    table_row_id   FROM   table_index   WHERE   ten-
ant_id=1000  AND  db_table_id=17  AND  (table_column_id=61
OR table_column_id=58) LIMIT 1
```

[7] The program listing of the query which retrieved the indexes of the ‘sales_fact’ VET from ‘table_index’ extension table.

```
SELECT  tr.table_column_id  ,tr.value  ,tr.table_row_id,
tr.serial_id FROM table_row tr WHERE tr.tenant_id =1000
AND tr.db_table_id = 17 AND tr.table_row_id IN (352871)
AND tr.table_column_id in (58,61)
ORDER BY 3,4 LIMIT 2 OFFSET 0
```

[8] The program listing of the query which retrieved the tuples of the ‘sales_fact’ VET.

(a)

product_id	tenant_id	product_bus_id	standard_cost	price	…

(b)

sales_fact_id	tenant_id	product_id	customer_id	sales_person_id	unit_price	…

(c)

product_id	price
100	5714.87

(d)

table_row_id
352871

(e)

table_column_id	value	table_row_id	serial_id
58	100	352871	4
61	15786	352871	7

(f)

	0	1
0	column 1	column 2
1	100	5714.87
2	100	15786

Fig. 3. The ‘product’ CTT and the ‘sales_fact’ VET data structures.

5 Performance Evaluation

After developing the EETPS, we carried out four types of experiments to verify the practicability of our service. These experiments were classified according to the complexities of the queries which used in these experiments including: simple, simple-to-medium, medium, and complex. The four experiments show comparisons between the response time of retrieving data from CTTs, VETs, or both CTTs and VETs. We have evaluated the response time through accessing the EETPS which converts multi-tenant queries into normal database queries, instead of accessing the database directly.

5.1 Experimental Data Set

The EETPS has designed and developed to serve multi-tenants in one instance application. However, in this paper the aim of the experiments is to evaluate the performance differences between retrieving the data of CTTs, VETs, or both CTTs and VETs together for one tenant. In our experiment settings we used one machine and we ran the following four types of experiments:

- Simple query experiment (Exp. 1): In this experiment we called a function which retrieved data from a CTT by executing Query 1 (Q1), and retrieved the same data from a VET by executing Query 2 (Q2).
- Simple-to-medium query experiment (Exp. 2): In this experiment we called a function which retrieved data from two CTTs by executing Query 3 (Q3), two VETs by executing Query 4 (Q4), and CTT-and-VET by executing Query 5 (Q5). Each of these two tables combination has got one-to-many relationship between them.
- Medium query experiment (Exp. 3): In this experiment we called a function which retrieved data from two tables by using a union operator for two CTTs by executing Query 6 (Q6), for two VETs by executing Query 7 (Q7), and for CTT-and-VET by executing Query 8 (Q8).
- Complex query experiment (Exp. 4): In this experiment we called a function which uses a left join operator that joined two CTTs by executing Query 9 (Q9), two VETs by executing Query 10 (Q10), and CTT-and-VET by Query 11 (Q11).

In these four experiments we ran the test on eleven queries twice, the first test was to retrieve only 1 tuple, and the second test was to retrieve a 100 of tuples by using the same queries. The queries that we ran on CTTs are the same queries we ran on VETs, and CTT-and-VET in order to have accurate comparisons. The structures of these queries are shown in Fig. 4. We recorded the execution time of these queries experiments based on six data sets for all the four types of experiments that we ran. The first data set contained 500 tuples, the second data set contained 5,000 tuples, the third data set contained 10,000 tuples, the fourth data set contained 50,000 tuples, the fifth data set contained 100,000 tuples, and the last data set contained 200,000 tuples. All of these data sets were for one tenant.

