A Multi-tenant Database Framework for Software and Cloud Computing Applications

A Thesis Submitted for the Degree of
Doctor of Philosophy in Computing Sciences

By

Haitham Yaish

Faculty of Engineering and Information Technology
UNIVERSITY OF TECHNOLOGY, SYDNEY
Australia
July 2014

CERTIFICATE OF AUTHORSHIP/ ORIGINALITY

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of the requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Haitham Yaish

Date: 14/07/2014

ACKNOWLEDGMENT

I wish to express my great thanks to all who gave me tremendous support and help during my PhD study primarily to the following.

First and foremost, I would like to express my ultimate acknowledgment to God, who has given me the inspiration and strength to accomplish this thesis beside the time that I spent working in the IT industry during the years of this study. Being a full-time PhD student and at the same time, full-time employee would be impossible without the assistance and the help of God.

I would like to express the deepest gratitude to my principal supervisor Dr. Madhu Goyal, and co-supervisor Dr. George Feuerlicht for their academic guidance, insight, and encouragement throughout every milestone in this study.

I would like to acknowledge my highest appreciation to my parents, Hashim and Sanaa, my brother Abdulatif, and my sisters Mona, Maha and May for their constant support, prayers, inspiration, and best wishes during this study and through all my life.

I would like to express my great appreciations to my wife, Heba. I could never have accomplished this thesis without her love, encouragement, patience, understanding, and prayers. She shared me all the good and difficult times during my study. I also would like to express my thanks and love to my sons, Yazan and Mohammad for doing their best to understand the situation of their father who had to be busy between study and work for such a long time.

I would like to express my appreciations of the support and help that I got from the Centre for Quantum Computation and Intelligent Systems (QCIS), School of Software, Faculty of Engineering and Information Technology (FEIT) at the University of Technology, Sydney.

Finally, I should not forget to thank all the teachers and lecturers who taught me during the school years, Bachelor degree, Master degree, and PhD degree. The knowledge that I gained from all of them was essential to complete this study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Cloud Computing is a new computing paradigm that transforms accessing computing resources from internal data centres to external service providers. This approach is rapidly becoming a standard for offering cost effective and elastic computing services that are used over the internet. Software as a service (SaaS) is one of the Cloud Computing service models that exploits economies of scale for SaaS service providers by offering the same software and computing environment for multiple tenants. This contemporary multi-tenant service requires a multi-tenant database design that can accommodate data for multiple tenants in one single database schema. Due to multi-tenant database resource sharing in this service, the multi-tenant schema should be highly secured, optimized, configurable, and extendable during runtime execution to fulfil the applications' requirements of different tenants. However, traditional Relational Database Management Systems (RDBMS) do not support such multi-tenant database schema capabilities, and it is a significant challenge to enable RDBMS to support these capabilities. Therefore, one solution is using an intermediate software layer that mediates multi-tenant applications and RDBMS, to convert multi-tenant queries into regular database queries, and to execute them in a RDBMS. Developing such a multi-tenant software layer to manage and access tenants' data is a hard and complex problem to solve and has significant complexities that involve longer development lifecycle.

There are two main contributions of this thesis. Firstly, a proposal for a novel multi-tenant schema technique called Elastic Extension Tables (EET). Secondly, a proposal for a multi-tenant database framework prototype to implement EET schema

in a RDBMS. This approach can be used to develop a software layer that mediates software applications and a RDBMS. This software layer aims to facilitate the development of software applications, and multi-tenant SaaS and Big Data applications for both cloud service providers and their tenants.

Extensive experiments were conducted to evaluate the feasibility and effectiveness of EET multi-tenant database schema by comparing it with Universal Table Schema Mapping (UTSM), which is commercially used. Significant performance improvements obtained using EET when compared to UTSM, makes the EET schema a good candidate for implementing multi-tenant databases and multi-tenant applications. Furthermore, the prototype of the EET framework was developed, and several experiments were performed to verify the practicability and the effectiveness of using this framework that based on EET multi-tenant database schema. The results of the experiments indicate that the EET framework is suitable for the development of software applications in general, and multi-tenant SaaS and Big Data applications in particular.

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

Cloud computing has recently emerged as a new computing paradigm that transforms the IT industry, making the computing software and hardware more appealing to use as a service over the internet (Brian et al. 2012). This new computing paradigm began to flourish for two reasons. First, the internet has become affordable and its speed has significantly increased (Wang et al. 2008). Second, the wide growth in computer usage, in areas such as businesses, governments, health services, educational purposes, social media networks, mobile applications, and other computational aspects (Brian et al. 2012). Such evolution in internet speed and the computer usage brought a demand to maximize the use of computational resources and to minimize the cost. Cloud Computing is considered a solution to fulfil this demand by moving applications and their data from desktop and portable Personal Computers into large data centres (Dikaiakos et al. 2009). Cloud Computing is no longer just hype, in contrast it is rapidly evolving, and the prospects that it will be one day the fifth used utility after water, electricity, gasoline, and telephone. (Agrawal, Das & Abbadi 2010; Buyya et al. 2009; Zhang et al. 2010) Cloud Computing has introduced several technologies to the industry such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) (Chang, Walters, & Wills 2013; Dikaiakos et al. 2009; Louridas 2010; Mohammed & Fiaidhi 2010). Multi-tenancy is the fundamental characteristic of Cloud Computing services, such characteristic allows SaaS vendors to run a single application that support

multiple tenants on the same software and hardware infrastructure (Fiaidhi et al. 2012; Kwok, Thao & Linh 2008). The term "tenant" is widely used in Cloud Computing to refer to organisation, customer, or user who is using a cloud service; this term also used in this thesis.

It is a common practice in the data architecture of SaaS applications to use a multi-tenant database where there is one database schema shared between all tenants (Aulbach 2011; Martinez 2012). Cloud database service providers have considered such a database as an effective resource sharing storage. That is because it reduces the costs on them by co-locating multiple tenants' databases into a single database schema. It also allows to reduce the total cost of ownership on the tenants of the service. Such data architecture consists of two data types: shared data and tenant's isolated data; in combining these two types of data together, tenants can have a complete data view that fit their business requirements (Domingo et al. 2010; Liu 2010).

The majority of modern Relational Database Management Systems (RDBMS) have designed to accommodate data for a single tenant. Nevertheless, single-tenant databases are not suitable to be used by multiple tenants, because they do not consider the unique characteristics of each tenant's data, which in return is leading to incorrect assumptions and query plans about the tenants' data (Aulbach 2011; Weissman & Bobrowski 2009). Therefore, various multi-tenant database schema techniques have been studied and implemented to overcome this challenge, including Private Tables, Extension Tables, Universal Table, Pivot Tables, Chunk Table, Chunk Folding, and XML Table (Aulbach et al. 2008; Aulbach et al.2009; Du, Wen & Yang 2010; Foping et al. 2009; Heng et al. 2012; Liao et al. 2012). The design of such multi-tenant schema techniques based on traditional RDBMS (Domingo et al. 2010; Martinez 2012). However, these multi-tenant schema techniques still have remaining challenges that need to be addressed (Agrawal, Das & Abbadi 2010; Aulbach et al. 2009; Aulbach 2011; Gorti, Shiri & Radhakrishnan 2013).

Overcoming multi-tenant schema technique challenges is one of the topics that has received much attention from academic and industrial fields to build multi-tenant software applications in general, and multi-tenant SaaS and Big Data applications in particular. Five of these challenges are listed below.

- **Managing Multi-tenant Database**: Configuration is the main characteristic in multi-tenant applications that allows to run a single instance application, which support multiple tenants with configurable metadata. Such application requires a multi-tenant aware design with single codebase and metadata service. This design allows to share resources across tenants, and to configure how multi-tenant application appears and behaves with the ability of isolating and differentiating data, information, configurations, and settings that belong to different tenants. Multi-tenant aware applications allow each tenant to design and manage different parts of the application, and automatically adjust and configure its behaviour during runtime execution without redeploy the application (Chong 2006). Consequently, managing and configuring multi-tenant aware application is a tenant self-service that typically performs while application in operation to minimize the system downtime, and allows the tenant to feel as if he is the only one using the application (Mietzner et al. 2009b). Furthermore, multi-tenant application requires a special multi-tenant schema technique, and a codebase solution that has two capabilities: (1) transforming virtual (logical) multi-tenant queries to physical database queries and execute them in a RDBMS; and (2) managing multiple tenants' data in a single instance application. Developing such a data management solution is a hard and complex problem to solve and has significant complexities that involve longer development lifecycle.
- **Extending RDBMS to Support Multiple Tenants:** Traditional RDBMSs have limitations in supporting multi-tenant database extensibility. That is because each individual tenant cannot modify the multi-tenant shared tables', since the shared tables commonly used between tenants, and changing their structure will affect all the tenants (Aulbach 2011). Thus, as mentioned before a number of multi-tenants

database schemas have introduced to overcome this challenge. However, such a multi-tenant database schemas have challenges include, but are not limited to: firstly, the existing multi-tenant database schema techniques store the tenant's extension data in non-relational format. In this way, either tenants who have structured data will refuse to adopt the multi-tenant database or they will adopt it by sacrificing their data normalization and data integrity. Secondly, some of these multi-tenant schema techniques introduce a number of issues by sharing tables with too many spare columns that resulting in (1) having an extensive number of null values, and (2) eliminating the usage of column indexes (Martinez 2012).

- **Optimizing Multi-tenant Database Query:** Multi-tenant database is designed to deliver database functionalities for multiple database users to create, store, and access their data over the internet. However, such approach raises an issue in database performance, because the multi-tenant database is shared between multiple tenants. The majority of modern RDBMSs have query optimizers to optimize the query execution of a single-tenant database, for example, Oracle, SQL Server (Raza et al. 2010), and PostgreSQL (Dash et al. 2010). Nevertheless, such query optimizers are leading to incorrect assumptions and query plans if they are used for the multi-tenant database. Therefore, the multi-tenant database requires a special query method to optimize different queries for multiple tenants who are using the same resources of a multi-tenant database.
- **Multi-tenant Database Access Control:** The growth of Cloud Computing multi-tenant services draws attention to security challenges, which are emerging due to the cloud vendor's resource sharing (Takabi, Joshi & Ahn 2010). It is highly unlikely that the tenants would risk their data by storing and accessing it over the cloud in favour of reducing the total cost of ownership, or using a flexible cloud service, unless the service providers offer reliable and secure services (Ren et al. 2012). Consequently, the multi-tenant database demands a special multi-tenant access control model, which provides access control not only for multiple tenants, but also for multiple users for each individual tenant. However, the existing

traditional RDBMSs do not differentiate between the data of different tenants' users (Brodersen et al. 2004), which in return is leading to incorrect assumptions and query plans about the tenants' data and the data of tenants' users.

- **The challenges of RDBMS and NoSQL Challenges**: Big Data is a new Cloud Computing paradigm that is defined in a simplistic form by four dimensions known as the 4Vs: Volume, Variety, Velocity, and Veracity (Demchenko et al. 2013; Kim, Trimi & Chung 2014). It emerged from the growth of data that collected from different applications (e.g. Enterprise, web, e-mail, social media, mobile, computational science), wireless sensor networks, surveillance and video cameras, and other sources. It has been argued that RDBMS are not suitable to store and manage Big Data for two reasons: (1) they are limited in offering good performance and scalability properties; and (2) they do not allow to extend on the existing database schema by adding, modifying, and deleting tables and tables' columns during the application's runtime execution (Leavitt 2010; Cattell 2011). NoSQL (Not Only SQL) storage technology has emerged significantly and used for Big Data applications, because NoSQL offers high scalability, elasticity, and availability (Agrawal, Das & Abbadi 2010). Andlinger[1] reports that the RDBMSs are dominating the database management systems in the industry, but there is an obvious trend and potential for the evolution and expansion of NoSQL in the database industry. However, NoSQL has a number of issues and concerns such as: (1) it does not support ACID (Atomicity, Consistency, Isolation, and Durability) transactional properties that the traditional RDBMSs support; (2) it requires manual query programming which adds extra overhead and complexity in software development; (3) the vast majority of organisations are unfamiliar with this technology, which leads to overwhelming them with difficulties in evaluating the feasibility of applying it or using it (Leavitt 2010; Cattell 2011); (4) it avoids joining operations, filtering on multiple properties, and filtering of data based on subqueries results (Dimovski 2013; Sakr et al. 2011). Such

1 http://db-engines.com/en/blog_post/23; Accessed July, 2014

limitations in RDBMS and NoSQL lead to a point that the optimal multi-tenant Big Data storage model does not exist yet.

1.2 RESEARCH OBJECTIVES

In view of the limitations described above about the existing multi-tenant schema techniques that are used as storage for multi-tenant applications, this study aims to achieve the following six research objectives. The primary objective is stated first and is followed by five other specific objectives.

1. Propose a new multi-tenant schema technique to improve and overcome the challenges of multi-tenant software applications. The aim of this objective is to propose a novel way of designing and creating a multi-tenant schema technique. This technique enables multiple tenants to share a single database schema instance in RDBMS, and in this schema builds each individual tenant's schema to fulfil the tenant's business requirements. Moreover, it allows tenants to store different data types of Big Data including structured, semi-structured, and unstructured in RDBMS.
2. Develop multi-tenant database management method to extend the RDBMS during the application's runtime execution. The aim of this objective is to develop a method to extend the traditional RDBMS by adding the tenant's virtual relational tables (the tenant's isolated tables) to the existing multi-tenant physical relational tables (tenants shared tables), and creating virtual database relationships between these tables such as one-to-one, one-to-many, many-to-one, many-to-many, or self-referencing during the application's runtime execution.
3. Develop a multi-tenant query execution method to retrieve tenants' data from multi-tenant physical relational tables and virtual relational tables. The aim of this objective is to develop a method to facilitate executing queries from one table, two or more tables of multi-tenant physical relational tables, and multi-

tenant virtual relational tables using virtual database relationships between them. This method allows tenants to use three database models: (1) multi-tenant physical relational tables, (2) multi-tenant physical relational tables integrated with virtual relational tables, and (3) multi-tenant virtual relational tables. In these three database models, tenants can retrieve data by executing simple and complex queries. In embracing this method, a traditional RDBMS could become elastic and contemporary database storage that is suitable for Cloud Computing services.

4. Develop a method to optimize retrieving data for multiple tenants who are using the same resources of a single multi-tenant database. The aim of this objective is to develop a method to optimize and speed up query retrievals, and use the most efficient way to execute a query in multi-tenant database using virtual primary keys, virtual relationships between multi-tenant physical relational tables and multi-tenant virtual relational tables, virtual indexes, efficient execution plans for virtual relational database queries, and efficient logics. This method reduces the query execution time for each single tenant who is using the same software and hardware resources that are used by multiple tenants, and reduces the consumption of these resources in the multi-tenant database.

5. Develop a method that allows each tenant to grant his users with access control capabilities from the multi-tenant database level rather than from the application level. The aim of this objective is to develop a multi-tenant access control method that is suitable for multi-tenant database. This method should allow each tenant in the multitenant database to have multiple users, and each user to have different types of grants to access the tenant's data. It also improves multi-tenant applications, by granting the tenant's user data accesses from the database level instead of the application level.

6. Develop a multi-tenant database framework prototype that mediates software applications and RDBMS. The aim of this objective is to achieve all the above

five objectives by proposing a multi-tenant database framework architecture to be used as a database layer between software applications and RDBMS. Such a framework is designed to implement the novel multi-tenant database schema that stated in the first objective, and simplify the development of software applications and multi-tenant SaaS and Big Data applications.

1.3 RESEARCH CONTRIBUTIONS

By tackling the six stated objectives, this thesis contributes to the theory and practice of the software development in general and Cloud Computing, SaaS, and Big Data development in particular as in the following aspects:

1. It proposes a novel multi-tenant schema technique called Elastic Extension Tables (EET), which enables service providers to offer elastic relational database schema that has three database models: (1) multi-tenant physical relational tables; (2) multi-tenant physical relational tables integrated with virtual relational tables; and (3) multi-tenant virtual relational tables. In the presence of this multi-tenant relational database schema, service providers can offer their tenants any business domain database such as customer relationship management (CRM), human resources (HR), accounting, or any other business domain database. Then tenants can store and access their data by using the existing business domain database as offered by the service provider, extending on it, or creating their own database schema from scratch. Moreover, such schema mapping technique is capable to store traditional data types (e.g. NUMBER, BOOLEAN, VARCHAR, DATE-AND-TIME, and others) and non-traditional data types (e.g. Large text, image, audio, video, and others). These data types typically are stored in Big Data application in structured, semi-structured, or unstructured format. However, the proposed schema is exhaustive in a way that it can store the entire data format of Big Data in RDBMS.

2. It develops a multi-tenant schema handler method that is based on EET. This method manages multi-tenant physical relational tables and virtual relational tables. Such a method allows the tenants to extend on the business domain database that the service providers offer by using a traditional RDBMS. In using this method, tenants can create/update/delete virtual tables, columns, columns constraints, primary keys, foreign keys, indexes, table rows (records), while for the physical tables, tenants can create/update/delete only physical table rows, and database relationships between physical tables and virtual tables. The rest of the physical tables' database operations such as creating physical tables, columns, columns constraints, primary keys, foreign keys, indexes, and database relationships between two physical tables are managed from a traditional RDBMS. The advancement of this method is that it allows tenants to manage their tables during the application's runtime execution.
3. It develops a multi-tenant database proxy method that is based on EET. This method integrates, generates, and executes tenants' queries by using a codebase solution that converts multi-tenant queries into regular database queries and executes them in any of the available RDBMSs. Such a method solves the problem of integrating multi-tenant physical relational tables and virtual relational tables by making them work together and operate virtually as a single database schema for each tenant. This integration capability allows tenants to retrieve data from any of the three database models that mentioned above. Such a method also retrieves simple and complex queries including joining operations, filtering on multiple properties, and filtering of data based on subqueries' results.
4. It develops a multi-tenant query optimizer method that is based on EET. This method estimates the cost of different query execution plans, to determine the optimal plan for each tenant and each tenant's user. Then, the proxy method that mentioned in the previous point uses this plan to execute the tenants'

queries by converting them into traditional database queries, and finally executes them using the query optimizer of a RDBMS.

5. It develops a multi-tenant access control method for the tenant's users to grant them access to the tenant's data that is stored in EET. Such an access control method permits the tenant's users to access table's columns and rows based on a number of groups or roles, which are assigned to these users.

6. It proposes framework architecture for multi-tenant database, and it develops a prototype for this architecture that is based on EET. This framework designed to be used as a database layer between software applications (e.g. SaaS, Big Data, mobile, web) and any of the available RDBMSs. In addition, it overcomes the above mentioned challenges and offers the following advancements: (1) allowing RDBMS vendors to improve their databases by accommodating multiple tenants, and storing Big Data in a single database; (2) allowing RDBMS vendors to offer three database models that stated above in the first contribution, and allowing their tenants to choose from any of these database models; (3) allowing cloud service providers to offer multi-tenant database and multi-tenant application services; (4) simplifying the development of software application in general, and the development of multi-tenant SaaS and Big Data applications in particular for both cloud service providers and their tenants; (5) permitting tenants to evaluate and adopt cloud database services; and (6) reducing the cost on both the cloud service providers and their tenants.

1.4 THESIS ORGANISATION

This thesis is structured in nine chapters as follows.

Chapter 1 introduces the topic of this thesis, opens with the topic background, then states the research objectives, contributions, the organisation of this thesis, and finally lists the publications related to this thesis.

Chapter 2 presents a literature review related to the research topic. Firstly, defines Cloud Computing and Software as a Service. Secondly, defines multi-tenancy. Thirdly, presents the multi-tenant database management. Fourthly, reviews the existing multi-tenant database schema designs. Fifthly, presents a number of multi-tenant database query optimizers. Sixthly, presents a number of multi-tenant database access controls. Seventhly, defines the Big Data, RDBMS, and NoSQL, and addresses the issues of RDBMS and NoSQL. Finally, summarises the literature review.

Chapter 3 proposes the multi-tenant database framework architecture. Firstly, presents the overview architecture of the EET framework. Secondly, presents the conceptual architecture design of EET framework. This chapter addresses the sixth objective of this thesis. Parts of this chapter have been published in CLOSER 2012 (Yaish, Goyal & Feuerlicht 2012), and CSE 2013 (Yaish & Goyal 2013a).

Chapter 4 proposes the novel multi-tenant schema technique. Firstly, proposes the EET multi-tenant schema technique. Secondly, proposes the three database models of EET. Thirdly, compares the existing multi-tenant schema techniques with EET. Fourthly, gives an evaluation and discussion of EET by comparing it with UTSM. Finally, summarises the chapter. This chapter addresses the first objective of this thesis. Parts of this chapter have been published in DASC 2011 (Yaish, Goyal & Feuerlicht 2011), Procedia Computer Science 2014 (Yaish, Goyal & Feuerlicht 2014a), and JoCCASA 2014 (Yaish, Goyal & Feuerlicht 2014d).

Chapter 5 proposes the multi-tenant schema handler method. Firstly, proposes the architecture of the multi-tenant schema handler method. Secondly, develops the schema handler concept and its algorithms. Thirdly, conducts experimental results and discussions on the schema handler method that is based on the EET. Finally, summarises the chapter. This chapter addresses the second objective of this thesis. Parts of this chapter have been published in Procedia Computer Science 2014 (Yaish, Goyal & Feuerlicht 2014b).

Chapter 6 proposes the multi-tenant database proxy method. Firstly, proposes the architecture of the proxy method. Secondly, develops the proxy method algorithms. Thirdly, conducts experimental results and discussions on the proxy method. Finally, summarises the chapter. This chapter addresses the third objective of this thesis. Parts of this chapter have been published in LNCS 2013 (Yaish, Goyal & Feuerlicht 2013a), and submitted to TLDKS (Yaish, Goyal & Feuerlicht 2014c).

Chapter 7 proposes the multi-tenant query optimizer method. Firstly, proposes the architecture of the query optimizer method. Secondly, conducts experimental results and discussions on the query optimizer method that is based on the EET and the proxy method. Finally, summarises the chapter. This chapter addresses the fourth objective of this thesis. Parts of this chapter have been published in LNCS 2013 (Yaish, Goyal & Feuerlicht 2013b).

Chapter 8 proposes the multi-tenant access control method. Firstly, proposes the multi-tenant access control data architecture that is based on EET. Secondly, presents the EET multi-tenant access grants. Thirdly, develops the multi-tenant access grants algorithms. Fourthly, conducts experimental results and discussions on the access grants of columns and rows related to the tenants' users. Finally, summarises the chapter. This chapter addresses the fifth objective of this thesis. Parts of this chapter have published in CSE 2013 (Yaish & Goyal 2013b).

Chapter 9 summarizes the thesis and draws conclusions, and provides new research directions that could be persuaded in the future.

The structure of the thesis is graphically described in Figure 1-1.

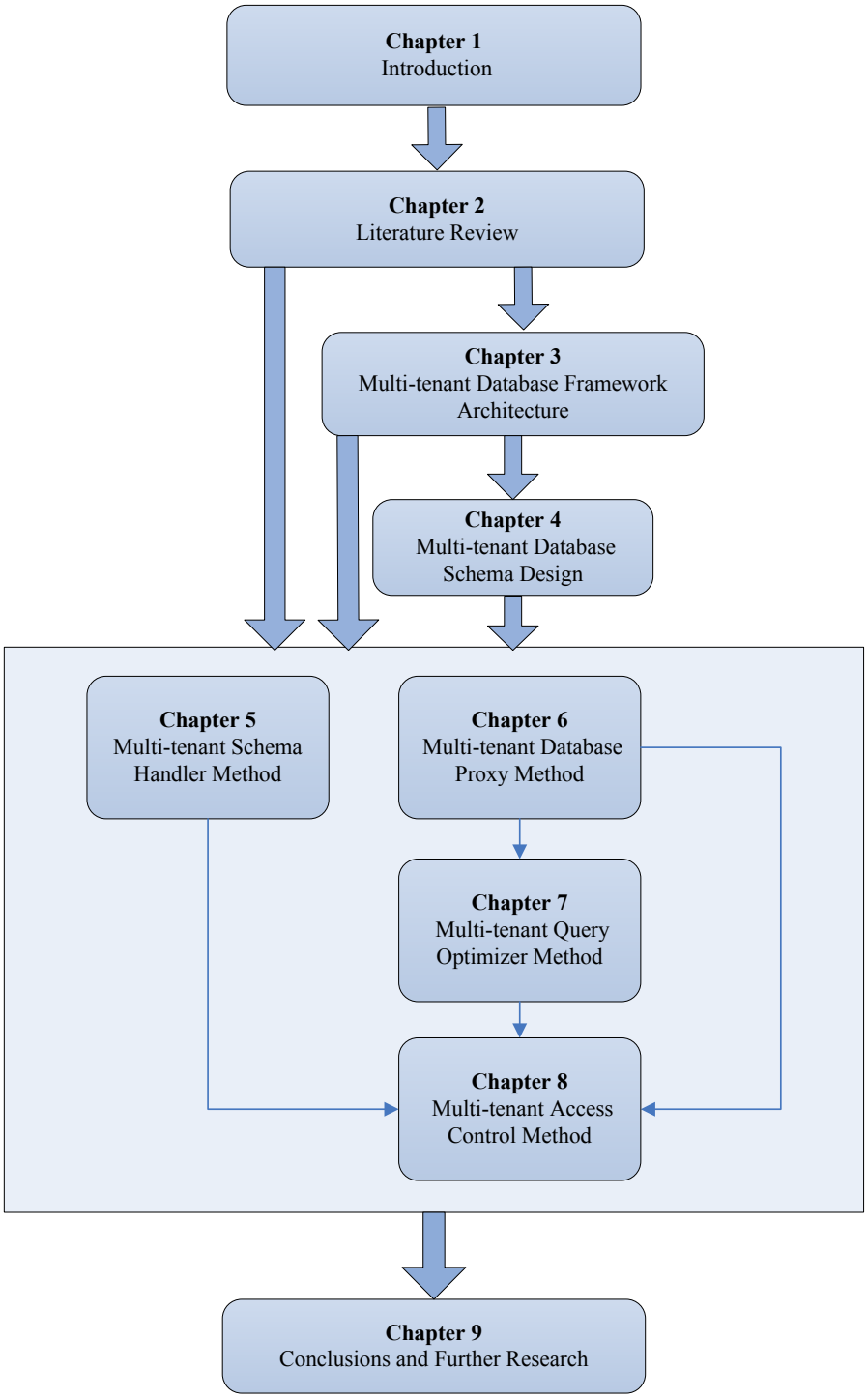


Figure 1-1: The overall structure of the thesis

1.5 PUBLICATIONS RELATED TO THIS THESIS

1. **Yaish, H.**, Goyal, M. & Feuerlicht, G. 2011, 'An elastic multi-tenant database schema for software as a service', *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, Sydney, Australia, pp. 737-743.
2. **Yaish, H.**, Goyal, M. & Feuerlicht, G. 2012, 'A novel multi-tenant architecture design for Software as a Service applications', *The 2nd International Conference on Cloud Computing and Services Science (CLOSER)*, Porto, Portugal, pp. 82-88.
3. **Yaish, H.**, Goyal, M. & Feuerlicht, G. 2013a, 'Proxy service for multi-tenant database access', *Lecture Notes in Computer Science (LNCS)*, vol. 8127, pp. 100-117.
4. **Yaish, H.**, Goyal, M. & Feuerlicht, G. 2013b, 'A method of optimizing multi-tenant database query access', *Lecture Notes in Computer Science (LNCS)*, vol. 8182, pp. 194-212.
5. **Yaish, H.** & Goyal, M. 2013a, 'A multi-tenant database architecture design for software applications', *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, Sydney, Australia, pp. 933-940.
6. **Yaish, H.** & Goyal, M. 2013b, 'Multi-tenant database access control', *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, Sydney, Australia, pp. 870-877.
7. **Yaish, H.**, Goyal, M. & Feuerlicht, G. 2014a, 'Evaluating the performance of multi-tenant elastic extension tables', *Procedia Computer Science,* vol. 29, pp. 614-626.
8. **Yaish, H.**, Goyal, M. & Feuerlicht, G. 2014b, 'Multi-tenant elastic extension tables data management', *Procedia Computer Science*, vol. 29, pp. 2168-2181.

9. **Yaish, H.**, Goyal, M. & Feuerlicht, G. 2014c, 'A proxy service for multi-tenant Elastic Extension Tables', *submitted to Advanced Techniques for Cloud Data Management (TLDKS),* vol. 8420, p. 25.

10. **Yaish, H.**, Goyal, M. & Feuerlicht, G. 2014d, 'Elastic extension tables multi-tenant database schema', *submitted to Journal of Cloud Computing: Advances, Systems and Applications (JoCCASA).*

CHAPTER 2

LITERATURE REVIEW

This literature review presents relevant works in connection with this research. Section 2.1 defines Cloud Computing and SaaS. Section 2.2 presents the multi-tenant architecture design for multi-tenant applications. Section 2.3 presents the multi-tenant database management. Section 2.4 reviews the existing multi-tenant database schema designs. Section 2.5 presents single-tenant and multi-tenants query optimizers. Section 2.6 presents different multi-tenant role based access control methods. Section 2.7 defines the Big Data, RDBMS and NoSQL, and addresses the issues of RDBMS and NoSQL. Finally, Section 2.8 summarises this literature review.

2.1 CLOUD COMPUTING AND SOFTWARE AS A SERVICE

Cloud Computing is a recent technology paradigm that allows to deliver internet applications and their infrastructure as services over the internet, by moving applications and their data from desktop and portable PCs (Personal Computers) into a large data centres (Brian et al. 2012; Dikaiakos et al. 2009). It enables tenants to share various computing services in an efficient way and cost-effective manner (Xu 2010; Ratametha & Veeragandham 2008). Cloud Computing Services allow to reduce the cost of adding more capacity at the peak demand of these services, and minimizing this capacity when the demand decreases (Dikaiakos et al. 2009). This service creates an elastic environment that expands and contracts depending on the workload and performance, as per the agreement of pay-by-use that guarantees a

minimum level of service that is offered by a cloud service provider (Carolan et al. 2009). Virtualization is the main characteristic of Cloud Computing that is applied in a number of aspects such as hardware, software, operating system, and storage in the Cloud Computing platform, instead of a physical platform. The Cloud Computing operations such as resource expansion, migration, and backup operate through a virtualization level (Shuai et al. 2010). IT organisations use virtualization to create multiple instances of an existing environment on the same application server easily and quickly. This technique uses pay-as-use model where a software environment can exist to run a job or many jobs on the server for a few minutes, hours, or long-term basis (Carolan et al. 2009). Cloud Computing introduced technologies such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) (Chang, Walters, & Wills 2013; Dikaiakos et al. 2009; Louridas 2010; Mohammed & Fiaidhi 2010). The details of these services are listed below:

- **Infrastructure as a Service:** IaaS provides the functionality of hosting virtual machines and other abstract hardware, infrastructures and operating systems (Dikaiakos et al. 2009; Ambrose, Dagland & Athley 2010) that is controlled through Application Programming Interface (API) (Ambrose, Dagland & Athley 2010). This service is provided by data centres to allow their tenants to deploy and run their operating systems on top of virtualized software. Such software is used in IaaS for distribution, automation, deployment, and installation. Amazon EC2 and S3, IBM BlueCloud, and Windows Live Skydrive are examples of IaaS (Ambrose, Dagland & Athley 2010; Mohammed & Fiaidhi 2010).

- **Platform as a Service:** PaaS is used to configure, deploy, and access software through a web browser, without any need for the end user to install or download the platform that is provided by the vendor (Ambrose, Dagland & Athley 2010). It offers access functionality to the developers, without offering any physical computing tools, in order to meet the application requirements, set the application server, and use virtual computing resources of IaaS (Chengtong et al. 2010). Further, it allows tenants to develop and deploy their websites and make them

available through the internet. Furthermore, it allows tenants to access different tools on the platform to configure their platforms during their websites' runtime execution (Ambrose, Dagland & Athley 2010). Such a platform makes the software implementation much easier and reduces the cost on the tenants (Chengtong et al. 2010). Microsoft Azure and Google App Engine are examples of PaaS (Ambrose, Dagland & Athley 2010; Mohammed & Fiaidhi 2010).

- **Software as a Service**: SaaS is defined as an online software delivery service that allows tenants to subscribe to a paid software service instead of paying for software licenses, and to share the service with multiple tenants (Ju et al. 2010). It is an emerging software service and one of the hot topics in the software industry (Burno 2006). Salesforce is the most common example of SaaS in the market, and there are many other examples, such as Google Apps (Carolan et al. 2009), NetSuite [2], and others.

The three Cloud Computing services, including IaaS, PaaS, and SaaS are described above; however, SaaS is the major focus of the rest of this section.

2.1.1 SaaS History

SaaS is an old concept from the mainframe era in the 1960s and into the minicomputer era in 1970s. Due to the high prices of acquiring computers at that time, the remote computer model emerged especially after introducing PCs, LANs, and client-server computing in 1980s and early 1990s (Landy 2008). SaaS first came in 1990s and it was called Application Service Provider (ASP). Nevertheless, it was almost lost because of the dotcom bubble bursting (Menken & Blokdijk 2009). In February 2000, SaaS started to return back when Salesforce launched its web-based service, and it became the early SaaS adopter. In February 2001, the term Software as a Service or SaaS published for the first time in a white paper called "*Software as a Service: Strategic Backgrounder*" (Hoch, Kerr & Griffith 2001, Nitu 2009). SaaS began to flourish in 2005-2006, because the

2 http://www.netsuite.com/portal/home.shtml; Accessed July, 2014

internet speed had significantly increased, had become affordable, and the tenants had started to be more comfortable to establish business over the internet (Wang et al. 2008).

2.1.2 SAAS MODEL

The SaaS business model differs in many aspects from the ASP model, though it is often mistaken to be the same. SaaS model is the successor of the ASP model, and it is considered as an improvement in ASP model, because the SaaS model overcomes ASP model issues and limitations (Lopes 2009). SaaS is delivered as web-based software to allow tenants to subscribe in services offered over the internet. The concept of SaaS saves software payers much money to acquire the software, and overcome the desktop software issues such as system compatibilities, installation difficulties, and manual updating. In fact, this model obviates tenants from buying and maintaining their ICT (Information and communications technology) infrastructure of their own. In contrast, they obtain tenant-focused and tenant-driven service approach offered by a service provider (Bezemer et al. 2010; Hoogvliet 2008).

2.1.3 SAAS CHARACTERISTICS

SaaS has four main characteristics: Multi-tenancy, Shared Services, Built-in Feedback Mechanism, and Pay-as-you-use only service (Hoogvliet 2008). *Multi-tenancy* is an important concept of SaaS that enables different tenants to run same program version, in the same software environment (Mathew & Spraetz 2009). *Shared Service* allows the tenant to link up SaaS program with other services available online. The *Built-in Feedback Mechanism* in SaaS program helps tenants to report problems or difficulties encountered while using the program, and on the other side helps the vendors to improve their program (Hoogvliet 2008). *Pay-as-you-use* characteristic gives the tenant flexibility in changing the usage of software, by increasing or decreasing the number of users at any time (Ju et al. 2010). In this

thesis, the focus will be on two characteristics of SaaS, the *Multi-tenancy* and the *Shared Service*.

2.1.4 SaaS Maturity Model

Multi-tenancy is applied in four software layers: application, middleware, virtual machine, and operating system (Kwok, Thao & Linh 2008). The application layer has four levels of the SaaS maturity model. Firstly, *Level 1 - Ad Hoc/Custom*, in which each tenant in this level has a separate custom instance of SaaS application that is hosted on the vendor servers, the source code of this instance can be customized according to the tenants' needs. Secondly, *Level 2 – Configurable*, in which the vendor hosts separate instances for each tenant, each of these instances is using the same source of the code. However, each instance can be configured differently to meet the tenants' needs. Thirdly, *Level 3 – Configurable and Multi-Tenant-Efficient*, which allows the vendor to run a single instance application that supports multiple tenants, this application can be configured differently by each single tenant, and each configuration belongs to the tenant who configured it. Finally, *Level 4 – Scalable Configurable, and Multi-Tenant-Efficient*, which is similar to Level 3, but the only difference in this level, is that the vendor hosts multiple tenants with a high level of scalability (Chong & Carraro 2006; Chong 2006; Hudli et al. 2009; Kwok, Thao & Linh 2008; Shao 2011).

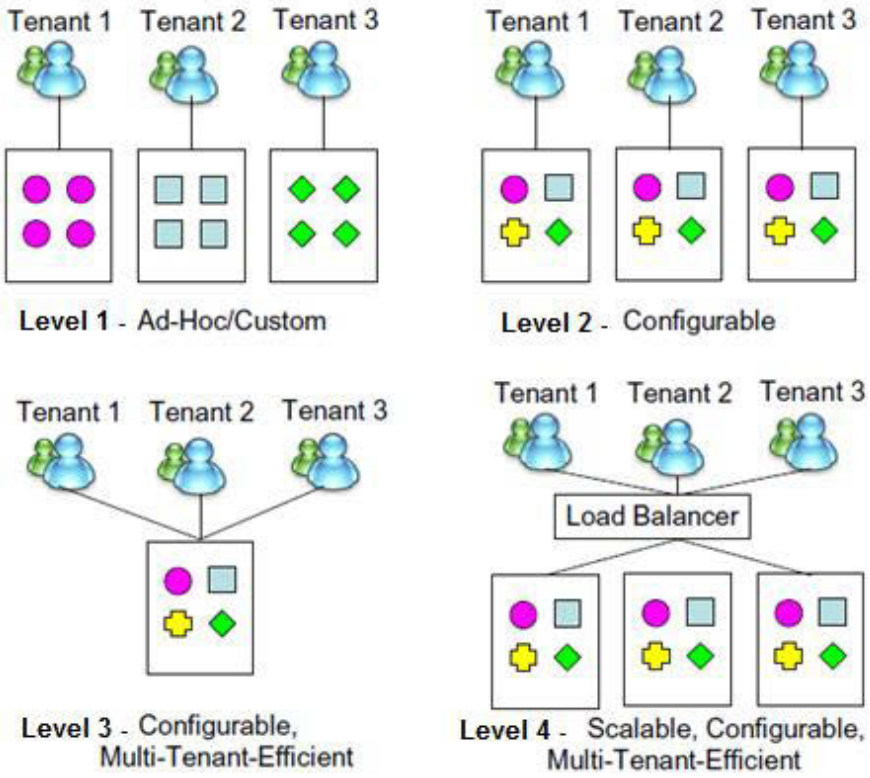


Figure 2-1: SaaS Maturity Levels (Shao 2011)

2.2 MULTI-TENANCY

Multi-tenancy is the fundamental design approach for SaaS service providers that improves and facilitates the manageability of SaaS applications (Jansen, Houben & Brinkkemper 2010), and allows each tenant who is using SaaS application to feel as if he is the only tenant using this application, where in fact, many tenants using it (Bezemer et al. 2010). It is an important characteristic of SaaS application that is used by multiple tenants at the same time on a single instance of software code (Menken & Blokdijk 2009; Lopes 2009; Shao 2011). Moreover, it can be applied in four software

layers, including application, middleware, virtual machine, and operating system (Kwok, Thao & Linh 2008).

2.2.1 MULTI-TENANT ARCHITECTURE

Shao (2011) states that SaaS tenants usually share the same software and in some cases the same database. Accordingly, once multi-tenant vendors need to increase multi-tenant architecture they should look after the Quality of Service (QoS) for each tenant and ensure that he is not affected from the rest of the tenants. Shao's study describes multi-tenant architecture in four aspects including Resource isolation, Configuration, Security, and Scalability. The resource isolation aspect is significant for multi-tenant application, because tenants share the same infrastructure and software code. The configuration aspect ensures that multi-tenant application is a highly configurable application. The security aspect is considered as a significant issue because of sharing software code and data between tenants. In terms of the scalability aspect, as discussed in the SaaS maturity model in the previous section, in order to make Level 1 and Level 2 scalable levels, a significant software design and implementation need to be done. However, this is not the case for Level 3, since it allows all SaaS tenants to use the same single instance.

2.2.2 MULTI-TENANT CONFIGURATION

In multi-tenant applications, the configuration capability allows SaaS vendors to run a single instance application that supports multiple tenants with configurable metadata, which provides a means of configuration for multi-tenant applications that satisfy the tenants' business needs, and resolves the problem of different requirements for several tenants who may use a particular business domain application. This maturity level requires a multi-tenant aware design with a single code base and metadata service, which allows to share resources across multiple tenants, and to configure how multi-tenant application appears and behaves with the ability of

isolating and differentiating data, information, configurations, and settings that belong to different tenants. A multi-tenant aware application allows each tenant to design different parts of the application, and automatically adjust and configure its behaviour during the application's runtime execution without redeploy the application (Chong 2006). Consequently, configuring multi-tenant aware applications is a tenant self-service that typically performs while applications are in operation to minimize system downtime, and allows the tenant to feel as if he is the only one using the application (Mietzner et al. 2009b). The relational database of multi-tenant applications has challenges in supporting well manageable database schema and providing configurable database fields (Foping et al. 2009; Du, Wen & Yang 2010; Kwok, Thao & Linh 2008; Mietzner et al. 2009a).

This section presents the architecture of multi-tenant applications that support multiple tenants in one single application, which has configurable metadata that provides a means of configuration to satisfy the tenants' business requirements.

2.3 MULTI-TENANT DATABASE MANAGEMENT

Managing multi-tenant database differs from traditional RDBMS in four aspects: (1) isolating tenants' data by ensuring that each tenant can access only his own data, (2) ensuring that each tenant's data is secured, (3) building robust multi-tenant database structure, and (4) optimizing the performance of each tenant database (Bezemer et al. 2010; Chong, Carraro & Wolter 2006; Lazarov 2007). Multi-tenant data architecture has two types: shared data and tenant's isolated data. Integrating these data together, tenants could have a complete data that they need. The multi-tenant database has three data isolation approaches. The first approach is called *Separate Database*, which is the simplest data isolation approach, which stores each tenant data in a separate database. Figure 2-2 shows the first approach. The second approach is called *Shared Database - Separate Schema*, which hosts all the tenants in

the same database instance, but each tenant has his own database schema. Figure 2-3 shows the second approach. The third approach is called *Shared Database - Shared Schema*, which allows tenants to store their data in the same database and the same schema, this means a given table can store different rows for different tenants, and a tenant ID column differentiates and isolates tenants' data (Domingo et al. 2010). Figure 2-4 shows the third approach.

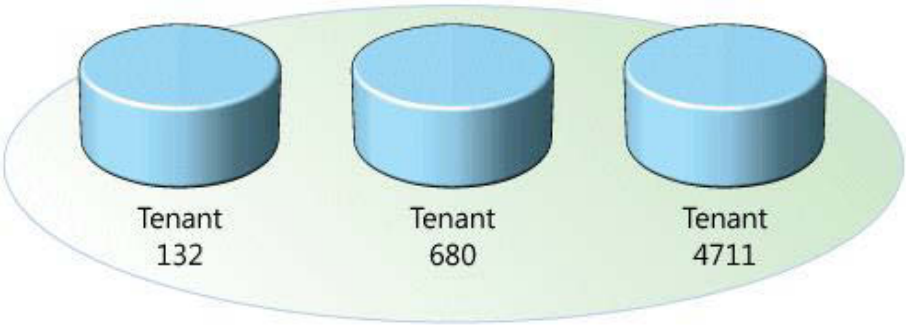


Figure 2-2: Separate Database Approach (Chong, Carraro & Wolter 2006)

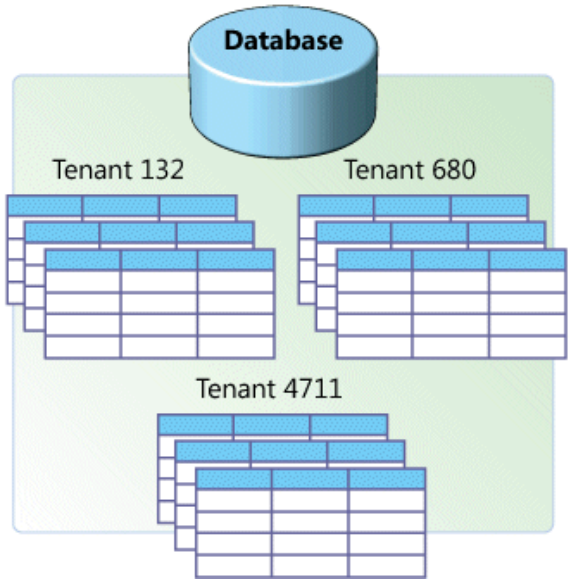


Figure 2-3: Shared Database - Separate Schema Approach (Chong, Carraro & Wolter 2006)

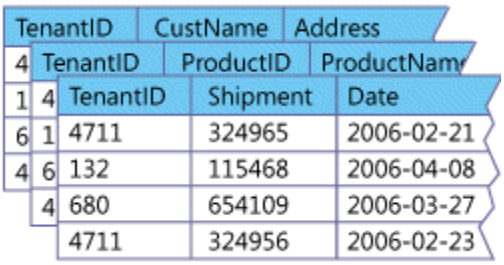


Figure 2-4: Shared Database - Shared Schema Approach (Chong, Carraro & Wolter 2006)

The multi-tenant shared data architecture needs large development efforts, time, and money. In contrast, the development of the isolated data architecture requires less effort, time, and money compared with the shared data architecture. Nevertheless, the ongoing maintenance and operational cost for the shared data architecture is less than the isolated data architecture. The decision on which data architecture implementation should be chosen depends on, firstly, the number of tenants who are going to use the data storage and, secondly, on the efficiency and the cost considerations of SaaS implementation (Chong, Carraro & Wolter 2006).

Overall, it is a significant challenge to maintain a traditional software application, change its requirements, and then deploy its new changes to the production server. In accordance with the extra complexity that involved in the architecture of the multi-tenant software, the maintenance process of such software will be more complex than the maintenance process of the traditional software. Unless, the multi-tenant application is highly configurable and can be managed and maintain during the application's runtime execution (Bezemer & Zaidman 2010). Such capability cannot be achieved unless the database of multi-tenant application is highly configurable database. The multi-tenant database schema that this thesis proposes is based on the *Shared Database - Shared Schema* approach.

2.4 MULTI-TENANT DATABASE SCHEMA DESIGNS

A number of multi-tenant database schema designs and techniques have studied and implemented to overcome multi-tenant database challenges. This section presents seven multi-tenant database schema techniques, including Private Tables, Extension Tables, Universal Table, Pivot Tables, Chunk Table, Chunk Folding, and XML Table (Aulbach et al. 2008; Aulbach et al.2009; Du, Wen & Yang 2010; Foping et al. 2009; Heng et al. 2012; Liao et al.2012). All of these seven multi-tenant database schema techniques are based on traditional RDBMS (Domingo et al. 2010; Martinez 2012).

2.4.1 PRIVATE TABLES

The Private Tables technique allows each tenant to have his own private tables, which can be extended and changed (Aulbach et al. 2008; Aulbach et al. 2009). The query of this technique can be transformed from one tenant to another by renaming tables, and metadata without using extra columns like 'tenant_id' to distinguish and isolate the tenants' data. In contrast, many tables are required to satisfy each tenant needs. Therefore, this technique can be used if there are fewer tenants using it, to produce sufficient database load and good performance (Aulbach et al. 2009).

2.4.2 EXTENSION TABLES

The Extension Tables are separated tables joined with the base tables by adding tenants' columns to construct logical source tables (Aulbach et al. 2008; Aulbach et al. 2009). This technique brought from the Decomposed Storage Model that splitting up n-columns table into n 2-column tables joined using surrogate values (Aulbach et al. 2008). It is used by multiple tenants, who can use the base tables, as well as the extension tables (Foping et al. 2009). Furthermore, it is considered better than the Private Tables that stated before. Nevertheless, in this design, the number of tables is

increased by increasing the number of tenants, and the variety of their different business requirements (Aulbach et al. 2008).

2.4.3 UNIVERSAL TABLE

The Universal Table is a table that contains additional columns of the base application schema columns, which enable tenants to store their required columns. It is structured with two main columns 'tenant_id' and 'table_id', and other generic data columns, which have a flexible VARCHAR data type in which different data types with different data values can be stored in these columns (Aulbach et al. 2008; Liao et al. 2012). It is a flexible technique that enables tenants to extend their tables in different ways according to their business needs. However, the rows of the universal table can be too wide with an overhead in the number of NULL values, which the database has to handle (Aulbach et al. 2008).

2.4.4 PIVOT TABLES

In the Pivot Tables technique, the application maps the schema into generic structure in the database, in which each column of each row in a logical source table is given its own row in the Pivot Table. The rows in the Pivot Table comprise of four columns, including tenant, table, column, and row that specifies which row in the logical source table they represent. As well as a single data type column that stores the values of the logical source table rows according to their data types in the designated pivot Table (Aulbach et al. 2008; Foping et al. 2009). For example, the Pivot Tables can have two pivot tables, the first table 'pivot_int' to store INTEGER values, and the second table 'pivot_str' to store STRING values. The performance benefits are achieved using this technique by avoiding NULL values and by selectively reading from smaller numbers of columns. In comparison with Pivot Tables (vertical tables) and horizontal tables, the first one performs better when it

allows columns selectively read in columns to improve the performance (Aulbach et al. 2008).

2.4.5 CHUNK TABLE

The Chunk Table is another generic structure technique that is similar to Pivot Table. Except, it has a set of data columns with a mixture of data types that replacing the column 'col' in the Pivot Table with 'chunk' column in the Chunk Table (Aulbach et al. 2008). This technique partitions the logical source table into groups of columns. Each group assigned to a chunk ID and mapped into an appropriate Chunk Table. This technique has four advantages over Pivot Table, including (1) Reducing metadata storage ratio, (2) reducing the overhead of reconstructing the logical source tables, (3) reducing the number of columns, and (4) providing indexes. This technique is flexible, but it adds complexity to the database queries (Aulbach et al. 2008).

2.4.6 CHUNK FOLDING

The Chunk Folding is a schema mapping technique that partition logical source tables into chunks vertically (Aulbach et al. 2008; Foping et al. 2009). These chunks are folded in different physical tables and joined together, where a chunk of columns is partitioned into a group of columns and each group has a chunk id (Foping et al. 2009). Aulbach et al. (2008) perform experiments to measure the efficiency of Chunk Table and Chunk Folding techniques, and they found that Chunk Folding technique outperform the Chunk Table technique. In addition, they state that the performance of this technique is enhanced by mapping the most used tenants' columns of the logical schema into conventional tables, and the remaining columns in the Chunk Tables are not used by the majority of tenants. However, the main limitation and weakness of the Chunk Folding technique is that the common schema that is used by multiple tenants must be known in advance, which is not a practical solution for multi-tenant databases. This issue also exists in Extension Tables, Pivot Tables, and Chunk Table.

2.4.7 XML TABLE

The XML Table database extension technique is a combination of relational database systems and Extensible Markup Language (XML) (Aulbach et al. 2009; Du, Wen & Yang 2010; Foping et al. 2009). The extension of XML can be provided as native XML data type, or storing the XML document in the database as a Character Large Object (CLOB) or Binary Large Object (BLOB) (Aulbach et al. 2009). XML data type facilitating the creation of database tables, columns, views, variables and parameters, and isolating the application from the relational data model (Du, Wen & Yang 2010). This technique satisfies tenants' needs because their data can be handled without changing original database relational schema, and XML data type can be supported by several relational database products (Du, Wen & Yang 2010; Foping et al. 2009). In contrast, this technique reduces the data access performance using XML files (Aulbach et al. 2009), and Heng et al. (2012) state that this technique has the highest response time, in other words, it was the slowest technique in comparison with Private Tables, Universal Tables, Pivot Tables, Chunk Table and Chunk Folding techniques.

Although Heng et al. (2012) use the Elastic Extension Tables (EET) name of the multi-tenant database schema that proposed in (Yaish, Goyal & Feuerlicht 2011), and call the storage model of Salesforce by this name, which is incorrect. However, this paper (Heng et al. 2012) conducted a number of significant experiments to evaluate retrieving data from five different multi-tenant schemas used in multi-tenant SasS applications, including Private Tables, Universal Tables, Pivot Tables, Chunk Table, Chunk Folding, and XML Table. The results of these experiments show that retrieving data from Universal Table is faster than the other schemas except the Private Tables schema. Aulbach et al. (2009) conducted experiments between Private Tables schema and the Universal Table (Spare Columns) schema. The results of these experiments show that the Universal Table schema has the same performance or even better than the Private Tables schema when retrieving or inserting data, except when

inserting a large number of data, the Universal Table schema is slower than the Private Tables schema. Such experimental results lead to the fact that the performance of retrieving data from Universal Table schema is considered the optimal performance out of the five multi-tenant schemas, because as mentioned before the Private Tables schema is only suitable for a small number of tenants, but not for a large number. Overall, the experimental results make the Universal Table schema the optimal schema to use for a multi-tenant database when it is compared to Pivot Tables, Chunk Table, Chunk Folding, and XML Table. Nevertheless, as mentioned before the Universal Table can be too large in a way that introducing overhead with the number of NULL values, which the database has to handle. Ultimately, this suggests that the current available multi-tenant database schemas still have remaining challenges, and the optimal multi-tenant schema does not exist yet. Chapter 4 presents an example that shows and clarifies how the data is populated in the seven multi-tenant database schema designs that discussed in this section.

2.5 Multi-tenant Database Query Optimizer

Monitoring the performance of executing queries in RDBMS by database administrators is costly and difficult. Therefore, the Query Optimizer module emerged to improve manual tuning of queries to automatic query optimization (Raza et al. 2012). The Query optimizer is a query processing technique that uses statistical properties to select an efficient execution query plan (Farahani, Sharifnejad & Sharifi 2006). There are a large number of related works, which have done on the query optimizer for single-tenant database, however, such query optimizers are suitable for single-tenant applications but not for multi-tenant applications. This section discusses a number of related works that show how single tenant database query optimizers work for Oracle, SQL Server, and PostgreSQL. In addition, it discusses how Salesforce the pioneer of SaaS, optimizes its multi-tenant database.

2.5.1 ORACLE QUERY OPTIMIZER

Oracle Query Optimizer's major three components are Query Transformer, Estimator, and Plan Generator. The *Query Transformer* Component consists of three techniques including View Merging, Sub Query Unnesting, and Materialized Views. Any of these techniques can be used by Query Transformer or a combination of them. The *Estimator* component estimates the cost of the query by using statistics that are created through dynamic sampling. The *Plan Generator* component generates the best plan with the lowest cost of query execution (Raza et al. 2010). Figure 2-5 shows the architecture of the Oracle query optimizer.

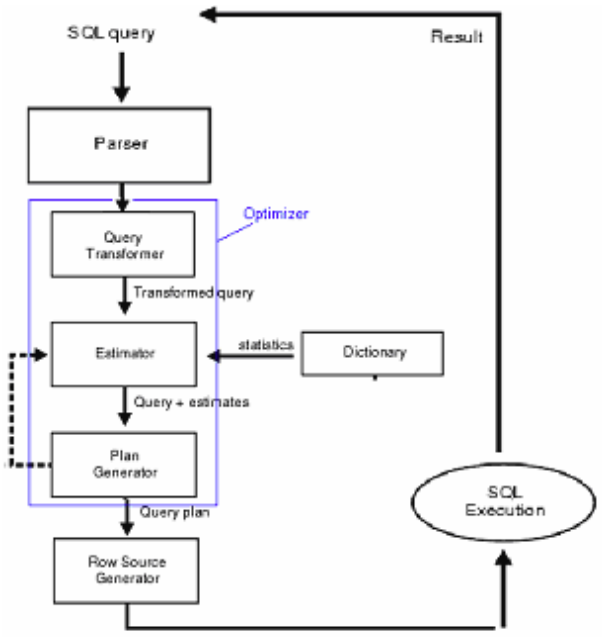


Figure 2-5: The architecture of Oracle query optimizer (Raza et al. 2010)

2.5.2 SQL SERVER QUERY OPTIMIZER

The query optimizer of SQL Server has six steps. First, query analysis, in which the search arguments and join clauses get identified. Second, index selection, in which it assesses the search arguments and joins selectivity, and comparing each index cost. Third, performing join selection, in which a different join processing strategies are considered. So far, in these three steps, statistics are loaded and query is simplified. Fourth, performing the optimizer in two phases: carrying out transaction processing and generating plans. Fifth, evaluating the generated plans, and then the Estimator will select the best query execution plan. Finally, the last step the optimizer will execute the selected plan (Raza et al. 2010). Figure 2-6 shows the architecture of SQL Server query optimizer.

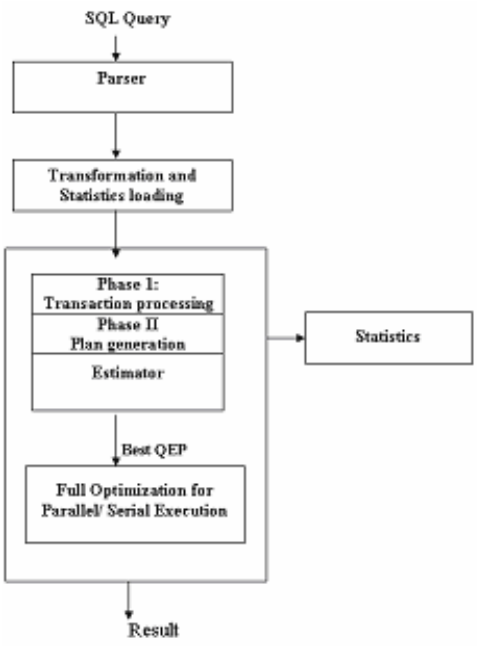


Figure 2-6: The architecture of SQL Server query optimizer (Raza et al. 2010)

2.5.3 POSTGRESQL QUERY OPTIMIZER

PostgreSQL Query Optimizer's components are Query Preprocessor, Sub-query Planner, Grouping Planner, Access Path Collector, Catalog, and Join Planner. The *Query Preprocessor* component rewrites the query to optimize the identified opportunities that produced from the analysed statistics. The *Sub-query Planner* component optimizes each sub-query that cannot be merged into the top-level query. The *Grouping Planner* component identifies a query ordering and grouping and isolates its columns. The *Access Path Collector* component iterates tables that starting from the FROM clause, and estimates the costs of accessing table's statistics by using table scans, index scans, or seeks. It looks up the statistics of the table as well as indexes from the *Catalog* schema and then estimates the costs of accessing them. Moreover, it attempts to eliminate the complexity of next components by avoiding inefficient access paths. The *Join Planner* component identifies the join methods and joins orders (Dash et al. 2010). Figure 2-7 shows the architecture of the PostgreSQL query optimizer.

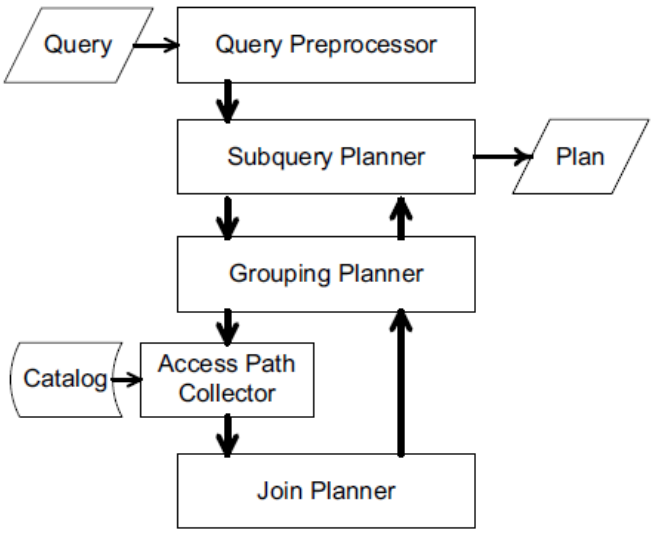


Figure 2-7: The architecture of PostgreSQL Query Optimizer (Dash et al. 2010)

2.5.4 SALESFORCE QUERY OPTIMIZER

Weissman & Bobrowski (2009) state that modern databases query optimizers like the ones described earlier designed for single-tenant applications. However, these query optimizers are not suitable for the multi-tenant environment. The Salesforce Query optimizer considers accessing data partitions that contain tenants' data rather than an entire table or index, accessing statistics of tenants, and group and user-level for each virtual multi-tenant object. This query optimizer considers the user who is executing a given application function by using related tenant-specific metadata with system pivot tables to build and execute optimized database queries. Moreover, Salesforce uses other types of statistics to help with any particular queries as custom indexes to show the total number of not null and unique values in the corresponding field, and histograms for pick list fields, which show the cardinality of each list value. However, when statistics is not helpful to generate optimal query a FallbackIndex pivot table efficiently used to find the requested results as a secondary search mechanism, instead of returning a disappointing error message.

This section listed different single-tenant database query optimizers that are suitable for a single-tenant database schema, but not for multi-tenant database schema. They are presented in this section as background examples of RDBMSs' query optimizers. Moreover, this section discussed how Salesforce optimizes its multi-tenant database that based on a multi-tenant database schema that consists of a set of metadata, universal data table, and pivot tables. Salesforce proposed a multi-tenant optimization method that based on their multi-tenant data storage. While this thesis proposes a multi-tenant optimization method that is based on EET multi-tenant database schema that is also proposed in this thesis.

2.6 MULTI-TENANT DATABASE ACCESS CONTROL

Access control is a security topic which was started back in the 1960s (Ren et al. 2012), since then various access control models have proposed such as Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role Based Access Control (RBAC) (Du, Wen & Yang 2010; Lang et al. 2009). David Ferraiolo and Richard Kuhn are the first who proposed the RBAC model in 1992, which introduces the role as a new concept to associate users to one or more roles that are associated with one or more permissions (Ferraiolo & Kuhn 1992; Du, Wen & Yang 2010). The growth of multi-tenant Cloud Computing services draws attention to security challenges that emerged due to the cloud vendor's resource sharing (Takabi, Joshi & Ahn 2010). It is unlikely that the cloud users would risk their data and their computing applications over the cloud in favour of reducing the Total Cost of Ownership (TCO), or using a flexible cloud service, unless the cloud service providers provide reliable and secure services. Outsourcing data to the cloud is one of the critical security challenges because this data is accessed among a large number of users from different organisations (Brian et al. 2012; Ren et al. 2012). As stated in section 2.3 multi-tenant database has three data isolation approaches, and this thesis focuses on the *Shared Database - Shared Schema* approach. This multi-tenant data isolation approach has challenges in supporting a secure database schema (Bezemer et al. 2010; Foping et al. 2009; Kwok, Thao & Linh 2008). These challenges include (1) isolating tenants data by ensuring that each tenant can access only his own data; (2) ensuring that the tenants' data is robust and secure; (3) optimizing database performance; (4) fulfilling different tenants' security business requirements by using a tenant-aware data management based on *Shared Database - Shared Schema approach* (Bezemer et al. 2010; Brian et al. 2012; Lazarov 2007; Schiller et al. 2011). In this section, three multi-tenant access control models are presented, including Siebel Systems, IBM DB2, and Salesforce.

2.6.1 SIEBEL SYSTEMS ACCESS CONTROL

Brodersen et al. (2004) state that the present single-organisation access control model is not suitable for multi-tenant database. Accordingly, it has proposed a multi-tenant role based access control method, which allows to have a plurality of tenants, where each tenant is the owner of a separate virtual database. This method supports an access control subsystem for multiple users who are seeking a data access, where each of the users has at least one organizational access attribute, and the data are stored in the underlying database. The database is divided into files; the files are divided into records within the file, and the individual records are divided into fields. This method is based on partitioning the individual database files in the database, which are based upon an attribute of ownership and/or a granted access control.

2.6.2 IBM DB2, ACCESS CONTROL

Arnold et al (2012) state that IBM DB2 has provided several approaches of data access in the database management systems level, including views, label-based access (LBAC), and row and column access control (RCAC). The views approach adds more management overhead because this approach uses views instead of tables. The LBAC approach creates labels on tables and columns, and these labels are granted to users or groups. IBM introduced in DB2 V10 the RCAC approach, which represents a second layer of security that works with the current table security model. This approach permits groups and users to access particular rows in a table and specifies the data accessed from some or all the table's columns. Furthermore, some of the column's data are masked with nulls, a user defined mask, or a column mask that restricts a user from accessing data within a column.

2.6.3 SALESFORCE ACCESS CONTROL

Salesforce 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

that are converted to objects, objects' fields and relationships, and other object definition characteristics that are tracked by Universal Data Dictionary (UDD) (Weissman & Bobrowski 2009). Salesforce is using an access control method wherein each tenant may have one or more users. Each user or group of users can have different types of access grants, which permit them to access different rows, including (1) the user rows, (2) rows for users below the user in a role hierarchy, (3) rows that are shared by a group that the user belongs to, and (4) rows that are manually shared by another user or group of users (Salesforce 2013; Weissman et al. 2012).

This section discussed different multi-tenant role based access control methods, and different approaches to access data from table columns and rows. However, these access control methods and approaches designed for multi-tenant database design other than EET multi-tenant database schema. Therefore, this thesis introduces an access control method that is suitable for the EET multi-tenant database schema.

2.7 BIG DATA

Big data is a popular term used to describe the growth, availability, and data collected from different digital information sources (Dobosz 2013; Kim, Trimi & Chung 2014). The tendency to Big Data increases and the amount of computational resources that required for storing and processing this data grows rapidly. Big Data is characterized on four dimensions of data growth known as the 4Vs: Volume, Velocity, Variety, and Veracity. *Volume* implies the size of the data in terabytes, petabytes, or even more. *Velocity* implies the speed of storing, retrieving, and processing data. *Variety* implies storing, retrieving, and processing different types of data, such as structured data (e.g. Data stored in a RDBMS), unstructured data (e.g. Audio, video, images, and large text), and semi-structured data (e.g. XML) (Kim, Trimi & Chung 2014). *Veracity* implies two aspects including consistency of the data, and trustworthiness of accessing and storing the data (Demchenko et al. 2013). An important challenge in the design of multi-tenant databases that support Big Data

application is to provide data storage that manages large volumes of data. Such a challenge is derived from the fact that traditional RDBMSs do not have the data management capabilities that suit the requirements of the contemporary multi-tenant cloud applications (Agrawal, Das & Abbadi 2010). Due to this fact, NoSQL data management systems started to flourish and become the data storage method for such applications, because it offers high scalability, elasticity, and availability (Agrawal, Das & Abbadi 2010). Nevertheless, there is a school of thought says that the RDBMSs have been leading IT (Information Technology) industry for decades and it is reliable and robust database management systems. Consequently, if RDBMSs can support the capabilities that fulfil the requirements of multi-tenant cloud applications, which the NoSQL support, then why would organisations choose NoSQL rather than RDBMS to store their Big Data? (Cattell 2011).

2.7.1 RDBMS AND SQL

In 1970, the IBM employee Edgar Codd was the first who presented the idea of the relational database model, which was designed to resolve the data storage redundancy and inconsistency (Codd 1970). In 1985, he identified twelve rules along with the fundamental rule that called Rule Zero. These rules defined how the design of the relational database model should look like. These rules published in (Codd 1985a, Codd 1985b) and listed below:

- **Rule 0 – The Information Rule:** A relational database management system must manage its stored data using only its relational capabilities. The remaining 12 rules are based on this fundamental rule.
- **Rule 1 - Information Representation:** The table format is the standard method of storing and presenting the relational data.
- **Rule 2 - The Guaranteed Access Rule:** Table name, column name, and the table primary key of a table row must address the values that are stored in the database.

- **Rule 3 - Systematic Treatment of Null Values:** Null values represent the missing information of a table column.
- **Rule 4 - Dynamic Online Catalog Based on the Relational Model**: The relational database should allow to access the database metadata (Data Catalog) using the same query language of the traditional database's data.
- **Rule 5 - The Comprehensive Data Sublanguage Rule:** A unified relational language should support the tasks in RDBMS.
- **Rule 6 - The View Updating Rule**: The database management system must be able to update the database views that are virtual tables based on the results of database queries.
- **Rule 7- High-level insert, update, and delete:** The insert, update, and delete operations should support dealing with a set of data rather than just a single row in a single table.
- **Rule 8 - Physical Data Independence**: Storing and retrieving operations should be independent from the underlying physical data storage of the database management system.
- **Rule 9 - Logical Data Independence:** The data structure of the database tables is independent from the application level and its user interfaces.
- **Rule 10 - Integrity Independence**: The database entity constraints such as entity integrity and relational integrity have to be specified in the database level instead of the application level.
- **Rule 11 - Distribution Independence:** The RDBMS must enable an application to work whenever the data is physically centralized or distributed.
- **Rule 12 – Nonsubversion**: If the RDBMS provides a low-level (record-at-a-time) interface, then that interface should not be used to challenge the system, by sacrificing the relational security or integrity constraint.

Since 1970s, the RDBMS has been the leading database management system in the IT industry (Cattell 2011), that is because the RDBMS can ensure the reliability and integrity of its data by sustaining its four data transaction properties, including

atomicity, consistency, isolation, and durability. The acronym ACID refers to these four properties, and they are defined as follows (Date 1990): *Atomicity* means either all the database transaction operations are executed successfully or none of them is executed. *Consistency* means that all the database transaction operations should be fully executed and transformed from one state to another, and during this process the new state should not be visible to the users or any other running transactional operations until the process is completed. *Isolation* means that different database transactions should work independently and without any interference from other concurrent running transactions. *Durability* means that a database transaction, which is completed successfully, should remain stored safely even if the database crashes or any other error occurred.

After introducing the relational database model in 1970 by Edgar Codd, in 1974, Chamberlin and Boyce from IBM introduced the Structured Query Language (SQL) (Chamberlin & Boyce 1974). IBM implemented SQL for the first time in System R, which is IBM's RDBMS. Since then, SQL has become the standard query language for RDBMS (Wikipedia)[3] SQL is used as a database query language for database systems such as Oracle, Sybase, DB2, SQL Server, Access, PostgreSQL, MySQL and others.

2.7.2 NoSQL

NoSQL is a non-relational databases that quickly gaining popularity (Leavitt 2010), due to its high scalability, elasticity, and availability (Agrawal, Das & Abbadi 2010). It offers APIs to manage and access the data, instead of using SQL. Moreover, its performance outperforms traditional RDBMS (Stonebraker 2010). In 1998, Carl Strozzi was the first one who used the term NoSQL, when he proposed his non-relational open source database that do not use SQL interface (Lith & Mattsson 2010). In 2007, Amazon published a paper describing their internal storage system that called Dynamo, which is a NoSQL storage system that is used for Amazon's internal systems (DeCandia et al. 2007; Leavitt 2010). The NoSQL data management

3 http://en.wikipedia.org/wiki/IBM_System_R; Accessed July, 2014

system is categorized into four types: (1) key-value stores (e.g. Redis, Memcached), (2) column-family stores (e.g. BigTable, Cassandra), (3) document stores (e.g. MongoDB, CouchDB, Riak), and (4) graph stores (e.g. Neo4j, DEX) (Sakr et al. 2011; Tweed & James 2010). As mentioned earlier RDBMSs sustain ACID properties, in contrast NoSQL does not. The unstructured approach of NoSQL data requires an alternative to the ACID model, this model known as BASE that is stands for Basically Available, Soft state, and Eventually Consistent (Cattell 2011). NoSQL focuses on achieving BASE through supporting partial failures instead of total system failures. In other words, this means that the BASE model focuses on perceiving higher availability of the system by having a high level of scalability that cannot be obtained by ACID. However, such a high level of scalability in the BASE model, sacrificing the data accuracy of a small percentage number of users who are using one particular host server from a large number of servers that are used for one software application (Pritchett 2008).

2.7.3 ISSUES IN RDBMS AND NOSQL

There are two possible reasons why it is being argued that RDBMSs are not suitable to store and manage Big Data: (1) RDBMSs are limited in offering good performance and scalability properties; and (2) it is uncommon to extend relational database schemas by adding, modifying, and deleting tables and tables' columns during the application's runtime execution (Leavitt 2010; Cattell 2011). Although, NoSQL has significantly emerged and used for Big Data applications (Agrawal, Das & Abbadi 2010), however, it has a number of issues and concerns that RDBMS does not have such as (1) it does not support ACID transactional properties; (2) most of the organisations are unfamiliar with this technology, which leads to adding difficulties to them in evaluating the feasibility of applying it or using it; (3) it requires manual query programming tasks that add extra overhead and complexity in software development lifecycle; (Leavitt 2010; Cattell 2011); (4) it is uncommon to have separation in the roles (e.g. System administrator, database administrator, and

developer) of the personnel who work on NoSQL data management systems, in contrast, such personnel should have a variety of skills and knowledge, including distributed database systems, special programming languages, and special analytical algorithms (Simmonds 2013); (5) the majority of NoSQL data management systems do not support joining operations, filtering on multiple properties, and filtering of data based on subqueries results (Dimovski 2013; Sakr et al. 2011); and (6) unless configuring NoSQL consistency models in protective modes of operation, NoSQL will not assure the data consistency and it might sacrifice query performance and scalability (Bobrowski 2011).

To sum up, NoSQL databases do not replace RDBMSs, but they are complimentary solutions to RDBMSs to provide enhanced data management capabilities (Indrawan-Santiago 2012; Leavitt 2010; Simmonds 2013), which manage unstructured and semi-structured data, while RDBMSs manage structured data (Kim, Trimi & Chung 2014; Leavitt 2010). Both RDBMS and NoSQL have a number of limitations in managing Big Data applications, and the optimal multi-tenant Big Data storage model does not exist yet. Simmonds (2013) states that there are around 150 implementations of NoSQL data management systems available in the market including open-source and commercial releases. Nevertheless, each of these data management systems has different design patterns, query methods, and implementations that make evaluating, adopting, and implementing these database management systems is a significant challenge (Simmonds 2013). In contrast, RDBMSs have a unified approach in dealing with the data using common SQL langue, but with minor differences between different database products (Cattell 2011). This fact attracts the attention to find ways to improve the existing RDBMSs by overcoming two of their issues. Firstly, the scalability issue, and secondly, the issue of extending RDBMS during the application's runtime execution. The first issue can be resolved by using any of the available distributed software products in the market that scale and optimize RDBMSs on the cloud, such as MySQL Cluster, VoltDB, Clustrix, ScaleDB, NuoDB, ScaleBase (Cattell 2011), and others. The

second RDBMS issue will be resolved in the rest of the chapters of this thesis, as well as the issues of NoSQL that listed in this section.

2.8 SUMMARY

This chapter reviewed the research areas that are related to this study. *First*, it reviewed SaaS that is an emerging Cloud Computing service and one of the significant topics in the software industry. *Second*, it reviewed multi-tenancy, which is the main characteristic of SaaS that allows SaaS service providers to run a single instance application, which supports multiple tenants on the same software and hardware infrastructure. Such an application should be highly configurable to improve the manageability of SaaS multi-tenant applications. It is widely known that many efforts are required to develop a highly configurable multi-tenant application. *Third*, it reviewed the multi-tenant database management, and from this review, it can be concluded that it is a common practice in SaaS applications to use the *Shared Database - Shared Schema* approach, which means a single database schema is used to accommodate all the tenants' data. *Fourth*, it reviewed seven multi-tenant database schema designs that were introduced to accommodate multiple tenants' data using a RDBMS. From this review, two conclusions can be drawn. (1) The main limitation and weakness of the seven multi-tenant database schemas except the Private Tables is that the common schema that is used by multiple tenants must be known in advance. While the multi-tenant database schema that this thesis proposes, which called EET offers three database models that overcome this issue; and (2) the Universal Table multi-tenant schema is considered the optimal schema to be used for multi-tenant applications. Based on this conclusion, in Chapter 4 of this thesis, the feasibility and effectiveness of EET were measured by comparing EET with Universal Table schema, which is commercially used by Salesforce. *Fifth*, it reviewed different single-tenant database query optimizers that are suitable for a single-tenant database schema, but not for multi-tenant database schema, including Oracle, SQL Server, and PostgreSQL, on the other hand, it reviewed Salesforce multi-tenant query optimizer.

Sixth, it reviewed different multi-tenant role based access control methods, including Siebel Systems, IBM DB2, and Salesforce.

Finally, it reviewed Big Data, its 4Vs characteristics, including Volume, Velocity, Variety and Veracity, and its three types of data including structured, unstructured, and semi-structured. It is a significant challenge to develop a multi-tenant Big Data application that is based on multi-tenant database, which provide a data storage that manages large volumes of data. There are two types of database management systems that are used for multi-tenant databases, first RDBMS, and second NoSQL. Both of these data management systems have clear benefits and limitations. NoSQL is gaining wide acceptance while relational databases have maintained a concrete place in the market for decades. NoSQL emerged in the market because the RDBMSs have limitations (discussed in Section 2.7.3) in meeting the demands of multi-tenant and Cloud Computing applications. However, by overcoming these limitations, RDBMSs could be suitable to store multi-tenant data as well as the three different data types of Big Data including structured, unstructured, and semi-structured. Furthermore, they could be scaled and performed well for modern online applications, such as SaaS, Big Data, web, mobile, social media, computational science, and other applications.

CHAPTER 3

MULTI-TENANT DATABASE FRAMEWORK ARCHITECTURE

SaaS model exploits economies of scale for SaaS service providers by offering the same software and hardware infrastructure for multiple tenants. This multi-tenant service requires a multi-tenant database design that delivers database functionalities for multiple tenants to create, store, and access their databases over the internet. Due to multi-tenant database resource sharing in this service, the multi-tenant schema should be highly secured, optimized, configurable, and extendable during the application's runtime execution to fulfil different tenants' business requirements. Nevertheless, the capabilities of this contemporary multi-tenant database schema are not supported by traditional RDBMSs. To overcome this issue, an intermediate software layer that mediates multi-tenant applications and RDBMS need to be used, to convert multi-tenant queries into regular database queries, and to execute them in a RDBMS. Developing such a multi-tenant software layer to manage and access tenants' data is a significant problem to solve and has significant complexities that involve longer development lifecycle. This chapter proposes an architecture design to build a multi-tenant database framework prototype to implement a novel multi-tenant database schema called Elastic Extension Tables (EET) in a RDBMS, and develop an intermediate software layer to be used between software applications and RDBMSs to store and access multiple tenants' data from EET multi-tenant database schema. This database layer integrates multi-tenant relational tables and virtual relational tables and

makes them operate virtually as a single database schema for each tenant. This multi-tenant database framework prototype is called EET framework. It is suitable for multi-tenant database environment that can run any business domain database. Moreover, this framework can be used as a base to build software applications in general and SaaS and Big Data applications in particular.

Cost effective scalability is very significant for multi-tenant applications. The maximum number of tenants that can be supported by a multi-tenant application can be increased as long as the resources increased while keeping the performance metrics of each tenant at an acceptable level (Foping et al. 2009; Liu 2010). The same case can be applied to the multi-tenant database. However, before start thinking to scale-up or scale-out multi-tenant database to optimize its performance. The multi-tenant database performance should be optimized in each single server instance by applying a multi-tenant architecture design, which includes a proper multi-tenant database schema design, and a proper multi-tenant query optimizer method, then any of the scale-out or scale-up approaches can be applied afterwards. Accordingly, the EET framework architecture design is focusing on how to optimize multi-tenant query performance in a single server instance and scalability will be out of this thesis scope. Nevertheless, it is one of the future research directions of this study.

The remainder of this chapter is structured as follows. Section 3.1 describes the EET overview architecture. Section 3.2 describes the EET conceptual architecture design. Section 3.3 concludes this chapter.

3.1 EET FRAMEWORK OVERVIEW ARCHITECTURE

This section presents the overview architecture design of EET framework that guides database vendors on how to design and develop a single database application, which supports multiple tenants on the same software and hardware infrastructure. This architecture is based on the EET multi-tenant database schema, and the *Shared*

Database - Shared Schema data isolation approach of multi-tenant database and Level 3 of SaaS Maturity Model that reviewed in Chapter 2. The EET framework can be used to implement any business domain database such as CRM, HR, Accounting, or any other business domain. In addition, it can be used to store information that collected from social media networks, e-mails, blogs, news, online texts and documents, and other data sources. Moreover, it is exposed to be used by the tenant's developers by accessing its APIs to store and retrieve the tenants' data over the internet, and build multi-tenant applications without worrying about the infrastructure database. Accordingly, the architectural design of this framework can be used as multi-tenant database cloud service to offer database storage for multiple tenants who can access this service by calling functions from the APIs of the EET framework.

The overview architecture of the EET frame is proposed in Figure 3-1 that shows the main six layers of EET framework architecture, including the presentation layer, the API layer, the service layer, the Data Access Object (DAO) layer, the Object Relational Mapping (ORM) layer, and the domain layer. The presentation layer represents the applications that can access EET database architecture such as SaaS, Big Data, mobile, web, and stand-alone software applications. The API layer consists of two APIs, including the EET Data Management APIs, and the EET Data Retrieval APIs. The service layer consists of four services, including the EET Access Control Service (EETACS), the EET Proxy Service (EETPS), the EET Query Optimizer Service (EETQOS), and EET Schema Handler Service (EETSHS). The DAO layer consists of two DAOs, including the Common Tenant Tables Data Access Object (CTTDAO), and the EET Data Access Object (EETDAO). The ORM layer is a virtual object database that can be used by the DAO layer to access the domain layer.

The architectural design of EET framework is based on a three-tier architecture design. The presentation layer and the API layer represent the presentation tier. The service layer represents the application tier. While, the DAO layer, the ORM layer, and the domain layer represent the data tier.

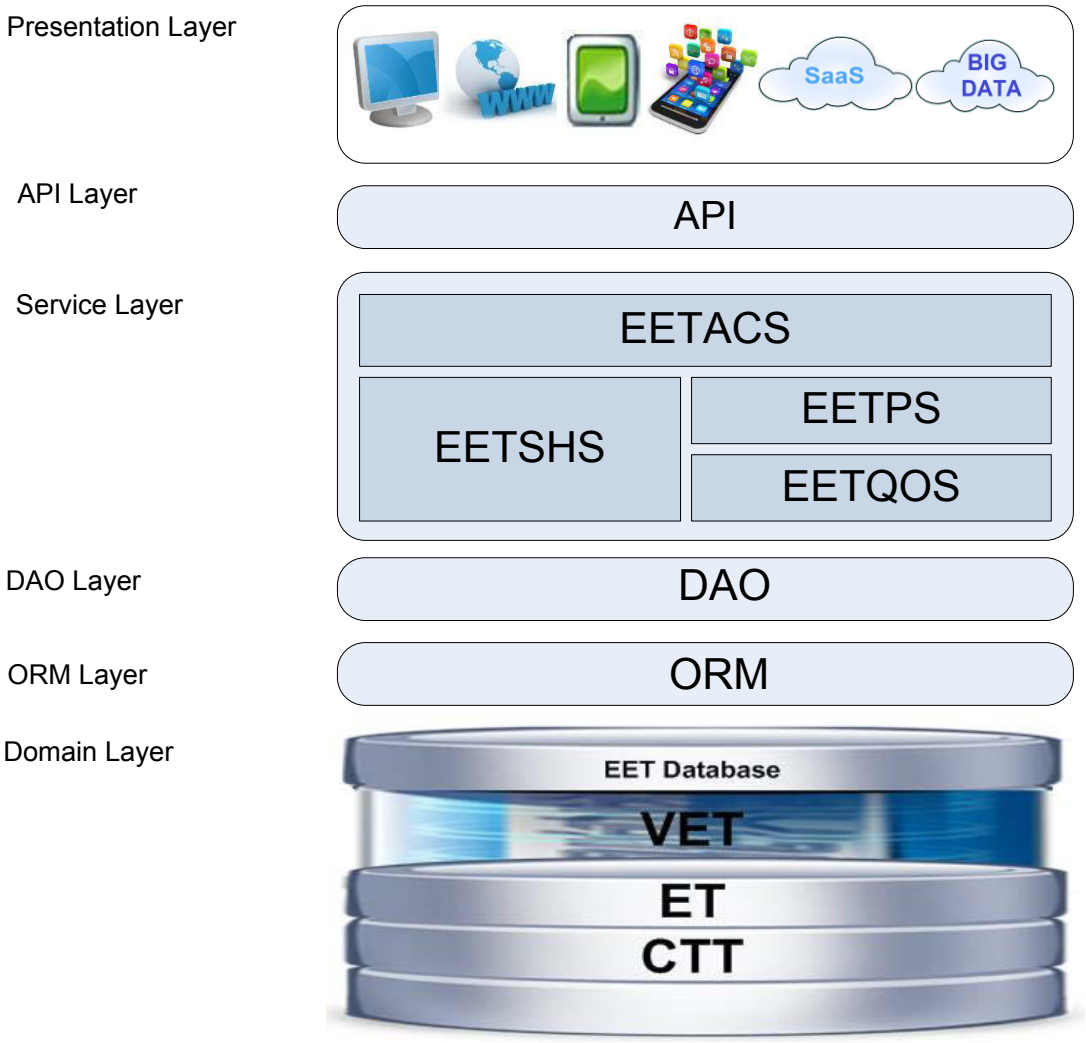


Figure 3-1: EET overview architecture

3.2 EET Framework Conceptual Architecture Design

The architecture design of EET framework comprises of eight artefacts: EET, EETPS, EETQOS, EETSHS, EETACS, DAO, ORM, and EET APIs. Figure 3-2 shows the essential elements of the EET conceptual architectural design.

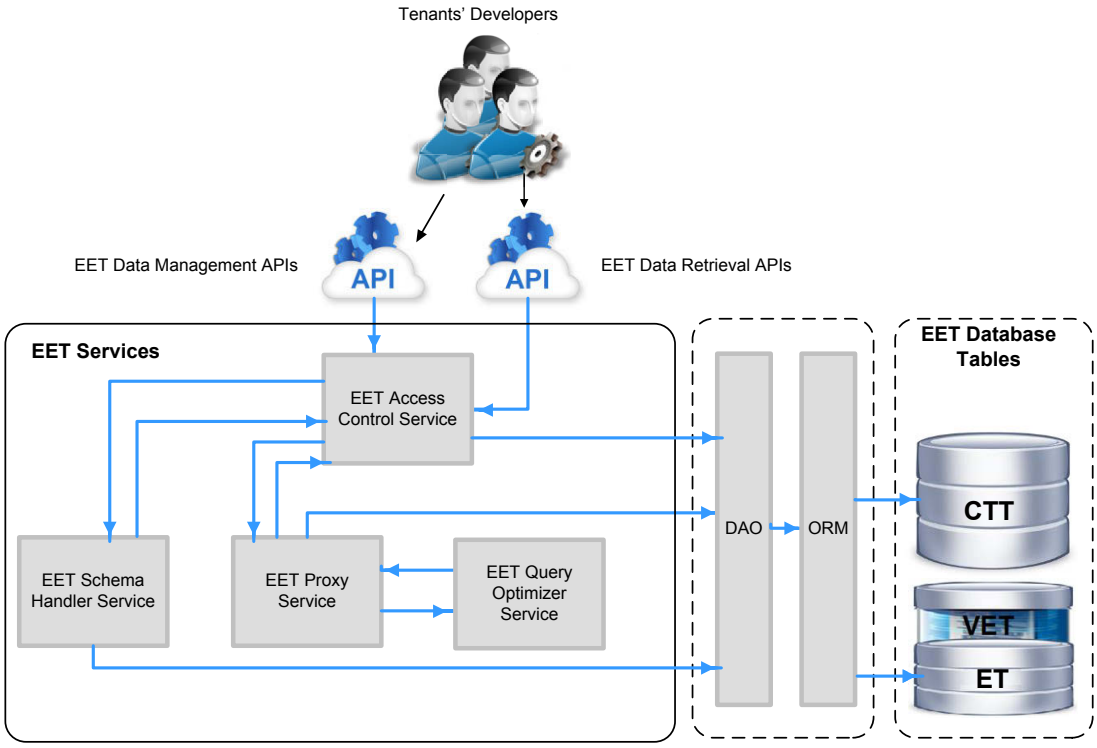


Figure 3-2: EET conceptual architecture design

3.2.1 Elastic Extension Tables

This section summarises the EET multi-tenant database schema that is the base of the EET Framework, and Chapter 4 presents EET in details. The EET consists of Common Tenant Tables (CTT), Extension Tables (ET), and Virtual Extension Tables (VET). The data architecture details of the eight ETs of EET are shown in Figure 4-1 of chapter 4 and listed as follows: (1) the 'db_table' ET allows tenants to create virtual tables and give them unique names. (2) The 'table_column' ET allows tenants to create virtual columns for a virtual table stored in the 'db_table' ET. (3) The table row ETs store records of virtual extension columns in three separate tables. These tables are separated to store small data values in the 'table_row' ET such as NUMBER, DATE-and-TIME, BOOLEAN, VARCHAR and other data types. On the other hand, the large data values are stored in two other tables. First, the

'table_row_blob' ET, which stores a URI for virtual columns of BLOB data type. Second, the 'table_row_clob' ET, which stores CLOB values for virtual columns with TEXT data type. These three types of tables are capable to store all the data types of Big Data, including traditional relational data, texts, audios, images, and videos in structured, semi-structured, and unstructured format. (4) The 'table_relationship' ET allows tenants to create virtual relationships for their virtual tables with any of CTTs or VETs. (5) The 'table_index' ET is used to add indexes to virtual columns. These indexes reduce the query execution time when tenants retrieve data from a VET. (6) The 'table_primary_key_column' ET allows tenants to create single or composite virtual primary key for virtual extension columns that are stored in the 'table_column' ET.

3.2.2 EET SCHEMA HANDLER SERVICE

The EET Schema Handler Service (EETSHS) proposes a method to manage multiple tenants' data in EET multi-tenant database schema. This service enables tenants to do the followings: (1) creating the required number of tables and columns; (2) creating virtual database relationships; (3) assigning suitable data types and constraints for table columns; (4) managing CTT and VET rows during multi-tenant application's runtime execution. This service is exposed to be used by EET Data Retrieval APIs via EETACS. It consumes the EETACS to grant different types of access control to the tenants' users. Moreover, it accesses the DAO layer to create or delete VETs, and retrieve and/or modify CTTs and VETs data structures and fields' details. The details of this service are presented in Chapter 5.

3.2.3 EET PROXY SERVICE

The EET Proxy Service (EETPS) integrates, generates, and executes tenants' queries by using a codebase solution, which converts multi-tenant queries into traditional database queries. This service has two objectives, firstly, allowing tenants'

applications to retrieve table rows from CTTs, retrieve integrated table rows from two or more CTTs and VETs, or retrieve rows from VETs. Secondly, avoiding tenants from spending money and efforts on writing SQL queries and backend data management codes by 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. Such functions convert multi-tenant queries into traditional database queries and execute them in a RDBMS. This service gives tenants the opportunity of satisfying their different business needs and requirements by choosing from any of the EET database models that are stated in chapter 4. This service consumes EETACS and EETQOS, and accesses the DAO layer to retrieve data from CTTs and VETs. In addition, it is exposed to be used by EETACS, EETQOS, EETSHS, and EET Data Retrieval APIs via EETACS. The details of this service are presented in Chapter 6.

3.2.4 EET QUERY OPTIMIZER SERVICE

The EET Query Optimizer Service (EETQOS) optimizes the performance, speeds up query retrievals, and uses the most efficient way to execute a multi-tenant queries in multi-tenant database, by estimating the cost of different query execution plans to determine the optimal plan, by using: (1) Virtual primary key indexes; (2) Virtual foreign key indexes; (3) Custom indexes; (4) A tenant's user access control methods; (5) Separating the tenant's data in three row ETs that store small data values in the 'table_row' ET and large data values in two other ETs, including the 'table_row_blob' and the 'table_row_clob' ETs. Then, this plan executes a tenant's query by consuming EETPS that converts multi-tenant queries into traditional database queries, and then executes them by using a query optimizer of any RDBMS. This service is exposed to be used by EETPS and consumes the EETPS. The details of this service are presented in Chapter 7.

3.2.5 EET ACCESS CONTROL SERVICE

The EET Access Control Service (EETACS) proposes an access control method, which permits each tenant in a multi-tenant database to have several users with different types of grants to access the tenant's data. The concept of retrieving data from the multi-tenant database is slightly different from the single-tenant database. The single-tenant database does not differentiate between the data of different tenants. While, the data of the multi-tenant database is partitioned to differentiate between data owned by multiple tenants, to access table rows that are granted to a tenants' users based on a number of groups or roles assigned to them. This service is granting access to users who are accessing the EET Data Retrieval APIS, to retrieve data from CTTs and/or VETs by consuming the EETPS. In addition, it is granting access to users who are accessing the EET Data Management APIs, to manage the data in CTTs and VETs by consuming EETSHS and EETPS. In addition, this service is exposed to be used by EETPS and EETSHS. The details of this service are presented in Chapter 8.

3.2.6 DATA ACCESS OBJECT

The Data Access Object (DAO) (Wikipedia) [4] is a software layer that is used and included in the EET architecture design, to decouple accessing data from its underlying RDBMS storage. In the EET architectural design, this software layer is composed of two DAOs. The first DAO is the CTTDAO, which is the interface that accesses the CTTs, and the second DAO is the EETDAO, which is the interface that accesses the EETs, and through accessing EETs, it accesses VETs. This DAO layer consumes the ORM layer to access through it the EET, and it is exposed to be used by EETACS, EETPS, and EETSHS. This software layer have included in the.

4 http://en.wikipedia.org/wiki/Data_access_object; Accessed July, 2014

3.2.7 OBJECT RELATIONAL MAPPING

Object Relational Mapping (ORM) mediates between object oriented architecture system and relational database environment (Xia, Yu & Tang 2009). It is an abstraction layer that is used over RDBMS such as PostgreSQL, MySQL, Oracle, Microsoft SQL Server, and other RDBMS. The benefit of this database layer is the notion of database portability that allows to migrate the database layer from one database vendor to another. Hibernate is an ORM library that is used for Java programming language. It is used in the architectural design of EET framework to access data from EET, and is exposed to be used by CTTDAO and EETDAO. Such a software layer has included in the EET architecture design, in order to make the database layer that is proposed in this chapter a portable layer that can be orchestrated with any RDBMS.