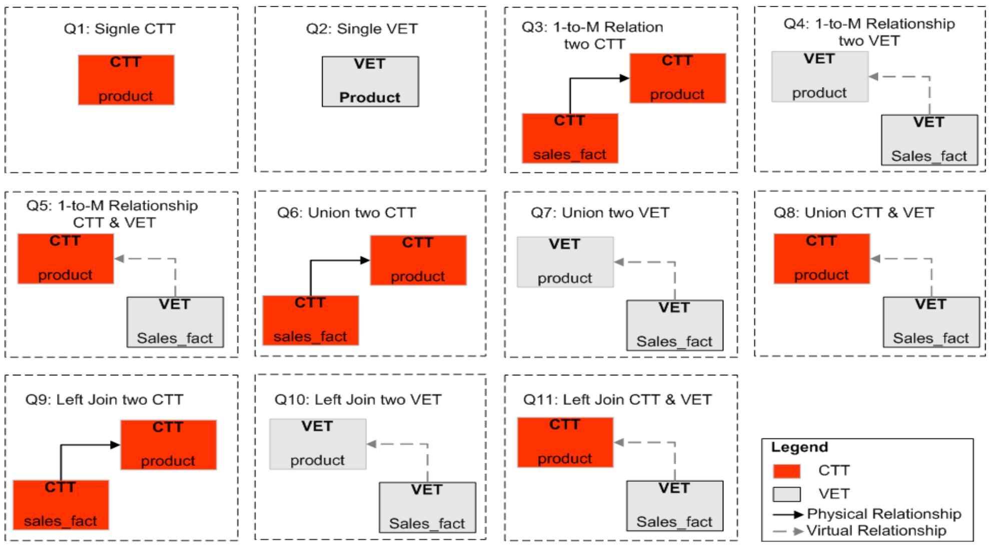


Fig. 4. The structures of the queries executed in our experiments.

5.2 Experimental Setup

Our EETPS was implemented in Java 1.6.0, Hibernate 4.0, and Spring 3.1.0. The database is PostgreSQL 8.4 and the application server is Jboss-5.0.0.CR2. Both of database and application server is deployed on the same PC. The operating system is windows 7 Home Premium, CPU is Intel Core i5 2.40GHz, the memory is 8GB, and the hard disk is 500G.

5.3 Experimental Results

In all the experimental diagrams we provided in this section the vertical axes which are the execution time in seconds, and the horizontal axes which are the total number of tuples that stored in a tenant's tables. Each of the four experiments retrieves 1 tuple and 100 of tuples, and we will show in this section the average execution time of the six data sets of these tuples which are related to CTTs, VETs, and CTTs and VETs, and show the differences between them. These experimental diagrams are shown in Fig 5-12.

We found in our experimental results that the average performance of the CTT and the VET for Exp.1 can be considered the same, and the VETs, the CTTs and the CTT-and-VET for Exp. 2 can be considered the same as well. In addition, we found that the average performance for Exp. 3 for the VETs, and the CTT-and-VET can be considered slightly higher than the CTTs, but the average performance of the VETs is the highest difference between the three types of tables. The average performance difference between the CTTs and the VETs for retrieving 1 tuple is 280 milliseconds, and for retrieving 100 tuples is 396 milliseconds. In the last experimental results Exp. 4 we found that the average performance for the CTT-and- VET can be considered higher than the CTTs by approximately 1.2 seconds, and for the VETs can be considered higher than the CTTs by approximately 1.5 seconds. The details of the experimental results summary are shown in Table 1 and 2.

Table 1. This table shows the experimental results of retrieving 1 tuple in milliseconds.

Retrieving 1 Tuple	CTT	VET	CTT-and-VET	Difference Between CTT–and-VET	Difference Between CTT and CTT-and-VET
Exp. 1	**Q 1**	**Q 2**			
	117	161		44	
Exp. 2	**Q 3**	**Q 4**	**Q 5**		
	146	155	149	9	3
Exp. 3	**Q 6**	**Q 7**	**Q 8**		
	231	511	340	280	109
Exp. 4	**Q 9**	**Q 10**	**Q 11**		
	403	1930	1632	1527	1229

Table 2. This table shows the experimental results of retrieving 100 tuples in milliseconds.

Retrieving 100 Tuples	CTT	VET	CTT-and-VET	Difference Between CTT–and-VET	Difference Between CTT and CTT-and-VET
Exp. 1	Q 1	Q 2			
	204	206		2	
Exp. 2	Q 3	Q 4	Q 5		
	331	355	343	24	12
Exp. 3	Q 6	Q 7	Q 8		
	245	641	388	396	143
Exp. 4	Q 9	Q 10	Q 11		
	560	2112	1856	1552	1296

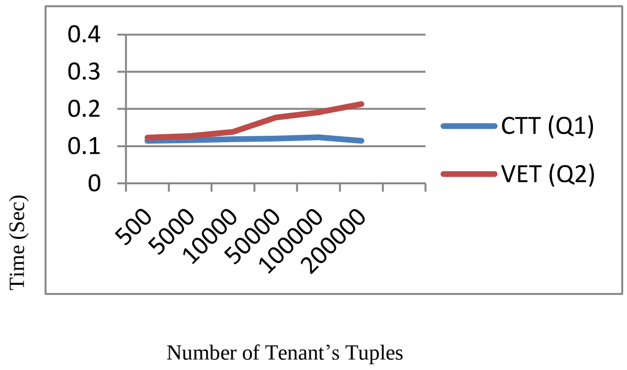


Fig. 5. Single Table 1 Tuple

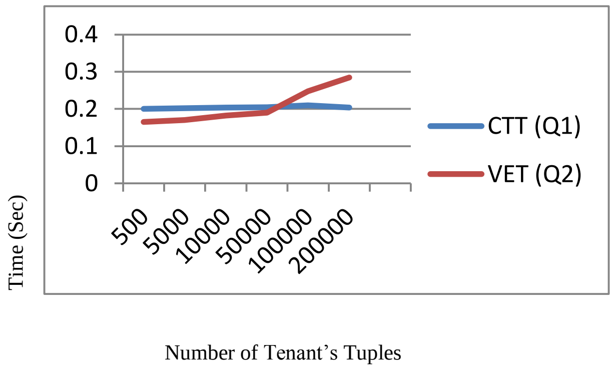


Fig. 6. Single Table 100 Tuples

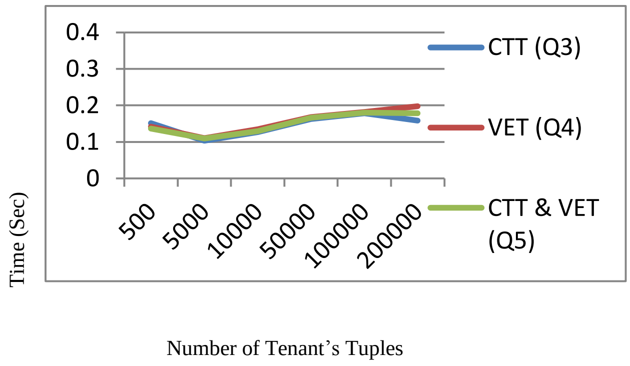


Fig. 7. 1-to-M 1 Tuple

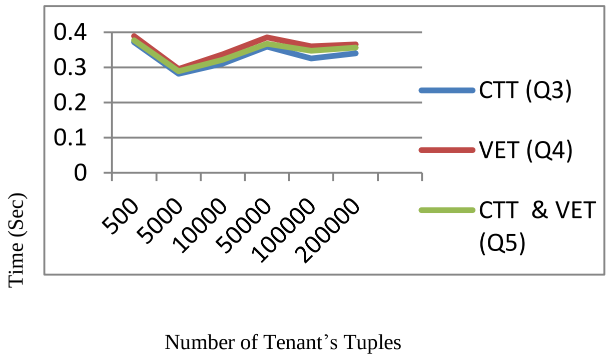


Fig. 8. 1-to-M 100 Tuples

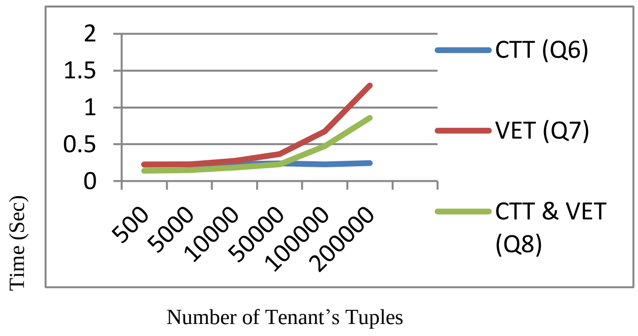


Fig. 9. Union 1 Tuple

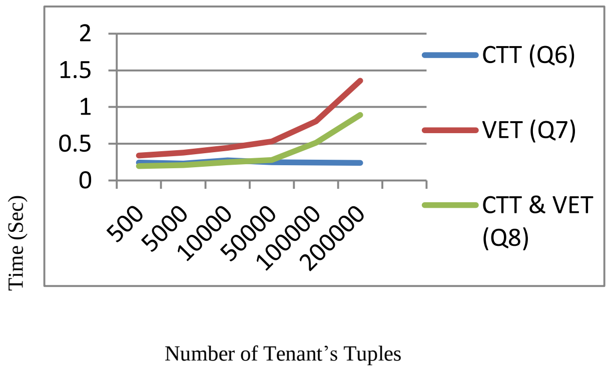


Fig. 10. Union 100 Tuples

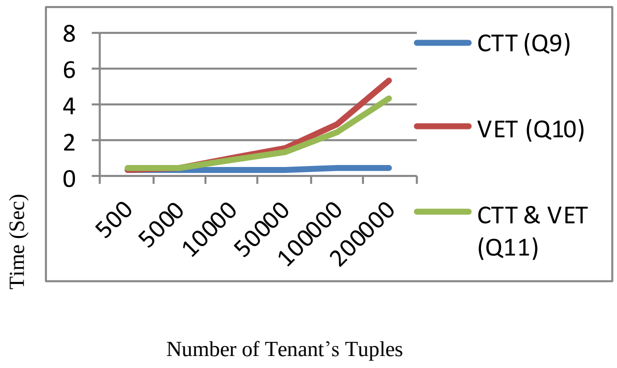


Fig. 11. Left Join 1 Tuple

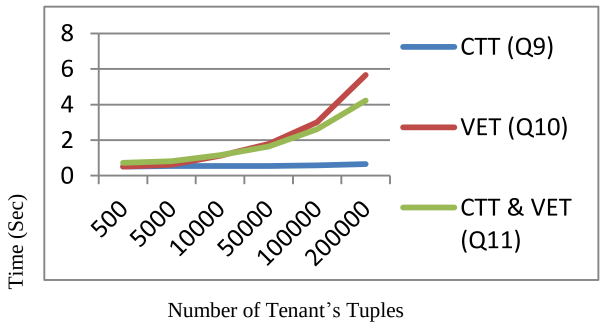


Fig. 12. Left Join 100 Tuples

6 Conclusion

In this paper, we are proposing a multi-tenant proxy service for the EET to combine, generate, and execute tenants' queries by using a codebase solution that converts a multi-tenant query into a normal database query. This service has two objectives, first, allowing tenants to choose from three database models: multi-tenant relational database, combined multi-tenant relational database and virtual relational database, and virtual relational database. Second, sparing tenants from spending money and efforts on writing SQL queries and backend data management codes by calling our service functions which retrieve simple and complex queries including join operations, filtering on multiple properties, and filtering of data based on subqueries results. In our paper, we explored two sample algorithms for two functions, and we carried out four types of experiments to verify the practicability of our service. These experiments were classified according to the complexities of the queries which used in these experiments including: simple, simple-to-medium, medium, and complex. The four experiments show comparisons between the response time of retrieving data from CTTs, VETs, or both CTTs and VETs. In our experimental results we found that the average performance of CTTs, VETs, and CTT- and-VET for the simple queries and the simple-to-medium queries are considered almost the same. Also, we found that the average performance of the medium queries for VETs, and CTT-and-VET is considered slightly higher than CTTs, but VET are the highest between the three types of tables. In the last experimental results of complex query we found that the average performance for CTT-and-VET is considered higher than CTTs by approximately 1.2 seconds, and for VETs is considered higher than CTTs by approximately 1.5 seconds. The cost of complex query is acceptable in favor of obtaining a combined relational database and virtual relational database for multi-tenant applications, which in return these combined databases provide a means of configuration for multi-tenant applications, reduce the Total Cost of Ownership (TCO) on the tenants, and reduce the ongoing operational costs on the service providers.