3.2.8 EET APIS

EET consists of two types of APIs. First, the EET Data Retrieval API that provides EET database web service interface that retrieves tenants' data over the internet integrates them with other applications, and combines them with multiple internet or cloud services to build mashups applications or services. This APIs consumes the EETPS via the EETACS, and it is exposed to be used by tenants' developers. Second, the EET DATA MANAGEMENT API that provides EET database web service interface that creates, updates, and deletes the tenants' VET, and tenants' data over the internet. This APIs consumes the EETSHS via EETACS, and it is exposed to be used by tenants' developers. Building these APIs is out of this thesis scope. However, it is one of the future work directions of this study. The APIs artefacts were introduced in the EET framework architecture to show a complete scenario on how

EET framework works and how the EET APIs orchestrate with the other artefacts of the framework.

3.3 SUMMARY

Designing and developing a configurable multi-tenant database that generates and executes tenants' queries by using a codebase solution, and converts multi-tenant queries into regular database queries, then execute them in a RDBMS is hard, complicated, and requires extra work and time to be achieved. This chapter has presented a multi-tenant database architecture design of EET framework that based on the EET multi-tenant database schema, and *Shared Database - Shared Schema* data isolation approach of multitenant database and level 3 of SaaS Maturity Model that reviewed in Chapter 2. This framework simplifies and speeds up the development of multi-tenant database solutions. It allows database service providers to create a single database application that supports multiple tenants on the same software and hardware infrastructure. Moreover, it overcomes multi-tenant database challenges from technical and business perspectives and reduces the TCO from the tenants' perspective. That is because, it avoids the tenants from spending money and efforts on writing SQL queries and backend data management code, by accessing APIs that manages tenant's data and retrieving simple and complex queries including join operations, filtering on multiple properties, and filtering of data based on subqueries results. Furthermore, it allows tenants to store different data types of Big Data including structured, semi-structured, and unstructured data, which are collected from various online sources of information. Whereas, from the database service provider perspective, it reduces the ongoing operational costs, by providing a database self-service to configure and manage the tenants' data by the tenants themselves, rather than the database service provider. This database solution is suitable to be used by tenants' developers, to store and access the tenants' data from the cloud to build their applications, or integrate this data with other applications or online data sources without spending much time and efforts on managing their database. Consequently,

the database layer that the EET framework provides can be used as a base to build software applications in general and SaaS and Big Data applications in particular. In this framework prototype, all the artefacts have implemented except the APIs, which is not part of the thesis objectives, and it is out of this thesis scope, but it is one of this study future work directions. Nevertheless, the APIs artefact introduced in the architecture of the EET framework to show a complete scenario on how EET framework works.

CHAPTER 4

MULTI-TENANT DATABASE SCHEMA DESIGN

Nowadays, a large number of companies are offering their web-based business application by adopting the SaaS model. Multi-tenancy is the primary characteristic of SaaS, it allows SaaS vendors to run a single instance application which supports multiple tenants on the same software and hardware infrastructure. This application should be highly configurable to meet the tenants' expectations and their business requirements. Such an application requires a highly elastic and configurable multi-tenant database that can be used to store different tenants' data in a single database schema. This chapter proposes a novel multi-tenant database schema called Elastic Extension Tables (EET) that consists of Common Tenant Tables (CTT), Extension Tables (ET), and Virtual Extension Tables (VET). This multi-tenant database schema gives tenants the opportunity to address their individual business requirements by choosing from three database models: Multi-tenant Relational Database, Integrated Multi-tenant Relational Database and Virtual Relational Database, and Virtual Relational Database. In addition, it allows tenants to store different data types of Big Data in structured, semi-structured, and unstructured format

The remainder of this chapter is structured as follows. Section 4.1 proposes the Elastic Extension Tables. Section 4.2 proposes the Elastic Extension Tables database models. Section 4.3 presents an example to compare multi-tenant database schema designs with the Elastic Extension Table design. Section 4.4 compares the

performance of accessing data from EET and Universal Table Schema Mapping (UTSM) (Liao et al. 2012). Liao et al. (2012) state that the UTSM data architecture is similar to Salesforce data architecture. In addition, a number of database queries examples presented in (Liao et al. 2012; Liao et al. 2013) that are used to retrieve data from this data architecture. Some of these queries are used in the experiments of this chapter, in addition to other queries that are used to show the difference in accessing data from EET and UTSM. The UTSM technique had to be chosen to compare it with EET technique, because as reviewed and concluded in the Literature Review (Chapter 2), the Universal Table that is used in UTSM, is considered as the optimal schema design for multi-tenant applications. Moreover, it is one of the multi-tenant database schema techniques implemented commercially by Salesforce that the American business magazine Forbes [5,6] selected it as the most innovative company in the world in the year 2011, 2012, and 2013. Section 4.5 concludes this chapter.

4.1 Elastic Extension Tables

The EET multi-tenant database schema proposes a novel way of designing and creating an elastic database that consists of three table types, the first type is CTT, the second type is ET, and the third type is VET. Figure 4-1 shows the details of EET multi-tenant schema. The design of this schema enables tenants to build their own virtual database schema by creating the required number of tables, columns, rows, virtual database relationships, and assigning suitable data types and constraints for table columns during the runtime execution of a multi-tenant application.

4.1.1 Common Tenant Tables

The Common Tenant Tables are the tables that can be shared between tenants who are using a multi-tenant single database schema. These tables are traditional physical tables that are based on RDBMS, and are used as a business domain database schema that is shared between multiple tenants. For example, a multi-tenant application of a sales business domain may have a database schema with sales tables, such as

5 http://www.forbes.com/innovative-companies/list; Accessed July, 2014
6 http://www.salesforce.com/company/awards/most-innovative-companies-salesforce-no1-forbes.jsp; Accessed July, 2014

salesperson, customer, product, sales-fact, and any other sales tables. These tables have columns that are used by most of the tenants, and the column tenant ID is used to differentiate between the tenants' rows. For example, the 'sales_person' CTT in Figure 4-11 shows some common columns, such as 'first_name', and 'last_name', while the 'tenant_id' column is used to differentiate between the tenants' rows.

4.1.2 EXTENSION TABLES

The Extension Tables are metadata tables that are used to create virtual tables for multiple tenants who are using a single multi-tenant database schema during the application's runtime execution. The ET consists of the following eight physical tables:

- **Db_table Extension Table:** The 'db_table' ET allows tenants to create virtual (logical) tables and give them unique names. The structure of this table has a composite primary key that consists of 'db_table_id' and 'tenant_id' columns. The 'db_table_id' column is a unique primary key of the table, while the 'tenant_id' column is a foreign key refers to the 'tenant' CTT and at the same time is a combined primary key with 'db_table_id' for this table. In addition, this table has the 'db_table_name' column that stores the virtual tables' names. In using this table, each tenant can have unique table names. For example, tenant-A can create a VET name 'sales_person', but cannot create the same VET name again for his VETs. However, tenant-B can create the 'sales_person' name even if tenant-A already created this VET's name.
- **Table_column Extension Table:** The 'table_column' ET allows tenants to create virtual columns for a VET that created in the 'db_table' ET. The structure of this table has a composite primary key consists of 'table_column_id', 'tenant_id', and 'db_table_id'. The 'table_column_id' is a unique primary key for this ET, while the other two columns 'tenant_id' and 'db_table_id' are primary keys in this table, and foreign keys that refer to primary key columns of the 'tenant' CTT, and the 'db_table' ET. Moreover, this table has other columns,

including 'table_column_name', 'default_value', 'data_type', 'is_indexed', 'is_null', 'is_relationship', 'is_primary_key_column', and 'is_unique_column'. The 'table_column_name' column has UNIQUE constraint, and VARCHAR data type. The 'default_value' column stores already defined value to be used once the database saves a table row, when there is no value specified to be stored in this column. The 'data_type' column specifies the data type of a virtual column that is stored into any of the three row ETs, which are presented in the following point. The 'is_indexed' column specifies whether a column has an index or not. The 'is_null' column specifies whether a column accepts to store NULL values or not, and if it does not, then this column is considered a mandatory column that must have a value. The 'is_relationship' column specifies whether a column has at least one relationship with any of the CTTs or the VETs. The 'is_primary_key_column' column specifies whether the column is a primary key or not. The 'is_unique_column' column specifies whether a column has a UNIQUE constraint or not.

- **The Row Extension Tables**: The row ETs store virtual table rows for virtual extension columns in three separate ETs. Such ETs are separated in three tables in order to store small data values in the 'table_row' ET, which stores values such as NUMBER, DATE-and-TIME, BOOLEAN, VARCHAR and other data types. While large data values are stored in other two ETs, the first ET is the 'table_row_blob' that stores BLOB values of virtual columns that stores BLOB data type (e.g. Images, Audio, Video), and the second ET is the 'table_row_clob' that stores CLOB values for virtual columns that store TEXT data type (e.g. E-mails, web pages). The EET design separates these three ETs to reduce the impact of BLOB and CLOB values from slowing down virtual schema queries. These three tables have the same columns, except the table row ID column, which is called differently in the three tables. In the 'table_row' ET called 'table_row_id', in the 'table_row_blob' ET called 'table_row_blob_id', and in the 'table_row_clob' called 'table_row_clob_id'. A table row ID can be given for

several columns that map to one row in a VET. Figure 4-14 shows an example of this mapping. The corresponding columns in these three tables include, first, the 'serial_id' column that is a composite primary key in these tables. This column stores a serial number of a virtual column that maps to a row in the virtual table. Second, the foreign key columns, including 'tenant_id', 'db_table_id', and 'table_column_id' that at the same time are composite primary keys with the Table Row ID column and the 'serial_id' column. Third, the 'value' column that stores the virtual column values, however, the data types of these columns vary in each of the three row tables according to the data types that supposed to be stored in each table. These three row ETs are capable to store all the Big Data types, including traditional relational data, texts, audios, images, videos, and XML in structured, unstructured, and semi-structured format. The structured data, such as traditional relational data can be stored in CTTs and VETs as it is presented in the EET design in Section 4.3. The unstructured data files such as images, audios, videos can be stored in EET, by storing the Uniform Resource Identifier (URI) of a file in the 'table_row_blob' ET. Then the actual physical file can be stored in a folder of a file system, and then this file can be accessed using the URI that stored in the 'table_row_blob' ET and mapped to the physical file that stored in a folder. While the semi-structured data such as XML files can be used in two ways. Firstly, using the same way of storing the unstructured data, then accessing the XML file using the URI that stored in the 'table_row_blob' ET and mapped to the physical XML file that stored in a folder. Secondly, an XML file can be stored as text in the 'table_row_clob' ET as a CLOB file, and then this XML file is accessed from the 'table_row_clob' ET.

- **Primary Key Extension Table:** The 'table_primary_key_column' ET allows tenants to create virtual primary keys for the virtual extension columns which are stored in the 'table_column' ET. The structure of this table has a composite primary key consists of 'table_primary_key_column_id', 'tenant_id', 'db_table_id', and 'table_column_id'. The 'table_primary_key_column_id'

column is a unique primary key of the table, while the other three columns 'tenant_id', db_table_id', and 'table_column_id' are primary keys and foreign keys. The 'is_auto_increment' column specifies whether a primary key can be auto-incremented or not. The 'is_composite_key' column is used to specify whether a virtual primary key that is stored in a table is a single primary key or a composite primary key.

- **Relationship Extension Table:** The 'table_relationship' ET allows tenants to create virtual relationships between their VETs and CTTs. The table structure has a composite primary key consists of 'table_relationship_id', 'tenant_id', 'db_table_id', and 'table_column_id'. The 'table_relationship_id' column is a unique primary key of the table, while the other three columns 'tenant_id', 'db_table_id', and 'table_column_id' are primary keys and foreign keys. The 'table_type' column specifies whether the relationship is with a CTT or a VET. The 'target_table_id' column is used to create a master-detail relationship between two VETs, by storing into it the table ID of the master VET that is stored in the 'db_table' ET, while the 'targeted_column_id' column is used to store into it the primary key ID of the master VET for the same relationship. The 'shared_table_name' column is used to create a master-detail relationship between a CTT and a VET, by storing into it the name of the master CTT while the name of the 'shared_column_name' column is used to store the primary key column name of the CTT for the same relationship. Furthermore, this ET can create a master-detail relationship between two VETs, or a CTT and a VET, even if the master table has composite primary keys. Such a relationship can be achieved by storing multiple table rows into the 'table_relationship' for the relationship that is between the master table that has a composite primary key, and the details VET. Each of these table rows denotes one of the primary key columns of the composite primary key that relates to the master table. The following are the database relationships that can be created using the 'table_relationship' ET between two

VETs, two CTTs, or one VET and one CTT, including One-to-One, One-to-Many, Many-to-One, Many-to-Many, and Self-referencing.

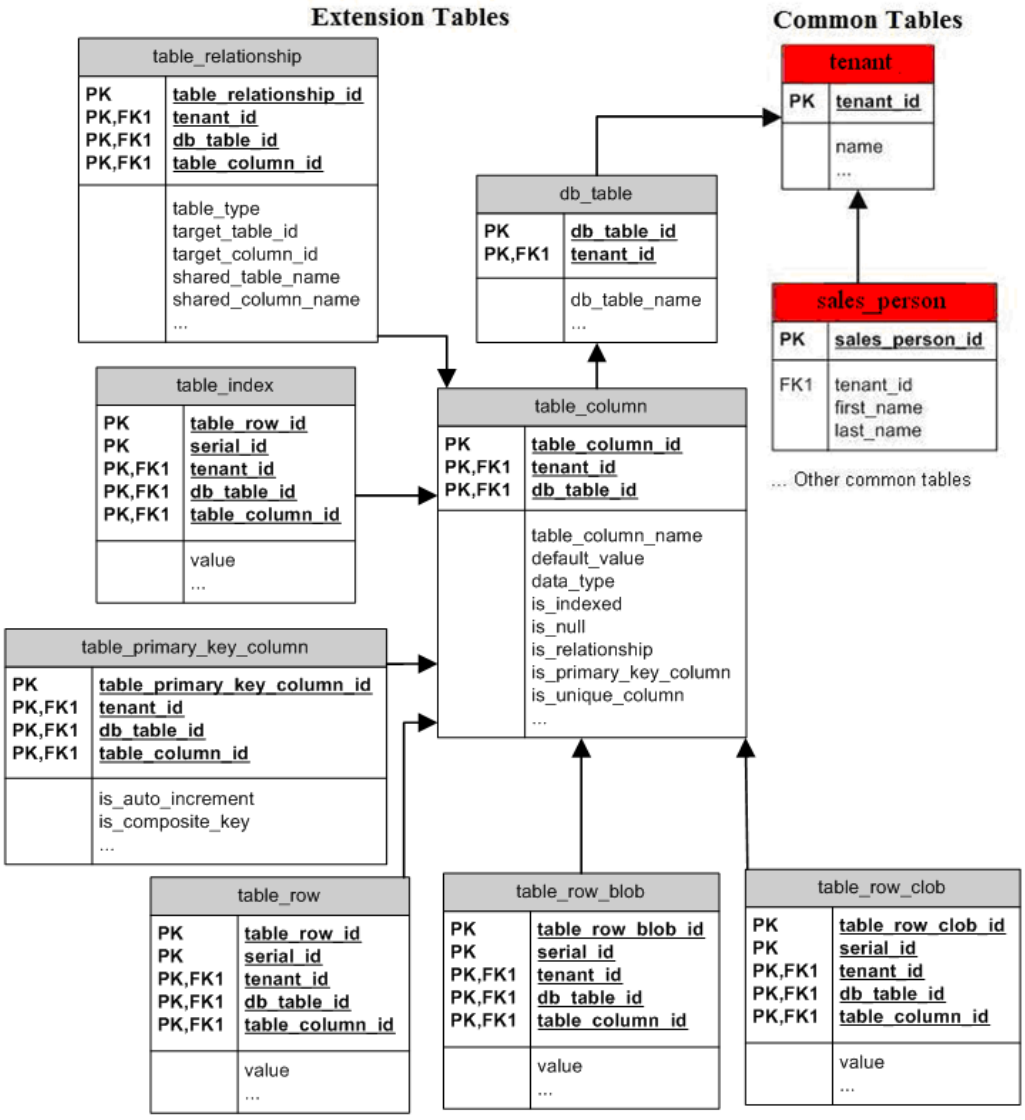


Figure 4-1: Elastic Extension Tables

- **Index Extension Table:** The 'table_index' ET is used to add indexes for virtual columns of a VET to improve and speed up the query execution time when retrieve data from this VET. The structure of this table has a composite primary key consists of 'table_row_id', 'serial_id', 'tenant_id', 'db_table_id', and

'table_column_id'. The column 'table_row_id' and 'serial_id' are unique primary keys that are referred to values stored into 'table_row_id' and 'serial_id' columns in the 'table_row' ET. While the other three columns 'tenant_id', 'db_table_id' and 'table_column_id' are primary keys and foreign keys for this table. The 'value' column stores a value that is stored in the 'table_row' ET and this value relates to an indexed virtual column, which is specified as an index in the 'table_column' ET by storing the necessary value in the 'is_indexed' column.

4.1.3 VIRTUAL EXTENSION TABLES

The Virtual Extension Tables are the tables that tenants can create during the application's runtime execution to extend an existing business domain database schema, or they can create their own virtual database schema from the scratch to fulfil their business needs. In Section 4.3, a detailed example is presented to explain how the tenants can create their VETs. In EET, VETs are created as a metadata into the eight ETs. In using this approach, the service provider who is offering a business domain database, can accommodate a huge number of virtual tables by allowing tenants to populate their data in these eight ETs. Such an approach allows the multi-tenant database service providers to manage their services in an efficient way and cost-effective manner, and simultaneously allows each tenant to configure his database schema and makes him feel as if he is the only tenant using the EET schema.

4.2 ELASTIC EXTENSION TABLES DATABASE MODELS

The EET multi-tenant database schema allows service providers to offer three database models, which give tenants the opportunity of satisfying their various business requirements by choosing from any of these three database models (Figure 4-2):

- **Multi-tenant relational database:** This database model allows tenants to use a ready relational database structure for a particular business domain database without any need of extending on the existing database structure. This business domain database, can be shared between multiple tenants and differentiate between them by using a Tenant ID column in the CTTs (physical tables). This model can be applied to any business domain database such as CRM, Accounting, HR, or other business domains.
- **Integrated multi-tenant relational database and virtual relational database:** This database model allows tenants to use a ready relational database structure of a particular business domain with the ability of extending on this relational database by adding more VETs, and to integrate these tables with the CTTs (existing database structure) by creating virtual relationships between them.
- **Multi-tenant virtual relational database**: This database model allows tenants to create their virtual database structures from the scratch, by creating VETs, virtual database relationships between the VETs, and other database constraints to satisfy the tenants' special business requirements of the tenants' business domain applications.

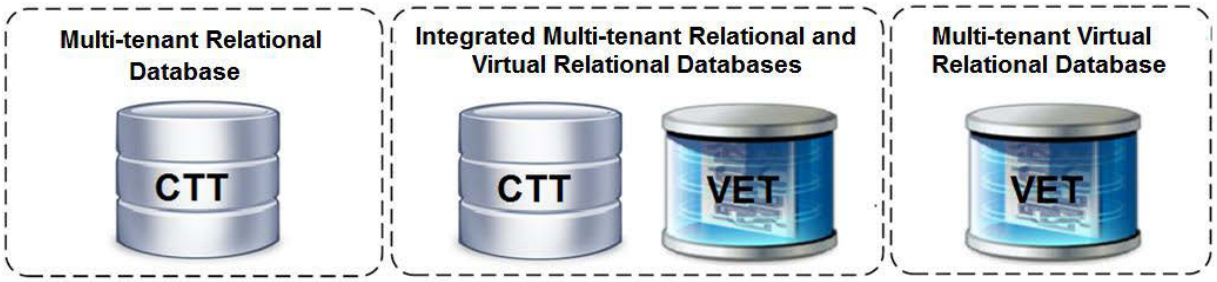


Figure 4-2: The Three EET Database Models

For example, if a service provider offers a sales database schema to be used by multiple tenants, and with this database schema the service provider uses the EET, then this service provider can provide the three database models listed above that fulfil various tenants' business requirements. This example assumes that the service provider has three tenants. The first tenant evaluated the sales database, and he found that this database suits his business requirements. Therefore, this tenant was interested to use the sales database schema as originally provided by the service provider as shown in Figure 4-3 (a). The second tenant evaluated the sales database schema and found that he needs to add extra tables to fulfil his business requirements. Thus, this tenant created VET 1, VET 2, and VET 3, and then, created virtual database relationships between these VETs and the already existing physical tables (CTTs) in the sales database schema. The database model that this tenant used is shown in Figure 4-3 (b). The third tenant evaluated the same database schema and found that it did not suit his business requirements. Therefore, he decided to not use the sales database schema of the service provider, and instead he created virtual relational tables from scratch and established database relationships between them as shown in Figure 4-3 (c). This example summarises the three database models of EET multi-tenant schema. Using these database models, tenants can design their databases and automatically configure their behaviours during their application’s runtime execution.

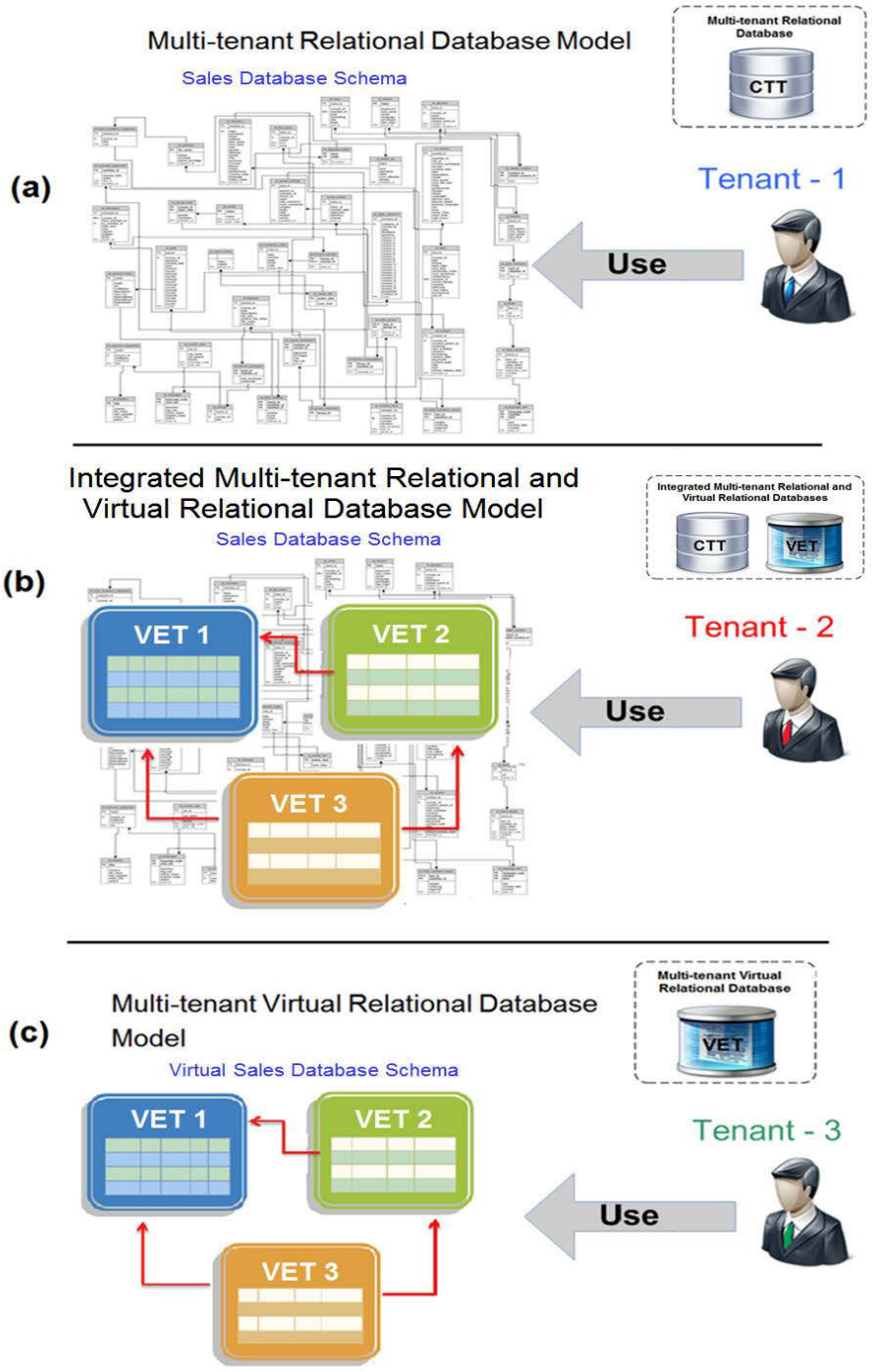


Figure 4-3: The EET Three Database Models Example.

4.3 AN EXAMPLE TO COMPARE MULTI-TENANT DATABASE SCHEMA DESIGNS WITH ELASTIC EXTENSION TABLES

This section presents an example that clarifies the seven multi-tenant database schema designs that presented in the literature review in chapter 2, and clarifies the differences between these designs and the EET multi-tenant schema design. This example shows three different tenants, including Tenant-A, Tenant-B, and Tenant-C. Each of these tenants uses a multi-tenant database, and in this database, they configure their sales database structure according to their different business needs. For simplicity, this example illustrates only one sales table that stores a sales person's information by using different multi-tenant database schema designs. Moreover, this example presents how the EET enables tenants to create their own database schema by extending an existing database schema based on RDBMS, including the required number of tables and columns, rows, virtual database relationships with any of the CTTs or VETs, primary keys for the columns, indexes for the columns, and assigning suitable data types for columns during multi-tenant application run-time execution. Furthermore, in this example, the data that is stored in the tables are the same across all the designs, in order to show the difference between their tables' structures and how data is populated in these structures.

The Private Tables in Figure 4-4 show three tenants who each of them has a different sales person table that fulfil their business requirements. Tenant-A has the 'sales_person_tenant_a' table, which consists of six columns, including 'sales_person_id', 'first_name', 'last_name', 'phone', 'age', and 'gender'. Tenant-B has the 'sales_person_tenant_b' table, which consists of four columns, including 'sales_person_id', 'first_name', 'last_name', and 'business_id'. Tenant-C has the 'sales_person_tenant_c' table; the columns in this table are the same as 'sales_person_tenant_a' table. The same data that populated in the private table is

populated in the rest of the multi-tenant database schema designs and EET schema, which are presented in the example of this section.

sales_person_tenant_a

sales_person_id	first_name	last_name	phone	age	gender
100	Joseph	Richard	02123456789	25	male
101	Sarah	Smith	02123456788	34	female

Tenant-a (table 1)

sales_person_tenant_b

sales_person_id	first_name	last_name	business_id
200	David	John	123456

Tenant-b (table 2)

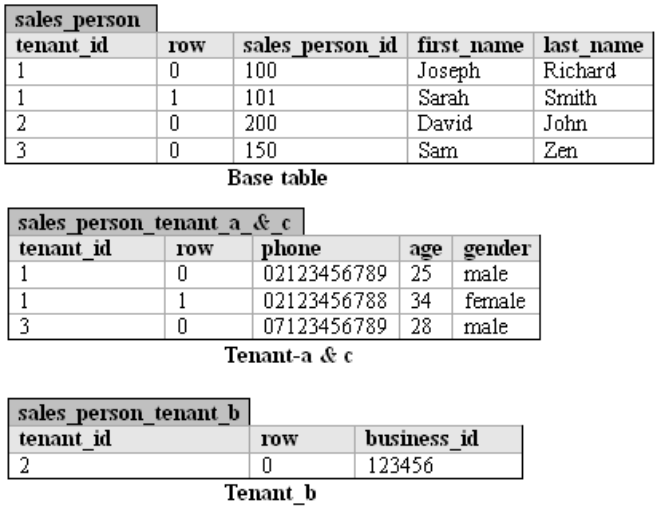
sales_person_tenant_c

sales_person_id	first_name	last_name	phone	age	gender
150	Sam	Zen	07123456789	28	male

Tenant-c (table 3)

Figure 4-4: Private Tables

The Extension Tables in Figure 4-5 show how the columns of the sales person tables for the three tenants split-up between the base table ‘sales_person’ and two extension tables ‘sales_person_tenant_a_&_c’ and ‘sales_person_tenant_b’. All of these three tables have two fixed common columns, including ‘tenant_id’ and ‘row’. The ‘tenant_id’ column is used to map data rows in the base table and the extension tables with the tenant who owns these rows. The ‘row’ column is used to give each row in the base table a row number and map it with other rows in the extension tables. The ‘sales_person’ base table has five columns, including ‘tenant_id’, ‘row’, ‘sales_person_id’, ‘first_name’, and ‘last_name’. All the tenants share the last three columns. The extension table ‘sales_person_tenant_a_&_c’ has five columns, including ‘tenant_id’, ‘row’, ‘phone’, ‘age’, and ‘gender’. This table is shared by two tenants Tenant-A and Tenant-C, due to the similarity in the extension columns that both tenants need. The ‘sales_person_tenant_b’ is used by Tenant-B, which has three columns ‘tenant_id’, ‘row’, and ‘business_id’.



sales_person

tenant_id	row	sales_person_id	first_name	last_name
1	0	100	Joseph	Richard
1	1	101	Sarah	Smith
2	0	200	David	John
3	0	150	Sam	Zen

Base table

sales_person_tenant_a_&_c

tenant_id	row	phone	age	gender
1	0	02123456789	25	male
1	1	02123456788	34	female
3	0	07123456789	28	male

Tenant-a & c

sales_person_tenant_b

tenant_id	row	business_id
2	0	123456

Tenant_b

Figure 4-5: Extension Tables

The Universal Table in Figure 4-6 shows how the tenants' data are stored in the universal table. This table has a number of columns, including 'tenant_id', 'table_id', and 'col_1' until 'col_n'. The 'tenant_id' column is used to map rows with their tenants. The 'table_id' column is used to map rows to a particular table. The columns, including 'col_1' until 'col_n' are the universal columns that store any data the tenants wish to store to fulfil their business requirements.

universal

tenant_id	table_id	col_1	col_2	col_3	col_4	col_5	col_6	col_n
1	1	100	Joseph	Richard	02123456789	25	male	NULL
1	1	101	Sarah	Smith	02123456788	34	female	NULL
2	1	200	David	John	123456	NULL	NULL	NULL
3	1	150	Sam	Zen	07123456789	28	male	NULL

Figure 4-6: Universal Table

The Pivot Tables in Figure 4-7 show how the tenants' data with a specific data type is stored in a specific pivot table. In this example, we have two pivot tables, the first table is 'pivot_int' that stores INTGER data values, and the second table is 'pivot_str' that stores STRING data values. Each pivot table has standard columns, including 'tenant_id', 'table', 'col', and 'row'. In addition to a column that can vary in each pivot table according to the data type that is specified for that table. For

instance, the pivot table that stores STRING values will have a column that stores STRING values, and the column name could be called 'str'. The 'tenant_id' column is used to map each row in a pivot table with a tenant. The 'table' column is used to map a data type value to a particular table. The 'col' column is used to map a data type value to a particular column in a particular table. The 'row' column is used to map a data type value to a particular row in a particular table.

pivot_int

tenant_id	table	col	row	int
1	1	0	0	100
1	1	3	0	02123456789
1	1	4	0	25
1	1	0	1	101
1	1	3	1	02123456788
1	1	4	1	34
2	2	0	0	200
2	2	1	0	123456
3	3	0	0	150
3	3	3	0	07123456789
3	3	4	0	28

pivot_str

tenant_id	table	col	row	str
1	1	1	0	Joseph
1	1	2	0	Richard
1	1	5	0	male
1	1	1	1	Sarah
1	1	2	1	Smith
1	1	5	1	female
2	2	1	0	David
2	2	2	0	John
3	3	1	0	Sam
3	3	2	0	Zen
3	3	5	0	male

Figure 4-7: Pivot Tables

The Chunk Table in Figure 4-8 shows how a set of data columns with a mixture of data types is structured. The 'chunk_int_str' table has six columns, including 'tenant_id', 'table', 'chunk', 'row', 'int1', and 'str1'. The 'tenant_id' column is used to map each table row in a chunk table with a tenant. The 'table' column is used to map a table row to a particular table. The 'chunk' column is used to compound data for more than one logical column for a particular table. The 'row' column is used to map a data value to a particular row in a particular table. The 'int1' column is used to store all the INTEGER data values for different columns of different tables. The 'str1' column is used to store all the STRING data values for different columns of different tables.

chunk_int_str

tenant_id	table	chunk	row	int1	str1
1	1	0	0	100	Joseph
1	1	0	1	101	Sarah
1	1	1	0	02123456789	Richard
1	1	1	1	02123456788	Smith
1	1	2	0	25	male
1	1	2	1	34	female
2	2	0	0	200	David
2	2	1	0	123456	John
3	3	0	0	150	Sam
3	3	1	0	07123456789	Zen
3	3	2	0	28	male

Figure 4-8: Chunk Table

The Chunk Folding tables in Figure 4-9 show how the most commonly used tenants' columns are structured in the 'account_row' table, while the remaining columns are structured into Chunk Folding table called 'chunk_row'. The remaining columns that are used by tenants have extra business requirements, which are not applied in the common columns in the 'account_row' table. The 'tenant_id' column in both tables is used to map each table row with a tenant. The 'row' column in both tables is used to map a data value in a particular row of a particular table. The table 'account_row' consists of five columns, including 'tenant_id', 'row', 'sales_person_id', 'first_name', and 'last_name'. The last three columns in this table are the common columns that are shared by the three tenants (Tenant-A, Tenant-B, and Tenant-C). The 'chunk_row' table consists of six columns, including 'tenant_id', 'table', 'chunk', 'row', 'int1', and 'str1'. The 'table' column is used to map a row to a particular table. The 'chunk' column is used to compound data for more than one column for a particular table. The 'int1' column is used to store all the INTEGER data values for different columns of different tables. The 'str1' column is used to store all the STRING data values for different columns of different tables.

account_row

tenant_id	row	sales_person_id	first_name	last_name
1	0	100	Joseph	Richard
1	1	101	Sarah	Smith
2	0	200	David	John
3	0	150	Sam	Zen

Account row

chunk_row

tenant_id	table	chunk	row	int1	str1
1	1	0	0	25	02123456789
1	1	1	0	NULL	male
1	1	0	1	34	02123456788
1	1	1	1	NULL	Female
2	2	0	0	123456	NULL
3	3	0	0	28	07123456789
3	3	1	0	NULL	male

Chunk row

Figure 4-9: Chunk Folding

The XML Table in Figure 4-10 shows how this technique combines RDBMS and XML, by having fixed columns shared by all tenants, including 'tenant_id', 'sales_person_id', 'first_name', 'last_name'. The 'tenant_id' column is used to map each table row in the 'account_row' table with a tenant. The rest of the columns are sales person columns that are shared by all tenants. The fifth column is 'ext_xml', this column is used to store an XML structure includes the rest of the logical columns that tenants may need to fulfil their extra business needs. For instance, as shown in the first table row in the 'account_row' table, there are three values stored using XML structure in the 'ext_xml' column, including phone, age, and gender.

account_row

tenant_id	sales_person_id	first_name	last_name	ext_xml
1	100	Joseph	Richard	<ext> <phone>02123456789</phone> <age>25</age> <gender>male</gender> </ext>
1	101	Sarah	Smith	<ext> <phone>02123456788</phone> <age>34</age> <gender>female</gender> </ext>
2	200	David	John	<ext> <bus_id>123456</ bus_id > </ext>
3	150	Sam	Zen	<ext> <phone>07123456789</phone> <age>28</age> <gender>male</gender> </ext>

Figure 4-10: XML Table

Figure 4-11 is showing an example of the EET, which have three VETs that created using the ETs. These three VETs are the tenants' tables that presented in the Private Tables in Figure 4-4. In this example, the 'sales_person' table is a CTT shared by all the three tenants and has predefined columns that are commonly used by these tenants. The Tenant-A has a business requirement to have a sales person table that includes the columns that predefined in the 'sales_person' CTT, in addition to three extra columns, including 'phone', 'age', and 'gender'. This business requirement can be fulfilled by creating the 'sales_person_tenant_a' VET, and adding to this table these extra three columns. In addition to, adding the 'sales_person_id' column that is a virtual foreign key, which builds the virtual relationship between 'sales_person_tenant_a' VET and the 'sales_person' CTT. The Tenant-B has a business requirement to have a sales person table that includes the columns that are predefined in the 'sales_person' CTT, in addition to the 'business_id' column as an extra column to the CTT. This business requirement can be fulfilled for this tenant by creating the 'sales_person_tenant_b' VET, in addition, adding the 'sales_person_id' column that is a virtual foreign key, which builds the virtual relationship between 'sales_person_tenant_b' VET and the 'sales_person' CTT. The Tenant-C has a business requirement the same as the business requirement of Tenant-A. Therefore, the 'sales_person_tenant_c' VET of the Tennant-C has a similar structure and relationship of the 'sales_person_tenant_a' VET. The shared columns of the 'sales_person' CTT store the three tenants' data, while the rest of the tenants' data is stored in VETs by using the ETs, including 'db_table', 'table_column', 'table_row', 'table_relationship', table_index', and 'table_primary_key _column'. The details of this data are shown in Figure 4-12 – 4-18.

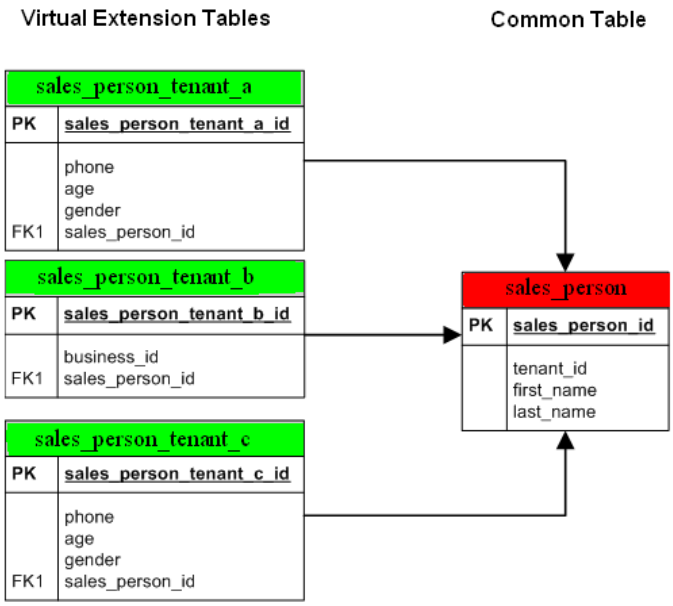


Figure 4-11: Virtual Extension Tables (VET)

sales_person_id	tenant_id	first_name	last_name
1	1	Joseph	Richard
2	1	Sarah	Smith
3	2	David	John
4	3	Sam	Zen

Figure 4-12: The data stored in the 'sales_person' CTT

db_table_id	tenant_id	db_table_name
4	1	sales_person_tenant_a
5	2	sales_person_tenant_b
6	3	sales_person_tenant_c

Figure 4-13: The data stored in the 'db_table' ET

table_column_id	tenant_id	db_table_id	table_column_name	default_value	data_type	is_indexed	is_null	is_relationship	is_primary_key_column	is_unique_column
28	1	1	sales_person_id		1	TRUE	FALSE	TRUE	FALSE	FALSE
1	1	1	age		1	FALSE	TRUE	FALSE	FALSE	FALSE
24	1	1	phone		1	FALSE	TRUE	FALSE	FALSE	FALSE
14	1	1	gender		1	FALSE	TRUE	FALSE	FALSE	FALSE
31	1	1	sales_person_tenant_a_id		1	TRUE	FALSE	FALSE	TRUE	TRUE
4	2	2	business_id		1	FALSE	TRUE	FALSE	FALSE	FALSE
29	2	2	sales_person_id		1	TRUE	FALSE	TRUE	FALSE	FALSE
32	2	2	sales_person_tenant_b_id		1	TRUE	FALSE	FALSE	TRUE	TRUE
33	3	3	sales_person_tenant_c_id		1	TRUE	FALSE	FALSE	TRUE	TRUE
2	3	3	age		1	FALSE	TRUE	FALSE	FALSE	FALSE
15	3	3	gender		1	FALSE	TRUE	FALSE	FALSE	FALSE
25	3	3	phone		1	FALSE	TRUE	FALSE	FALSE	FALSE
30	3	3	sales_person_id		1	TRUE	FALSE	TRUE	FALSE	FALSE

Figure 4-14: The data stored in the 'table_column' ET

table_row_id	serial_id	tenant_id	db_table_id	table_column_id	value
1	1	1	1	31	1
1	2	1	1	1	25
1	3	1	1	14	male
1	4	1	1	24	02123456789
1	5	1	1	28	1
2	1	1	1	31	2
2	2	1	1	1	34
2	3	1	1	14	female
2	4	1	1	24	02123456788
2	5	1	1	28	2
3	1	2	2	32	1
3	2	2	2	4	123456
3	3	2	2	29	3
4	1	3	3	33	1
4	2	3	3	25	07123456789
4	3	3	3	2	28
4	4	3	3	15	male
4	5	3	3	30	4

Figure 4-15: The data stored in the 'table_row' ET

table_relationship_id	tenant_id	db_table_id	table_column_id	table_type	target_table_id	target_column_id	shared_table_name	shared_column_name
1	1	4	28	1			sales_person	sales_person_id
2	2	5	29	1			sales_person	sales_person_id
3	3	6	30	1			sales_person	sales_person_id

Figure 4-16: The data stored in the 'table_relationship' ET

table_row_id	serial_id	tenant_id	db_table_id	table_column_id	value
1	1	1	4	31	1
1	2	1	4	31	2
1	1	2	5	32	3
1	1	3	6	33	4

Figure 4-17: The data stored in the 'table_index' ET

table_primary_key_column_id	tenant_id	db_table_id	table_column_id	is_auto_increment	is_composite_key
1	1	4	31	t	f
2	2	5	32	t	f
3	3	6	33	t	f

Figure 4-18: The data stored in the 'table_primary_key_column' ET

4.4 PERFORMANCE EVALUATIONS

Liao et al. (2012) have used in their work the Universal Table Schema Mapping (UTSM). The design of this schema is similar to the schema Salesforce is using (Liao et al. 2012), and originated from the Universal Relations (Maier & Ullman 1983). The data architecture of UTSM is shown in Figure 4-19. The 'Data' table is the universal table that stores all tenants' data, and it has fixed number of data columns. The number of columns of this table should be a large number to fit a different number of columns required by different tenants (e.g. Salesforce uses 500 columns for this table). These columns store data that maps to objects and fields created in the 'Objects' and 'Fields' tables. The data type of these columns is VARCHAR, which allows to store different data types (STRING, NUMBER, DATE, etc.). The 'Objects', 'Fields', and 'Relationships' tables are used to construct virtual tables and their virtual columns, and build relationships between these virtual tables. Whereas the 'Index' and 'Uniquefields' tables are used to optimize the query execution time of retrieving data from the 'Data' universal table (Liao et al. 2012; Weissman & Bobrowski 2009).

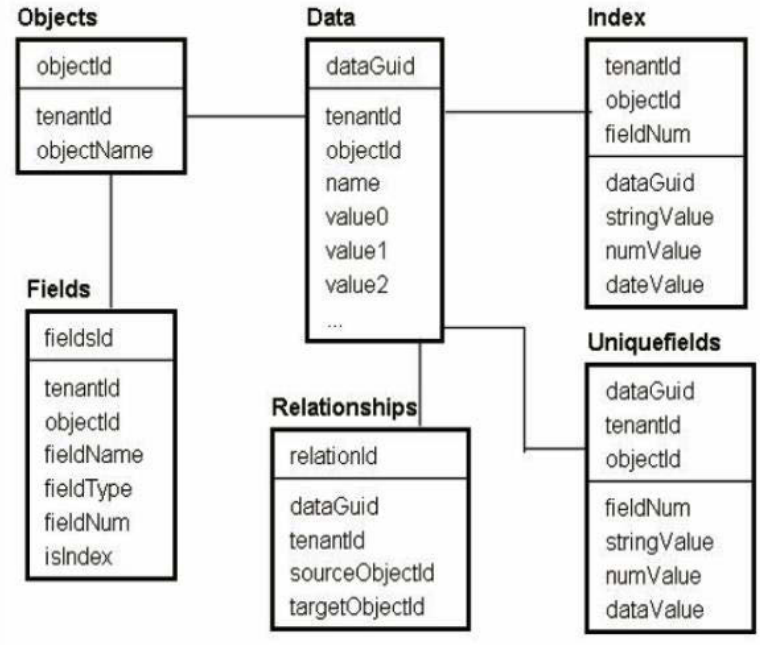


Figure 4-19: Universal Table Schema Mapping (Liao et al. 2012)

In this performance evaluation, the focus is on comparing the performance of accessing data from EET and UTSM directly from the database level, irrespective of the software solution which built on top of these two multi-tenant database schemas for two reasons: (1) The most significant challenge in multi-tenant applications is designing the multi-tenant database schema that improves multi-tenant query processing. This schema design influences the software design, which built on top of it and its performance. (2) Comparing the performance of two multi-tenant software solutions under the same conditions, and on the same hardware resources is hard to be achieved, especially when the other software is not available to be installed on the same application server.

4.4.1 Experimental Data Set and Setup

Typically, multi-tenant databases store massive data volumes across multiple servers to optimize the performance of data retrieval. However, before considering scale-up or scale-out of multi-tenant databases to optimize its performance, we

believe that we should perform a comparison between EET and UTSM using a single server instance. In order to test the effectiveness of accessing data from these two multi-tenant database architecture designs without affecting their performance by using any scalability. In our experiments, we focus on benchmarking the performance of the main tables of both data architectures where most of the tenants' data is stored, and we discard the lookup queries. For example, in EET, we discard the queries which check whether a virtual column is indexed or not from the 'table_column' ET. On the other hand, we discard the queries which check whether a column is indexed or not from the 'fields' table of UTSM. In this case, our focus in EET is on 'table_row', and 'table_index' ETs, and in UTSM is on 'Data', 'Index', and 'Uniquefields' tables. Furthermore, in order to run comparative experiments, exactly the same data was populated in the 'table_row', and 'table_index' ETs of EET in a separate database, and the 'Data', 'Index', and 'Uniquefields' tables of UTSM in another database. No indexes were used other than the default indexes of each schema, which are the primary keys and the foreign keys indexes that are automatically generated in the RDBMS once the primary key and foreign key constraints are specified. The number of virtual rows that were already populated in 'table_row' ET is 200,000 rows and the same number of rows in the 'Data' universal table. These rows belong to the 'product' virtual table, and the structure of this table in EET and UTSM is shown in Figure 4-20. There was no data populated in these two databases other than the populated 200,000 rows.

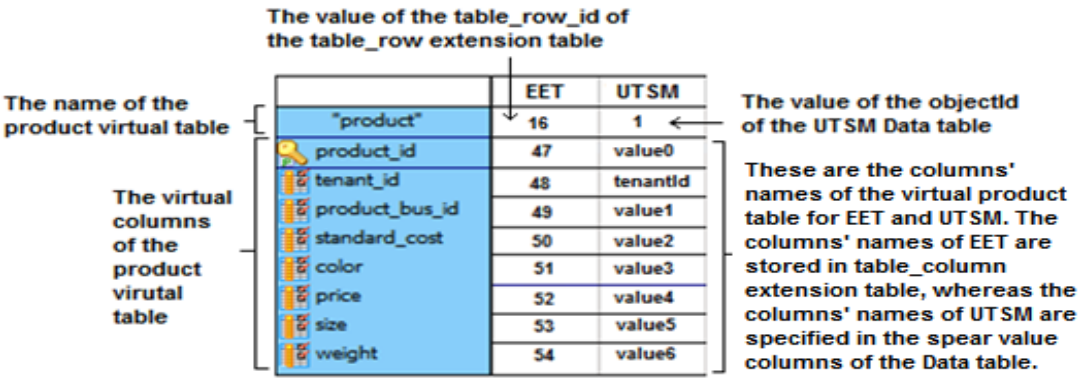


Figure 4-20: The virtual 'product' table structure.

In the multi-tenant database, each tenant's data is isolated in a table partition. Therefore, the experiments are performed for one tenant to evaluate the effectiveness of retrieving data for each single tenant from the multi-tenant database. These experiments are divided into four types that are sharing the details of this data set. Each query of these experiments is performed ten times, and the average execution time of these queries is shown in Figure 4-21 – 4-28 and Table 4-1 – 4-7. The queries that are related to EET and UTSM are shown in Table 4-8. The input and the output of EET and UTSM queries are the same. However, the structures of these queries are different because the data architectures of the two schemas are different. The four experiments details are listed below:

1) **Retrieving Rows Experiment (Exp.4-1):** The aim of this experiment is to benchmark the query execution time of retrieving rows from EET and UTSM. This experiment is divided into four experiments including:

   Retrieving Rows without Using Query Columns Filters Experiment (Exp.4-1.1): In this experiment, Query 4-1 (Q4-1) and Query 4-2 (Q4-2) are executed. The Q4-1 retrieves rows from the 'table_row' ET of EET without specifying any query filters other than the tenant ID, and the 'project' table ID. Whereas the Q4-2 retrieves rows from the 'Data' universal table without specifying any query filters other than the tenant ID and the 'project' object ID. In this study, eight tests using these two queries are performed to retrieve 1, 10, 50, 100, 500, 1000, 1500, and 2000 rows.

   Retrieving Rows Using Columns Query Filters Experiment (Exp.4-1.2): In this experiment, Query 4-3 (Q4-3) is executed on the 'table_row' ET of EET and Query 4-4 (Q4-4) is executed on the 'Data' universal table. Both queries are filtered by specifying particular numbers of product IDs stored in the 'product' virtual table. In this study, three tests using these two queries are performed to retrieve rows by specifying 1 product ID for the first test, 10 product IDs for the

second test, and 50 product IDs for the third test. The structure of Q4-4 has presented in (Liao et al. 2013) but with different value settings.

Retrieving Rows Using Primary Key Indexes Experiment (Exp.4-1.3): In this experiment, Query 4-5 (Q4-5) is executed on the 'table_row' and 'table_index' ETs of EET and Query 4-6 (Q4-6) is executed on the 'Data' and 'Uniquefields' tables of UTSM. In this experiment, a primary key index is used to retrieve rows from the 'product' virtual table from the 'table_row' ET and from the 'Data' table. In this study, three tests using these two queries are performed to retrieve 1, 10, and 50 rows. The structure of Q4-6 has presented in (Liao et al. 2012), but with different value settings.

Retrieving Rows Using Custom Index Experiment (Exp.4-1.4): In this experiment, Query 4-7 (Q4-7) is executed on the 'table_row' and 'table_index' ETs of EET and Query 4-8 (Q4-8) is executed on the 'Data' and 'Index' tables of UTSM. In this experiment, a custom index is used, which is a selective filter in the tenant's query. This index should be other than the primary key and foreign key indexes. This custom index retrieves rows from the 'product' virtual table for both 'table_row' and 'Data' tables. The 'standard_cost' virtual column is chosen to filter the queries by looking up for all the products, which have a standard cost greater or equal '$ 9000' from the 'product' virtual table. In this study, four tests using these two queries are performed to retrieve 1, 10, 50, and 100 rows.

2) **Inserting Rows Experiment (Exp.4-2):** The aim of this experiment is to benchmark the query execution time of inserting rows into EET and UTSM. Query 4-9 (Q4-9) is executed on the 'table_row' and 'table_index' ETs of EET and Query 4-10 (Q4-10) is executed on the 'Data', 'Index', and 'Uniquefields' tables of UTSM. In this study, four tests using these two queries are performed to insert 1, 10, 50, and 100 rows.
3) **Updating Rows Experiment (Exp.4-3)**: The aim of this experiment is to benchmark the query execution time of updating rows into EET and UTSM. Query 4-11 (Q4-11) is executed on the 'table_row' and 'table_index' ETs of EET

and Query 4-12 (Q4-12) is executed on the 'Data', and 'Index' tables of UTSM. In this study, four tests using these two queries are performed to update 1, 10, 50, and 100 rows.

4) **Deleting Rows Experiment (Exp.4-4):** The aim of this experiment is to benchmark the query execution time of deleting rows from EET and UTSM. Query 4-13 (Q4-13) is executed on the 'table_row' and 'table_index' ETs of EET, and Query 4-14 (Q4-14) is executed on the 'Data', 'Index', and 'Uniquefields' tables of UTSM. In this study, four tests using these two queries are performed to delete 1, 10, 50, and 100 rows.

The experiments were performed on PostgreSQL 8.4 database, using the default configuration setup. This database installed on a PC with 64-bit Windows 7 Home Premium operating system, Intel Core i5 2.40GHz CPU, 8 GB RAM memory, and 500 GB hard disk storage.

4.4.2 EXPERIMENTAL RESULT

This section gives four experimental results as follows:

1) **Retrieving Rows Experimental Results:** This experimental result was divided into four results as follows. The experimental study of Exp.4-1.1 shows that the execution time of Q4-1 that perform on the 'table_row' ET of EET is approximately 76% faster on average than the execution time of Q4-2 that perform on the 'Data' universal table when 1, 10, 50, 100, 500, 1000, 1500, and 2000 rows were retrieved. The details results of this experiment are shown in Figure 4-21 – 4-22 and Table 4-1. The experimental study of Exp.4-1.2 shows that the execution time of Q4-3 that perform on the 'table_row' ET of EET is approximately 94% faster on average than the execution time of Q4-4 that perform on the 'Data' universal table when 1, 10, and 50 rows were retrieved. The details results of this experiment are shown in Figure 4-23 and Table 4-2. The experimental study of Exp.4-1.3 shows that the execution time of Q4-5 that

perform on the 'table_row' and 'table_index' ETs of EET is approximately 88% faster on average than the execution time of Q4-6 that perform on the 'Data' and 'Uniquefields' tables of UTSM when 1, 10, and 50 rows were retrieved. The details results of this experiment are shown in Figure 4-24 and Table 4-3. The experimental study of Exp.4-1.4 shows that the execution time of Q4-7 that perform on the 'table_row' and 'table_index' ETs of EET is approximately 60% faster on average than the execution time of Q4-8 that perform on the 'Data' and 'Index' tables of UTSM when 1, 10, 50, and 100 rows were retrieved. The details results of this experiment are shown in Figure 4-25 and Table 4-4.

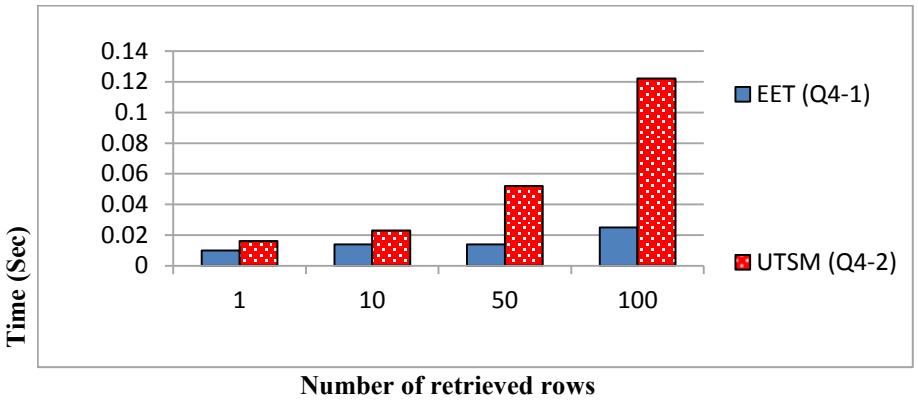


Figure 4-21: Retrieving small numbers of rows (Exp. 4-1.1)

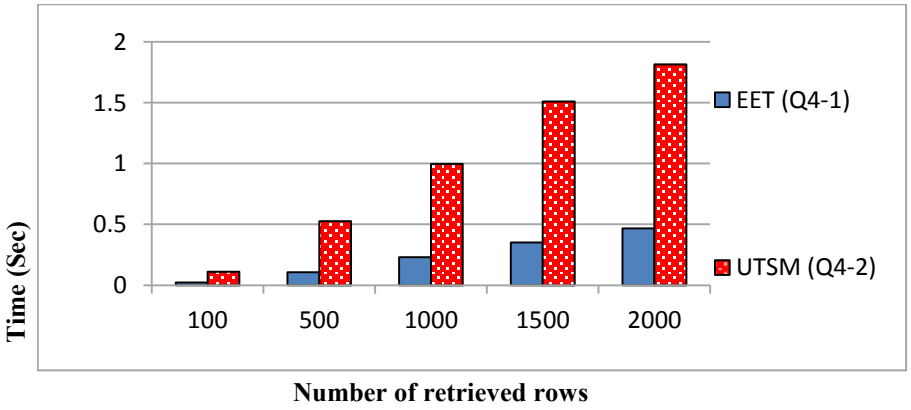


Figure 4-22: Retrieving large numbers of rows (Exp. 4-1.1)

Table 4-1: The query execution times of retrieving rows without using query columns filters experiment (Exp. 4-1.1)

Number of retrieved rows	EET (Q4-1) Time in seconds	UTSM (Q4-2) Time in seconds
1	0.010	0.016
10	0.014	0.023
50	0.014	0.052
100	0.025	0.122
500	0.107	0.527
1000	0.230	0.998
1500	0.352	1.510
2000	0.468	1.814

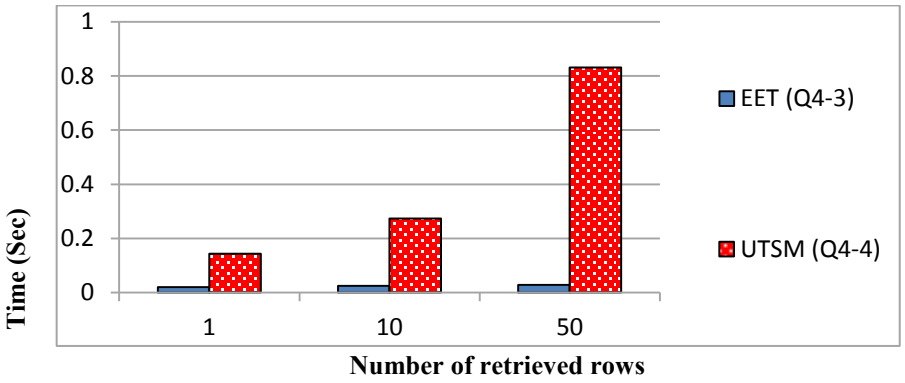


Figure 4-23: Retrieving rows using columns query filters (Exp.4-1.2)

Table 4-2: The query execution times of retrieving rows using columns query filters experiment (Exp. 4-1.2)

Number of retrieved rows	EET (Q4-3) Time in seconds	UTSM (Q4-4) Time in seconds
1	0.020	0.143
10	0.025	0.273
50	0.029	0.831

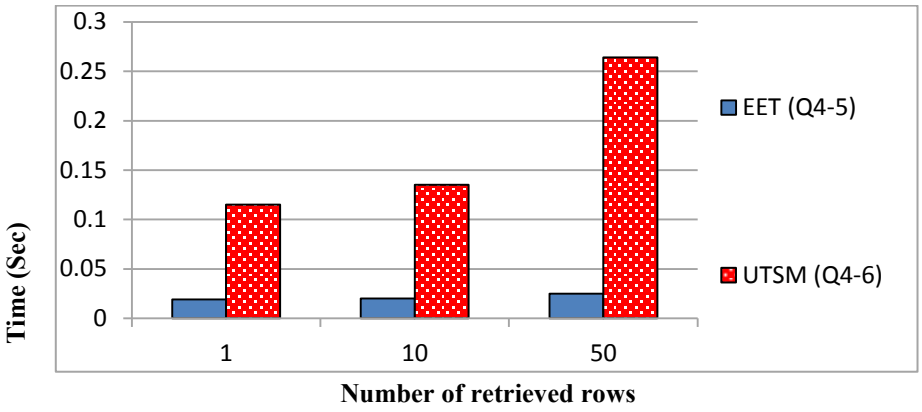


Figure 4-24: Retrieving rows using PK indexes (Exp. 4-1.3)

Table 4-3: The query execution times of retrieving rows using primary key indexes experiment (Exp. 4-1.3)

Number of retrieved rows	EET (Q4-5) Time in seconds	UTSM (Q4-6) Time in seconds
1	0.019	0.115
10	0.020	0.135
50	0.025	0.264

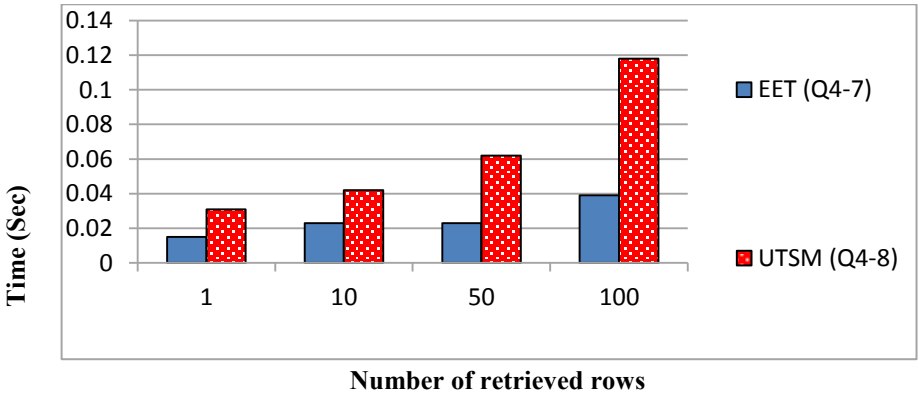


Figure 4-25: Retrieving rows using a custom index (Exp. 4-1.4)

Table 4-4: The query execution times of retrieving rows using custom index experiment (Exp. 4-1.4)

Number of retrieved rows	EET (Q4-7) Time in seconds	UTSM (Q4-8) Time in seconds
1	0.015	0.031
10	0.023	0.042
50	0.023	0.062
100	0.039	0.118

2) **Inserting Rows Experimental Results:** The experimental study of Exp.4-2 shows that the execution time of Q4-9 that perform on the 'table_row' and 'table_index' ETs of EET is approximately 19% slower on average than the execution time of Q4-10 that perform on the 'Data', 'Index', and 'Uniquefields' tables of UTSM when 1, 10, 50, and 100 rows were inserted. The details results of this experiment are shown in Figure 4-26 and Table 4-5.

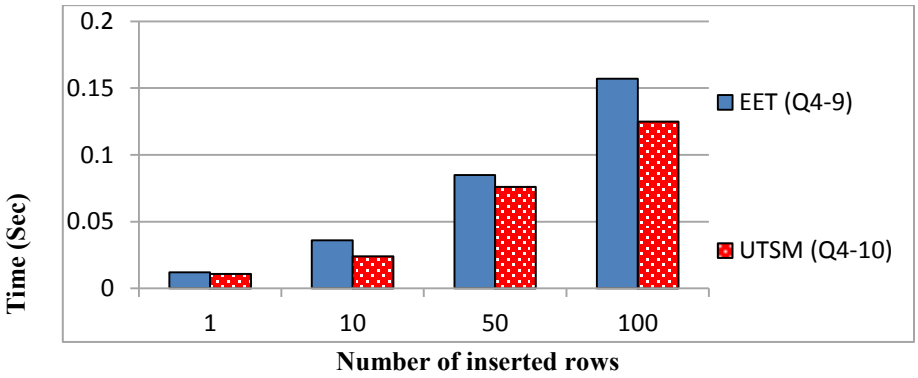


Figure 4-26: Inserting rows (Exp.4-2)

Table 4-5: The query execution times of inserting rows experiment (Exp. 4-2)

Number of inserted rows	EET (Q4-9) Time in seconds	UTSM (Q4-10) Time in seconds
1	0.012	0.011
10	0.036	0.024
50	0.085	0.076
100	0.157	0.125

3) **Updating Rows Experimental Results:** The experimental study of Exp.4-3 shows that the execution time of Q4-11 that perform on the 'table_row' and 'table_index' ETs of EET is approximately 51% faster on average than the execution time of Q4-12 that perform on the 'Data', and 'Index' tables of UTSM when 1, 10, 50, and 100 rows were updated. The details results of this experiment are shown in Figure 4-27 and Table 4-6.

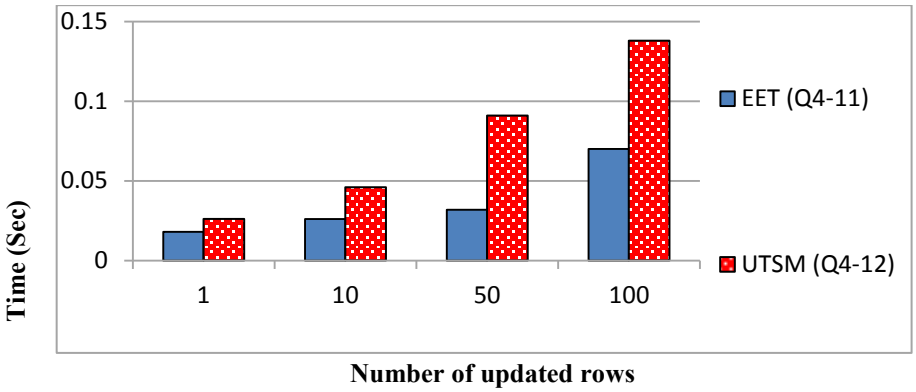


Figure 4-27: Updating rows (Exp. 4-3)

Table 4-6: The query execution times of updating rows experiment (Exp. 4-3)

Number of updated rows	EET (Q4-11) Time in seconds	UTSM (Q4-12) Time in seconds
1	0.018	0.026
10	0.026	0.046
50	0.032	0.091
100	0.070	0.138

4) **Deleting Rows Experimental Results**: The experimental study of Exp.4-4 shows that the execution time of Q4-13 that perform on the 'table_row' and 'table_index' ETs of EET is approximately 32% faster on average than the execution time of Q4-14 that perform on the 'Data', 'Index', and 'Uniquefields' tables of UTSM when 1,

10, 50, and 100 rows were deleted. The details results of this experiment are shown in Figure 4-28 and Table 4-7.

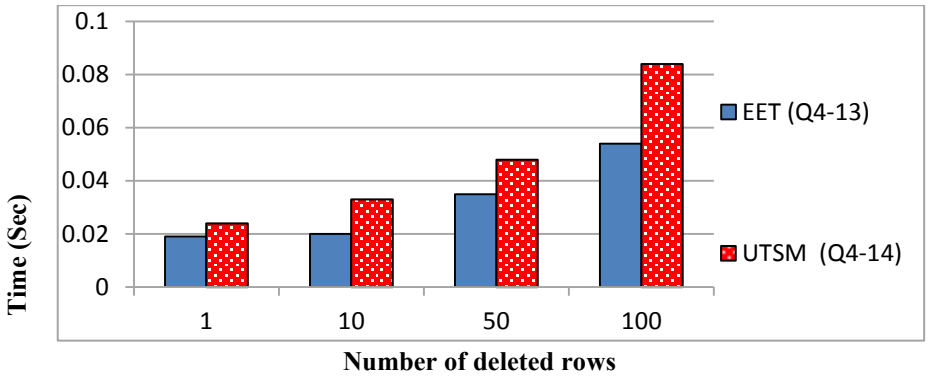


Figure 4-28: Deleting rows (Exp.4-4)

Table 4-7: The query execution times of deleting rows experiment (Exp. 4-4)

Number of deleted rows	EET (Q4-13) Time in seconds	UTSM (Q4-14) Time in seconds
1	0.019	0.024
10	0.020	0.033
50	0.035	0.048
100	0.054	0.084

Table 4-8: The experiments queries

Query No.	Query Details
Q4-1	SELECT * FROM table_row tr WHERE tr.table_row_id in (SELECT distinct(tr2.table_row_id) FROM table_row tr2 where tr2.db_table_id = 16 and tr2.tenant_id = 1000 **LIMIT 1**) ;
Q4-2	SELECT * FROM data WHERE tenantid = 1000 and objectId = 1 **LIMIT 1**;
Q4-3	SELECT * FROM table_row tr WHERE tr.tenant_id =1000 and tr.db_table_id = 16 and tr.table_column_id IN (50,52,54) and tr.table_row_id IN (SELECT table_row_id FROM table_row tr2 WHERE tr2.tenant_id =1000 and tr2.db_table_id = 16 and **(tr2.table_column_id =47 and tr2.value = '163336')**);
Q4-4 **(Liao et al. 2013)**	SELECT price, cost, weight FROM (SELECT value0 AS id, value4 AS price , value2 AS cost, value6 AS weight FROM data WHERE objectid = 1 and tenantid = 1000) AS product WHERE **id = '163336'**;
Q4-5	SELECT * FROM table_row tr WHERE tr.tenant_id =1000 and tr.db_table_id = 16 and tr.table_row_id IN (SELECT ti.table_row_id FROM table_index ti WHERE ti.tenant_id =1000 and ti.db_table_id = 16 and **ti.table_column_id =47 and ti.value = '163337'**);
Q4-6	SELECT * FROM data WHERE objectid =1 and tenantId = 1000 and dataguid in

(Liao et al. 2012)	(SELECT dataguid FROM uniquefields WHERE objectid = 1 and tenantId = 1000 and **numvalue IN (163337)**);
Q4-7	SELECT * FROM table_row tr WHERE tr.tenant_id =1000 and tr.db_table_id = 16 and tr.table_row_id IN (SELECT ti.table_row_id FROM table_index ti WHERE ti.tenant_id = 1000 and ti.db_table_id = 16 and ti.table_column_id = 50 and **(cast (ti.value as numeric)** >= **'9000') LIMIT 1**);
Q4-8	SELECT * FROM data WHERE objectid =1 and tenantId = 1000 and dataguid in (SELECT dataguid FROM index WHERE objectid = 1 and tenantId = 1000 and fieldNum =3 and **numvalue > = 9000 LIMIT 1**);
Q4-9	INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000061,1,1000, '50000000',16,47); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000061,2,1000, '1000',16,48); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000061,3,1000, '50000',16,49); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000061,4,1000, '222.50',16,50); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000061,5,1000, 'Red',16,51); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000061,6,1000, '242.50',16,52); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000061,7,1000, '40',16,53); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000061,8,1000, '300',16,54); INSERT into table_index (tenant_id, value, table_row_id, serial_id, db_table_id, table_column_id) values (1000, '50000000',50000061,1,16,47); INSERT into table_index (tenant_id, value, table_row_id, serial_id, db_table_id, table_column_id) values (1000, '222.50',50000061,4,16,50);
Q4-10	INSERT into data (dataguid, tenantid, objectid ,name, value0, value1, value2, value3,value4, value5 ,value6) values(50000061,1000,1,'name', '50000000', '50000', '222.50','Red', '242.50', '40', '300'); INSERT into uniquefields values (50000061, 1000, 1, 1,'',50000000,'2013-12-12'); INSERT into index values (50000061, 1000, 1, 3,'', '222.50','2013-12-12');
Q4-11	UPDATE table_row set value = '230.50' WHERE tenant_id = 1000 and db_table_id = 16 and table_column_id = 52 and table_row_id =50000061; UPDATE table_index set value = '230.50' WHERE tenant_id = 1000 and db_table_id = 16 and table_column_id = 52 and table_row_id =50000061;
Q4-12	UPDATE data set value2 = '230.50' WHERE tenantid = 1000 and objectid = 1 and dataguid =50000061; UPDATE index set numvalue = 230.50 WHERE tenantid = 1000 and objectid = 1 and fieldnum =3 and dataguid =50000061;
Q4-13	DELETE from table_index WHERE tenant_id = 1000 and db_table_id = 16 and table_row_id =50000061; DELETE from table_row WHERE tenant_id = 1000 and db_table_id = 16 and table_row_id = 50000061;
Q4-14	DELETE from index WHERE tenantid = 1000 and objectid = 1 and fieldnum =3 and dataguid =50000061; DELETE from uniquefields WHERE tenantid = 1000 and objectid = 1 and fieldnum =1 and dataguid =50000061; DELETE from data WHERE tenantid = 1000 and objectid = 1 and dataguid =50000061;

4.5 SUMMARY

In this chapter, a novel configurable multi-tenant database schema design called EET that consists of CTT, ET, and VET is proposed. EET allows tenants to create their own virtual database schema, including the required number of tables and columns, rows, virtual database relationships with any of CTTs or VETs, and to assign suitable data types and constraints for columns during multi-tenant application run-time execution. EET is a single multi-tenant database schema that has a flexible way of creating database schemas for multiple tenants, by extending a business domain database that is based on RDBMS, or by creating a tenant's business domain database from the scratch. It improves the multi-tenant database performance by avoiding NULL values, assigning primary keys to unique columns, providing indexes to table columns, and storing BLOB and CLOB data types in separate designated tables. In addition, it allows to store different data types of Big Data including structured, semi-structured, and unstructured data. However, in this chapter and this thesis the empirical tests are conducted on the structured data, but empirical tests of semi-structured, and unstructured data are out of the scope of this thesis. For two reasons, first, as reviewed in chapter 2, storing and retrieving data in XML files (semi-structured data) has the highest response time between the reviewed seven multi-tenant database schema designs (Aulbach et al. 2009; Heng et al. 2012). Thus, the semi-structured data can be stored in EET, but it is not recommended to be used as storage for multiple tenants. Second, there are many techniques for storing and retrieving different data types of Big Data. Accordingly, to compare all of these techniques with EET is hard, complex, and time consuming task, which it is hard to be achieved during the time frame and size of a PhD thesis. Thus, comparing EET with other data types and other techniques is one of the future research directions of this study.

Moreover, the EET allows to create virtual relationships between the tenants' shared physical tables (CTT) and the tenants' virtual tables (VET), and allows tenants to choose from three database models: Multi-tenant Relational Database, Integrated Multi-tenant Relational Database and Virtual Relational Database, and Virtual Relational Database. Nevertheless, this capability not applied in UTSM or any other multi-tenant database schema design yet. Furthermore, this chapter compared and evaluated the performance of EET and UTSM. The EET avoids storing rows with NULL values. In contrast, the Universal Table of UTSM can be large with overheads cause a large number of NULL values. The experimental study that conducted shows an improvement gained when retrieving, updating, and deleting data from EET over the UTSM. Especially when retrieving data from EET, it is much faster than UTSM. However, the execution time of inserting rows in EET is slightly slower than UTSM. Overall, this experimental study makes the EET schema a good candidate for implementing multi-tenant databases and multi-tenant SaaS and Big Data applications.

CHAPTER 5

MULTI-TENANT SCHEMA HANDLER METHOD

Multi-tenant database is a new database solution that is significant for SaaS and Big Data applications in the context of cloud computing paradigm. This multi-tenant database has significant design challenges to develop a solution that insures a high level of data quality, configurability, accessibility, and manageability. Configuration is one of the significant characteristics of multi-tenant applications that allows SaaS vendors to run a single instance application that can be configured by multiple tenants. 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 behaviour during the application's runtime execution without redeploying the application (Bezemer & Zaidman 2010). Multi-tenant data has two types, shared data and tenant's isolated data. Integrating these two types of data together, gives a complete data view for the tenants to fit their business requirements (Domingo 2010; Liu 2010). Developing such a data management solution is a complex problem to solve that involves huge efforts and longer development lifecycle. This chapter proposes a multi-tenant data management service called Elastic Extension Tables Schema Handler Service (EETSHS), which is based on the EET multi-tenant database schema. This data management service creates, manages, organises, and administrates multiple tenants' data in a single database schema. Moreover, it integrates traditional

relational data with virtual relational data in a single database schema, and allows tenants to manage this data by calling functions from this service.

The remainder of this chapter is structured as follows. Section 5.1 proposes the EET schema handler service. Section 5.2 presents algorithms for the frequently used functions of EETSHS. Section 5.3 presents a set of experiments to compare the performance of managing CTTs (traditional physical RDBMS tables) and VETs. These experiments verify that EET schema is a good candidate for the management of multi-tenant data for SaaS and Big Data applications. Section 5.4 concludes this chapter.

5.1 Elastic Extension Tables Schema Handler Service

There are several commercial cloud data management systems (e.g. BigTable[7], SimpleDB[8], HyperTable[9], CouchDB[10]) that allow end users to manage their data storage using APIs (Sakr et al. 2011). A similar approach is applied in designing the EETSHS. This service provides functions that allow tenants to manage their data without having to write SQL queries and backend data management code, by calling data management functions from EET data management APIs. Figure 5-1 shows the overview architecture of the EETSHS, and shows how this service is interacting with the rest of the EET framework artefacts. The details of these interactions are stated in Chapter 3, while this section will present the main functions of EETSHS in the following subsections, and these functions are also shown in the same figure.

7 http://en.wikipedia.org/wiki/BigTable; Accessed July, 2014
8 https://aws.amazon.com/simpledb; Accessed July, 2014
9 http://hypertable.org/; Accessed July, 2014
10 http://couchdb.apache.org; Accessed July, 2014

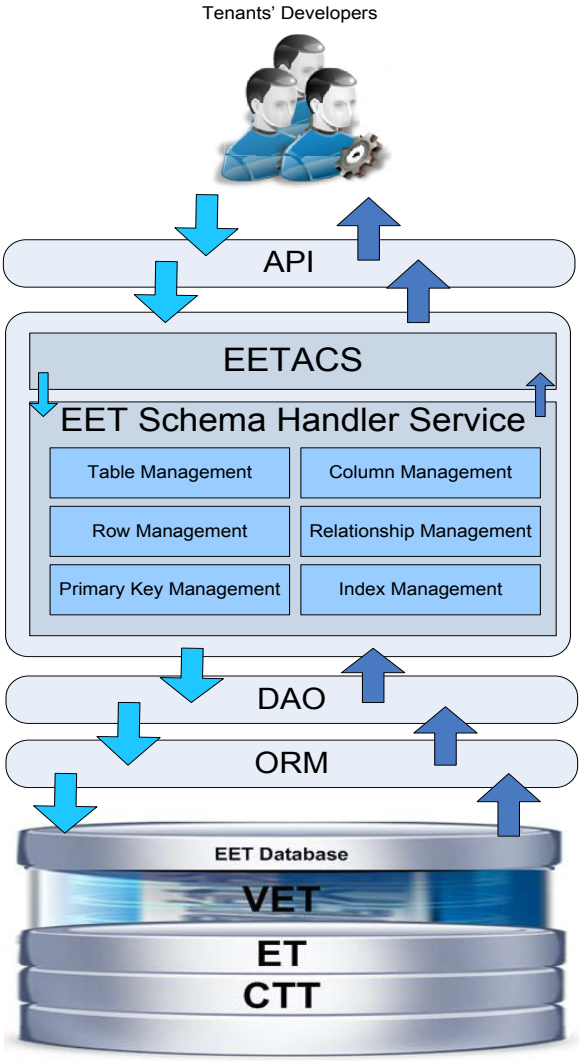


Figure 5-1: EET Schema Handler Service overview architecture

5.1.1 TABLE MANAGEMENT

The EETSHS has three data management functions to manage VETs, whereas CTTs are managed by the RDBMS.

- **Create Virtual Tables Function**: This function creates a VET's name for a tenant, and this name is a unique name only for this tenant. For example, tenant-A can create a VET name called 'product', but cannot create the same VET name

again. However, tenant-B can create the 'product' name even if tenant-A already created this VET's name. This function avoids the redundancy of individual tenant tables, because the 'db_table_name' column of the 'db_table' ET has UNIQUE constraint.

- **Update Virtual Tables Function**: After creating the table name of the tenant's VET, this name can be updated by calling this function. The updated VET name remains unique for every individual tenant, because of the UNIQUE constraint of the 'db_table_name' column.
- **Delete Virtual Tables Function**: After creating the tenant's VET, the tenant can delete this table by calling this function. Deleting a VET means that the table name and its virtual columns that are stored in the 'table_column' ET and related to this VET have to be deleted. In addition to the rows, the indexes, and the constraints that are linked to these columns and stored in other ETs have to be deleted. However, the only case the tenant cannot delete a VET, is when it has a master-detail relationship with other VET, and the primary key of the master VET that need to be deleted is a foreign key in other details VET.

5.1.2 COLUMN MANAGEMENT

The EETSHS has three data management functions to manage VETs' columns, whereas CTTs' columns are managed by the RDBMS.

- **Create Virtual Columns Function:** This function creates a virtual column for a VET and specifies its properties by storing the necessary column properties values in the columns of the 'table_column' ET. The column properties include (1) the default value of a column that need to be inserted when no data specified for it while creating or updating a virtual row, (2) the data type of the column, (3) index column flag, (4) null column flag, (5) foreign key column flag, (6) primary key column flag, and (7) unique key column flag.
- **Update Virtual Columns Function:** After creating virtual columns for a tenant's VET, the tenant can update any column properties by calling this function.

- **Delete Virtual Columns Function:** After creating virtual columns for a tenant's VET, the tenant can delete any column even if it is a primary key, as long as this column is not a primary key that has foreign keys in any other table pointing to it. This function deletes a column from a VET, and simultaneously deletes the entire rows associate to this column that may be stored in the other ETs that store rows, relationships, indexes, and primary keys.

5.1.3 ROW MANAGEMENT

The EETSHS has three data management functions to manage the rows of CTTs and VETs.

- **Create Physical and Virtual Rows Function**: This function creates a tenant table row for a CTT or a VET. The physical rows of CTTs are created in the physical tables of the RDBMS, whereas the virtual rows of VETs are created in the 'table_row', 'table_row_blob', 'table_row_clob', and 'table_index' ETs. Algorithm 5-1 and Section 5.3 present the details of this function.
- **Update Physical and Virtual Rows Function**: After creating a tenant's table row in a CTT or a VET, the tenant may update this row by calling this function. Algorithm 5-2 and Section 5.3 present the details of this function.
- **Delete Physical and Virtual Rows Function**: After creating a tenant's table row in a CTT or a VET, the tenant may delete this row by calling this function. Algorithm 5-3 and Section 5.3 present the details of this function.

5.1.4 RELATIONSHIP MANAGEMENT

The EETSHS has two data management functions to manage virtual relationships between CTTs and VETs.

- **Create Virtual Relationships Function**: This function creates a virtual relationship between CTT and VET, or two VETs. The virtual relationships that are created using this function allows a tenant to choose from any of the three

EET database models that stated in Chapter 4. This function stores for a tenant, a master-detail relationship between two tables into the 'table_relationship' ET. Simultaneously, it creates in the details VET foreign key columns that refer to the primary key columns of the master CTT or VET.

- **Delete Virtual Relationships Function**: After the tenant creates a virtual master-details relationship between two tables, he can delete this relationship by calling this function. This function deletes the relationship from the 'table_relationship' ET, deletes from the details VET all the foreign key columns that refer to the primary key columns of the master CTT or VET, and deletes any VET's rows stored in the 'table_row' and 'table_index' ETs.

In traditional RDBMS, the database administrator cannot update a relationship between two physicals tables. The same case applies for EETSHS; it does not have a function to update virtual relationships. Nevertheless, the tenant can update a relationship by deleting an existing relationship and then creating a new relationship by calling the two functions described in this section.

5.1.5 PRIMARY KEY MANAGEMENT

The EETSHS has two data management functions to manage virtual primary keys of VETs, whereas the primary keys of CTTs are managed by the RDBMS.

- **Create Virtual Primary Keys Function**: This function creates a virtual PRIMARY KEY constraint for a VET column by changing the value of the 'is_primary_key_column' column in the 'table_column' ET to 'true', and storing the detail settings of the primary key column in the 'table_primary_key_column' ET. If the column has already data stored in the 'table_row' ET, then this function copies all of the column's data of the primary key to the 'table_index' ET. As long as the column is a primary key column, then it should be indexed. Otherwise, if the column does not have data, then no any data need to be copied to the 'table_index' ET. This function allows to create single and composite primary

keys. In the case when a tenant wants to create a new primary key and the VET has at least one primary key, then this function will store the value 'true' into the 'is_composite_key' column of the 'table_primary_key_column' ET for the new primary key and the already existing primary keys. Moreover, this function specifies if a primary key is auto incremented, which means that a unique number is generated when a new row is inserted into a VET. However, this function avoids adding a PRIMARY KEY constraint to any column has redundant data.

- **Update Virtual Primary Keys Function**: This function is used for two cases, if a column already has a primary key constraint, or if it has not. In the first case, this function deletes the primary key constraint by changing the value of 'is_primary_key_column' in the 'table_column' ET to 'false', and deletes the details of the primary key column from the 'table_primary_key_column' ET. If the column has data stored into the 'table_row' ET, then this function deletes all of the column's data of the primary key from the 'table_index' ET as long as the column is not anymore primary key column then it should not be indexed. Otherwise, if the column does not have data, then no any data need to be deleted from the 'table_index' ET. Nevertheless, when a tenant deletes a primary key constraint of a column that is part of a composite primary key, then this function changes the value of the 'is_composite_key' to 'false' for the primary key that will not be deleted from the 'table_primary_key_column' ET. In the second case, when the column is not already a primary key, then this function calls the Create Virtual Primary Key Function. Moreover, this function can update the auto increment property of the primary key by either activating or deactivating it in the 'table_primary_key_column' ET.

5.1.6 INDEX MANAGEMENT

The EETSHS has no specific functions to manage VETs' indexes, including primary key, foreign key and custom indexes. However, these indexes are managed in the other functions that discussed in this section. The details are listed below:

- **Create Virtual Indexes:** There are four cases to create rows in the 'table_index' ET, including (1) when a tenant creates a virtual master-detail relationship between CTT and VET, or two VETs. In this case, if the master table has table rows, then the primary key of these rows get inserted into the 'table_index' ET as foreign keys for the details table. This situation occurs in the Create Virtual Relationships Function. (2) When a tenant adds a PRIMARY KEY constraint to a column that already exist in a VET and this column has data, then this data is inserted into 'table_index' ET. This situation occurs in the Create Virtual Primary Keys Function. (3) When a tenant makes a column a custom index column that is a selective filter in the tenant's query and has data, then this data is inserted into the 'table_index' ET. This case occurs in the Update Virtual Columns Function. (4) Once a VET row inserted in the 'table_row' ET and this VET has indexed columns, including primary key, foreign key and custom indexes. Then the values of these indexed columns are inserted into the 'table_index' ET. This situation occurs in the Create Physical and Virtual Rows Function.
- **Update Virtual Indexes:** There is only one case to update virtual indexes in the 'table_index' ET when the value of a virtual custom index column of a virtual row is updated in the 'table_row' ET. Then the same value is updated in the 'table_index' ET. This situation occurs in the Update Physical and Virtual Rows Function.
- **Delete Virtual Indexes:** There are three cases to delete rows from the 'table_index' ET, including (1) when a tenant deletes a virtual relationship between CTT and VET, or two VETs. In this case, if the master and the details tables have data, then the data value in the 'table_index' ET that corresponds to the foreign key of the details VET get deleted. This situation occurs in the Delete Virtual Relationships Function. (2) When a tenant updates a custom index column and makes it not indexed column, and this column has data, then this data is deleted from the 'table_index' ET. This case occurs in the Update Virtual Columns Function. (3) Once a VET row is deleted from the 'table_row' ET and

this row has indexed columns, including any of the primary keys, foreign keys, or custom indexes. Then the corresponding index values that related to the deleted row and stored into the 'table_index' ET get deleted. This situation occurs in the Delete Physical and Virtual Rows Function.

5.2 SAMPLE ALGORITHMS OF ELASTIC EXTENSION TABLES SCHEMA HANDLER SERVICE

In this section, we present three EET data management sample algorithms that are used to allow tenants to insert, update, and delete rows within CTTs and VETs.

5.2.1 CREATING PHYSICAL AND VIRTUAL ROWS ALGORITHM

This data management algorithm inserts rows in CTTs and VETs by passing six parameters to it, including the tenant ID, table name, table type (CTT or VET), table row matrix, table BLOB matrix, and table CLOB matrix. More details of this algorithm are presented in Definition 5-1 and Algorithm 5-1.

Definition 5-1 (Creating Physical and Virtual Rows): T denotes a tenant ID. B denotes a table name. $CPVR_{type}$ denotes the table type whether it is a CTT or a VET. $CPVR_{row}$ denotes a row matrix with 2 rows and n columns. The first row stores a $CPVR_{row\ 0,i}$ that denotes a column name of a CTT or a VET, and the second row stores a $CPVR_{row\ 1,i}$ that denotes a column value of a CTT or a VET. $CPVR_{rowSize}$ denotes the size of $CPVR_{row}$. $CPVR_{blob}$ denotes a BLOB row matrix with 2 rows and n columns. The first row stores a $CPVR_{blob\ 0,j}$ that denotes a BLOB column name of a CTT or a VET, and the second row stores a $CPVR_{blob\ 1,j}$ that denotes a BLOB column value of a CTT or a VET. $CPVR_{blobSize}$ denotes the size of $CPVR_{blob}$. $CPVR_{clob}$ denotes a CLOB row matrix with 2 rows and n columns. The

first row stores a $CPVR_{clob\ 0,k}$ that denotes a CLOB column name of a CTT or a VET, and the second row stores a $CPVR_{clob\ 1,k}$ that denotes a CLOB column value of a CTT or a VET. $CPVR_{clobSize}$ denotes the size of $CPVR_{clob}$. $CPVR_{rowID}$ denotes the 'table_row_id' primary key of 'table_row', 'table_row_blob', and 'table_row_clob' ETs. $CPVR_{serialID}$ denotes the 'serial_id' column in the 'table_row', 'table_row_blob', and 'table_row_clob' ETs.

Algorithm 5-1: Creating Physical and Virtual Rows (CPVR)

Input: T, B, $CPVR_{type}$, $CPVR_{row}$, $CPVR_{blob}$, and $CPVR_{clob}$

1. **if** $CPVR_{type}$ = 'CTT' **then**
2. Insert the table row into B in RDBMS for T
3. **else if** $CPVR_{type}$ = 'VET' **then**
4. **if** $CPVR_{row} \notin$ B **then** /* When the row is not already exist in B */
5. $CPVR_{rowID} \leftarrow$ get max(table_row_id) from **table_row** ET + 1
6. **for** $i \leftarrow 0$ **to** $CPVR_{rowSize}$ **do**
7. $CPVR_{serialID} \leftarrow i$
8. Insert $CPVR_{rowID}$ **,** $CPVR_{serialID}$, T, B, $CPVR_{row\ 0,i}$, $CPVR_{row\ 1,i}$ into **table_row** ET
9. **if** $CPVR_{row\ 0,i}$ is indexed column **then**
10. Insert $CPVR_{rowID}$ **,** $CPVR_{serialID}$, T, B, $CPVR_{row\ 0,i}$, $CPVR_{row\ 1,i}$ into **table_index** ET
11. **end if**
12. $i \leftarrow i + 1$
13. **end for**
14. **for** $j \leftarrow 0$ **to** $CPVR_{blobSize}$ **do**
15. $CPVR_{serialID} \leftarrow j$
16. Insert $CPVR_{rowID}$ **,** $CPVR_{serialID}$, T, B, $CPVR_{blob\ 0,j}$, $CPVR_{blob\ 1,j}$ into **table_row_blob** ET
17. Store the BLOB file in its designated URI
18. $j \leftarrow j + 1$
19. **end for**
20. **for** $k \leftarrow 0$ **to** $CPVR_{clobSize}$ **do**
21. $CPVR_{serialID} \leftarrow k$
22. Insert $CPVR_{rowID}$ **,** $CPVR_{serialID}$, T, B, $CPVR_{clob\ 0,k}$ **,** $CPVR_{clob\ 1,k}$ into **table_row_clob** ET
23. $k \leftarrow k + 1$
24. **end for**
25. **end if**
26. **end if**

5.2.2 UPDATING PHYSICAL AND VIRTUAL ROWS ALGORITHM

This data management algorithm updates rows in CTTs and VETs by passing seven parameters to it, including the tenant ID, table name, table type, table row matrix, table BLOB matrix, table CLOB matrix, and the table row ID of a VET in the case when a tenant updates a VET row. More details of this algorithm are presented in Definition 5-2 and Algorithm 5-2.

Definition 5-2 (Updating Physical and Virtual Rows): T denotes a tenant ID. B denotes a table name. $UPVR_{type}$ denotes the table type whether it is a CTT or a VET. $UPVR_{row}$ denotes a row matrix with 2 rows and n columns. The first row stores a $UPVR_{row\ 0,i}$ that denotes a column name of a CTT or a VET, and the second row stores a $UPVR_{row\ 1,i}$ that denotes a column value of a CTT or a VET. $UPVR_{rowSize}$ denotes the size of $UPVR_{row}$. $UPVR_{blob}$ denotes a row BLOB matrix with 2 rows and n columns. The first row stores a $UPVR_{blob\ 0,j}$ that denotes a BLOB column name of a CTT or a VET, and the second row stores a $UPVR_{blob\ 1,j}$ that denotes a BLOB column value of a CTT or a VET. $UPVR_{blobSize}$ denotes the size of $UPVR_{blob}$. $UPVR_{clob}$ denotes a row CLOB matrix with 2 rows and n columns. The first row stores a $UPVR_{clob\ 0,k}$ that denotes a CLOB column name of a CTT or a VET, and the second row stores a $UPVR_{clob\ 1,k}$ that denotes a CLOB column value of a CTT or a VET. $UPVR_{clobSize}$ denotes the size of $UPVR_{clob}$. $UPVR_{rowID}$ denotes the 'table_row_id' primary key of 'table_row', 'table_row_blob', and 'table_row_clob' ETs. In each virtual table row, this ID is the same row ID for these three ETs.