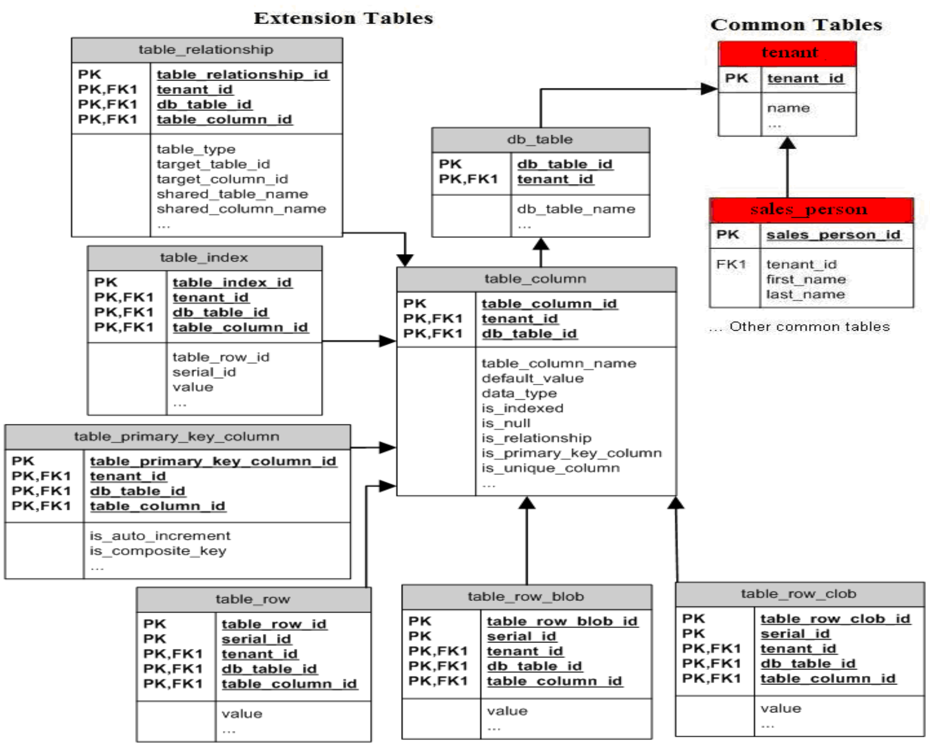
Our future work will focus on optimizing virtual data retrieval from our EET for simple and complex queries by using a highly-optimized executing query plans and logic, and add more functions to insert, updated, delete tuples from CTT and VET.