Algorithm 5-2: Updating Physical and Virtual Rows (UPVR)

Input: T, B, $UPVR_{type}$, $UPVR_{row}$, $UPVR_{blob}$, $UPVR_{clob}$, and $UPVR_{rowID}$

1. **if** $UPVR_{type}$ = 'CTT' **then**
2. Update the table row in the **CTT** in RDBMS using T and B query filters
3. **else if** $UPVR_{type}$ = 'VET' **then**

4. **if** $UPVR_{row} \notin$ B **then** /* When the row is not already exist in B */
5. **for** $i \leftarrow 0$ **to** $UPVR_{rowSize}$ **do**
6. update $UPVR_{row\ 1,i}$ in **table_row** ET using T, B, $UPVR_{row\ 0,i}$, and $UPVR_{rowID}$ query filters
7. **if** $UPVR_{row\ 0,i}$ is custom index column **then**
8. Update $UPVR_{row\ 1,i}$ in **table_index** ET using T, B, $UPVR_{row\ 0,i}$, and $UPVR_{rowID}$ query filters
9. **end if**
10. $i \leftarrow i + 1$
11. **end for**
12. **for** $j \leftarrow 0$ **to** $UPVR_{blobSize}$ **do**
13. Update $UPVR_{blob\ 1,j}$ in **table_row_blob** ET using T, B, $UPVR_{blob\ 0,j}$, and $UPVR_{rowID}$ query filters
14. Delete the existing BLOB file in its designated URI
15. Insert the new BLOB file in its designated URI
16. $j \leftarrow j + 1$
17. **end for**
18. **for** $k \leftarrow 0$ **to** $UPVR_{clobSize}$ **do**
19. Update $UPVR_{clob1,k}$ in **table_row_clob** ET using T, B, $UPVR_{clob\ 0,k}$, and $UPVR_{rowID}$ query filters
20. $k \leftarrow k + 1$
21. **end for**
22. **end if**
23. **end if**

5.2.3 DELETING PHYSICAL AND VIRTUAL ROWS ALGORITHM

This data management algorithm deletes rows from CTTs and VETs by passing five parameters to it including the tenant ID, table name, table type, table row matrix, and the table row ID of a VET in the case when a tenant deletes a VET row. More details of this algorithm are presented in Definition 5-3 and Algorithm 5-3.

Definition 5-3 (Deleting Physical and Virtual Rows): T denotes a tenant ID. B denotes a table name. $DPVR_{type}$ denotes the table type whether it is a CTT or a VET. $DPVR_{row}$ denotes a row matrix with 2 rows and n columns. The first row stores a $DPVR_{row\ 0,i}$ that denotes a column name of a CTT or a VET, and the second row stores a $DPVR_{row\ 1,i}$ that denotes a column value of a CTT or a VET.

$DPVR_{rowID}$ denotes the 'table_row_id' primary key of 'table_row', 'table_row_blob', and 'table_row_clob' ETs. $DPVR_{MDR}$ denotes a master-detail relationship. $DPVR_{detail}$ denotes a details VET of the master table that this algorithm deletes its row. $DPVR_{detailsRow}$ denotes a row in the details table refers to the row that this algorithm aims to delete from the master table. $DPVR_{Relation}$ denotes a list of database relationships that a master CTT may have with details VETs, or a list of relationships that a master VET may have with details CTTs or VETs. $DPVR_{RelationSize}$ denotes the size of $DPVR_{Relation}$. $DPVR_{CTTrelation}$ denotes a list of CTT relationships that a master CTT may have with other details CTTs. $DPVR_{CTTrelationSize}$ denotes the size of $DPVR_{CTTrelation}$. $DPVR_{largeObj}$ denotes a row matrix with 1 row and n columns, each element of this matrix may contains 'BLOB' or 'CLOB' string. $DPVR_{PK}$ denotes a list of primary keys of CTTs or VETs.

Algorithm 5-3: Deleting Physical and Virtual Rows (DPVR)

Input: T, B, $DPVR_{type}$, $DPVR_{row}$, and $DPVR_{rowID}$

1. **if** $DPVR_{type}$ = 'CTT' **then**
2. $DPVR_{PK}$ ← get the primary keys of B from INFORMATION_SCHEMA.TABLE_CONSTRAINTS and INFORMATION_SCHEMA.KEY_COLUMN_USAGE views using T and B query filters
3. **else if** $DPVR_{type}$ = 'VET'
4. $DPVR_{PK}$ ← get the primary keys of B from table_column ET using T and B query filters
5. **end if**
6. **if** $DPVR_{PK} \neq$ Nil **then** **/* checking if any of the tables that have relationship with B have any row with references to the row that need to be deleted */**
7. **if** $DPVR_{type}$ = 'CTT' **then**
8. $DPVR_{CTTRelation}$ ← get the relationships of B from the INFORMATION_SCHEMA.KEY_COLUMN_USAGE view using T and B query filters
9. **for** $i \leftarrow 0$ **to** $DPVR_{CTTRelationSize}$ **do**
10. **if** $DPVR_{MDR} = DPVR_{CTTRelation\ i} \wedge DPVR_{detailsRow} \in DPVR_{details}$ **then**
11. return /* Exit Algorithm */
12. **end if**
13. $i \leftarrow i + 1$

14. **end for**
15. **end if**
16. $DPVR_{relation}$ ← get the relationships of B from the table_relationship ET using T and B query filters
17. **for** $j \leftarrow 0$ **to** $DPVR_{relationSize}$ **do**
18. **if** $DPVR_{MDR} = DPVR_{relation\ j} \wedge DPVR_{detailsRow} \in DPVR_{details}$ **then**
19. return /* Exit Algorithm */
20. **end if**
21. $j \leftarrow j + 1$
22. **end for**
23. **end if**
24. **if** $DPVR_{type}$ = 'CTT' **then**
25. Delete the row from the RDBMS using T, B, and $DPVR_{rowID}$ query filters
26. **else if** $DPVR_{type}$ = 'VET' **then**
27. $DPVR_{largeObj}$ ← get from the table_column ET the BLOB and CLOB objects using T and B query filters
28. **for all** $DPVR_{largeObj}$ **do**
29. **if 'BLOB'** ∈ $DPVR_{largeObj}$ **then**
30. Delete all BLOB rows from the table_row_blob ET using T, B, and $DPVR_{rowID}$ query filters
31. Delete the BLOB file from its designated URI
32. **end if**
33. **if 'CLOB'** ∈ $DPVR_{largeObj}$ **then**
34. Delete all CLOB rows from the table_row_clob ET using T, B, and $DPVR_{rowID}$ query filters
35. **end if**
36. **end for**
37. Delete rows from table_row ET using T, B, and $DPVR_{rowID}$ query filters
38. **end if**

5.3 PERFORMANCE EVALUATIONS

In this thesis, the EET multi-tenant database schema and EET framework architecture are designed to serve multiple tenants in one application instance. However, the aim of the experiments of this chapter is evaluating the performance of EETSHS for one tenant. As long as in the multi-tenant database the data of each tenant's user is isolated in a table partition, these experiments can evaluate the effectiveness of managing data for each single tenant from the multi-tenant database.

The multi-tenant database performance needs to be tested in one single server instance before considering scale-up or scale-out multi-tenant databases. This approach is applied to test the effectiveness of running database operations for CTTs and VETs using EETSHS. These experiments compare the performance of the query execution time from a CTT (traditional physical table), and VET that is stored in the ETs.

5.3.1 EXPERIMENTAL DATA SET AND SETUP

In this experiment, three of the EETSHS functions are invoked to insert, update, and delete 1, 10, 50, and 100 rows from the 'product' table. This table structure is used for both the 'product' CTT and VET. There are 200,000 rows stored in these tables that belong to a tenant whose 'tenant_id' equals 1000, and the 'db_table_id' of the 'product' VET in the 'db_table' ET equals 16. The 'product' CTT has a master-detail relationship with the 'sales_fact' CTT, whereas the 'product' VET has a master-detail relationship with the 'sales_fact' VET. The 'db_table_id' of the 'sales_fact' VET in the 'db_table' ET equals 17. The 'product_id' for both the 'product' CTT and VET equal '300000'. The 'table_row_id' of the 'product' VET equals '50000001'. Figure 5-2 shows the 'product' and 'sales_fact' tables. In both the 'product' CTT and VET, the values, including 300000, 1000, 123123, 11.5, Red, 100, 10 cm, 140 g are inserted respectively in the following columns 'product_id', 'tenant_id', 'product_bus_id', 'standard_cost', 'color', 'price', 'size', and 'weight'. In addition, the values, including 444333, 12.5, Blue, 105, 105 cm, 155 g are updated respectively in the following columns 'product_bus_id', 'standard_cost', 'color', 'price', 'size', and 'weight'. This data set presents the values that used in the experiments to test inserting, updating, and deleting one row from the 'product' table. However, other values were used to manage the rest of the rows. This section presents the three experiments and the queries of these experiments are shown in Table 5-4.

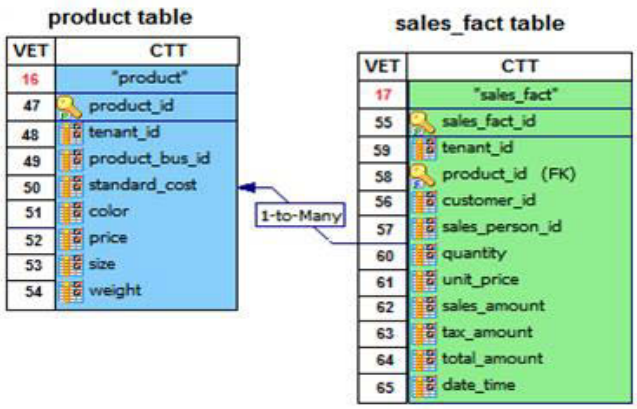


Figure 5-2: The product and the sales_fact tables' structures.

1) **Inserting Physical and Virtual Rows Experiment (Exp. 5-1):** The aim of this experiment is benchmarking the query execution time of inserting rows into the 'product' CTT and VET. The Create Physical and Virtual Rows Function is invoked from EETSHS that executes Query 5-1 (Q5-1) on the 'product' VET. This query comprises of four subsidiary queries, the first query retrieves the maximum number of 'table_row_id' from the 'table_row' ET. The second query retrieves records from 'table_index' ET to check if the virtual column name value of the 'product_id' primary key that equals 47 and its value that equals '300000' has already existed or not before inserting the row. The third query inserts eight column values of the 'product' VET in the 'table_row' ET. The fourth query inserts the values of three column indexes, including primary key, foreign key, and custom index into the 'table_index' ET. Whereas, the same function executes Query 5-2 (Q5-2) on the 'product' CTT to insert the same row values that are inserted in Q5-1.

2) **Updating Physical and Virtual Rows Experiment (Exp. 5-2):** The aim of this experiment is benchmarking the query execution time of updating rows in the 'product' CTT and VET. The Update Physical and Virtual Rows Function is invoked from EETSHS that executes Query 5-3 (Q5-3) on the 'product' VET. This query comprises of three subsidiary queries, the first query retrieves records from 'table_index' ET to check if the column name ID of the virtual 'product_id' primary

key equals 47 and the value of this column equals '300000' is already exist or not before updating the row. The second query updates six column values of the 'product' VET in the 'table_row' ET, excluding the primary key and foreign key values. The third query updates the custom index value in the 'table_index' ET. Whereas the same function executes Query 5-4 (Q5-4) on the 'product' CTT to update the same row values that are updated in Q5-3.

3) **Deleting Physical and Virtual Rows Experiment (Exp. 5-3):** The aim of this experiment is benchmarking the query execution time of deleting rows from the 'product' CTT and VET. The Delete Physical and Virtual Rows Function is invoked from EETSHS that executes Query 5-5 (Q5-5) on the 'product' VET. This query comprises of five subsidiary queries, the first query retrieves the database relationships that the 'product' VET has with the other VETs and CTTs from the 'table_relationship' ET. The second query retrieves only the column of BLOB and CLOB data type from a VET, and in this experiment the structure of the 'product' VET does not have any of them. The results of the first query indicated that the 'sales_fact' VET has a master-detail relationship with the 'product' VET. Therefore, the third query checks if the 'sales_fact' VET that is a details table of the master 'product' VET has a row refers to the row that this function aims to delete or not. The fourth query deletes all the rows of the indexed columns' rows that related to the 'product' VET from the 'table_index' ET. The fifth query, deletes all the rows of the columns that related to the 'product' VET from the 'table_row' ET, and as long as this table does not have BLOB or CLOB columns, then no rows are deleted from the 'table_row_blob' and 'table_row_clob' ETs. Whereas the same function executes Query 5-6 (Q5-6) on the 'product' CTT. This query comprises of five subsidiary queries, the first query retrieves the physical primary keys of the 'product' CTT using a query that joins the INFORMATION_SCHEMA. TABLE_CONSTRAINTS and INFORMATION_SCHEMA.KEY_COLUMN_USAGE views. The second query retrieves the details CTTs' names that have a master-detail relationship with the 'product' master CTT from the INFORMATION_-

SCHEMA.KEY_COLUMN_USAGE view. The third query checks if the 'sales_fact' CTT that is a details table of the master 'product' CTT has a row refers to the row that the function aims to delete or not. The fourth query retrieves the details VETs' names that have a master-detail relationship with the 'product' CTT. However, since the 'product' CTT does not have any relationship with any VET, therefore no any further queries executed to check if a details table has a row refer to the primary key of the 'product' CTT such as what is done in the third query of Q5-5. The fifth subsidiary query deletes the row from the 'product' CTT.

The EETSHS service 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, the database and the application server are deployed on the same PC. The operating system is Windows 7 Home Premium, with Intel Core i5 2.40GHz CPU, 8 GB of RAM memory, and 500 GB of hard disk storage.

5.3.2 EXPERIMENTAL RESULT

This section shows the three experimental results of inserting, updating, and deleting rows from the 'product' CTT or VET as follows:

1) **Inserting Physical and Virtual Rows Experimental Result:** The experimental study of Exp.5-1 is showing that the execution time of Q5-1 that performed on the 'Product' VET is approximately 16% slower on average than the execution time of Q5-2 that performed on the 'product' CTT when 1, 10, 50, and 100 rows are inserted. The results of this experiment are shown in Figure 5-3 and Table 5-1.

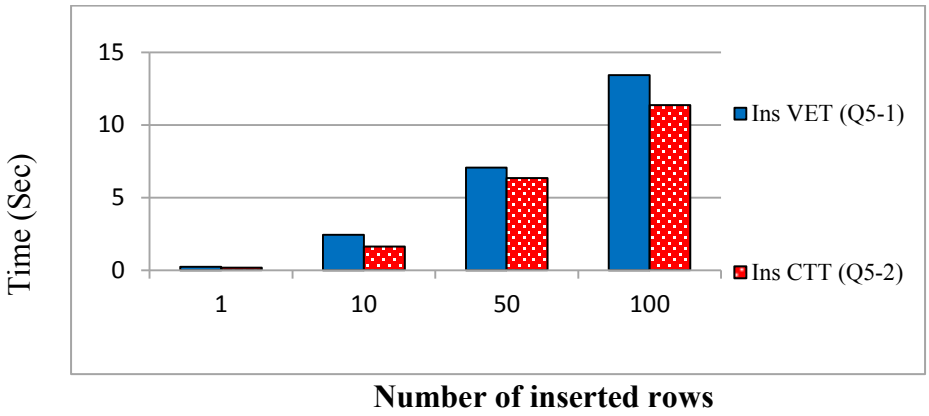


Figure 5-3: Inserting rows experiment

Table 5-1: The query execution times of inserting rows experiment (Exp. 5-1)

Number of inserted rows	VET (Q5-1) Time in seconds	CTT (Q5-2) Time in seconds
1	0.242	0.162
10	2.427	1.624
50	7.065	6.330
100	13.434	11.356

2) **Updating Physical and Virtual Rows Experimental Result:** The experimental study of Exp.5-2 is showing that the execution time of Q5-3 that performed on the 'Product' VET is approximately 12% slower on average than the execution time of Q5-4 that performed on the 'product' CTT when 1, 10, 50, and 100 rows are updated. The results of this experiment are shown in Figure 5-4.

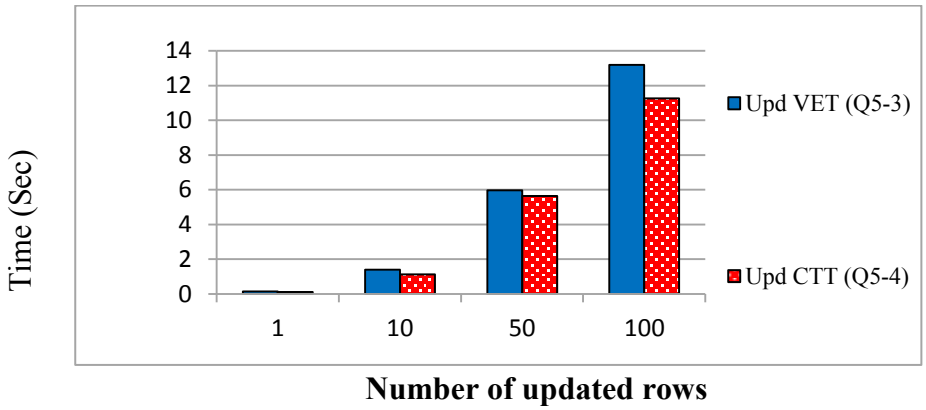


Figure 5-4: Updating rows experiment

Table 5-2: The query execution times of updating rows experiment (Exp. 5-2)

Number of updated rows	VET (Q5-3) Time in seconds	CTT (Q5-4) Time in seconds
1	0.140	0.111
10	1.400	1.116
50	5.971	5.633
100	13.186	11.25

3) **Deleting Physical and Virtual Rows Experimental Result:** The experimental study of Exp.5-3 is showing that the execution time of Q5-5 that performed on the 'Product' VET is approximately 73% faster on average than the execution time of Q5-6 that performed on the 'product' CTT when 1, 10, 50, and 100 rows are deleted. The results of this experiment are shown in Figure 5-5.

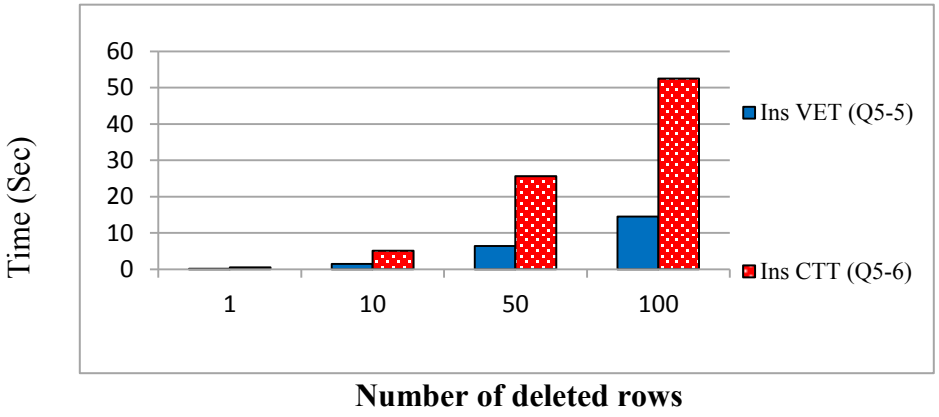


Figure 5-5: Deleting rows experiment

Table 5-3: The query execution times of deleting rows experiment (Exp. 5-3)

Number of deleted rows	VET (Q5-5) Time in seconds	CTT (Q5-6) Time in seconds
1	0.143	0.503
10	1.430	5.037
50	6.374	25.633
100	14.498	52.506

Table 5-4: The experiments queries

Query No.		Query Details
Q1	**1**	**SELECT** max(table_row_id) From table_row;
	2	**SELECT *** FROM **table_index** WHERE tenant_id=1000 and db_table_id=16 and table_column_id=47 and row_value='300000' order by table_row_id ASC;
	3	**INSERT** into **table_row**(table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,1,1000, '300000',16,47); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,2,1000, '1000',16,48); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,3,1000, '123123',16,49); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,4,1000, '11.5',16,50); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,5,1000, 'Red',16,51); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,6,1000, '100',16,52); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,7,1000, '10 cm',16,53); INSERT into table_row (table_row_id, serial_id, tenant_id, value, db_table_id, table_column_id) values (50000001,8,1000, '140 g',16,54);

	4	**INSERT** into **table_index** (tenant_id, value, table_row_id, serial_id, db_table_id, table_column_id) values (1000, '300000',50000001,1,16,47); INSERT into table_index (tenant_id, value, table_row_id, serial_id, db_table_id, table_column_id) values (1000, '1000',50000001,2,16,48); INSERT into table_index (tenant_id, value, table_row_id, serial_id, db_table_id, table_column_id) values (1000, '11.5',50000001,4,16,50);
Q2	**1**	INSERT into product (product_id, tenant_id, product_bus_id, standard_cost, color, price, size, weight) values (300000,1000,'123123',11.5,'Red',100,'10 cm','140 g');
Q3	**1**	**SELECT** * FROM **table_index** WHERE tenant_id=1000 and db_table_id=16 and table_column_id=47 and row_value='300000' order by table_row_id ASC;
	2	**UPDATE table_row** set value = '444333' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 49 AND table_row_id =50000001; UPDATE table_row set value = '12.5' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 50 AND table_row_id =50000001; UPDATE table_row set value = 'Blue' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 51 AND table_row_id =50000001; UPDATE table_row set value = '105' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 52 AND table_row_id =50000001; UPDATE table_row set value = '105 cm' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 53 AND table_row_id =50000001; UPDATE table_row set value = '155 g' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 54 AND table_row_id =50000001;
	3	**UPDATE table_index** set value = '12.5' WHERE tenant_id = 1000 AND db_table_id = 16 AND table_column_id = 50 AND table_row_id =50000001;
Q4	**1**	UPDATE **product** SET product_bus_id = '444333', standard_cost = 12.5, color = 'Blue', price = 105, size = '105 cm', weight = '155 g' WHERE tenant_id = 1000 and product_id = 300000;
Q5	**1**	**SELECT** * FROM **table_relationship** WHERE tenant_id=1000 and (db_table_id=16 or target_table_id=16) order by table_relationship_id;
	2	**SELECT** * FROM **table_column** WHERE data_type= 2 or data_type = 3 and db_table_id=16 order by table_column_id;
	3	**SELECT** * FROM **table_index** WHERE tenant_id=1000 and db_table_id=17 and table_column_id=58 and row_value='300000' order by table_row_id ASC;
	4	**DELETE** from **table_index** WHERE tenant_id = 1000 AND db_table_id = 16 AND table_row_id =50000001;
	5	**DELETE** from **table_row** WHERE tenant_id = 1000 AND db_table_id = 16 AND table_row_id =50000001;
Q6	**1**	**SELECT** c.COLUMN_NAME FROM **INFORMATION_SCHEMA.TABLE_CONSTRAINTS** pk , **INFORMATION_SCHEMA.KEY_COLUMN_USAGE** c where pk.TABLE_NAME = 'product' and CONSTRAINT_TYPE = 'PRIMARY KEY' and c.TABLE_NAME = pk.TABLE_NAME and c.CONSTRAINT_NAME = pk.CONSTRAINT_NAME;
	2	**SELECT** distinct table_name from **INFORMATION_SCHEMA.KEY_COLUMN_USAGE** where column_name in ('product_id');
	3	**SELECT** sales_fact_id FROM **sales_fact** WHERE product_id=300000 limit 1;
	4	**SELECT** * FROM **table_relationship** WHERE tenant_id=1000 and shared_table_name = 'product' order by table_relationship_id;
	5	**DELETE** FROM **product** WHERE tenant_id = 1000 and shr_product_id = 300000;

5.4 SUMMARY

This chapter proposes a multi-tenant data management service based on EET, which called EETSHS. This service provides functions that allow tenants to manage their data by calling the service functions, without the need of writing SQL queries and backend data management code. In using this service, tenants can create VETs and create VETs' columns, rows, relationships, primary keys, indexes, and other columns constraints. In addition, tenants can create CTTs' rows, and database relationships between CTTs and VETs, while the rest of the CTT database operations, including creating CTTs, CTTs' columns, database relationships between two CTTs, primary keys, indexes, and other columns constraints can be managed from a traditional RDBMS instead of EETSHS. That is because CTTs are shared between multiple tenants, and changing any of these operations affect all the tenants. Moreover, this service ensures a high level of multi-tenant data quality, configurability, consistency, accessibility, and manageability. In this chapter, three algorithms were developed to manage CTTs and VETs rows, and several experiments were performed using these algorithms to measure the feasibility and effectiveness of managing data using this service that based on EET. The experimental results show that the query execution time of inserting and updating rows in the tenants' CTTs is slightly faster than in the tenants' VETs. This increase in the query execution time of VET is not significant compared to the benefits that this service brings to SaaS and Big Data applications. The experimental results of deleting rows from the tenants' CTTs are approximately four times slower than deleting them from the tenants' VETs. This increase in the query execution time occurs in CTTs that are the traditional physical tables of EET, due to the process of deleting a CTT row is more complicated than VET. As long as EETSHS checks before deleting a CTT row, if the CTT has a master-detail relationship with other CTTs or VETs, and it checks if any of these tables have any row with references to the row that need to be deleted. In general, these experimental results make this service and EET schema a suitable

candidate for the management of multi-tenant data for software applications in general, and SaaS and Big Data applications in particular.

CHAPTER 6

MULTI-TENANT DATABASE PROXY METHOD

An important challenge in the design of multi-tenant databases that support SaaS or Big Data application is to provide a solution that manages large volumes of data collected from different data sources. It is a difficult and complex task to integrate and utilize such data and to find ways to use it in businesses and operational strategies. These various collections of data require a multi-tenant data service to accommodate these collections of data together and make them operate as one database. This chapter proposes a multi-tenant database proxy service, called Elastic Extension Tables Proxy Service (EETPS). This service is based on EET, and it integrates, generates, and executes tenants' queries by using a codebase solution that converts multi-tenant queries into traditional database queries and execute them in a RDBMS.

The remainder of this chapter is structured as follows. Section 6.1 proposes the EET proxy service. Section 6.2 presents sample algorithms for five functions of EETPS. Section 6.3 presents a set of experiments to compare the performance of retrieving data from CTTs, integrated CTTs and VETs, and VETs. Section 6.4 concludes this chapter.

6.1 ELASTIC EXTENSION TABLES PROXY SERVICE

The Elastic Extension Tables Proxy Service combines, generates, and executes tenants' queries by using a codebase solution which converts multi-tenant queries into traditional database queries, then executes them in a RDBMS. This service has three objectives: (1) allowing the tenants' applications to retrieve table rows from CTTs, VETs, or both CTTs and VETs; (2) allowing tenants to extend a business domain database that based on traditional RDBMS during their applications' runtime execution; (3) avoiding tenants from spending money and efforts on writing SQL queries, learning special programming languages, and writing backend data management code by 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. These functions return a two dimensional array (Object [α] [β]), where α is the number of array rows that represents a number of retrieved rows, and β is the number of array columns that represent a number of retrieved columns for a particular CTT or VET. These functions are designed to retrieve tenants' data from the following tables:

- One table, either a CTT or a VET.
- Two tables have One-to-One, One-to-Many, Many-to-One, Many-to-Many, or Self-referencing relationships. These relationships can be between two VETs, two CTTs, or one VET and one CTT.
- Two tables based on a common field 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 operations can be used between two VETs, two CTTs, or a VET and a CTT.
- Two tables or more may have or may 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 that have relationships between them, by using filters on multiple tables, or filtering data based on the results of subqueries.

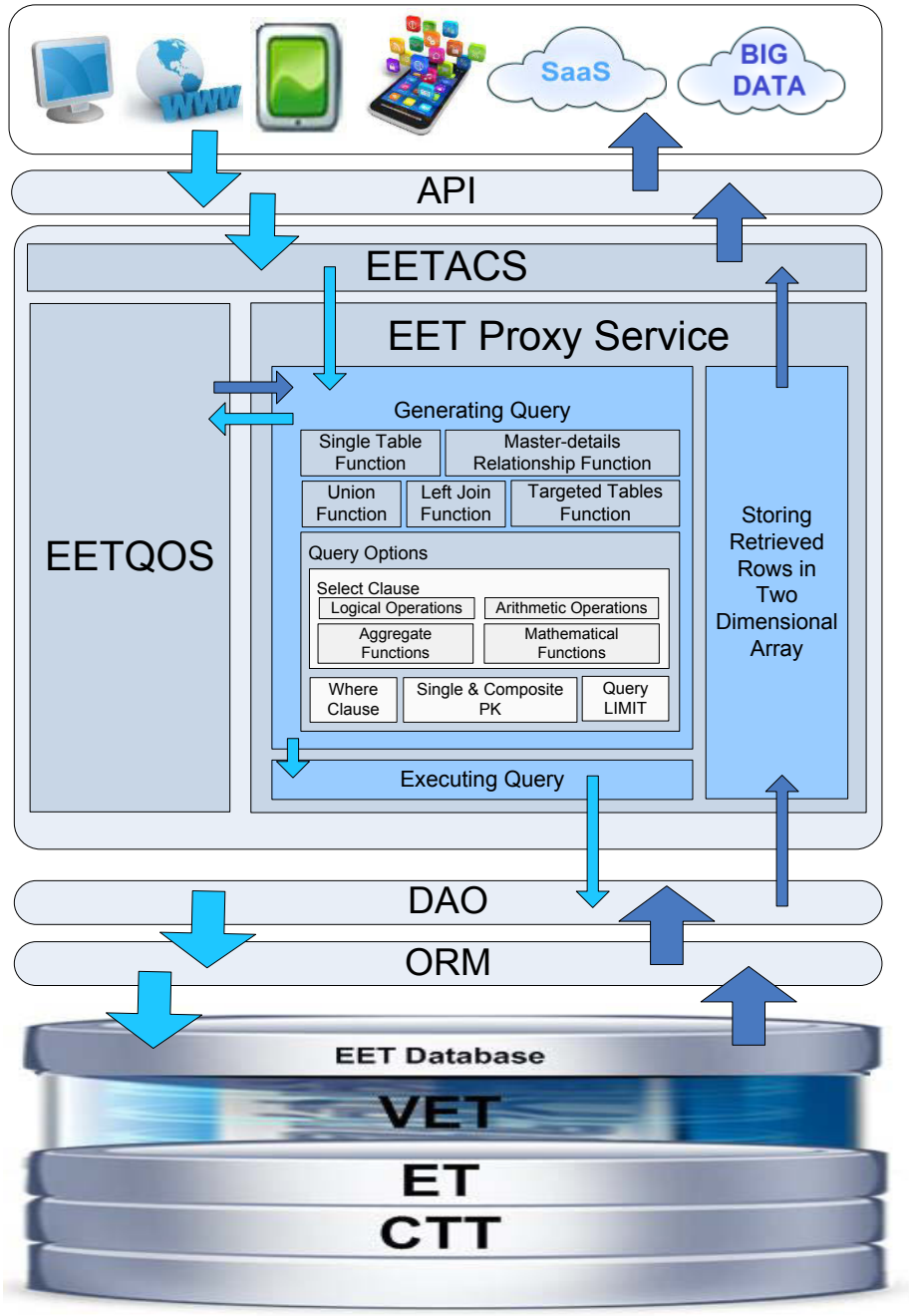


Figure 6-1: EETPS overview architecture

Moreover, the EETPS functions have the capabilities of retrieving data from CTTs or VETs by using query options, including 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. The overview architecture of the EETPS is shown in Figure 6-1. This architecture shows the main six layers of EETPS, including the presentation layer, the API layer, the service layer, DAO layer, ORM layer, and the domain layer. The presentation layer represents the applications that may access EET schema through the EETPS such as Big Data, SaaS, mobile, web, and stand-alone applications. The API layer consists of EET Data Retrieval API. The service layer presents the details of EETPS and the other two services that interact with it including, EETACS and EETQOS. The details of these interactions are stated in Chapter 3. The DAO layer consists of the CTTDAO and EETDAO. The ORM layer consists of the ORM objects of EET schema. Finally, the Domain layer consists of the multi-tenant EET schema. The presentation layer allows the tenants of the service, to retrieve data by calling functions from the EETPS through the API layer, and passing parameters to these functions. The invoked EETPS function generates a tenant query from CTTs and/or VETs by using the ET and a number of query options, and then executes the generated query in a RDBMS. The RDBMS returns the retrieved table rows from EET and passes these rows back to the EETPS to store them in an array. Finally, the EETPS returns the tenant's requested data in an array to the tenant through the API layer.

6.2 Elastic Extension Tables Proxy Service Algorithms

In this section, we present the main algorithms of EETPS functions and some subsidiary algorithms of these main algorithms.

6.2.1 SINGLE TABLE QUERY ALGORITHM

This section presents the main algorithm and some subsidiary algorithms of the Single Table function that retrieves table rows from a CTT or a VET. This algorithm has three different cases to retrieve table rows from a VET. Firstly, retrieving rows from a VET by specifying a set of primary keys. Secondly, retrieving rows from a VET by specifying a set of table row IDs that are stored in 'table_row' ET. Thirdly, retrieving all rows of a CTT or a VET without specifying any primary key or row ID.

1) **Single Table Main Algorithm:** This algorithm is used to retrieve table rows from one single table either a CTT or a VET, and it is outlined in Definition 6-1 and Algorithm 6-1.

Definition 6-1 (Single Table Query Main Algorithm): T denotes a tenant ID. B denotes a table name. STQ_{rowID} denotes a set of table row IDs. STQ_{PK} 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 the first result number of a query limit. M denotes the maximum amount number of a query limit. STQ_{type} denotes the table type (CTT or VET). STQ_{index} denotes a set of VET indexes. $STQ_{PKIndex}$ denotes a primary key indexes of a VET. STQ_{CTT} denotes a set of retrieved rows from a CTT. STQ_{VET} denotes a set of retrieved rows from a VET. STQ_{array} denotes a two dimensional array that stores the retrieved rows.

Algorithm 6-1: SingleTableQuery (STQ)

Input: T, B, STQ_{rowID}, STQ_{PK}, S, W, F, M, and STQ_{type}
Output: STQ_{array}
1. **if** STQ_{type} = 'CTT' **then**
2. W ← Concatenate (W, STQ_{PK})
3. STQ_{CTT} ← **CovertCTTtoVETStructure**(T, B, S, W, F, M) /* Algorithm 6-4 */
4. **else if** STQ_{type} = 'VET' **then**
5. **if** $STQ_{PK} \neq$ null **then**
6. STQ_{VET} ← **GetTableRowQuery**(T, B, Nil, STQ_{PK}, S, W, F, M) /*Algorithm 6-2 */
7. **else if** $STQ_{rowID} \neq$ null **then**

8. STQ_{VET} ← **GetTableRowQuery**(T, B, STQ_{rowID}, Nil, S, W, F, M) /*Algorithm 6-2 */
9. **else**
10. $STQ_{PKIndex}$ ← retrieve the primary key index column from table_column ET using T, and B query filters
11. STQ_{index}← retrieve indexes from table_index ET using T, B, and $STQ_{PKIndex}$ query filters
12. **if** STQ_{index} ∈ B **then** /* If B has indexes */
13. STQ_{VET} ← **GetTableRowQuery**(T, B, STQ_{index}, Nil, S, W, F, M)
14. /* Algorithm 6-2 */
15. **else**
16. STQ_{VET} ← **GetTableRowQuery**(T, B, Nil, Nil, S, W, F, M)
17. /* Algorithm 6-2 */
18. **end if**
19. **end if**
20. **end if**
21. **if** STQ_{type} = 'CTT' **then**
22. STQ_{array} ← **StoreRowsInArray(**T, B, STQ_{CTT}) /* Algorithm 6-3 */
23. **else if** STQ_{type} = 'VET' **then**
24. STQ_{array} ← **StoreRowsInArray(**T, B, STQ_{VET}) /* Algorithm 6-3 */
25. **end if**
26. **Return** STQ_{array}

2) **Get Table Row Query Algorithm:** This subsidiary query algorithm is used to retrieve tenant's table rows from a VET. The database query of this algorithm uses UNION operator keyword to combine the result-set of three SELECT statements for three tables, including 'table_row', 'table_row_blob', and 'table_row_clob' ETs when the VET contains BLOB and/or CLOB. However, if the VET does not contain BLOB and CLOB then this algorithm do not use the UNION operator in the query and instead it retrieve data from only the 'table_row' ET. The details of this algorithm are outlined in Definition 6-2 and Algorithm 6-2.

Definition 6-2 (Get Table Row Query Algorithm): T denotes a tenant ID. B denotes a table name. $GTRQ_{rowID}$ denotes a set of table row IDs. $GTRQ_{PK}$ denotes a primary key row matrix with 2 rows and n columns. The first row stores a $GTRQ_{PK\ 0,i}$ that denotes a value of virtual primary key column ID stored in the 'table_column_id' column of any of the three ETs, including the 'table_row', the 'table_row_blob', and the 'table_row_clob'. The second row stores a $GTRQ_{PK\ 1,i}$ that

denotes the value of the $GTRQ_{PK\,0,i}$. S denotes a string of the SELECT clause. W denotes a string of the WHERE clause. F denotes the first result number of a query limit. M denotes the maximum amount number of a query limit that is retrieved from the 'table_row' ET. $GTRQ_{queryStr}$ denotes a string contains the structure of a select statement that is executed in this algorithm. $GTRQ_{rows}$ denotes the retrieved rows from RDBMS after executing the $GTRQ_{queryStr}$.

Algorithm 6-2: GetTableRowQuery (GTRQ)

Input: T, B, $GTRQ_{rowID}$, $GTRQ_{PK}$, S, W, F, and M
Output: $GTRQ_{rows}$
1. W ← Concatenate(W, $GTRQ_{PK}$)
/* Store the select statement string into $GTRQ_{queryStr}$ */
2. $GTRQ_{queryStr}$ ← Retrieve rows from table_row ET using T, B , $GTRQ_{rowID}$, S, and W ∪
Retrieve rows from table_row_blob ET using T, B, $GTRQ_{rowID}$, S, and W ∪
Retrieve rows from table_row_clob ET using T, B, $GTRQ_{rowID}$, S, and W using a limit of the rows result between F and M
3. $GTRQ_{queryStr}$ ← execute $GTRQ_{queryStr}$ in RDBMS
4. **Return** $GTRQ_{rows}$

---

3) **Store Rows 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 rows, and the number of array columns represents a number of retrieved columns. The column names are stored in the first element of this two dimensional array, and the column values are stored in the rest of the array elements. The details of this algorithm are outlined in Definition 6-3 and Algorithm 6-3.

Definition 6-3 (Store Rows in Array Algorithm): T denotes a tenant ID. B denotes a table name. $SRA_{rowsList}$ denotes a set of retrieved rows from a CTT or a VET where each of these rows is denoted by SRA_{row}. Each SRA_{row} is a set of columns and each column is denoted by SRA_{col}. $SRA_{row\,n}$ ($SRA_{col\,m}$) denotes a value stored in the $SRA_{col\,m}$ of the $SRA_{row\,n}$. $SRA_{rowsListSize}$ denotes the size of the $SRA_{rowsList}$. $SRA_{rowSize}$ denotes the size of the SRA_{row}. $SRA_{colNames}$ denotes a set

of column names of a CTT or a VET. $SRA_{colNamesSize}$ denotes the size of the $SRA_{colNames}$. SRA_{array} denotes a two dimensional array to store the retrieved rows.

Algorithm 6-3: StoreRowsInArray (SRA)

Input: T, B, and $SRA_{rowsList}$
Output: SRA_{Array}
1. $SRA_{colNames} \leftarrow$ retrieve the column names from table_column ET using T and B query filters
2. Initialize SRA_{array} $[SRA_{rowsListSize}]$ $[SRA_{colNamesSize}]$
3. **for** $i \leftarrow 0$ **to** $SRA_{colNameSize}$ **do** /* This loop stores the row columns names */
4. $SRA_{array}[0][i] \leftarrow SRA_{colNames\,i}$
5. $i \leftarrow i + 1$
6. **end for**
7. **for** $n \leftarrow 0$ **to** $SRA_{rowsListSize}$ **do** /* This loop stores the rows columns values */
8. **for** $m \leftarrow 0$ **to** $SRA_{rowSize}$ **do**
9. $SRA_{array}[n+1][m] \leftarrow SRA_{row\,n}\ (SRA_{col\,m})$
10. $m \leftarrow m + 1$
11. **end for**
12. $n \leftarrow n + 1$
13. **end for**
14. **Return** SRA_{array}

4) **Convert CTT Structure to VET Structure Algorithm:** This subsidiary algorithm is used to convert the retrieved data from a CTT into VET structure that consists of two-dimensional array, the number of array rows represents a number of retrieved rows, and the number of array columns represents a number of retrieved columns of a CTT. The column names are stored in the first element of this two dimensional array, and the data in these columns are stored in the rest of the array elements. The details of this algorithm are outlined in Definition 6-4 and Algorithm 6-4.

Definition 6-4 (Convert CTT to VET Structure Algorithm): T denotes a tenant ID. B denotes a table name. S denotes a string of the SELECT clause parameter. W denotes a string of the WHERE clause. F denotes the first result number of a query limit. M denotes the maximum amount number of a query limit. $CCVS_{rowList}$ denotes a set of retrieved rows from a CTT where each row is denoted as $CCVS_{row}$. Each

$CCVS_{row}$ is a set of columns and each column is denoted by $CCVS_{col}$. $CCVS_{row\,n}$ ($CCVS_{col\,m}$) denotes a value stored in the $CCVS_{col\,m}$ of the $CCVS_{row\,n}$. $CCVS_{rowListSize}$ denotes the size of $CCVS_{rowList}$, and the size of each $CCVS_{row}$ in the $CCVS_{rowList}$ is denoted by $CCVS_{rowSize}$. $CCVS_{colNames}$ denotes a set of a CTT columns names. $CCVS_{colNamesSize}$ denotes the size of the $CCVS_{colNames}$. $CCVS_{array}$ denotes a two dimensional array that stores the retrieved rows from a CTT.

Algorithm 6-4: CovertCTTtoVETStructure (CCVS)

Input: T, B, S, W, F, and M
Output: $CCVS_{array}$

1. $CCVS_{colNames} \leftarrow$ retrieve the column names of a CTT from INFORMATION_SCHEMA.COLUMNS view using B query filter
2. $CCVS_{rowList} \leftarrow$ retrieve rows from B using T, S, W, F, and M query filters
3. **for** $i \leftarrow 0$ **to** $CCVS_{colNamesSize}$ **do**
4. $CCVS_{array}\ [0]\ [i] \leftarrow CCVS_{colNames\,i}$
5. $i \leftarrow i + 1$
6. **end for**
7. **for** $n \leftarrow 0$ **to** $CCVS_{rowListSize}$ **do**
8. **for** $m \leftarrow 0$ **to** $CCVS_{rowSize}$ **do**
9. $CCVS_{array}\,[n+1]\,[m] \leftarrow CCVS_{row\,n}\ (CCVS_{col\,m})$
10. $m \leftarrow m + 1$
11. **end for**
12. $n \leftarrow n + 1$
13. **end for**
14. **Return** $CCVS_{array}$

6.2.2 One-to-Many Query Algorithm

This algorithm retrieves table rows from two CTTs, two VETs, or one VET and one CTT. These two tables may have any of the following database relationships between them, including One-to-One, One-to-Many, Many-to-One, Many-to-Many, or Self-referencing. In this section, a sample algorithm of the One-to-Many relationship is presented as outlined in Definition 6-5 and Algorithm 6-5.

Definition 6-5 (One-to-Many Query Algorithm): T denotes a tenant ID. $OTMQ_{master}$ denotes the master table of the One-to-Many relationship. $OTMQ_{details}$ denotes the details table of the One-to-Many relationship. $OTMQ_{masterPK}$ denotes a

row matrix with 2 rows and n columns. The first row stores $OTMQ_{masterPK\ 0,i}$ that denotes a primary key column name of a table. The second row stores $OTMQ_{masterPK\ 1,i}$ that denotes a table primary key value of the $OTMQ_{masterPK\ 0,i}$. S denotes a string of the SELECT clause. W denotes a string of the WHERE clause. F denotes the first result number of a query limit. M denotes the maximum amount number of a query limit. $OTMQ_{type}$ denotes the $OTMQ_{details}$ type (CTT or VET). $OTMQ_{detailsFK}$ denotes a set stores foreign keys columns names of the details table. $OTMQ_{swapPK}$ denotes a row matrix with 2 rows and n columns to store . The first row stores a $OTMQ_{swapPK\ 0,i}$ that denotes the column name of a foreign key. This foreign key belongs to the details table and refers to a primary key in the master table. The second row stores $OTMQ_{swapPK\ 1,i}$ that denotes a value of the $OTMQ_{swapPK\ 0,i}$. $OTMQ_{rows}$ denotes a set of retrieved rows from a VET. $OTMQ_{array}$ denotes a two dimensional array to store the retrieved rows.

Algorithm 6-5: OneToManyQuery (OTMQ)

Input: T, $OTMQ_{master}$, $OTMQ_{details}$, $OTMQ_{masterPK}$ S, W, F, M, and $OTMQ_{type}$
Output: $OTMQ_{array}$

1. **if** $OTMQ_{type}$ = 'CTT' **then**
2. W ← Concatenate (W, $OTMQ_{masterPK}$)
3. $OTMQ_{array}$ ← **CovertCTTtoVETStructure**(T, B, S, W, F, M)
   /* Algorithm 6-4 */
4. **else if** $OTMQ_{type}$ = 'VET' **then**
5. $OTMQ_{detailsFK}$ ← retrieve the foreign keys of $OTMQ_{details}$ which has a master details relationship with $OTMQ_{master}$ from the table_relationship ET using T, B query filters
6. **for** $i \leftarrow 0$ **to** $OTMQ_{masterPK}$ **do**
7. **for** $j \leftarrow 0$ **to** $OTMQ_{detailsFK}$ **do**
8. **if** $OTMQ_{detailsFK\ j} \in OTMQ_{masterPK\ 0,i}$ **then**
9. $OTMQ_{swapPK\ 0,i} \leftarrow OTMQ_{detailsFK\ j}$
10. $OTMQ_{swapPK\ 1,i} \leftarrow OTMQ_{masterPK\ 1,i}$
11. **end if**
12. $j \leftarrow j + 1$
13. **end for**
14. $i \leftarrow i + 1$
15. **end for**
16. $OTMQ_{rows}$ ← **GetTableRowQuery(**T, B, Nil, $OTMQ_{swapPK}$,S, W, F, M)

/* Algorithm 6-2 */
17. $OTMQ_{array}$ ← **StoreRowsInArray(**T, B, $OTMQ_{rows}$) /* Algorithm 6-3 */
18. **end if**
19. **Return** $OTMQ_{array}$

6.2.3 UNION QUERY ALGORITHM

This algorithm retrieves a combined result-set of two or more tables, whether they are CTTs, VETs or a combination of CTTs and VETs, and then stores the result-set in an array. These tables may or may not have relationships between them. The details of this algorithm are outlined in Definition 6-6 and Algorithm 6-6.

Definition 6-6 (Union Query Algorithm): T denotes a tenant ID. UQ_{tables} denotes a set of CTTs and/or VETs tables names. $UQ_{colNames}$ denotes a matrix with 1 row and n columns. Each column in this matrix contains of a set of arbitrary table columns which are related to a table in UQ_{tables}. W denotes a set of WHERE clauses which are related to the UQ_{tables} and the columns are ordered according to the table orders of UQ_{tables}. F denotes the first result number of a query limit. M denotes the maximum amount number of a query limit. UQ_{rows} denotes a set of retrieved rows from a CTT or a VET where each row is denoted as UQ_{row}. Each UQ_{row} is a set of columns and each column is denoted by UQ_{col}. $UQ_{row\,n}$ ($UQ_{col\,m}$) denotes a value stored in $UQ_{col\,m}$ of $UQ_{row\,n}$. $UQ_{tablesSize}$ denotes the size of UQ_{tables}. $UQ_{rowSize}$ denotes the size of UQ_{row}. $UQ_{colNamesSize}$ denotes the size of $UQ_{colNames}$. UQ_{array} denotes a two dimensional array that stores the retrieved rows.

Algorithm 6-6: UnionQuery (UQ)

Input: T, UQ_{tables} , $UQ_{colNames}$,W, F, and M
Output: UQ_{array}
1. **for** i ← 0 **to** $UQ_{tablesSize}$ **do**
2. **if** $UQ_{table\,i}$ ∈ CTTs **then**
3. UQ_{row} ← retrieve rows from $UQ_{table\,i}$ by using $UQ_{colNames\,i}$, W_i , F, and M query filters
4. **else if** $UQ_{table\,i}$ ∈ VETs **then**
5. UQ_{row}← **GetTableRowQuery**(T, $UQ_{table\,i}$, Nil, Nil, $UQ_{colNames\,i}$, W_i , F, M x $UQ_{colNamesSize}$) /* Algorithm 6-2 */

6. **end if**
7. **for** $n \leftarrow 0$ to $UQ_{rowSize}$ **do**
8. **for** $m \leftarrow 0$ **to** $UQ_{rowSize}$ **do**
9. $UQ_{array}[n+1][m] \leftarrow UQ_{row\,n}(UQ_{col\,m})$
10. $m \leftarrow m+1$
11. **end for**
12. $n \leftarrow n+1$
13. **end for**
14. $i \leftarrow i+1$
15. **end for**
16. **Return** UQ_{array}

6.2.4 JOIN QUERY ALGORITHM

This algorithm retrieves a combined table rows from two CTTs, two VETs, or a VET and a CTT based on a common field between them using different types of joins including Left Join, Right Join, Inner Join, Outer Join, Left Excluding Join, Right Excluding Join, and Outer Excluding Join. In this section, a sample algorithm of the Left Join is outlined in Definition 6-7 and Algorithm 6-7.

Definition 6-7 (Left Join Query Algorithm): T denotes a tenant ID. $LJQ_{leftTable}$ denotes a left table of the left join operation. $LJQ_{rightTable}$ denotes a right table of the left join operation. $S_{leftTable}$ denotes a string of the SELECT clause for the left table. $S_{rightTable}$ denotes a string of the SELECT clause for the right table. $W_{leftTable}$ denotes a string of the WHERE clause for the left table. $W_{rightTable}$ denotes a string of the WHERE clause for the right table. F denotes a first result number of the query limit. M denotes the maximum amount number of the query limit. LJQ_{leftPK} denotes a set of primary keys of the left table. $LJQ_{rightPK}$ denotes a set of primary keys of the right table. $LJQ_{rightFK}$ denotes a set of foreign keys of the right table refrencing the primary keys of the left table. $LJQ_{twoVETs}$ denotes a row matrix with n rows and 2 columns. The first column stores a $LJQ_{twoVETs\,n,0}$ that denotes a 'table_row_id' of the $LJQ_{leftTable}$. The second column stores a $LJQ_{twoVETs\,n,1}$ that denotes a 'table_row_id' of the

$LJQ_{rightTable.}$ $LJQ_{CTTandVET}$ denotes a row matrix with n rows and 2 columns. The first column stores a $LJQ_{CTTandVET\,n,0}$ that denotes the primary key of the $LJQ_{leftTable}$ which is a CTT. The second column stores a $LJQ_{CTTandVET\,n,1}$ that denotes a 'table_row_id' of the $LJQ_{rightTable}$ which is a VET. $LJQ_{leftRows}$ denotes a set of rows of the left table. $LJQ_{rightRows}$ denotes a set of rows of the right table. LJQ_{set} denotes a set which consist of two elements. The first element is the $LJQ_{rightRows}$, and the second element is the $LJQ_{leftRows}$. LJQ_{rows} denotes a set of rows for a left CTT and right CTT. LJQ_{array} denotes a two dimensional array that stores the retrieved rows.

Algorithm 6-7: LeftJoinQuery (LJQ)

Input: T, $LJQ_{leftTable}$, $LJQ_{rightTable}$, $S_{leftTable}$, $S_{rightTable}$, $W_{leftTable}$, $W_{rightTable}$, F, and M
Output: LJQ_{array}
/* Left join for two CTTs */
1. **if** $LJQ_{leftTable}$ is CTT $\wedge$ $LJQ_{rightTable}$ is CTT **then**
2. $LJQ_{rightPK}$ ← get the primary keys of the CTT $LJQ_{rightTable}$ from the INFORMATION_SCHEMAviews
3. LJQ_{leftPK} ← get the primary keys of the CTT $LJQ_{leftTable}$ from the INFORMATION_SCHEMA views
4. LJQ_{rows} ← retrieve rows from $LJQ_{leftTable}$ and $LJQ_{rightTable}$ from RDBMS using the left join operator and using LJQ_{leftPK} , $LJQ_{rightPK}$, $S_{leftTable}$, $S_{rightTable}$, $W_{leftTable}$, $W_{rightTable}$, F, and M query filters
5. LJQ_{array} ← store LJQ_{rows} in two dimensional array
/* Left join for two VETs */
6. **else if** $LJQ_{leftTable}$ is VET $\wedge$ $LJQ_{rightTable}$ is VET **then**
7. LJQ_{leftPK} ← get the primary keys of the $LJQ_{leftTable}$ from table_column ET
8. $LJQ_{rightFK}$ ← get the foreign keys of the $LJQ_{rightTable}$ from table_relationship ET
9. $LJQ_{twoVETs}$ ← retrieve rows from the table_index ET using a join operator, and using LJQ_{leftPK} and $LJQ_{rightFK}$ query filters
10. $LJQ_{leftRows}$ ← **GetTableRowQuery**(T, $LJQ_{lefttTable}$, $\boldsymbol{LJQ_{twoVETs\,(1,\ 2,\ \ldots\ ,n),0}}$, Nil, $S_{leftTable}$, $W_{leftTable}$, F, M) /* Algorithm 6-2*/

11. $LJQ_{rightRows}$ ← **GetTableRowQuery**(T, $LJQ_{rightTable}$, $\boldsymbol{LJQ_{twoVETs(1,\ 2,\ \ldots\ ,n),1}}$, Nil, $S_{rightTable}$, $W_{rightTable}$, F, M)
/* Algorithm 6-2*/
12. LJQ_{set} ← Concatenate($LJQ_{leftRows}$, $LJQ_{rightRows}$)
13. LJQ_{array} ← store LJQ_{set} in two dimensional array

/* Left join for a CTT and a VET */

14. **else if** ($LJQ_{leftTable}$ is CTT ∧ $LJQ_{rightTable}$ is VET) **then**
15. LJQ_{leftPK} ← get the primary keys of the $LJQ_{leftTable}$ from the INFORMATION_SCHEMAviews
16. $LJQ_{rightFK}$ ← get the foreign keys of the $LJQ_{rightTable}$ from table_relationship ET
17. $LJQ_{CTTandVET}$ ← retrieve rows for $LJQ_{leftTable}$ and $LJQ_{rightTable}$ from the CTT $LJQ_{leftTable}$ and the table_index ET using a join operator and using LJQ_{leftPK}, $LJQ_{rightFK}$ query filters
18. $LJQ_{leftRows}$ ← retrieve rows from the CTT $LJQ_{leftTable}$ using $S_{leftTable}$, $W_{leftTable}$, F, and M query filters
19. $LJQ_{rightRows}$ ← **GetTableRowQuery**(T, $LJQ_{rightTable}$, $\boldsymbol{LJQ_{CTTandVET(1,\ 2,\ \ldots\ ,n),1}}$, Nil, $S_{rightTable}$, $W_{rightTable}$, F, M)
/* Algorithm 6-2*/
20. LJQ_{set} ← Concatenate($LJQ_{leftRows}$, $LJQ_{rightRows}$)
21. LJQ_{array} ← store LJQ_{set} in two dimensional array

/* Left join for a VET and a CTT */

22. **else if** ($LJQ_{leftTable}$ is VET ∧ $LJQ_{rightTable}$ is CTT) **then**
23. **[…]** /* Symmetric to lines 15 to 21, but the difference is that the left table is a VET and the right table is a CTT */
24. **end if**
25. **Return** LJQ_{array}

6.2.5 TARGETED TABLES QUERY ALGORITHM

This algorithm combines the result-set of two or more tables, whether they are CTTs, VETs or a combination of both types of tables. It uses query filters on multiple tables or filtering data based on the results of subqueries. This complex queries can be executed by calling this function that executes this algorithm. Figure 6-2 shows an example of a set of tables that have relationships between them. Table A (Root Table) and table C have a Many-to-Many database relationship, while table B is a join table that construct this relationship. Table C and table D have a One-to-Many relationship.

Finally, Table D and table E (Targeted Table) have a One-to-Many relationship. This algorithm filters the data in the Targeted Table E based on a number of query results that obtained from table A to table D. The details of this algorithm are outlined in Definition 6-8 and Algorithm 6-8.

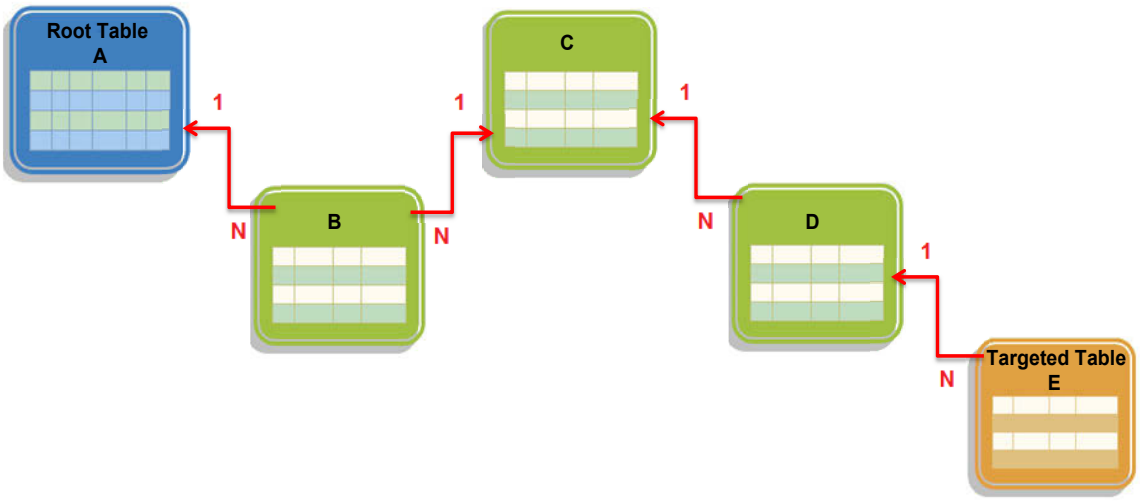


Figure 6-2: Targeted Tables example

Definition 6-8 (Targeted Tables Query Algorithm): T denotes a tenant ID. TTQ_{tables} denotes a set of CTTs and/or VETs names. $TTQ_{tablesSize}$ denotes the size of TTQ_{tables}. TTQ_{PK} denotes a set of row matrix with 2 rows and n columns. The first row stores the column name of a table primary key. The second row stores the value of a table primary key. TTQ_{select} denotes a set of SELECT clauses, where each table in TTQ_{tables} may have a SELECT clause. TTQ_{where} denotes a set of WHERE clauses, where each table in TTQ_{tables} may have a SELECT clause. F denotes the first result number of a query limit. M denotes the maximum amount number of a query limit. TTQ_{type} denotes a table type of a table in TTQ_{tables}. $TTQ_{tables\,i}$ denotes a current root table. $TTQ_{tables\ i+1}$ denotes a current targeted table. Figure 6-3 shows an example of Current Root Table and Current Targeted Table that this algorithm may reaches during iterating the targeted table sequence list. $TTQ_{relation}$ denotes a relationship between two tables. $TTQ_{PK\,i}$ denotes a primary key set for a current root table in the TTQ_{tables}. This set has only primary key IDs without values

that can be obtained while iterating the loop of the algorithm for the current root table. TTQ_{array} denotes a two dimensional array that stores the retrieved rows.

Algorithm 6-8: TargetedTablesQuery (TTQ)

Input: T, TTQ_{tables}, TTQ_{PK}, TTQ_{select}, TTQ_{where}, F, and M
Output: TTQ_{array}

1. **for** $i \leftarrow 0$ **to** $TTQ_{tablesSize}$ **do**
2. **if** $TTQ_{tablesSize} = (i + 2)$ **then** /* $\boldsymbol{TTQ_{tables}}$ **has 2 table** */
3. $TTQ_{relation} \leftarrow$ get the relationship between $TTQ_{tables\ i}$ and $TTQ_{tables\ i+1}$
4. **if** $TTQ_{relation}$ is One-to-Many **then**
5. $TTQ_{array} \leftarrow$ **OneToManyQuery**(T, $TTQ_{tables\ i}$, $TTQ_{tables\ i+1}$, $TTQ_{PK\ i}$, $TTQ_{select\ i}$, $TTQ_{where\ i}$, F, M, TTQ_{type}) /* Algorithm 6-5 */
6. **else if** $TTQ_{relation}$ is Many-to-One **then**
7. $TTQ_{array} \leftarrow$ **OneToManyQuery**(T, $TTQ_{tables\ i+1}$, $TTQ_{tables\ i}$, $TTQ_{PK\ i}$, $TTQ_{select\ i}$, $TTQ_{where\ i}$, F, M, TTQ_{type}) /* Algorithm 6-5 */
8. **end if**
9. **else if** $TTQ_{tablesSize} >= (i + 3)$ **then** /* $\boldsymbol{TTQ_{tables}}$ **has 3 or more tables** */
10. $TTQ_{PK\ i} \leftarrow$ **OneToManyPKQuery(…)** /* This algorithm is similar to Algorithm 6-5, but this algorithm returns only primary key IDs and does not store results in an array */
11. **end if**
12. $i \leftarrow i + 1$
13. **end for**
14. **Return** TTQ_{array}

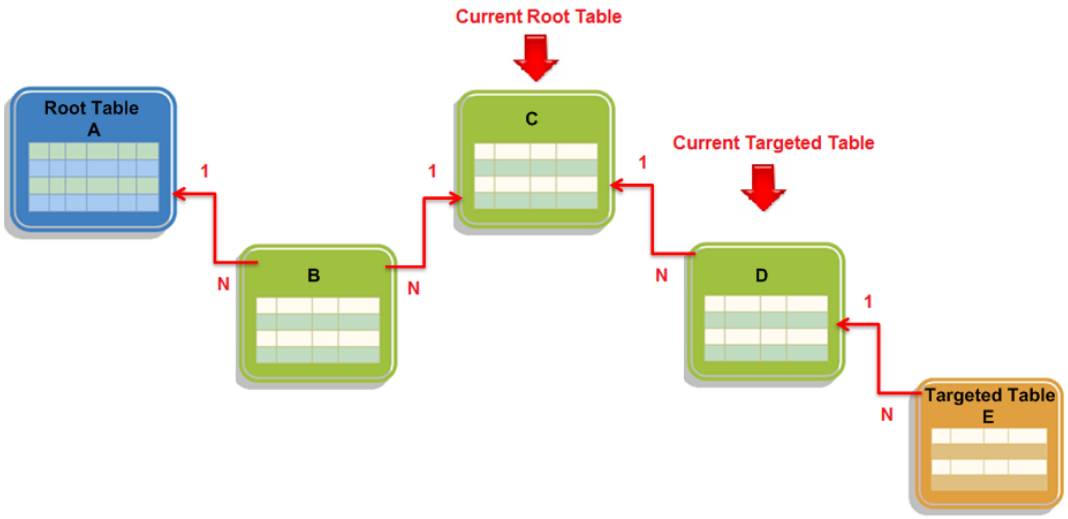


Figure 6-3: Current Root Table and Current Targeted Table

6.3 PERFORMANCE EVALUATION

The potentials of using EET multi-tenant database schema have discussed in Chapter 4, and several experiments were performed in the same chapter to measure the feasibility and effectiveness of EET by comparing it with UTSM, which is one of the multi-tenant database schema techniques that implemented commercially by Salesforce. Significant performance improvements obtained using EET when compared to UTSM, making the EET schema a good candidate for the management of multi-tenant data in SaaS and Big Data applications. Whereas in this chapter, five types of experiments are carried out to verify the practicability of EETPS. These experiments classified according to the complexity of the queries used in these experiments, including simple, simple-to-medium, medium, medium-to-complex, and complex. These five experiments show comparisons between the response time of retrieving data from CTTs, VETs, or both CTTs and VETs. The response time of retrieving data from EET is evaluated by accessing the EETPS functions.

6.3.1 EXPERIMENTAL SETUP

The 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, the database and the application server are deployed on the same PC. The operating system is Windows 7 Home Premium, with Intel Core i5 2.40GHz CPU, 8 GB of RAM memory, and 500 GB of hard disk storage.

6.3.2 EXPERIMENTAL DATA SET AND RESULTS

The EETPS designed and developed to serve multiple tenants in one instance application. However, in this chapter the aim of the experiments is to evaluate the performance and show the differences between retrieving data of CTTs, VETs, or both CTTs and VETs together for one tenant. As long as in the multi-tenant database, the data of each tenant is isolated in a table partition, these experiments can evaluate the

effectiveness of retrieving each single tenant's data from the EET multi-tenant database. Moreover, the experiments are performed in one single server instance, because before anyone starts thinking to scale-up or scale-out multi-tenant database to optimize its performance, the performance of the EETPS should be tested in a single instance application. In the five experiments, the test is performed on fourteen queries twice, the first test to retrieve only 1 row, and the second test to retrieve 100 rows by using the same queries. In order to have accurate comparisons, the same data input is used for CTTs, VETs and CTT-and-VET to retrieve the same data output. The execution times of these query experiments are recorded based on six data sets for all the five types of experiments. These six data sets contain, (1) 500 rows, (2) 5,000 rows, (3) 10,000 rows, (4) 50,000 rows, (5) 100,000 rows, and (6) 200,000 rows. In this section, the average execution time is recorded by executing ten tests on each of the six data sets to show accurate results. All of these data sets were for one tenant. In all the experimental diagrams, the vertical axis shows the execution time in milliseconds, and the horizontal axis shows the total number of rows that stored in a tenant's table. In the five experiments, the 'tenant_id' equals 1000. The CTTs that are used in the experiments, including 'product', 'sales_fact' and 'sales_details', and the corresponding 'db_table_id' of VETs for these tables are 16, 17, and 18 respectively. The data structure of the queries that is used in the experiments are shown in Figure 6-22, and listed below:

1) **Simple Query Experiment (Exp. 6-1):** In this experiment, the function of the Single Table Query Algorithm that retrieves data from a CTT is invoked by executing Query 6-1 (Q 6-1) that comprises of the Individual Query (IQ) 1, and retrieve the same data from a VET by executing Query 6-2 (Q 6-2) that comprises of IQ2 – IQ4. These experimental tests show how the Single Table Algorithm retrieves physical rows from a CTT and virtual rows from a VET. The three cases that this algorithm is handling are described in Section 6.2. This experiment studies the third case that retrieves all rows of the 'product' CTT and the 'product' VET from the Single Table function without specifying any primary keys or row IDs. The structure of the

'product' table is shown in Figure 6-4 (a). The 'db_table_id' of the 'product' VET equals 16. The experimental results of Exp. 6-1 shows that the performance of the query execution time of a VET is faster than a CTT when 1 or 100 of table rows are retrieved. The details of the queries are used in this experiment and shown in Table 6-13 – 6-14, the output of these queries is shown in Figure 6-5, and the throughputs of this experiment are depicted in Figure 6-6 – 6-7 and Table 6-1 – 6-2.

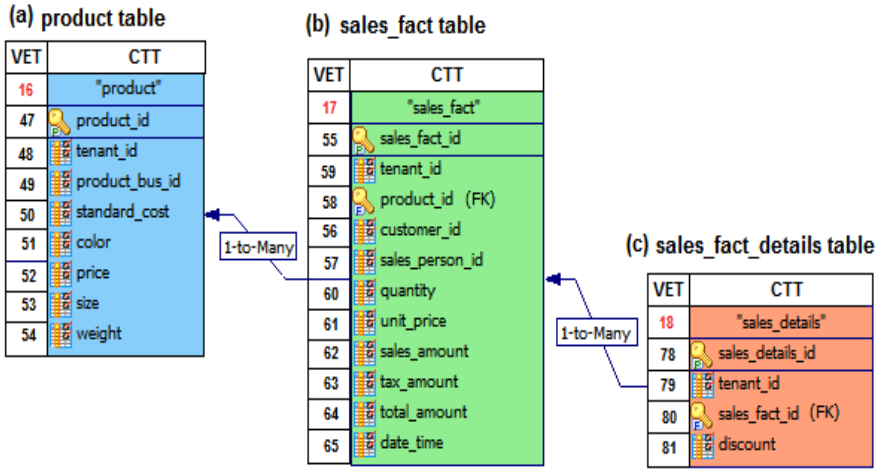


Figure 6-4: The tables structures used in the experiments

Q6-1 Ouptut

IQ 1

product_id	tenant_id	product_bus_id	standard_cost	color	price	size	weight
100	1000	60699	228.26	Green	5714.80	670	5.79

Q6-2 Ouptut

IQ 2

table_column_id
47

IQ 3

table_row_id
1100

IQ 4

table_column_id	value	table_row_id	serial_id
47	100	1100	1
48	1000	1100	2
49	60699	1100	3
50	228.26	1100	4
51	Green	1100	5
52	5714.80	1100	6
53	670	1100	7
54	5.79	1100	8

The Two Dimensional Array Output

product_id 0,0	tenant_id 0,1	product_bus_id 0,2	standard_cost 0,3	color 0,4	price 0,5	size 0,6	weight 0,7	} columns' names
100 1,0	1000 1,1	60699 1,2	228.26 1,3	Green 1,4	5714.80 1,5	670 1,6	5.79 1,7	} columns' values

Figure 6-5: The outputs of the Simple Query Experiment (Single Table)

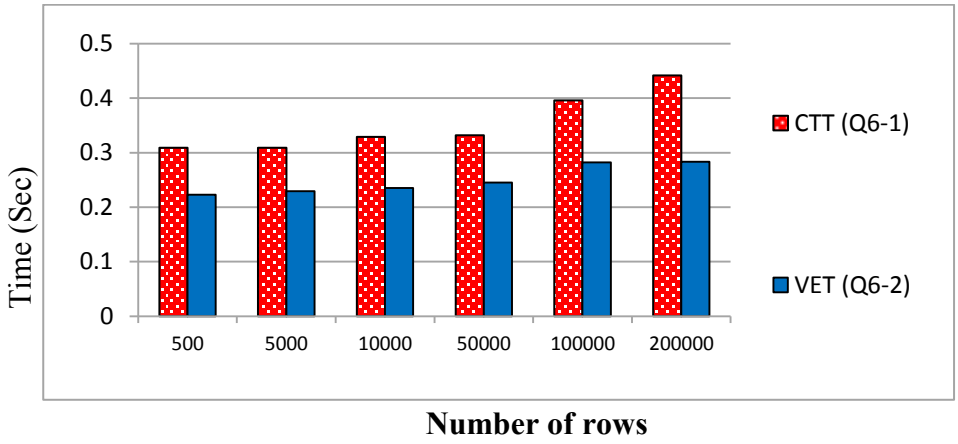


Figure 6-6: The experimental results of retrieving 1 row from the Single Table function

Table 6-1: The query execution times of retrieving 1 row from the Single Table experiment (Exp. 6-1)

Number of populated rows	CTT (Q 6-1) Time in seconds	VET (Q 6-2) Time in seconds
500	0.309	0.223
5000	0.309	0.229
10000	0.329	0.235
50000	0.332	0.245
100000	0.396	0.282
200000	0.442	0.283

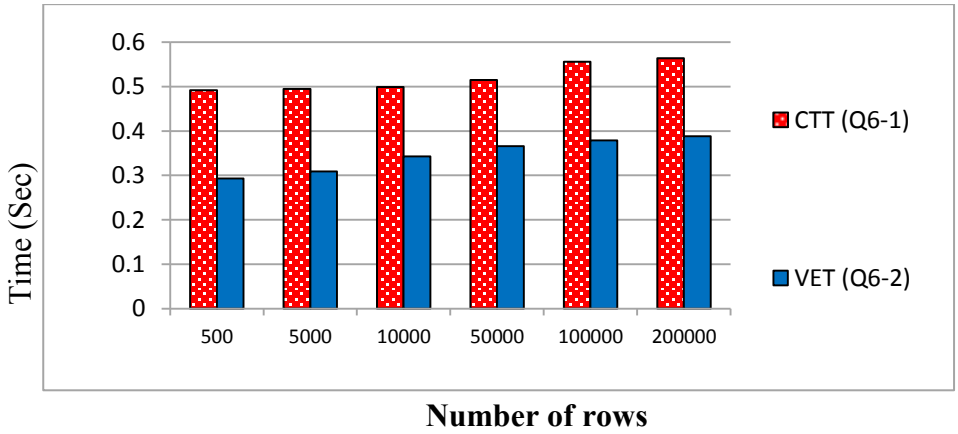


Figure 6-7: The experimental results of retrieving 100 rows from the Single Table function

Table 6-2: The query execution times of retrieving 100 rows from the Single Table experiment (Exp. 6-1)

Number of populated rows	CTT (Q 6-1) Time in seconds	VET (Q 6-2) Time in seconds
500	0.492	0.293
5000	0.495	0.309
10000	0.499	0.343
50000	0.515	0.366
100000	0.556	0.379
200000	0.564	0.388

2) **Simple-to-Medium Query Experiment (Exp. 6-2):** In this experiment, the function of One-to-Many Query Algorithm is invoked to retrieve data from two CTTs by executing Query 6-3 (Q 6-3) that comprises of IQ5, two VETs by executing Query 6-4 (Q 6-4) that comprises of IQ6 and IQ7, and CTT-and-VET by executing Query 6-5 (Q 6-5) that comprises of IQ8 and IQ7 respectively. The focus of this experiment is to study each of the two table combinations that have a One-to-Many relationship between them. The master table of this relation is the 'product' table, and the details table is the 'sales_fact' table. The structure of these two tables is shown in Figure 6-4 (a) and Figure 6-4 (b). The value of the 'product_id' column that is used in this experiment equals 100 for both the 'product' CTT and VET. The 'product_id' of the VET is represented as the number 58. The experimental results of Exp. 6-2 show that there is an approximate symmetry in the performance of the query execution time of VET and CTT-and-VET and they are one time faster than CTT when 1 row is retrieved. On the other hand, when 100 rows are retrieved, the query execution time of CTT is faster than VET, and CTT-and-VET is the fastest of the three queries. Moreover, the experimental results show that the execution time of CTT is approximately the same when 1 row and 100 rows are retrieved, whereas it increases for VET and CTT-and-VET when 100 rows are retrieved. The details of the queries used in this experiment are shown in Table 6-13 – 6-14, the output of these queries is

shown in Figure 6-8, and the throughputs of this experiment are depicted in Figure 6-9 – 6-10 and Table 6-3 – 6-4.

Q6-3 Output

IQ 5

sales_fact_id	tenant_id	product_id	customer_id	sales_person_id	quantity	unit_price	sales_amount	tax_amount	total_amount	date_time
1	1000	100	1	1	1415	98192	10	122	98314	2010-02-19 00:00:00

Q6-4 Output

IQ 6

table_column_id
58

IQ 7

table_column_id	value	table_row_id	serial_id
55	1	200001	1
56	1	200001	2
57	1	200001	3
58	100	200001	4
59	1000	200001	5
60	1415	200001	6
61	98192	200001	7
62	10	200001	8
63	122	200001	9
64	98314	200001	10
65	2010-02-19 00:00:00	200001	11

Q6-5 Output

IQ 8

table_column_id
58

IQ 7 as listed above.

The Two Dimensional Array Output

sales_fact_id	tenant_id	product_id	customer_id	sales_person_id	quantity	unit_price	sales_amount	tax_amount	total_amount	date_time	
0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	0,10	} columns' names
1	1000	100	1	1	1415	98192	10	122	98314	2010-02-19 00:00:00	} columns' values
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	

Figure 6-8: The outputs of the Simple-to-Medium Query Experiment (One-to-Many)

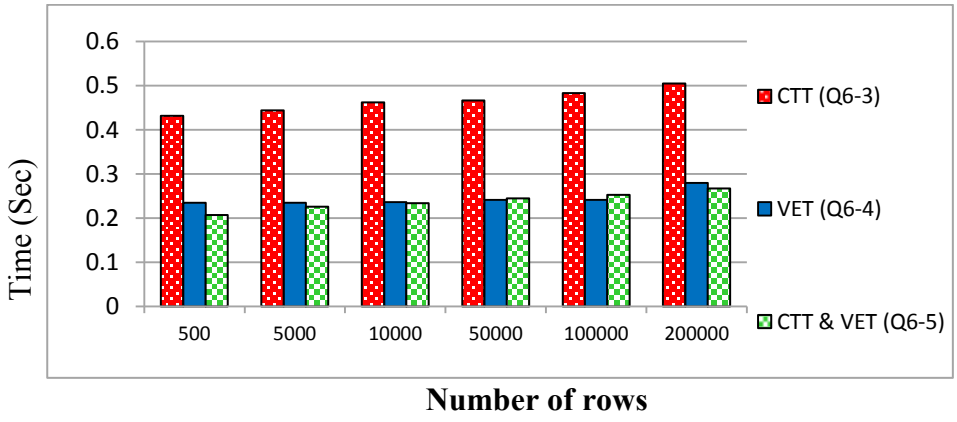


Figure 6-9: The experimental results of retrieving 1 row from the One-to-Many function

Table 6-3: The query execution times of retrieving 1 row from the One-to-Many experiment (Exp. 6-2)

Number of populated rows	CTT (Q 6-3) Time in seconds	VET (Q 6-4) Time in seconds	CTT & VET (Q 6-5) Time in seconds
500	0.432	0.235	0.207
5000	0.444	0.235	0.226
10000	0.462	0.236	0.234
50000	0.467	0.241	0.245
100000	0.483	0.241	0.253
200000	0.505	0.280	0.267

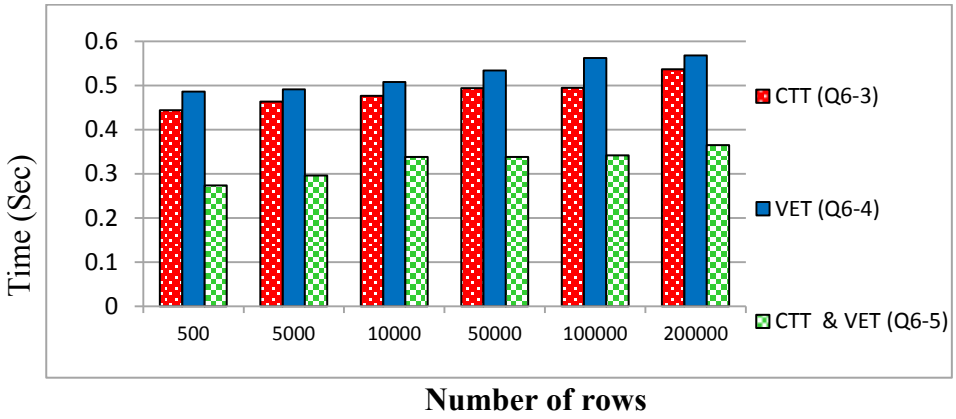


Figure 6-10: The experimental results of retrieving 100 rows from the One-to-Many function

Table 6-4: The query execution times of retrieving 100 rows from the One-to-Many experiment (Exp. 6-2)

Number of populated rows	CTT (Q 6-3) Time in seconds	VET (Q 6-4) Time in seconds	CTT & VET (Q 6-5) Time in seconds
500	0.444	0.486	0.274
5000	0.464	0.491	0.296
10000	0.477	0.508	0.337
50000	0.495	0.534	0.338
100000	0.494	0.562	0.342
200000	0.537	0.568	0.365

3) **Medium Query Experiment (Exp. 6-3):** In this experiment, the function of the Union Query Algorithm that retrieves data from two tables is invoked by using a

union operator for two CTTs by executing Query 6-6 (Q 6-6) that comprises of IQ9 and IQ10, for two VETs by executing Query 6-7 (Q 6-7) that comprises of IQ2, IQ3, IQ11, IQ12, IQ13, and IQ14, and for CTT-and-VET by executing Query 6-8 (Q 6-8) that comprises of IQ9, IQ12, IQ13, and IQ14 respectively. The aim of using this algorithm is to study, retrieving data from two tables. The first table is the 'product' table, and the second table is the 'sales_fact' table. The structures of these two tables are shown in Figure 6-4 (a) and 6-4 (b). On Q 6-6, in the SELECT clause two physical columns 'product_id' and 'price' are specified for the 'product' CTT, and two physical columns 'sales_fact_id' and 'unit_price' are specified for the 'sales_fact' CTT. On Q 6-7, in the SELECT clause two virtual columns are specified. The first column ID is 47, and the second column ID is 52 for the 'product' VET that equals 16. The column ID 47 corresponds to the 'product_id' column, and the column ID 52 corresponds to the 'price' column of the 'product' VET. In addition, in the SELECT clause two virtual columns are specified for the 'sales_fact' VET that equals 17. The first column is 55, and the second column is 61. The column ID 55 corresponds to the 'sales_fact_id' and the column ID 61 corresponds to the 'unit_price' of the 'sales_fact' VET. Finally, in Q 6-8, in the SELECT clause two physical columns 'product_id' and 'price' are specified for the 'product' CTT, and two virtual columns for the 'sales_fact' VET. The first column is 55, and the second column is 61. The experimental results of Exp. 6-3 shows that the query execution time of VET is faster than CTT, and CTT-and-VET is the fastest of the three queries when 1 and 100 rows are retrieved. Moreover, this experiment shows that the query execution times of the three types CTT, VET, and CTT-and-VET are approximately the same when 1 row and 100 rows are retrieved. The details of the queries used in this experiment are shown in Table 6-13 – 6-14, the output of these queries is shown in Figure 6-11, and the throughputs of this experiment are depicted in Figure 6-12 – 6-13 and Table 6-5 – 6-6.

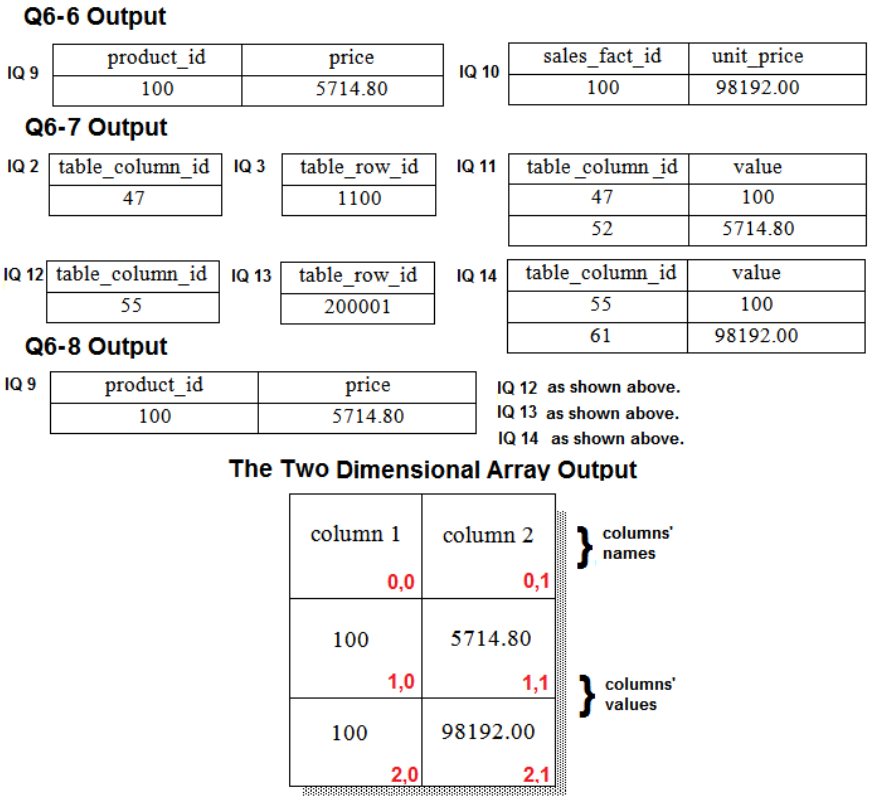


Figure 6-11: The outputs of the Medium Query Experiment (Union)

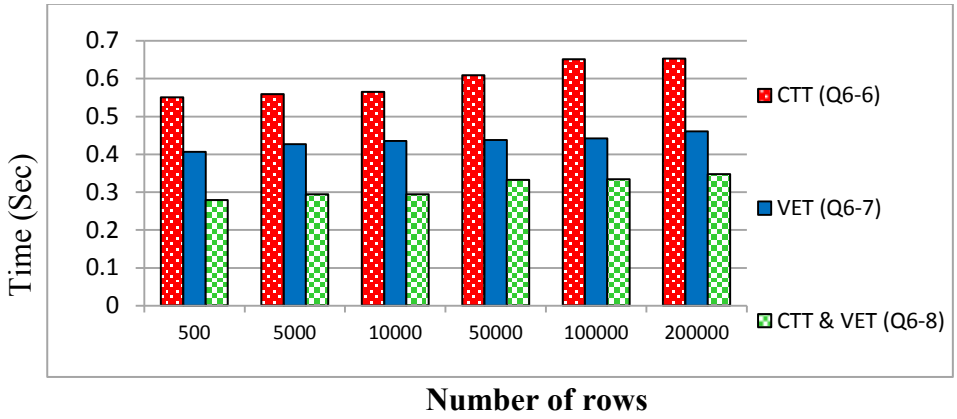


Figure 6-12: The experimental results of retrieving 1 row from the Union function

Table 6-5: The query execution times of retrieving 1 row from the Union Experiment (Exp. 6-3)

Number of populated rows	CTT (Q 6-6) Time in seconds	VET (Q 6-7) Time in seconds	CTT & VET (Q 6-8) Time in seconds
500	0.550	0.407	0.279
5000	0.559	0.427	0.294
10000	0.565	0.435	0.294
50000	0.609	0.438	0.332
100000	0.651	0.442	0.334
200000	0.653	0.461	0.347

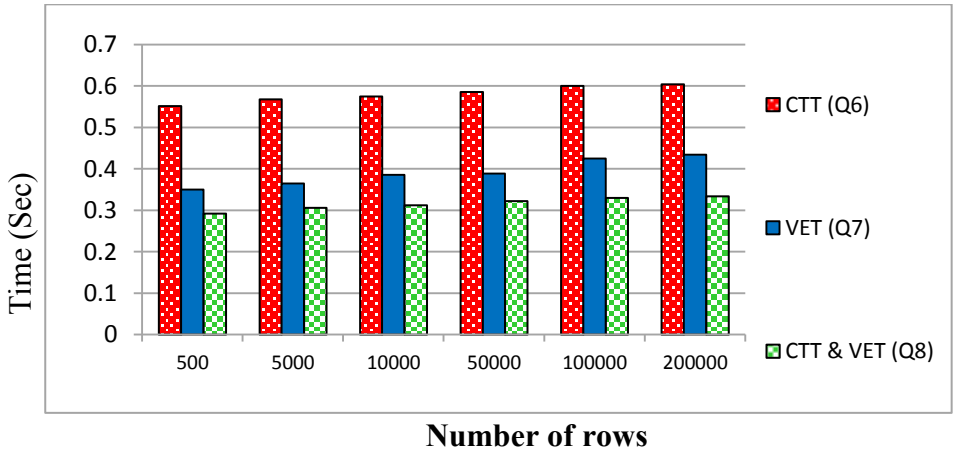


Figure 6-13: The experimental results of retrieving 100 rows from the Union function

Table 6-6: The query execution times of retrieving 100 rows from the Union experiment (Exp. 6-3)

Number of populated rows	CTT (Q 6-6) Time in seconds	VET (Q 6-7) Time in seconds	CTT & VET (Q 6-8) Time in seconds
500	0.552	0.350	0.292
5000	0.568	0.365	0.306
10000	0.575	0.386	0.312
50000	0.586	0.389	0.322
100000	0.600	0.425	0.330
200000	0.604	0.434	0.334

4) **Medium-to-Complex Query Experiment (Exp. 6-4):** In this experiment, the function of the Left Join Query Algorithm is invoked using a left join between three types of table combinations. First, two CTTs by executing Query 6-9 (Q 6-9) that comprises of IQ15 – IQ17. The two CTTs are the 'product' CTT and the 'sales_fact' CTT. Second, two VETs by executing Query 6-10 (Q 6-10) that comprises of IQ2, IQ18, IQ19, IQ4, and IQ21. The two VETs are the 'product VET that equals 16, and the 'sales_fact' VET that equals 17. Third, a CTT and a VET by executing Query 6-11 (Q 6-11) that comprises of IQ15, IQ8, IQ20, IQ1, and IQ21 respectively. These two tables are the 'product' CTT, and the 'sales_fact' VET that equals 17. Figure 6-14 shows the three Left Join operations that are used in this experiment. The experimental results of Exp. 6-4 shows that the query execution time of CTT-and-VET is faster than CTT, and the VET is the fastest of the three queries when 1 row and 100 rows are retrieved. The details of the queries used in this experiment are shown in Table 6-13 – 6-14, the output of these queries is shown in Figure 6-15, and the throughputs of this experiment are depicted in Figure 6-16 – 6-17 and Table 6-7 – 6-8.