References

1. Aulbach, S., Grust, T.,Jacobs, D., Kemper, A., Seibold, M.: A Comparison of Flexible Schemas for Software as a Service. In: Proceedings of the 35th SIGMOD International Conference on Management of Data, pp. 881--888. ACM, Rhode Island (2009)
2. Aulbach, S., Grust, T.,Jacobs, D., Kemper, A., Rittinger, J.: Multitenant Databases for Software as a Service: Schema Mapping Techniques. In: Proceedings of the 34th SIGMOD International Conference on Management of Data, pp. 1195 -- 1206. ACM, Vancouver (2008)
3. Bezemer, C., Zaidman, A.: Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare?. In: Proceedings of the Joint Workshop on Software Evolution and International Workshop on Principles of Software Evolution, pp. 88--92. ACM, Antwerp (2010)
4. Bobrowski, S.: Optimal Multitenant Designs for Cloud Apps. In: 4th International Conference on Cloud Computing, pp. 654--659. IEEE Press, Washington (2012)
5. Domingo, E.J., Nino, J.T., Lemos, A.L., Lemos, M.L., Palacios, R.C., Berbí, s, J.M.G.: CLOUDIO: A Cloud Computing-Oriented Multi-tenant Architecture for Business Information Systems. In: 3rd International Conference on Cloud Computing, pp. 532--533. IEEE Press, Madrid, (2010)
6. Dimovski, D.: Database management as a cloud-based service for small and medium organizations. Master Thesis, Masaryk University Brno (2013)
7. Du, J., Wen, H. Y.,Yang, Z. J.: Research on Data Layer Structure of Multi-tenant E-commerce System. In: IEEE 17th International Conference on Industrial Engineering and Engineering Management, pp. 362 – 365, Xiamen (2010)
8. Foping, F. S., Dokas, I. M., Feehan, J., Imran, S.: A New Hybrid Schema-sharing Technique for Multitenant Applications. In: Fourth International Conference on Digital Information Management, pp. 1 -- 6, IEEE Press, Michigan (2009)
9. Force.com, http://www.salesforce.com/us/developer/docs/soql_sosl/salesforce_soql_sosl.pdf

10. Google Developers, https://developers.google.com/appengine/docs/python/datastore/overview#Comparison_with_Traditional_Databases.
11. Guoling, L.: Research on Independent SaaS Platform. In: The 2nd IEEE International Conference on Information Management and Engineering, pp. 110--113. IEEE Press, Chengdu (2010)
12. Indrawan-Santiago M.: Database Research: Are We at a Crossroad? Reflection on NoSQL. In: 15th International Conference on Network-Based Information Systems, pp. 45--51. IEEE Press, Melbourne (2012)
13. Weissman C. D., Bobrowski S.: The design of the force.com multitenant internet application development platform. In: Proceedings of the 35th SIGMOD international conference on Management of data, pp. 889--896. ACM, Rhode Island (2009)
14. Yaish, H., Goyal, M., Feuerlicht, G.: An Elastic Multi-tenant Database Schema for Software as a Service. In: Ninth IEEE International Conference on Dependable, Autonomic and Secure Computing, pp. 737--743. IEEE Press, Sydney (2011)

Appendix: Elastic Extension Tables (EET)


Extension Tables
Common Tables
tenant
PK tenant_id
name ...
sales_person
PK sales_person_id
FK1 tenant_id first_name last_name
... Other common tables
table_relationship
PK table_relationship_id
PK,FK1 tenant_id
PK,FK1 db_table_id
PK,FK1 table_column_id
table_type target_table_id target_column_id shared_table_name shared_column_name ...
db_table
PK db_table_id
PK,FK1 tenant_id
db_table_name ...
table_index
PK table_index_id
PK,FK1 tenant_id
PK,FK1 db_table_id
PK,FK1 table_column_id
table_row_id serial_id value ...
table_column
PK table_column_id
PK,FK1 tenant_id
PK,FK1 db_table_id
table_column_name default_value data_type is_indexed is_null is_relationship is_primary_key_column is_unique_column ...
table_primary_key_column
PK table_primary_key_column_id
PK,FK1 tenant_id
PK,FK1 db_table_id
PK,FK1 table_column_id
is_auto_increment is_composite_key ...
table_row
PK table_row_id
PK serial_id
PK,FK1 tenant_id
PK,FK1 db_table_id
PK,FK1 table_column_id
value ...
table_row_blob
PK table_row_blob_id
PK serial_id
PK,FK1 tenant_id
PK,FK1 db_table_id
PK,FK1 table_column_id
value ...
table_row_clob
PK table_row_clob_id
PK serial_id
PK,FK1 tenant_id
PK,FK1 db_table_id
PK,FK1 table_column_id
value ...