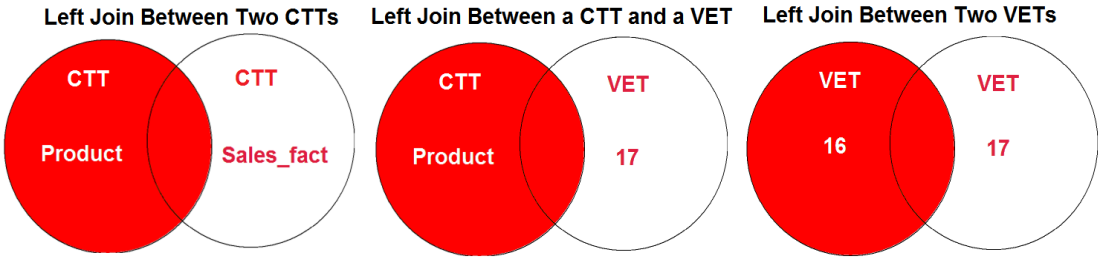


Figure 6-14: The three left joins of The Left Join experiment

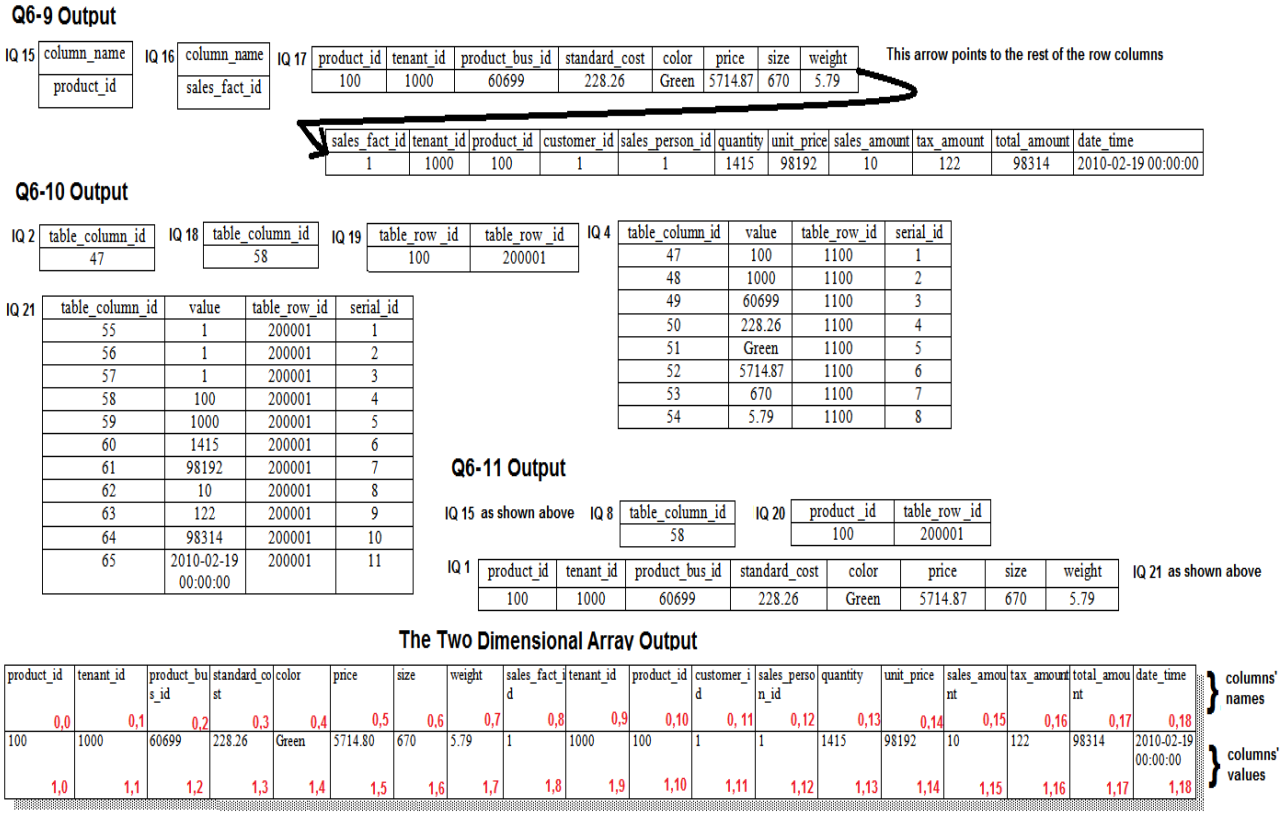


Figure 6-15: The output of the Medium-to-Complex Query Experiment (Left Join)

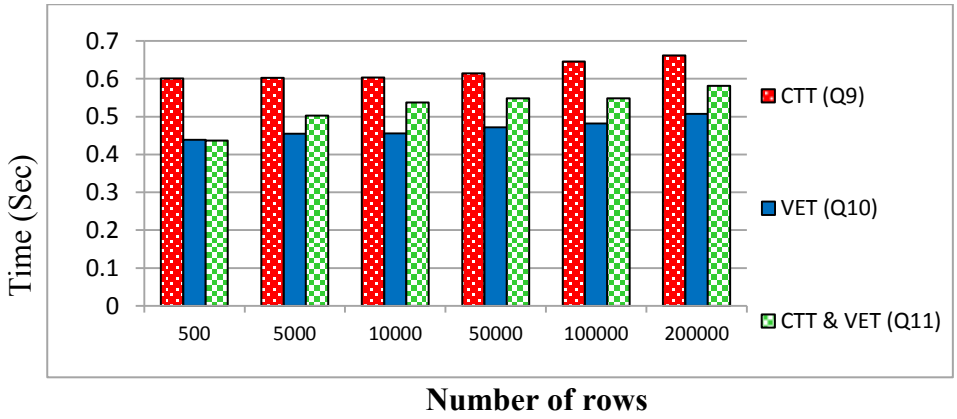


Figure 6-16: The experimental results of retrieving 1 row from the Left Join function

Table 6-7: The query execution times of retrieving 1 row from the Left Join experiment (Exp. 6-4)

Number of populated rows	CTT (Q 6-9) Time in seconds	VET (Q 6-10) Time in seconds	CTT & VET (Q 6-11) Time in seconds
500	0.601	0.439	0.437
5000	0.603	0.455	0.503
10000	0.604	0.456	0.538
50000	0.615	0.472	0.549
100000	0.646	0.482	0.549
200000	0.662	0.507	0.582

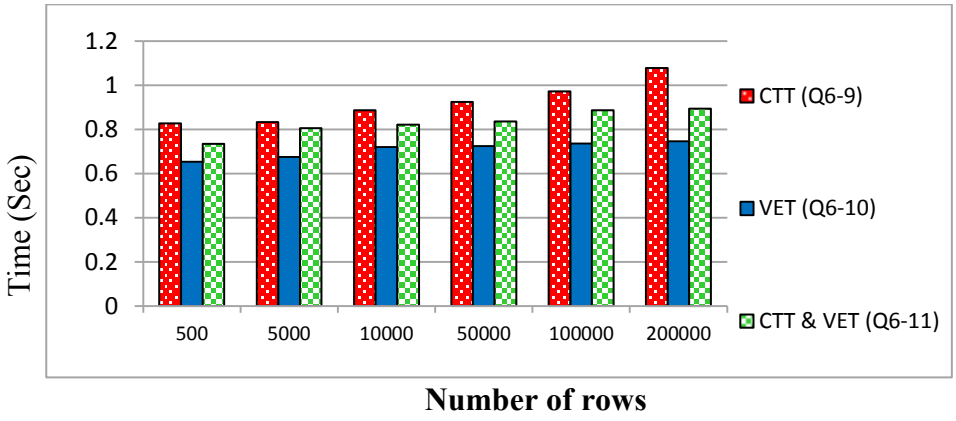


Figure 6-17: The experimental results of retrieving 100 rows from the Left Join 100 rows experimental results

Table 6-8: The query execution times of retrieving 100 rows from the Left Join experiment (Exp. 6-4)

Number of populated rows	CTT (Q 6-9) Time in seconds	VET (Q 6-10) Time in seconds	CTT & VET (Q 6-11) Time in seconds
500	0.827	0.653	0.734
5000	0.834	0.675	0.806
10000	0.887	0.720	0.821
50000	0.925	0.725	0.836
100000	0.973	0.736	0.887
200000	1.079	0.747	0.894

5) **Complex Query Experiment (Exp. 6-5):** In this experiment, the function of the Targeted Table Query Algorithm is invoked to join two CTTs by executing Query 6-12 (Q 6-12) that comprises of IQ22 and IQ23, two VETs by executing Query 6-13 (Q 6-13) that comprises of IQ24 – IQ27, and CTT-and-VET by Query 6-14 (Q 6-14) that comprises of IQ28, IQ22, IQ29, and IQ27 respectively. This experiment is used to study, retrieving data from three targeted tables. The first table is the 'product' table, the second table is the 'sales_fact' table, and the third table is 'sales_fact_details'. The structure of these three tables is shown in Figure 6-4 (a), 6-4 (b), and 6-4 (c). These tables have database relationships between them, and multiple query filters are used in each of these tables to filter data based on the results of subqueries, starting from the 'product' table (Root Table) until the 'sales_fact_details' table (Targeted Table). The 'product' table is filtered by retrieving only products with product IDs equal to 100. Then the 'sales_fact' table is filtered by retrieving the sales transactions that their product IDs match the sales IDs that are retrieved from the 'product' table, and the quantity values that are greater or equal than 9000. Finally, the 'sales_fact_details' table is filtered by retrieving the sales details that their sales IDs matches sales IDs retrieved from the 'sales_fact' table, and the sales discounts that are greater or equal 30%. Figure 6-18 shows how the queries are filtered from the three tables. On Q 6-12, the three CTTs are used as stated above. The Q 6-13 uses three VETs, including the 'product VET that equals 16, the 'sales_fact' VET that equals 17, and the 'sales_fact_details' VET that equals 18. The Q 6-14 uses two CTTs and one VET. The two CTTs are the 'product' CTT and the 'sales_fact' CTT, and the VET is the 'sales_fact_details' that equals 18. The experimental results of Exp. 6-5 shows that the query execution time of CTT is faster than VET, and CTT-and-VET is the fastest of the three queries that retrieve 1 row. On the other hand, 100 rows are retrieved, the query execution time of VET is faster than CTT, and CTT-and-VET is the fastest of the three queries. Most importantly, the query execution times when retrieve 1 row or 100 rows from a VET or a CTT-and-VET are

approximately the same, whereas the increase in retrieving 100 rows from CTT is approximately 70% on average higher than when 1 row is retrieved. The details of the queries used in this experiment are shown in Table 6-13 – 6-14, the output of these queries is shown in Figure 6-19, and the throughputs of this experiment are depicted in Figure 6-20 – 6-21 and Table 6-9 – 6-10.

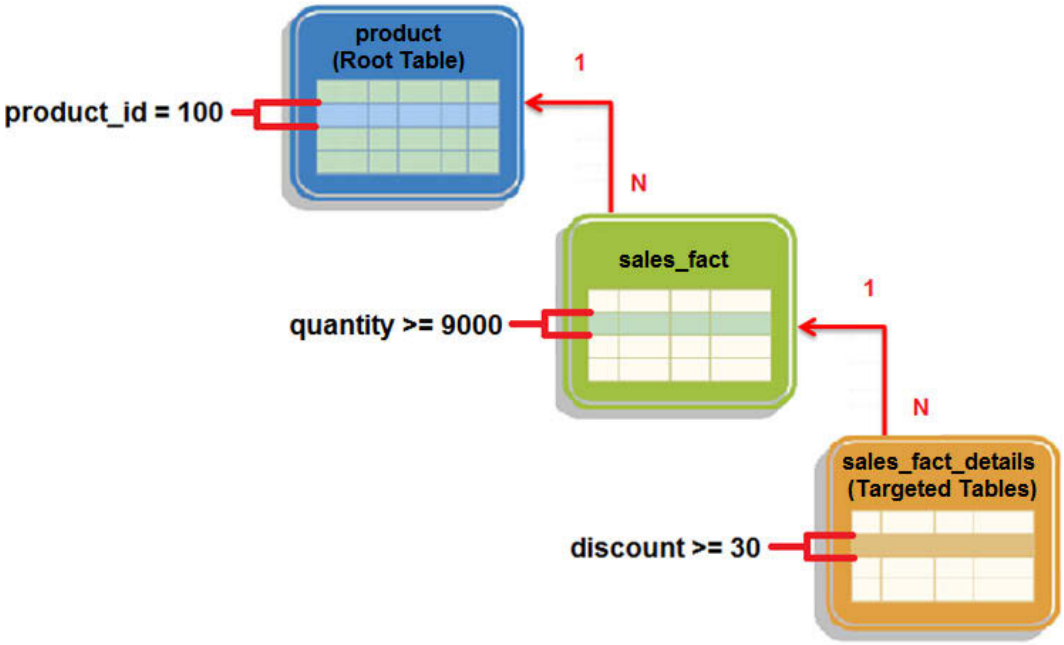


Figure 6-18: The query filters of the Targeted Tables experiment

Q6-12 Output

IQ 22

sales_fact_id
9
12
16
… and others

IQ 23

sales_details _id	tenant_id	sales_fact_id	discount
9	1000	9	60

Q6-13 Output

IQ 24

table_relationship_id	tenant_id	table_type	shared_table_name	db_table_id	table_column_id	target_table_id	shared_column_name	target_column_id
15	1000	2		17	58	16		47

IQ 25

table_column _id	value	table_row_id	serial_id
55	16	200872	1
55	18	200824	1
			… and others

IQ 27

table_column _id	value	table_row_id	serial_id
78	9	200680	1
79	1000	200680	2
80	9	200680	3
81	63.00	200680	4

IQ 26

table_relationship_id	tenant_id	table_type	shared_table_name	db_table_id	table_column_id	target_table_id	shared_column_name	target_column_id
15	1000	2		17	58	16		47
16	1000	1	product	17	58		Product_id	
17	1000	2		20	80	17		55

Q6-14 Output

IQ 28

table_relationship_id	tenant_id	table_type	shared_table_name	db_table_id	table_column_id	target_table_id	shared_column_name	target_column_id
16	1000	1	product	17	58		Product_id	

IQ 29

table_relationship_id	tenant_id	table_type	shared_table_name	db_table_id	table_column_id	target_table_id	shared_column_name	target_column_id
18	1000	1	sales_fact	20	80		Sales_fact_id	

IQ 22 as shown above IQ 27 as shown above

The Two Dimensional Array Output

sales_details_id 0,0	tenant_id 0,1	Sales_fact_id 0,2	discount 0,3	} columns' names
9 1,0	1000 1,1	9 1,2	60 1,3	} columns' values

Figure 6-19: The outputs of the Complex Query Experiment (Targeted Tables)

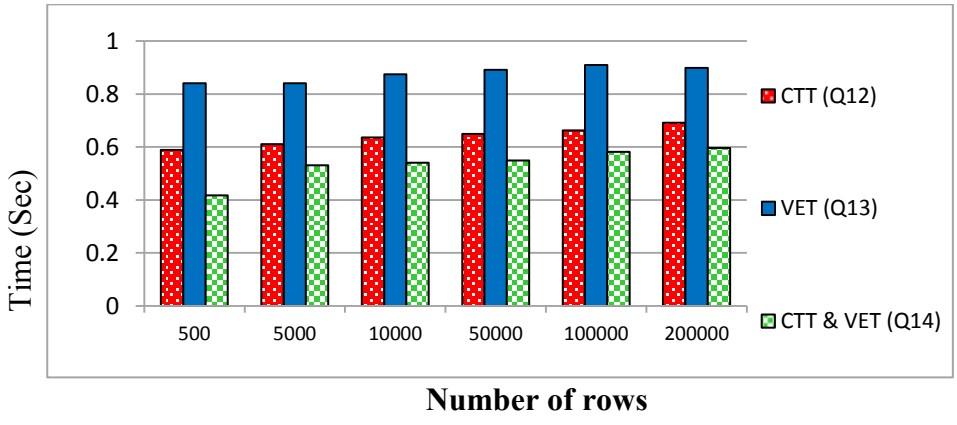


Figure 6-20: The experimental results of retrieving 1 row from the Targeted Tables function

Table 6-9: The query execution times of retrieving 1 row from the Targeted Tables experiment (Exp. 6-5)

Number of populated rows	CTT (Q 6-12) Time in seconds	VET (Q 6-13) Time in seconds	CTT & VET (Q 6-14) Time in seconds
500	0.589	0.841	0.418
5000	0.611	0.841	0.531
10000	0.636	0.875	0.541
50000	0.650	0.891	0.549
100000	0.663	0.910	0.582
200000	0.692	0.989	0.596

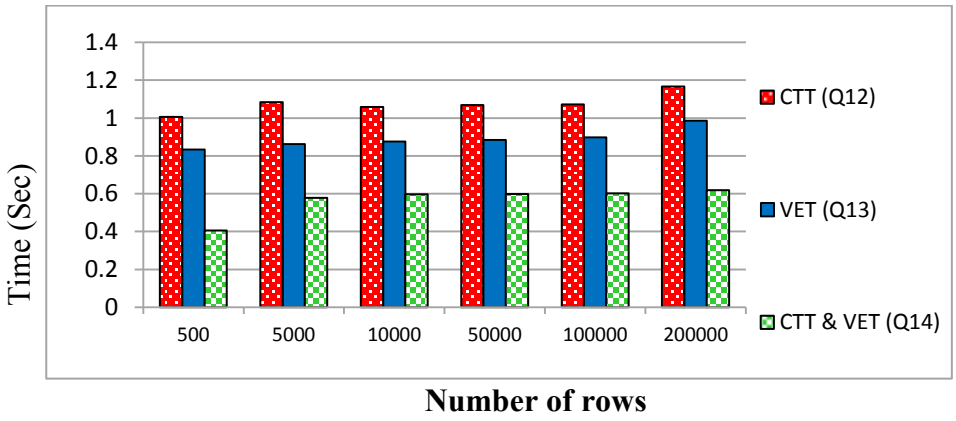


Figure 6-21: The experimental results of retrieving 100 rows from the Targeted Tables function

Table 6-10: The query execution times of retrieving 100 rows from the Targeted Tables experiment (Exp. 6-5)

Number of populated rows	CTT (Q 6-12) Time in seconds	VET (Q 6-13) Time in seconds	CTT & VET (Q 6-14) Time in seconds
500	1.006	0.833	0.406
5000	1.083	0.863	0.578
10000	1.059	0.875	0.596
50000	1.068	0.884	0.599
100000	1.072	0.897	0.601
200000	1.167	0.986	0.618

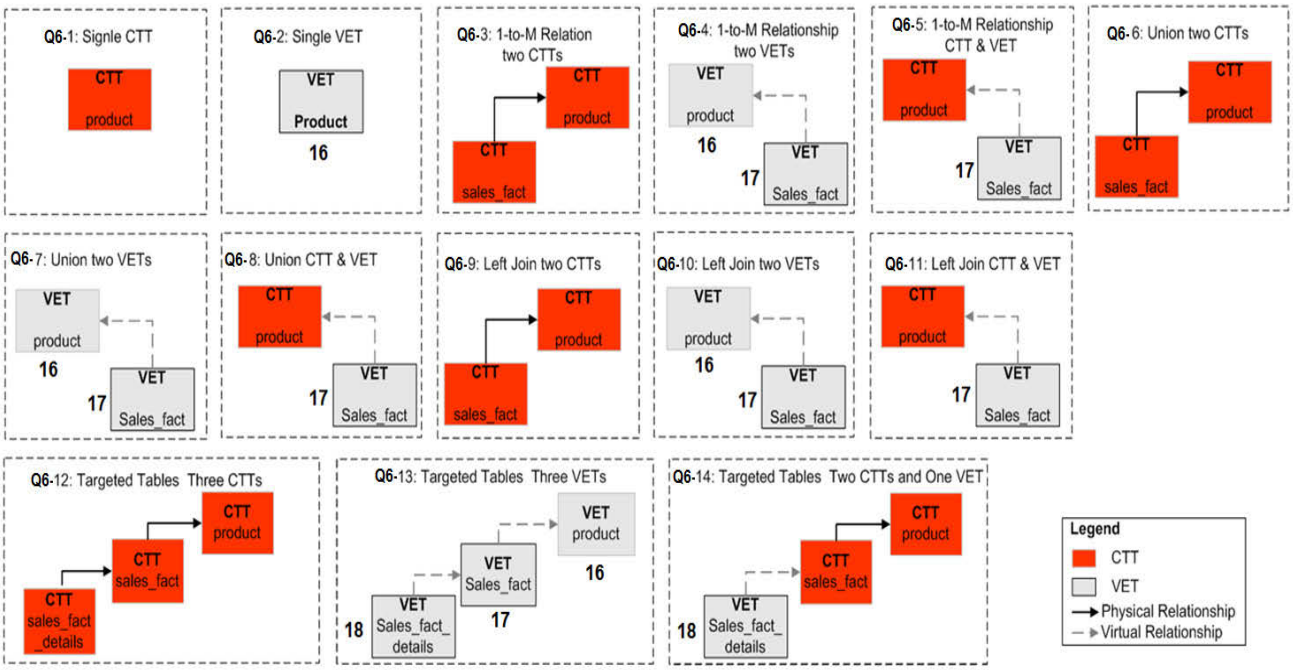


Figure 6-22: The structures of the queries used in the experiments

The five experiments that presented above are summarized in Figure 6-23 and Table 6-11, which show the average query execution time of the six data sets for each experiment when 1 row is retrieved, and Figure 6-24 and Table 6-12 that show the same when 100 rows are retrieved. The result of these experiments indicates that most of the experiments that are performed on the EETPS functions show that the query execution of retrieving data from VET and CTT-and-VET is faster than CTT (traditional physical tables). Except in two cases, first, when 100 rows are retrieved from the One-to-Many function, VET is slightly slower than CTT. Second, when 1 row is retrieved from the Targeted Tables function VET is slower than CTT and the average difference between them is 236 milliseconds.

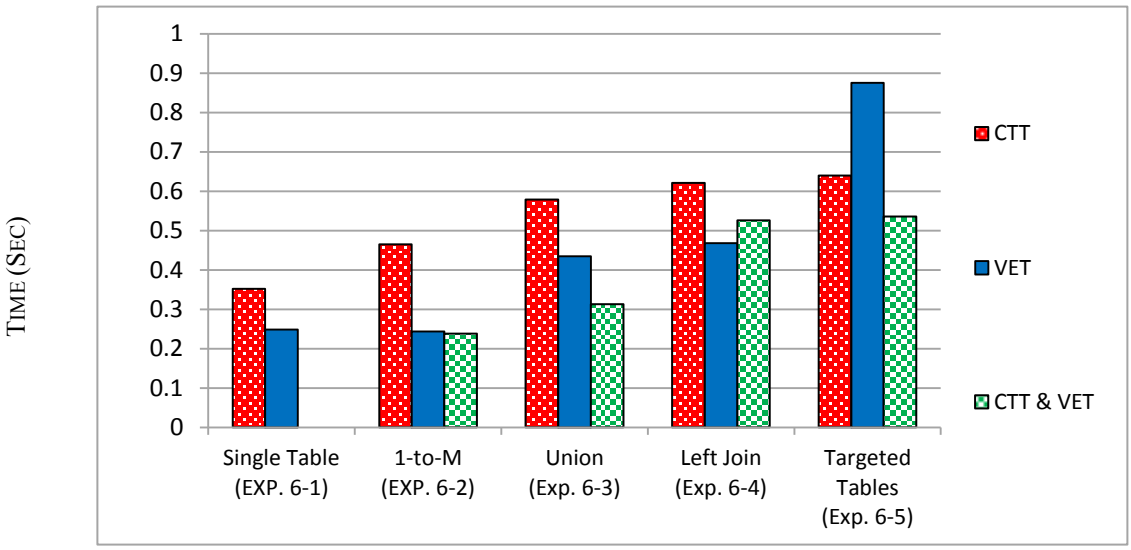


Figure 6-23: The average experimental results of retrieving 1 row

Table 6-11: The average experimental results of retrieving 1 row in milliseconds

Retrieving 1 Row	**CTT**	**VET**	**CTT-and-VET**
Single Table (Exp. 6-1)	**Q 6-1**	**Q 6-2**	
	352	249	
One-to-Many (Exp. 6-2)	**Q 6-3**	**Q 6-4**	**Q 6-5**
	465	244	238
Union (Exp. 6-3)	**Q 6-6**	**Q 6-7**	**Q 6-8**
	579	435	313
Left Join (Exp. 6-4)	**Q 6-9**	**Q 6-10**	**Q 6-11**
	621	468	526
Targeted Tables (Exp. 6-5)	**Q 6-12**	**Q 6-13**	**Q 6-14**
	640	876	536

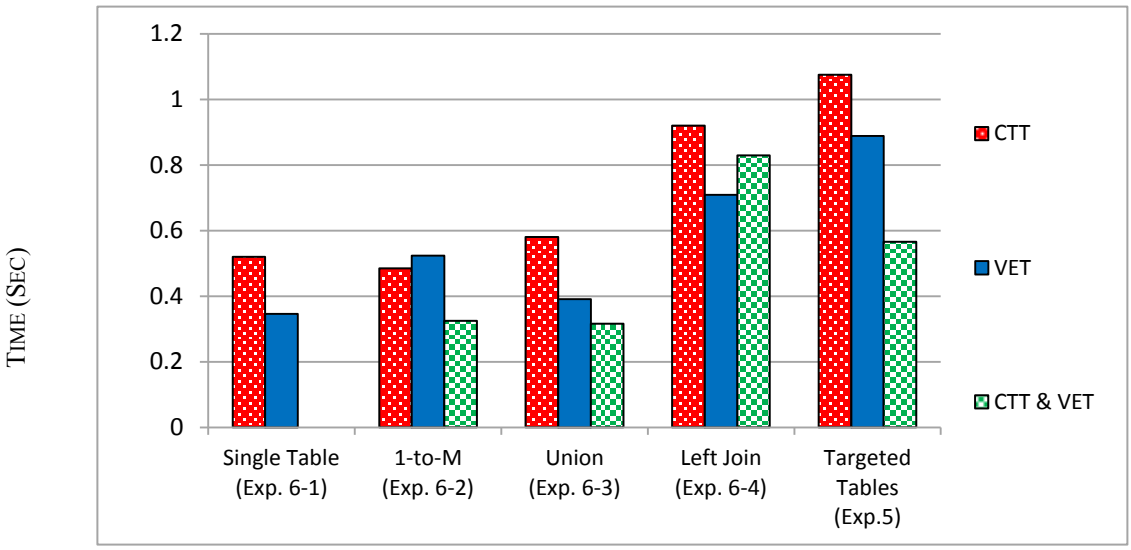


Figure 6-24: The average experimental results of retrieving 100 rows

Table 6-12: The average experimental results of retrieving 100 rows in milliseconds

Retrieving 100 Row	**CTT**	**VET**	**CTT-and-VET**
Single Table (Exp. 6-1)	**Q 6-1**	**Q 6-2**	
	520	346	
One-to-Many (Exp. 6-2)	**Q 6-3**	**Q 6-4**	**Q 6-5**
	485	524	325
Union (Exp. 6-3)	**Q 6-6**	**Q 6-7**	**Q 6-8**
	580	391	316
Left Join (Exp. 6-4)	**Q 6-9**	**Q 6-10**	**Q 6-11**
	920	709	829
Targeted Tables (Exp. 6-5)	**Q 6-12**	**Q 6-13**	**Q 6-14**
	1075	889	566

Table 6-13: The experiments queries

Query No.	Query Name	A set of Individual Query (IQ) Executed in an Algorithm Sequentially (The Details are in APPENDIX II)
Q 6-1	Single Table Query for a CTT.	IQ1.
Q 6-2	Single Table Query for a VET.	IQ2, IQ3, and IQ4.
Q 6-3	One-to-Many Query for two CTTs.	IQ5.
Q 6-4	One-to-Many Query for two VETs.	IQ6, and IQ7.
Q 6-5	One-to-Many Query for a CTT and a VET.	IQ8, and IQ7.
Q 6-6	Union Query for Two CTTs.	IQ9, and IQ10.
Q 6-7	Union Query for Two VETs.	IQ2, IQ3, IQ11, IQ12, IQ13, and IQ14.
Q 6-8	Union Query for a CTT and a VET.	IQ9, IQ12, IQ13, and IQ14.
Q 6-9	Left Join Query for two CTTs.	IQ15, IQ16, and IQ17.
Q 6-10	Left Join Query for two VETs.	IQ2, IQ18, IQ19, IQ4, and IQ21.
Q 6-11	Left Join Query for a CTT and a VET.	IQ15, IQ8, IQ20, IQ1, and IQ21.
Q 6-12	Targeted Tables Query for two CTTs.	IQ22, and IQ23.
Q 6-13	Targeted Tables Query for two VETs.	IQ24, IQ25, IQ26, and IQ27.
Q 6-14	Targeted Tables Query for a CTT and a VET.	IQ28, IQ22, IQ29, and IQ27.

Table 6-14: The experiments queries details

Individual Query (IQ)	Query Details
IQ1	SELECT * FROM product p WHERE p.tenant_id=1000 ORDER BY p.product_id LIMIT 1;

IQ2	SELECT tc.table_column_id FROM table_column tc WHERE tc.tenant_id=1000 and tc.db_table_id=16 and tc.is_primary_key_column = true ORDER BY tc.table_column_id;
IQ3	SELECT distinct ti.table_row_id FROM table_index ti WHERE ti.tenant_id=1000 and ti.db_table_id=16 and ti.table_column_id=47 LIMIT 1;
IQ4	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 = 16 and tr.table_row_id IN (1100) ORDER BY 3,4 LIMIT 8 OFFSET 0;
IQ5	SELECT * FROM sales_fact sf WHERE sf.tenant_id=1000 andsf.product_id = 100 ORDER BY sf.sales_fact_id LIMIT 1;
IQ6	SELECT trs.table_column_id FROM table_relationship trs WHERE trs.tenant_id=1000 and trs.db_table_id=17 and trs.table_type=2 and trs.target_table_id='16' and (trs.table_column_id=58 or trs.target_column_id=58) ORDER BY 1 ASC;
IQ7	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 (SELECT distinct tr.table_row_id From table_index tr WHERE tr.tenant_id =1000 and tr.db_table_id =17 and ((tr.table_column_id = '58' and tr.value ='100')) LIMIT 1 OFFSET 0) ORDER BY 3,4 ASC LIMIT 11 OFFSET 0;
IQ8	SELECT trs.table_column_id FROM table_relationship trs WHERE trs.tenant_id=1000 and trs.db_table_id=17 and trs.table_type=1 and trs.shared_table_name='product' and trs.shared_column_name='product_id' ORDER BY 1 ASC;
IQ9	SELECT p.product_id, p.price FROM product p WHERE p.tenant_id=1000 ORDER BY p.product_id LIMIT 1;
IQ10	SELECT sf.sales_fact_id , sf.unit_price FROM sales_fact sf WHERE sf.tenant_id=1000 ORDER BY sf.sales_fact_id LIMIT 1;
IQ11	SELECT tr.table_column_id, tr.value FROM table_row tr WHERE tr.tenant_id =1000 and tr.db_table_id = 16 and tr.table_row_id IN (1100) and table_column_id in (47,52) ORDER BY tr.table_row_id, tr.serial_id LIMIT 2 OFFSET 0;
IQ12	SELECT tc.table_column_id FROM table_column tc WHERE tc.tenant_id=1000 and tc.db_table_id=17 and tc.is_primary_key_column=true ORDER BY tc.table_column_id;
IQ13	SELECT distinct ti.table_row_id FROM table_index ti WHERE ti.tenant_id=1000 and ti.db_table_id=17 and ti.table_column_id=55 LIMIT 1;
IQ14	SELECT tr.table_column_id, tr.value FROM table_row tr WHERE tr.tenant_id =1000 and tr.db_table_id = 17 and tr.table_row_id IN (200001) and table_column_id in (55,61) ORDER BY tr.table_row_id, tr.serial_id LIMIT 2 OFFSET 0;

IQ15	SELECT c.COLUMN_NAME FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS pk ,INFORMATION_SCHEMA.KEY_COLUMN_USAGE c WHERE pk.TABLE_NAME = 'product' and CONSTRAINT_TYPE = 'PRIMARY KEY' and c.TABLE_NAME = pk.TABLE_NAME and c.CONSTRAINT_NAME = pk.CONSTRAINT_NAME;
IQ16	SELECT c.COLUMN_NAME FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS pk ,INFORMATION_SCHEMA.KEY_COLUMN_USAGE c WHERE pk.TABLE_NAME = 'sales_fact' and CONSTRAINT_TYPE = 'PRIMARY KEY' and c.TABLE_NAME = pk.TABLE_NAME and c.CONSTRAINT_NAME = pk.CONSTRAINT_NAME;
IQ17	SELECT * FROM product lt LEFT JOIN sales_fact rt ON lt.product_id = rt.product_id LIMIT 1 OFFSET 0;
IQ18	SELECT trs.table_column_id FROM table_relationship trs WHERE trs.tenant_id=1000 and trs.db_table_id=17 and trs.table_type=2 and trs.target_table_id='16' and (trs.table_column_id=47 or trs.target_column_id=47) ORDER BY 1 ASC;
IQ19	SELECT trl.table_row_id , trr.table_row_id FROM table_index trl, table_index trr WHERE trl.tenant_id = 1000 and trr.tenant_id = 1000 and(((trl.db_table_id = 16 and trl.table_column_id = 47) and(trr.db_table_id = 17 and trr.table_column_id = 58) and trl.value = trr.value)) LIMIT 1 OFFSET 0;
IQ20	SELECT cttl.product_id , trr.table_row_id as right_row_id FROM product cttl, table_index trr WHERE cttl.tenant_id = 1000 and trr.tenant_id = 1000 and((trr.db_table_id = 17 and trr.table_column_id = 58 and trr.value = CAST(cttl.product_id AS TEXT))) LIMIT 1 OFFSET 0;
IQ21	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 (200001) ORDER BY 3,4 LIMIT 11 OFFSET 0;
IQ22	SELECT sf.sales_fact_id FROM sales_fact sf WHERE sf.tenant_id=1000 and sf.quantity >= 9000 and product_id=100 ORDER BY sf.sales_fact_id;
IQ23	SELECT * FROM sales_details sd WHERE sd.tenant_id=1000 and sd.discount >= 30 and (sales_fact_id in (9 , 12 , 16 , … and other IDs)) ORDER BY sd.sales_details_id LIMIT 1;
IQ24	SELECT * FROM table_relationship trs WHERE trs.tenant_id=1000 and trs.db_table_id=16 or trs.target_table_id=16 ORDER BY trs.table_relationship_id;
IQ25	SELECT tr.table_column_id, tr.value, tr.table_row_id, tr.serial_id FROM table_row tr JOIN table_column tc ON tr.table_column_id = tc.table_column_id and tr.tenant_id = tc.tenant_id and tr.db_table_id =

	tc.db_table_id WHERE tc.is_primary_key_column= 't' and tr.tenant_id =1000 and tr.db_table_id =17 and tr.table_row_id IN (SELECT distinct tr.table_row_id FROM table_row tr WHERE tr.tenant_id = 1000 and tr.db_table_id = 17 and tr.table_column_id = 60 and(cast(value as numeric) >= '9000') and tr.table_row_id IN (SELECT tr.table_row_id FROM table_index tr WHERE tr.tenant_id =1000 and tr.db_table_id =17 and((tr.table_column_id = '58' and tr.value ='100')))) ORDER BY 3,4 ASC;
IQ26	SELECT * FROM table_relationship trs WHERE trs.tenant_id=1000 and trs.db_table_id=17 or trs.target_table_id=17 ORDER BY trs.table_relationship_id;
IQ27	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 =18 and tr.table_row_id IN (SELECT distinct tr.table_row_id FROM table_row tr WHERE tr.tenant_id = 1000 and tr.db_table_id = 18 and tr.table_column_id = 81 and(cast(value as numeric) >= '30') and tr.table_row_id IN (SELECT distinct tr.table_row_id From table_index tr WHERE tr.tenant_id =1000 and tr.db_table_id =18 and((tr.table_column_id = '80' and tr.value ='9') OR (tr.table_column_id = '80' and tr.value ='12') OR (... and other symetric query filters, but with different values)) LIMIT 1 OFFSET 0)) ORDER BY 3,4 ASC;
IQ28	SELECT * FROM table_relationship trs WHERE trs.tenant_id=1000 and trs.shared_table_name='product' ORDER BY trs.table_relationship_id;
IQ29	SELECT * FROM table_relationship trs WHERE trs.tenant_id=1000 and trs.shared_table_name='sales_fact' ORDER BY trs.table_relationship_id;

6.4 SUMMARY

In this chapter, a multi-tenant proxy service called EETPS is proposed. This service integrates, generates, and executes tenants' queries by using a codebase solution that converts multi-tenant queries into traditional database queries and execute them in a RDBMS. This service has three objectives. Firstly, it allows tenants to choose from the three database models of EET, including multi-tenant relational database, integrated multi-tenant relational database and virtual relational database, and virtual relational database. Secondly, it allows each single tenant to extend his database schema, by extending the existing business domain database schema that based on a traditional RDBMS during the application's runtime execution. Thirdly, it

avoids tenants from spending efforts on writing SQL queries and backend data management code by utilizing the service functions that execute simple and complex queries including join operations, filtering on multiple properties, and filtering of data based on subqueries results. These three objectives, overcome the RDBMS and NoSQL issues that discussed in the literature review in Chapter 2, except the scalability issue of RDBMS, which is one of the future research directions of this study. Moreover, in this chapter, five sample algorithms for five functions of EETPS were developed, and five experiments for these functions were carried out to verify the effectiveness of EETPS. These experiments were classified according to the complexity of the queries that were used in the experiments. The five experiments show comparisons between the response time of retrieving data from CTTs, VETs, and both CTTs and VETs. The result of these experiments shows that most of the experiments that performed by calling functions from the EETPS to retrieve data from VET and CTT-and-VET improves the performance when compared to CTT (traditional physical tables). These results verify the practicability and the effectiveness of using EETPS and EET multi-tenant database and their three types of database models. Moreover, these findings make the EET multi-tenant schema and EETPS suitable for the software applications in general and SaaS and Big Data applications in particular.

CHAPTER 7

MULTI-TENANT QUERY OPTIMIZER METHOD

Organisations often spend large amounts of their time, resources and money managing and supporting information stored in their on-premises databases, to ensure that the right information is available when it is needed. According to statistics, data management cost is 5 to 10 times more than the data gain cost (Alzain & Pardede 2011; Hacigümüş et al. 2002). Therefore, it is widely agreed that this is a significant issue for organisations in general and for small and medium size organisations in particular. Subsequently, the multi-tenant database is considered a solution for this issue because it provides database features such as data definition, storage, and retrieval that can be accessed from the service providers' premises on a subscription basis over the internet (Mateljan, Cisic & Ogrizovic 2010). However, such an approach raises an issue in database performance, because the multi-tenant database is shared between multiple tenants. Therefore, this contemporary database requires a special query method to optimize different query retrievals for multiple tenants who are using the same resources of a single multi-tenant database. The majority of modern RDBMS such as Oracle, SQL Server (Raza et al. 2010), and PostgreSQL (Dash et al. 2010) have a query optimizer to optimize the query execution of a single-tenant (single-user) database. Nevertheless, the multi-tenant database requires a special query optimizer method that plays a vital role in improving the multi-tenant query processing and solves the issues of multi-tenant database, including isolating tenant's data statistics, retrieving tenant's queries in a timely and cost efficient

manner, and making the best use of multi-tenant database resources. Salesforce states that modern database query optimizers are designed for single-tenant databases and they are not suitable for multi-tenant databases. That is because they are not taking in consideration the unique characteristics of each tenant's data, such as indexes and the gathered statistics for all tenants instead of specific statistics for a tenant or a tenant's user, which in return is leading to incorrect assumptions and query plans of tenants' data (Weissman & Bobrowski 2009; Weissman et al. 2012). Therefore, to implement a multi-tenant database on a single-tenant RDBMS, a multi-tenant query optimizer is required by special query execution plans, which assign indexes for the tenants' data and gather special statistics for each single tenant and each single tenant's user.

The main contribution of this chapter is proposing a multi-tenant optimizer service, called Elastic Extension Tables Query Optimizer Service (EETQOS). This service optimizes the tenant's data retrieval using EETPS that presented in the previous chapter (Chapter 6), through estimating the cost of different query execution plans, and then determining the optimal query execution plan based on the estimated cost and the structure of a given query. The query optimizer of this service does not replace the traditional RDBMS query optimizers. In contrast, it is used to optimize how to collect each single tenant and each single tenant's user statistics and choose the optimal query execution plan. Then this plan is used in EETPS to generate and execute the tenant query in a RDBMS using its traditional query optimizer and its powerful and advanced data management capabilities. Scalability is very significant for multi-tenant applications, however, before anyone starts thinking to scale-out or scale-up multi-tenant database to optimize its performance, the multi-tenant database performance should be optimized in each single server instance by applying a proper multi-tenant query optimizer, then any of the scale-out or the scale-up approaches can be applied afterwards. Accordingly, the focus of this chapter will be on how to optimize multi-tenant query performance in a single server instance and scalability will be out of this thesis scope. Nevertheless, it is one of the future research directions of this study. Moreover, the practicability and effectiveness of applying the EETQOS

on EETPS is verified by executing a number of different types of queries to retrieve a tenant's data from EET multi-tenant schema via using these two services of the EET framework.

The remainder of this chapter is structured as follows. Section 7.1 proposes the EET query optimizer service. Section 7.2 presents a set of experiments to evaluate the performance of retrieving data from a VET using different multi-tenant query optimization methods. Section 7.3 concludes this chapter.

7.1 ELASTIC EXTENSION TABLES QUERY OPTIMIZER SERVICE

The EETQOS Architecture is shown in Figure 7-1. This query optimizer has four aims, including gathering statistics of virtual rows that a query can potentially access, finding the fastest path to execute a query, estimating the cost of different query execution plans, and determining the optimal plan for execution. Then the determined optimal execution plans are used in the EETPS that constructs and generates multi-tenant queries, and executes them in a RDBMS by using its traditional query optimizer. The following seven points show the components of EETQOS architecture and the way that they are orchestrating with EETPS and EET.

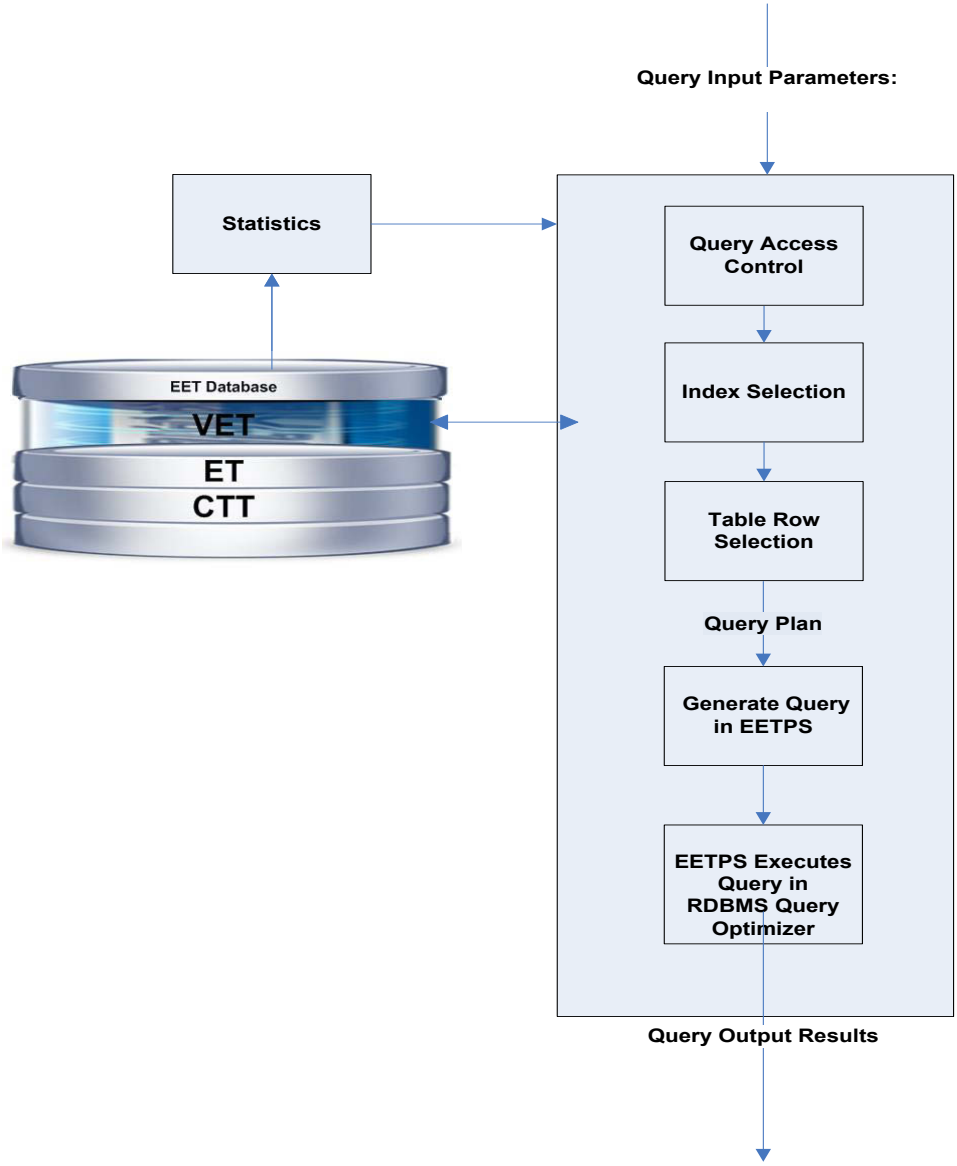


Figure 7-1: The EETQOS architecture and how it is orchestrated with EETPS and EET

7.1.1 Query Access Control

The Query Access Control component is the first component that is executed in the EETQOS, which controls the access of multi-tenant data in CTTs and VETs. As long as CTTs and VETs are using a 'tenant_id' to isolate the tenant's data in EET multi-

tenant schema and divide it into partitions, then each tenant can have his own partitions to store his own data and his tenant's data. These access control permissions are stored in the database, and before executing the users' queries these permissions are checked to optimize the query execution by generating the optimal query structure, which in return reduces the query execution time. In this component, two methods are applied to control the tenants' users in the EET multi-tenant schema, which in return allows EETQOS to determine the optimal query execution plan that reduces the query execution cost. These two methods are briefly presented below, and will be presented in details and evaluated in the following chapter (Chapter 8).

- **Accessing table columns:** This method allows tenants to grant their users permissions to access some or all columns of a CTT or a VET. These permissions can restrict the tenants' users from accessing some or all the table columns. In addition, these permissions can help EETQOS to determine the optimal query execution plans by knowing if a user can access all, some, or none of the table columns. In this method, the query optimization occurs in two cases. Firstly, when a user cannot access any of the table columns, in this case the query is not executed. Secondly, when a user can access some of the table columns, in this case, the data is retrieved from some of the table columns by generating a query structure that is different from the query structure that needs extra execution time to retrieve data from all the table columns.
- **Accessing table Rows:** This method allows tenants to grant their users permissions to access some or all rows of a CTT or a VET. These permissions can restrict tenants' users from accessing some or all rows of a table. These permissions help the EETQOS to choose the optimal query execution plan based on the number of rows that can be accessed by a user, and to generate a query structure to retrieve data from some of the table rows, which is different from the query structure that needs extra execution time to retrieve all the table rows.

In summary, the EETQOS allows to choose the optimal execution plan between different query execution paths based on the tenants' users and their different columns and rows that they can access. This plan reduces the overhead on the EET multi-tenant schema and accelerates the query execution time of its queries.

7.1.2 INDEX SELECTION

The Index Selection component is the second component that is executed in the EETQOS, which checks whether a VET has virtual indexes, then accordingly estimates the cost and chooses the optimal execution plan between different query execution paths. These virtual indexes are related to virtual columns of a VET, and they are typically stored in the 'table_index' ET. This ET stores, three types of indexes:

- **Primary Key Index:** This index is typically created for a single primary key column by having a single primary key index, or composite primary key columns by having multiple primary key indexes for a VET.
- **Foreign Key Index**: This index is typically created for a single foreign key column, or for composite foreign key columns in a VET.
- **Custom Index**: This index can be created for any virtual column of a VET that is used often as a selective filter in a tenant query. The virtual column should be any column other than the primary key and the foreign key columns.

When tenants create a VET and assign to it any of the three indexes listed above, then this VET can have three cases to retrieve table rows. The first case is retrieving rows from a VET by specifying primary keys. The second case is retrieving rows from a VET by specifying table row IDs that are stored in the 'table_row' ET. The third case is retrieving VET rows without specifying any primary keys or row IDs. These three cases can construct a SELECT clause and a WHERE clause in the query to retrieve data from a VET based on the following six execution plans:

- **Primary Key**: This execution plan does not use indexes; instead, it uses the following filters to execute a query, including tenant ID, user ID, VET ID, and specific Primary Keys.
- **Row ID**: This execution plan does not use indexes; instead, it uses the following filters to execute the query, including tenant ID, user ID, VET ID, and specific Row IDs.
- **Full Table**: This execution plan does not use indexes or any other filters except the standard filters, including tenant ID, user ID, and VET ID.
- **A Percentage of Custom Index**: This execution plan uses a percentage of custom index, tenant ID, user ID, VET ID, and a value belongs to the custom index IDs that filters the query.
- **All of Custom Index**: This execution plan uses all the indexes IDs of a custom index column, tenant ID, user ID, VET ID, and a value belongs to the custom index that filters the query.
- **None of Custom Index**: This execution plan uses none of the indexes IDs of Custom Index column, tenant ID, user ID, VET ID, and a particular value that relates to the custom index column.

The EET multi-tenant schema allows each tenant to have a unique data structure, tables, columns, and column constraints. These characteristics work side by side with the above listed index execution plans to support multi-tenant query execution strategies of the EETQOS.

7.1.3 TABLE ROW SELECTION

The Table Row Selection is the third component that is executed in the EETQOS. In EET multi-tenant schema, there are three row ETs that store virtual rows of virtual extension columns. These ETs store three different data types; therefore, they are separated in order to store small data values in the ‘table_row’ ET, and large data

values in the other two tables. First, the 'table_row_blob' ET that stores all BLOB, and second, the 'table_row_clob' ET that store CLOB values. More details of these row tables are presented in Chapter 4. The reason behind separating these three tables is to reduce the impact of BLOB and CLOB values from slowing down virtual schema queries. It is not always the case that all VETs have BLOB and/or CLOB data types. Therefore, this method eliminates the search in 'table_row_blob', and 'table_row_clob' ETs, if a VET has not got any of the BLOB or CLOB columns, or if a VET has got BLOB or CLOB columns, but these columns are not part of the query SELECT clause or WHERE clause. The Table Row Selection method helps the EETQOS in two aspects. Firstly, checking whether a table has a CLOB or BLOB data types, if any of these data types exists, then the query retrieves table rows from the 'table_row' ET, and both or either 'table_row_blob' or 'table_row_clob' ETs. Otherwise, the query retrieves rows from only the 'table_row' ET. The UNION operator keyword is used to combine the result-set of the three SELECT statements of the three row tables if the VET only contains BLOB and/or CLOB. However, if the VET does not contain BLOB and CLOB then the UNION operator does not use in the query. This approach minimizes the runtime optimization overhead on the EETQOS by avoiding using the UNION operator keyword unless it is necessary. More details about this approach are presented in Chapter 6 in Algorithm 6-2. Secondly, separating the data that's stored in the three row ETs, by storing small data values in the 'table_row' ET and large data values in two others ETs the 'table_row_blob' and the 'table_row_clob'. This approach minimizes the runtime optimization overhead on the EETQOS, by minimizing the size of the stored content in the 'table_row' ET that stores most of the tenants' data, and by accessing the BLOB and the CLOB data only on demand.

7.1.4 STATISTICS

The concept of statistics in modern RDBMS is gathering the amount and the data that is stored in the database. These statistics estimate the cost of different query

execution plans that determine the optimal plan. In multi-tenant databases, the statistical concept is slightly different from the single-tenant database. In the multi-tenant database, there are two ways of gathering statistics. Firstly, gathering statistics of each tenant, by differentiating between the tenants' rows. Secondly, gathering statistics of columns and rows that can be accessed by the tenant's user who is allowed to access and view a query result based on groups and/or roles assigned to this user. More details about the statistics gathering will be presented in the following chapter (Chapter 8).

7.1.5 MULTI-TENANT DATABASE

The EETQOS is based on retrieving data stored in EET multi-tenant schema that consists of three types of tables CTT, ET, and VET, which presented in details in Chapter 4. The tenant's and the tenant's users statistics are granted from this multi-tenant schema.

7.1.6 GENERATE QUERY

After executing the EETQOS and finding the best execution plans, the EETPS is invoked to generate a virtual multi-tenant query based on the selected query plans that decided from the EETQOS components.

7.1.7 EXECUTE QUERY

The EETPS executes multi-tenant database query according to the best and lowest cost execution plan that was selected from the query optimizer components. Then this multi-tenant query is converted into a traditional database query and is executed in RDBMS by using its traditional query optimizer and its powerful and advanced data management capabilities.

7.2 PERFORMANCE EVALUATION

After the EETPS of the EET framework prototype has developed, the EETQOS is implemented on the EETPS, and six types of experiments are carried out to verify the practicability of implementing the EETQOS on the EETPS. The aim of these experiments is to examine the response times when EETPS consumes the EETQOS to select the optimal tenant's query execution plan, then uses this plan to convert the tenant's queries into traditional database queries, and finally executes these queries using a traditional query optimizer of a RDBMS.

7.2.1 EXPERIMENTAL DATA SET AND SETUP

The EETPS has designed and developed to serve multiple tenants on one instance application. Nevertheless, in this chapter the aim of the experiments is evaluating the performance after applying the EETQOS on the EETPS for one tenant. Typically, multi-tenant databases store massive data volumes across multiple servers to optimize the performance of data retrieval. However, before anyone starts thinking to scale-up or scale-out multi-tenant database to optimize its performance, the multi-tenant database performance should be optimized in each single server instance by implementing a proper multi-tenant query optimizer, then either of the scale-up or the scale-out approaches can be applied afterwards. In this experiment, one machine is used, and the Single Table function of the EETPS (presented in Chapter 6 in Section 6.2.1) is invoked to retrieve a 100 of rows from the 'product' VET, which is shown in Figure 7-2. There are 200,000 rows stored in this table that belongs to a tenant whose 'tenant_id' equals 1000, and the 'db_table_id' of this table equals 16. The 'tenant_id', the 'db_table_id', and all the filters that are specified in the experiments filter all the queries that are implemented in these experiments. These experiments are executed for one tenant, because, in a multi-tenant database, each tenant's data is isolated in a table partition. Thus, the purpose of these experiments is to evaluate the effectiveness of retrieving data for a single tenant from the multi-tenant database by using

EETQOS. These experiments are divided into six types sharing the details of this data set. The queries of these experiments are shown in Table 7-2, and they are structured based on the Single Table algorithm that presented in the previous chapter (Chapter 6). The six experiments are listed below:

1) **None of Custom Index Experiment (Exp.7-1):** In this experiment, the aim is to filter a query by using a query filter, which is the 'standard_cost' column of the table. This filter retrieves all the rows that have a standard cost value greater than 9000 ('standard_cost' > 9000). In addition, it is assumed that the tenant did not create a custom index to tune the query execution, and no any other index of the table is used to execute Query 7-1 (Q7-1).
2) **A Percentage of Custom Index Experiment (Exp.7-2):** In this experiment, the aim is to have the same assumption of using the standard cost filter that stated in Exp.7-1. However, in this experiment, a custom index is created to tune the query execution. Accordingly, in this experiment, the custom indexes on 'standard_cost' column is used to execute Query 7-2 (Q7-2), and only a particular percentage of the table rows that match the filter criteria will be retrieved.
3) **All of Custom Index Experiment (Exp.7-3):** The aim of this experiment is to benchmark the effectiveness of using all the table indexes IDs in Query 7-3 (Q7-3), by using the same standard cost filter that is assumed in Exp.7-1.
4) **Full Table Experiment (Exp.7-4):** In this experiment, the aim is to benchmark the effectiveness of not using any query filters or indexes filters in Query 7-4 (Q7-4).
5) **Primary Key Index Experiment (Exp.7-5):** In this experiment, the aim is to benchmark the effectiveness of filtering Query 7-5 (Q7-5) by using three values of the primary key 'product_id', including 101, 102 and 103, and without using any indexes or query filters.

6) **Row ID Experiment (Exp.7-6):** The aim of this experiment is to benchmark the effectiveness of filtering Query 7-6 (Q7-6) by using three values of the 'table_row_id', including 22, 23 and 24, and without using other indexes or query filters.

We have included Exp.7-4, Exp.7-5, and Exp.7-6 to compare the query execution time of these experiments with the other experiments (Exp.7-1, Exp.7-2, and Exp.7-3) of the Custom Index.

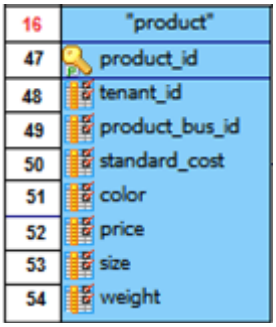


Figure 7-2: The table structure of the 'product' table

The EETQOS 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, the database and the application server are deployed on the same PC. The operating system is Windows 7 Home Premium, with Intel Core i5 2.40GHz CPU, 8 GB of RAM memory, and 500 GB of hard disk storage.

7.2.2 Experimental Results

The experimental study is showing that the average execution time for Q7-4, Q7-5, and Q7-6 equals approximately 240 milliseconds. In addition, the average execution time of Q7-2 is approximately 50% slower than the average execution time of Q7-4, Q7-5, and Q7-6. Whereas, the average execution time of Q7-1 is approximately 83% slower than the average execution time of Q7-4, Q7-5, and

Q7-6. Finally, the average execution time of Q7-3 is the slowest one, which is on average approximately 88% slower than the average execution time of Q7-4, Q7-5, and Q7-6. The practicability and the advancements of EET multi-tenant schema is verified by executing the queries of Exp.7-4, Exp.7-5, and Exp.7-6 in a short time. Whereas Exp.7-1, Exp.7-2, and Exp.7-3 compared the difference between three Custom Index cases, including *None of Custom Index* (Q7-1), *A Percentage of Custom Index* (Q7-2), and *All of Custom Index* (Q7-3). The interpretation of these three experiments leads to the following conclusions. Firstly, if a tenant filters a query by using a column value not indexed, and it is neither a primary key nor a foreign key, then the EETQOS chooses the execution plan of a *None of Custom Index* (Q7-1). Secondly, if a tenant uses the same column that stated in the first point, and this column is a Custom Index, then the EETQOS chooses the execution plan of the *Percentage of Custom Index* (Q7-2). Thirdly, the EETQOS does not choose the execution plan of the *All of Custom Index* (Q7-3), because the query execution cost of this query is high in comparison with the execution plan of a *None of Custom Index* (Q7-1) or the *Percentage of Custom Index* (Q7-2). The details results of this experiment are shown in Figure 7-3 and Table 7-1.

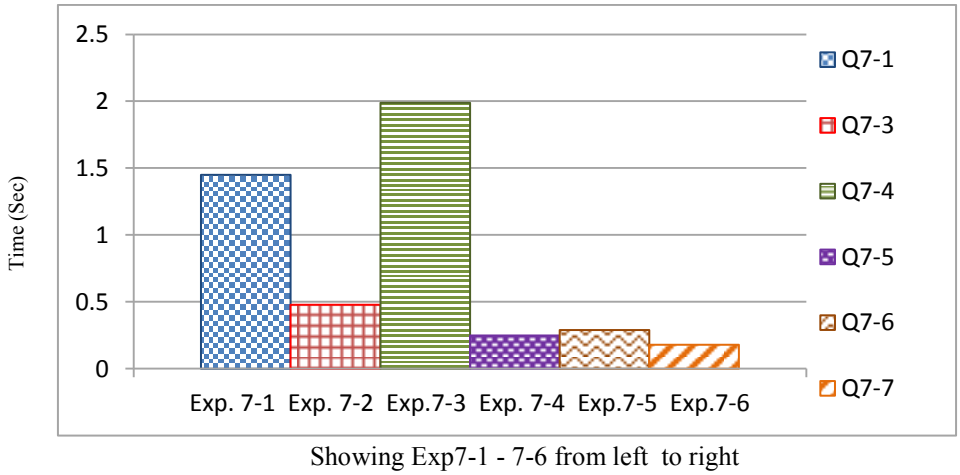


Figure 7-3: The experimental results of retrieving data using filters and indexes

Table 7-1**:** The query execution times of retrieving data using filters and indexes

Experiment	Query executed	Time in seconds
Exp. 7-1	Q7-1	1.45
Exp. 7-2	Q7-2	0.48
Exp. 7-3	Q7-3	1.99
Exp. 7-4	Q7-4	0.25
Exp. 7-5	Q7-5	0.29
Exp. 7-6	Q7-6	0.18

Table 7-2: The experiments queries

Query No.	Query Details
Q7-1	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 = 16 AND tr.table_row_id IN (SELECT distinct tr.table_row_id **FROM table_row** tr WHERE tr.tenant_id = 1000 AND tr.db_table_id = 16 AND tr.table_column_id = 50 **AND (cast (value as numeric) > '9000')**) ORDER BY 3, 4 LIMIT 800 OFFSET 0;
Q7-2	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 = 16 AND tr.table_row_id IN (SELECT distinct tr.table_row_id **FROM table_index** tr WHERE tr.tenant_id = 1000 AND tr.db_table_id = 16 AND tr.table_column_id = 50 **AND (cast (row_value as numeric) > '9000')**) ORDER BY 3, 4 LIMIT 800 OFFSET 0;
Q7-3	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 = 16 AND tr.table_row_id IN (SELECT distinct tr.table_row_id FROM table_row tr WHERE tr.tenant_id = 1000 AND tr.db_table_id = 16 AND tr.table_column_id = 50 **AND (cast (value as numeric) > '9000')**) **AND tr.table_row_id IN (SELECT distinct tr.table_row_id FROM table_index tr WHERE tr.tenant_id = 1000 AND tr.db_table_id = 16)** ORDER BY 3, 4 LIMIT 800 OFFSET 0;
Q7-4	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 = 16 ORDER BY 3, 4 LIMIT 800 OFFSET 0;
Q7-5	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 = 16 AND tr.table_row_id IN (SELECT DISTINCT tr.table_row_id FROM table_row tr WHERE tr.tenant_id = 1000 AND tr.db_table_id = 16 **AND ((tr.table_column_id =47 AND tr.value ='101') OR (tr.table_column_id =47 AND tr.value ='102') OR (tr.table_column_id =47 AND tr.value ='103'))** ORDER BY 3,4 ;
Q7-6	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 = 16 **AND tr.table_row_id IN (22, 23, 24)** ORDER BY 3, 4;

7.3 SUMMARY

A multi-tenant query optimizer service called EETQOS is proposed in this chapter. This service estimates the cost of different query execution plans to determine the optimal query execution plan. This query optimizer service reduces the query execution time of EETPS that access data from EET multi-tenant schema. Moreover, six types of experiments were carried out to verify the practicability of implementing the EETQOS on the EETPS that is based on EET multi-tenant schema. These experiments conclude that the cost of executing a query from a VET when using all the indexes of that VET is high in comparison with the cost of executing the same query without using any index. Nevertheless, the least query execution cost was recorded when a custom index with a percentage of a VET indexes is used to retrieve table rows from the same VET. Furthermore, these experiments confirmed the effectiveness of using the data structure of EET multi-tenant schema that is the base storage of EETQOS and EETPS, and the benefits of EETQOS query execution plans for EETPS and EET multi-tenant schema.

CHAPTER 8

MULTI-TENANT ACCESS CONTROL METHOD

This chapter focuses on the *Shared Database - Shared Schema* isolation approach that stated in chapter 2, which requires a high degree of data isolation to ensure the security and privacy of the tenants' shared data. This multi-tenant data approach consists of two data types, shared tenants' data and tenants' isolated data, by integrating these two types of data together, tenants can have the complete data they need (Domingo et al. 2010; Liu 2010). An access control service is proposed in this chapter, called Elastic Extension Tables Access Control Service (EETACS). This service is based on EET multi-tenant schema that designed to be used by multiple tenants and each tenant may have multiple users. Therefore, such a database demands a special multi-tenant access control model that provides an access control not only for multiple tenants, but also for multiple users per tenant. This method permits each tenant in the multi-tenant database to have several users with different types of grants to access the tenant's data.

The remainder of this chapter is structured as follows. Section 8.1 proposes the EET access control service. Section 8.2 presents the columns and rows access grant algorithms. Section 8.3 presents two experiments to verify the practicability of granting a tenant's user accessibility on a tenant's table columns and rows, by using the EETPS and EETACS that grant the tenants' users permissions to access their data. Section 8.4 concludes this chapter.

8.1 ELASTIC EXTENSION TABLES ACCESS CONTROL

An access control data architecture that is based on EET multi-tenant schema is defined in this section. In addition to EET access grants that are granted to tenants' users to access table columns and rows that are stored in the EET multi-tenant schema.

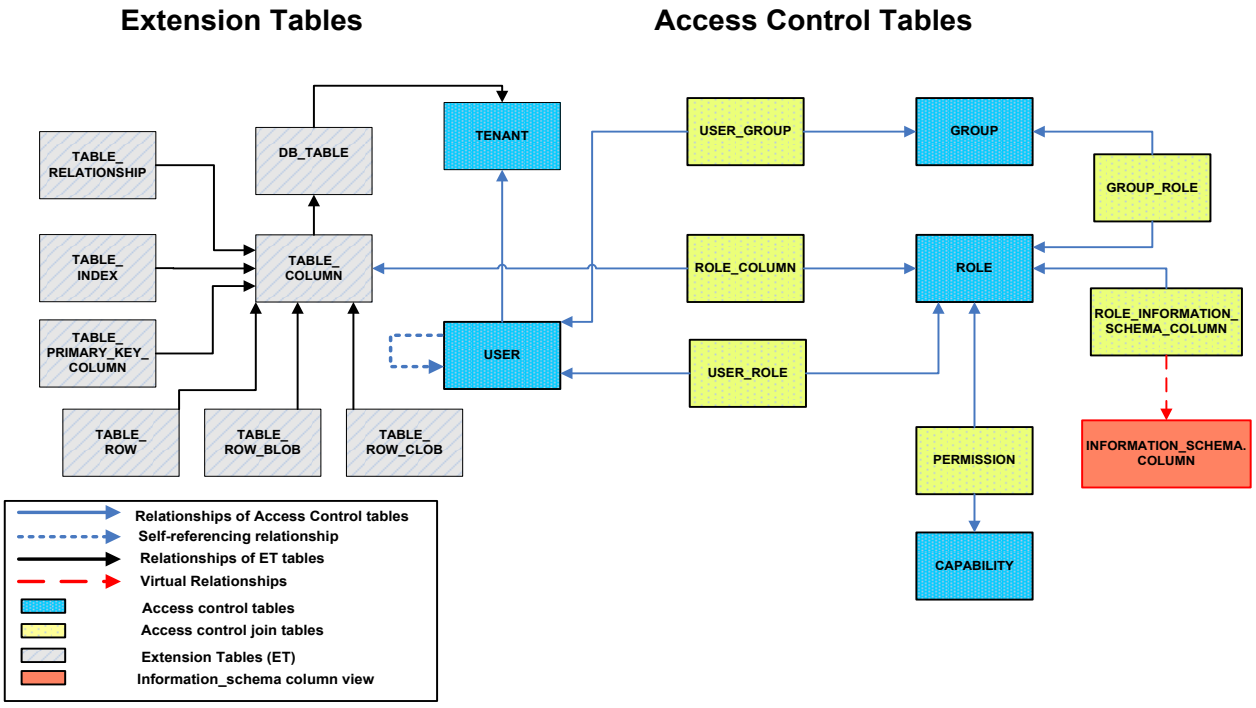


Figure 8-1: EET Access Control Data Architecture

8.1.1 ACCESS CONTROL TABLES

There are three types of EETACS tables that store the tenants' access control configurations. The first type is the main entity tables of the access control data

architecture. The second type is the join tables, and the third type is the Information_Schema view. These three types of tables are listed below:

1) **Access Control Main Entity Tables**: These tables are listed below and illustrated in a blue colour in Figure 8-1.

- **Tenant Table**: This table stores the tenants' information details. The Tenant ID column of this table is used to isolate the tenants' data that is stored in a CTT or a VET. Such isolation is applied by having a master-detail relationship between this table and a CTT or a VET. Where the Tenant ID column is used as a reference column to a CTT or a VET, to refer the data in any of these tables to an existing tenant who has a unique Tenant ID in the Tenant table.
- **User Table**: Each tenant in the multi-tenant database can have multiple users accessing the tenant's data. This table can store three types of these users. The first type is an admin or a super user. The second type is a single user. The third type is a parent-child user who allows a tenant to have an admin or a super user and assign to this user one or more users by using the self-referencing relationship that this table has. Each user type can have different levels of database accesses that based on groups and/or roles, which are associated with a user.
- **Group Table**: This table is used to define different levels of tenants' groups. Any of these groups is logically associating users with similar data access needs. Once a tenant group is defined, some roles that have granted permissions are assigned to that group. Then, any tenant user who is associated with this group inherits all of the permissions granted to that group.
- **Role Table**: The tenants' users and the tenants' groups can have roles that are granted permissions to perform database activities on a tenant's data among multiple tenants' data, which is stored in a multi-tenant database. In addition, this role table grants permissions for both types of tables CTTs and VETs.

- **Capability Table**: This table allows tenants to authorise their users privileges to any operation performed upon data. These operations have different access levels, including full access, read/write access, read access, and other access types.

2) **Access Control Join Tables**: These tables are used to establish Many-to-Many relationships between the access control main entities, and are listed below and illustrated in a yellow colour in Figure 8-1.

- **User_Group Table**: This join table is used to allocate an access classification level between groups and the tenants' users. Typically, this allocation is used to group users together, such as administrator users, super users, or public users.
- **Group_Role Table**: This join table is used to authorise a group of users to access one or more database access roles, and any user who is allocated to this group inherits all the permissions that are granted to the group.
- **User_Role Table**: This join table is used to authorise a user to access one or more database access roles.
- **Role_Column Table**: This join table is used to allocate a role to access some or all the tenant's columns of a VET. Once the tenant has this allocation, he can add business rules to access some or all rows from these VET columns. The details of these business rules are presented in Section 8.2.2.
- **Permission Table**: This join table is used to allocate roles to different kinds of database access capabilities such as full access, read/write access, read access, and other access types.
- **Role_Information_Schema_Column Table**: The purpose of this join table is similar to the purpose of the Role_Column Table. However, this join table allocates a role to access some or all the tenant's columns of a CTT, once the tenant has this allocation, he can add business rules to access some or all rows from these CTT columns.

3) **Information_Schema.column View**: This view allows to get information about columns for tables and views within the PostgreSQL RDBMS. This Information_Schema view is also used by databases, such as Oracle[11], Mysql[12], and others. This view is used in the EETACS data architecture to give access grants for tenants' users to access CTTs columns. This view is illustrated in a red colour in Figure 8-1.

8.1.2 ELASTIC EXTENSION TABLES ACCESS GRANTS

EETACS has two main types of grants. The first type is Group Access Grant, in which a user is assigned to a group, and this user inherits all of the roles granted to that group. The second type is Role Access Grant, in which a user is assigned to a role that assigned to a user directly or inherited via a group. To allow the user to access CTTs or VETs in the EET database schema. The two types of grants are shown in Figure 8-2. The group access grant is illustrated in the blue arrows, and the role access grant is illustrated in the grey arrow.

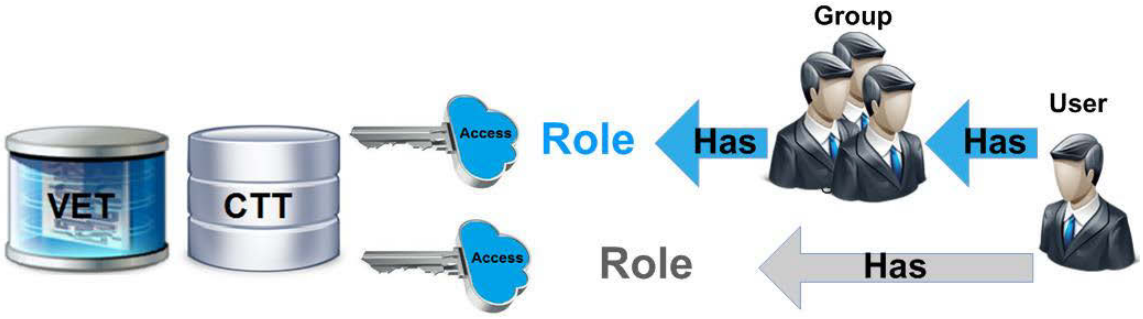


Figure 8-2: EET access control grants

These two main types of grants have two subtypes of grants. First, Table Columns Access Grant, and second, Table Rows Access Grant. These access grants control the access of multi-tenant data in CTTs and VETs. Since CTTs and VETs are using the Tenant ID to isolate the tenants' data in EET multi-tenant database and divide it into partitions, then each single tenant can have his own partitions to store their own data.

11 http://docs.oracle.com/cd/E17952_01/refman-5.1-en/information-schema.html; Accessed July, 2014
12 http://dev.mysql.com/doc/refman/5.0/en/columns-table.html; Accessed July, 2014

Moreover, these partitions are divided by the tenants' users according to these two grants, which are discussed in details, in the following two points:

- **Table Columns Access Grant**: In this grant, tenants are allowed to give user permissions access some or all columns of a CTT or a VET. These permissions can restrict tenants' users from accessing some or all columns of a table. For example, Figure 8-3 is showing two types of users, the first user is a super user called Adam, who has roles that can access all the table's columns of a table. The second user is Abraham, who has roles that can access only three columns of the same table that Adam can access. In addition, this grant helps in deciding the optimal query execution plans, by knowing whether a user can access all, or some of a table columns. In the case when a user can access some of the table's columns that can be retrieved from a table, this grant, generates a query structure different from the structure of retrieving all the columns.

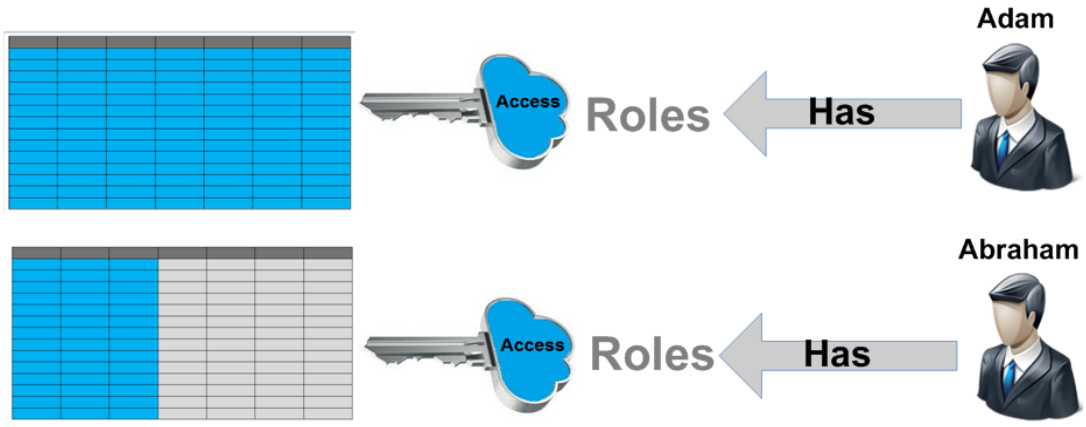


Figure 8-3: Table columns access grant

- **Table Rows Access Grant**: In this grant, tenants are allowed to offer user permissions to access some or all rows of a CTT or a VET. These permissions can restrict tenants' users from accessing some or all rows of a table. For example, Figure 8-4 is showing the same users who were shown in Figure 8-3,

but this time Adam has roles that can access all the table's columns and rows, while Abraham has roles grant him to access all columns, but only some rows of the same table that Adam can access. Moreover, this grant optimizes the query execution by considering the number of rows that are accessed by a user, and generating a query structure different from the query structure of retrieving all the table rows.

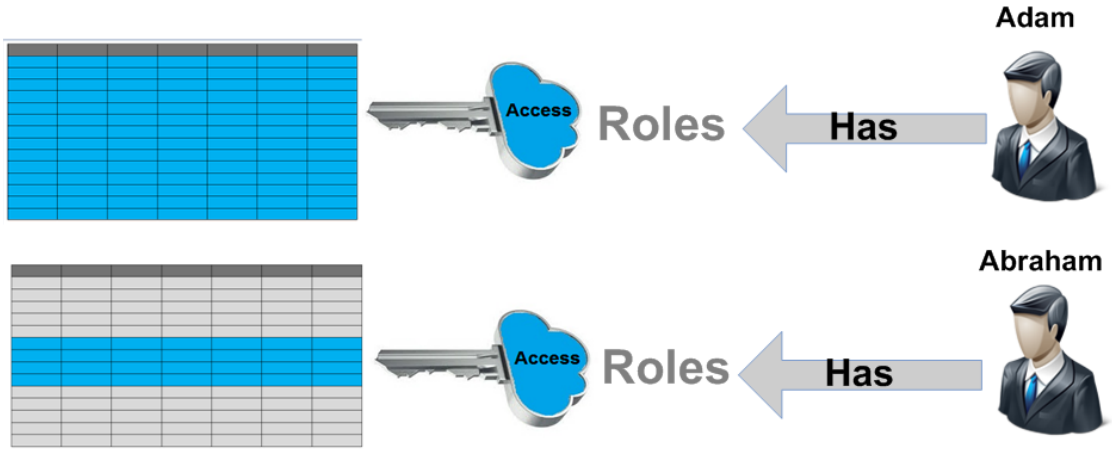


Figure 8-4: Table rows access grant

8.2 Columns and Rows Access Grant Algorithms

In this section, six access control algorithms are presented to allow the tenants' users to access data that are granted to them, when they assigned roles that are permitted to access columns and rows of a CTT or a VET. The first two algorithms (Algorithm 8-1 and 8-2) are used as subsidiary algorithms in Algorithm 8-3 - 8-6. These algorithms verify the table columns and rows that a tenant's user can insert, update, delete, or retrieve. The details of these algorithms are listed below.

8.2.1 Get User Roles Algorithm

The Get User Roles Algorithm retrieves the tenant's user roles that assigned to CTT or VET columns. The details of this algorithm are presented in Definition 8-1 and Algorithm 8-1.

Definition 8-1 (Get User Roles): T denotes a tenant ID. U denotes a tenant's user. B denotes a table name. S denotes a string of the SELECT clause parameter. Q denotes the table type, whether it is a CTT or a VET. R_{table} denotes a set of role ID values assigned for a CTT or a VET. $R_{tableSize}$ denotes the size of R_{table}. R_{group} denotes a set of role ID values assigned to the tenant's user groups. R_{user} denotes a set of role ID values assigned to the tenant's user. $R_{userSize}$ denotes the size of R_{user}. R_{id} denotes a role ID. R_{return} denotes a set of role ID values that the tenant's user can access. ∅ denotes an empty set.

Algorithm 8-1: GetUserRoles (T, U, B, Q)

Input: T, U, B, and Q
Output: R_{return}
1. **if** Q = 'CTT' **then**
2. R_{table} ← retrieve roles assigned to a CTT from role_information_schema_column table using T and B query filters
3. **else**
4. R_{table} ← retrieve roles assigned to a VET from the role_column table using T and B query filters
5. **end if**
6. R_{group}← retrieve roles assigned to U from the group_role table using T, and U query filters
7. R_{user} ← retrieve roles assigned to U from user_role table using T, and U query filters
8. $R_{return} \leftarrow \varnothing$
9. **for** $i \leftarrow 0$ to $R_{tableSize}$ **do**
10. $R_{id} \leftarrow R_{table\ i}$
11. **if** $R_{id} \in R_{group} \vee R_{id} \in R_{user}$ **then**
12. return /* Exit the algorithm */
13. **end if**
14. $i \leftarrow i + 1$
15. **end for**
16. $R_{return} \leftarrow R_{group}$
17. **for** $j \leftarrow 0$ to $R_{userSize}$ **do**
18. $R_{id} \leftarrow R_{user\ j}$

19. **if** $R_{id} \notin R_{return}$ **then**
20. $R_{return} \leftarrow R_{return} \cup R_{id}$
21. **end if**
22. $j \leftarrow j + 1$
23. **end for**
24. **Return** R_{return}

8.2.2 GET USER COLUMNS ALGORITHM

The Get User Columns Algorithm retrieves columns and columns rules that are granted to a tenant's user. The details of this algorithm are presented in Definition 8-2 and Algorithm 8-2.

Definition 8-2 (Get User Columns): T denotes a tenant ID. U denotes a tenant's user. B denotes a table name. Q denotes the table type whether it is a CTT or a VET. C_{user} denotes the tenant's user columns and columns rules retrieved from the 'role_column' access control table, and stored in a matrix with n rows and 2 columns. Where $C_{user\ i,0}$ is the first column of the matrix that represents the tenant's user columns, and $C_{user\ i,1}$ is the second column of the matrix that represents the tenant's user columns rules. C_{select} denotes a set that is storing a table columns values that constructs the tenant's user query SELECT clause, where $C_{select} = \{ C_{select\ 1}, C_{select\ 2}, \ldots C_{select\ n}\}$. Each element in this set represents a column name in B. C_{where} denotes a string that is storing the access control part of the query WHERE clause, which typically is used to grant rows access to tenant's users. R_{return} denotes a set of roles the tenant's user can access. C_{return} denotes a row matrix with 1 row and 2 columns that has two elements, the first element is $C_{return\ 0,0}$ that stores the value of C_{select}, and the second element is $C_{return\ 0,1}$ that stores the value of C_{where}.

Algorithm 8-2: GetUserColumns(T, U, B, Q, R_{return})

Input: T, U, B, Q, and R_{return}
Output: C_{return}
1. **if** Q = 'CTT' **then**
2. $C_{user} \leftarrow$ retrieve columns and columns rules for U who has R_{return} from role_information_schema_column table using T, U, B, and R_{return} query filters

3. **else**
4. C_{user} ← retrieve columns and columns rules for U who has R_{return} from role_column table using T, U, B, and R_{return} query filters
5. **end if**
6. $i \leftarrow 0$
7. **for all** C_{user} **do**
8. $C_{select\ i} \leftarrow C_{user\ i,0}$
9. $C_{where} \leftarrow C_{where} \cup C_{user\ i,1}$
10. $i \leftarrow i + 1$
11. **end for**
12. $C_{return\ 0,0} \leftarrow C_{select}$
13. $C_{return\ 0,1} \leftarrow C_{where}$
14. **Return** C_{return}

8.2.3 GET USER INSERT ACCESS ALGORITHM

The Get User Insert Access Algorithm defines the columns that a tenant's user is permitted to insert in a CTT or a VET. These columns are determined based on the intersection between the columns that are passed to this algorithm and the columns that are granted to be inserted by the user whose user ID is passed to this algorithm. This algorithm is invoked from Algorithm 5-1 of Chapter 5. Nevertheless, for simplicity, this access control algorithm was not included in Algorithm 5-1. The details of this algorithm are presented in Definition 8-3 and Algorithm 8-3.

Definition 8-3 (Get User Insert Access): T denotes a tenant ID. U denotes a tenant's user. B denotes a table name. UIA_{column} denotes a set of table columns that need to be inserted in a table. Q denotes the table type whether it is a CTT or a VET. B_{column} denotes a CTT or a VET columns. R_{return} denotes a set of user roles returned by calling GetUserRoles algorithm (Algorithm 8-1). C_{return} denotes a row matrix with 1 row and 2 columns that has two elements, the first element is $C_{return\ 0,0}$ that denotes the tenant's user SELECT clause attributes, and the second element is $C_{return\ 0,1}$ that denotes the access control part of the tenant's user Where clause. UIA_{return} denotes a set of table columns that a user is granted to insert in a table, which this algorithm returns.

Algorithm 8-3: GetUserInsertAccess (T, U, B, UIA_{column}, Q)

Input: T, U, B, UIA_{column} , and Q
Output: UIA_{return}
1. **if** Q = 'CTT' **then**
2. B_{column} ← retrieve the number of columns for a CTT from role_information_schema_column table using T, U, and B query filters
3. **else**
4. B_{column} ← retrieve number of columns for a VET from table_column ET using T, U, and B query filters
5. **end if**
6. R_{return} ← getUserRoles(T, U, B, Q) /* Algorithm 8-1 */
7. C_{return} ← getUserColumns (T, U, B, Q, R_{return}) /* Algorithm 8-2 */
8. UIA_{return} ← $UIA_{column} \cap C_{return\ 0,0}$
9. Return UIA_{return}

8.2.4 GET USER UPDATE ACCESS ALGORITHM

The Get User Update Access Algorithm defines the columns that a tenant's user is permitted to update in a CTT or a VET. These columns are determined based on the intersection between the columns that are passed to this algorithm and the columns that are granted to be updated by the user whose user ID is passed to this algorithm. This algorithm is invoked from Algorithm 5-2 of Chapter 5. Nevertheless, for simplicity, this access control algorithm was not included in Algorithm 5-2. The details of this algorithm are presented in Definition 8-4 and Algorithm 8-4.

Definition 8-4 (Get User Update Access): T denotes a tenant ID. U denotes a tenant's user. B denotes a table name. UUA_{column} denotes a set of table columns that need to be updated in a table. Q denotes the table type whether it is a CTT or a VET. B_{column} denotes a CTT or a VET columns. R_{return} denotes a set of user roles returned by calling GetUserRoles algorithm. C_{return} denotes a row matrix with 1 row and 2 columns which has two elements, the first one is $C_{return\ 0,0}$ that denotes the tenant's user SELECT clause attributes, and the second one is $C_{return\ 0,1}$ that denotes the access control part of the tenant's user Where clause. UUA_{update} denotes a set of table columns that derived from an intersection of two sets of table columns. The first

set is the table columns that passed to the algorithm to be update. The second set is the table columns that can be accessed by the user ID that passed to the algorithm. UUA_{return} denotes a row matrix with 1 row and 2 columns that has two elements. The first element is $UUA_{return\ 0,0}$ that stores into it the value of UUA_{update}. The second element is $UUA_{return\ 0,1}$ that stores into it the value of $C_{return\ 0,1}$.

Algorithm 8-4: GetUserUpdateAccess (T, U, B, UUA_{column}, Q)

Input: T, U, B, UUA_{column} , and Q
Output: UUA_{return}
1. **if** Q = 'CTT' **then**
2. B_{column} ← retrieve the number of columns for a CTT from role_information_schema_column table using T, U, and B query filters
3. **else**
4. B_{column} ← retrieve number of columns for a VET from table_column ET using T, U, and B query filters
5. **end if**
6. R_{return} ← getUserRoles(T, U, B, Q) /* Algorithm 8-1 */
7. C_{return} ← getUserColumns (T, U, B, Q, R_{return}) /* Algorithm 8-2 */
8. $UUA_{update} \leftarrow UUA_{column} \cap C_{return\ 0,0}$
9. $UUA_{return\ 0,0} \leftarrow UUA_{update}$
10. $UUA_{return\ 0,1} \leftarrow C_{return\ 0,1}$
11. **Return** UUA_{return}

8.2.5 GET USER DELETE ACCESS ALGORITHM

This access control algorithm defines the WHERE clause for a delete statement and returns its value, which determines the table rows that a tenant's user can delete from a CTT or a VET. This algorithm is invoked from Algorithm 5-3 of Chapter 5. Nevertheless, for simplicity, this access control algorithm was not included in Algorithm 5-3. The details of this algorithm are presented in Definition 8-5 and Algorithm 8-5.

Definition 8-5 (Get User Delete Access): T denotes a tenant ID. U denotes a tenant's user. B denotes a table name. Q denotes the table type, whether it is a CTT or a VET. B_{column} denotes a CTT or a VET columns. R_{return} denotes a set of user roles

returned by calling GetUserRoles algorithm. C_{return} denotes a row matrix with 1 row and 2 columns that has two elements, the first element is $C_{return\ 0,0}$ that denotes the tenant's user SELECT clause attributes, and the second element is $C_{return\ 0,1}$ that denotes the access control part of the tenant's user Where clause. UDA_{return} denotes a string of WHERE clause that this algorithm returns, which is used to filter table rows that a user is granted to delete.

Algorithm 8-5: GetUserDeleteAccess (T, U, B, Q)

Input: T, U, B, and Q
Output: UDA_{return}
1. **if** Q = 'CTT' **then**
2. B_{column} ← retrieve the number of columns for a CTT from role_information_schema_column table using T, U and B query filters
3. **else**
4. B_{column} ← retrieve number of columns for a VET from table_column ET using T, U and B query filters
5. **end if**
6. R_{return} ← getUserRoles(T, U, B, Q) /* Algorithm 8-1 */
7. C_{return} ← getUserColumns (T, U, B, Q, R_{return}) /* Algorithm 8-2 */
8. UDA_{return} ← $C_{return\ 0,1}$
9. **Return** UDA_{return}

8.2.6 GET USER QUERY ACCESS ALGORITHM

This access control algorithm defines the SELECT and the WHERE clauses and returns their values, by determining which columns and rows a user can access. These SELECT and WHERE clauses are used to construct the user's query statement that retrieves data from a CTT or a VET based on access grants assigned to the user. This algorithm is invoked from inside the algorithms (functions) of EETPS. However, for simplicity, this access control algorithm was not included in the algorithms of EETPS in Chapter 6. The details of this algorithm are presented in Definition 8-6 and Algorithm 8-6.

Definition 8-6 (Get User Query Access): T denotes a tenant ID. U denotes a tenant's user. B denotes a table name. S denotes a string of the SELECT clause

parameter. Q denotes the table type whether it is a CTT or a VET. B_{column} denotes a CTT or a VET columns. ∅ denotes an empty set. R_{return} denotes a set of user roles returned by calling GetUserRoles algorithm. C_{return} denotes a row matrix with 1 row and 2 columns that has two elements, the first element is $C_{return\ 0,0}$ that denotes the tenant's user SELECT clause attributes, and the second element is $C_{return\ 0,1}$ that denotes the access control part of the tenant's user Where clause. The values of C_{return} returned by calling GetUserColumns algorithm. UQA_{select} denotes a string of query SELECT clause. UQA_{return} denotes a row matrix with 1 row and 2 columns which has two elements, the first element is $UQA_{return\ 0,0}$ that stores into it the value of UQA_{select}, and the second element is $UQA_{return\ 0,1}$.

Algorithm 8-6: GetUserQueryAccess (T, U, B, S, Q)

Input: T, U, B, S, and Q
Output: UQA_{return}

1. **if** Q = 'CTT' **then**
2. B_{column} ← retrieve the number of columns for a CTT from role_information_schema_column table using T, U, and B query filters
3. **else**
4. B_{column} ← retrieve number of columns for a VET from table_column ET using T, U, and B query filters
5. **end if**
6. R_{return} ← getUserRoles(T, U, B, Q)
7. C_{return} ← getUserColumns (T, U, B, Q, R_{return})
8. **if** size of B_{column} = size of $C_{return\ 0,0}$ **then**
9. **if S =** ∅ **then**
10. UQA_{select} ← ∅
11. **else**
12. UQA_{select} ← **S**
13. **end if**
14. **else**
15. **if S =** ∅ **then**
16. UQA_{select} ← ∅
17. **else**
18. UQA_{select} ← S ∩ $C_{return\ 0,0}$
19. **end if**
20. **end if**
21. $UQA_{return\ 0,0}$ ← UQA_{select}

22. $UQA_{return\ 0,1} \leftarrow C_{return\ 0,1}$
23. **Return** UQA_{return}

8.3 PERFORMANCE EVALUATION

After the EETACS was developed, it was used by the EETPS. Two types of experiments are carried out to verify the practicability of applying the EETACS on the EETPS. In these experiments, the response times of retrieving the tenant's columns and rows are evaluated by invoking the Single Table function (Algorithm 6-1) of EETPS, which gets the tenant's user access grants from EETACS.

8.3.1 EXPERIMENTAL DATA SET AND SETUP

The EETPS has designed and developed to serve multiple tenants on one instance application. However, in this chapter the aim of the experiments is evaluating the performance after applying the EETACS method on the EETPS for one tenant. A number of experiments are executed for one tenant, because, in the multi-tenant database the data of each tenant's user is isolated in a table partition. Thus, these experiments can evaluate the effectiveness of retrieving data for each single tenant's user from the multi-tenant database. In these experiments, one machine is used and the Single Table function of the EETPS (presented in Chapter 6 in Section 6.2.1) is invoked to retrieve 100 of rows from the 'product' VET that is shown in Figure 8-5. There are 200,000 rows stored in this table that belongs to a tenant whose 'tenant_id' equals 1000, and the 'db_table_id' of this table equals 16. All the queries implemented in these experiments are filtered by 'tenant_id', 'db_table_id', and other filters specified in the below experiments. These experiments are divided in two types sharing the details of this data set, which are listed below, and the queries of these experiments are shown in Table 8-2.

1) Accessing Data from Table Columns Experiment (Exp.8-1): In this experiment, Query 8-1 (Q8-1) and Query 8-2 (Q8-2) are executed to benchmark the query execution time difference between a tenant's user who can access data from all

columns of a table by executing Q8-1, and another tenant's user who can access data from only three out of eight columns of the same table by executing Q8-2.

2) Accessing Data from Table Rows Experiment (Exp. 8-2): In this experiment Query 8-1 (Q8-1) and Query 8-3 (Q8-3) are executed to benchmark the query execution time difference between a tenant's user who can access data from all the table rows by executing Q8-1, and another tenant's user who can access 10% of the table data that equals approximately 20,000 rows by executing Q8-3.

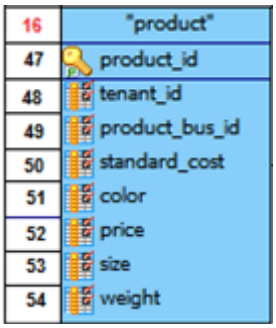


Figure 8-5: The table structure of the 'product' table

The EETACS 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, the database and the application server are deployed on the same PC. The operating system is Windows 7 Home Premium, with Intel Core i5 2.40GHz CPU, 8 GB of RAM memory, and 500 GB of hard disk storage.

8.3.2 Experimental Results

1) **Accessing Data from Table Columns Experimental Results**: Typically, users are granted access to table columns from the application level, because, in a single-tenant database, the tenants' users are not granted database access on the column level. While, the EETACS grants the tenants' users database accesses on the column

level. This capability reduces the query execution time in the multi-tenant database. The experimental study of Exp.8-1shows that the execution time of Q8-2 for a user who can access fewer numbers of columns of a table is less than the execution time of Q8-1 for a user who can access all of the table columns. The details results of this experiment are shown in Figure 8-6 and Table 8-1.

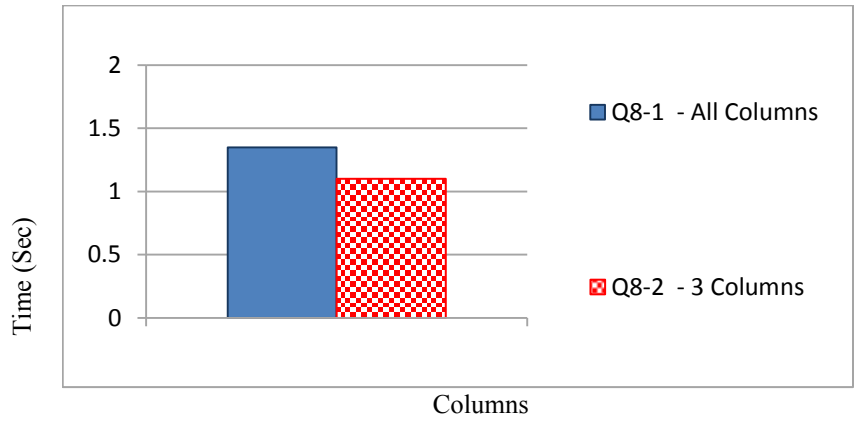


Figure 8-6: Accessing data from the table columns experiment (Exp.8-1)

2) **Accessing Data from Table Rows Experimental Results**: Normally, the tenants' users cannot be granted a database access to table rows from the database. While, the EETACS grants the tenants' users database accesses on the row level. This capability reduces the query execution time in the multi-tenant database. The experimental study of Exp.8-2 shows that the execution time of Q8-3 for a user who can access a percentage of a table rows is less than the execution time of Q8-1 for a user who can access all the table rows. The details results of this experiment are shown in Figure 8-7 and Table 8-1.

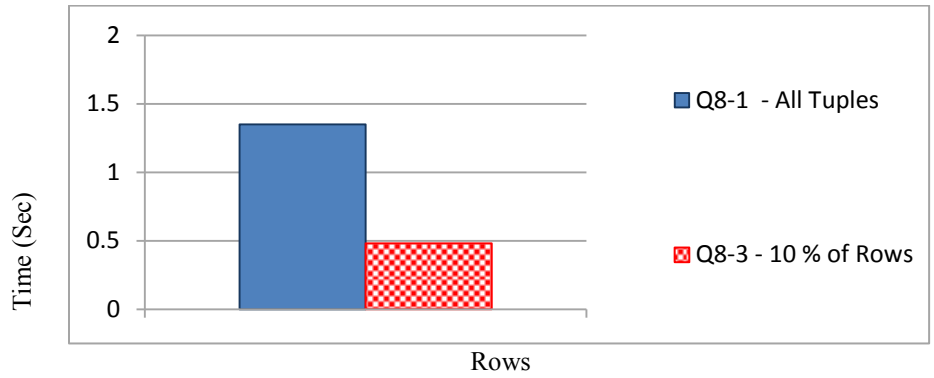


Figure 8-7: Accessing data from the table rows experiment (Exp.8-2)

TABLE 8-1: The query execution times of Exp.8-1 and Exp.8-2

Experiment	Query executed	Time in seconds
Exp. 8-1	Q8-1	1.35
Exp. 8-1	Q8-2	1.10
Exp. 8-2	Q8-1	1.35
Exp. 8-2	Q8-3	0.48

Table 8-2: The experiments queries

Query No.	Query Details
Q8-1	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 = 16 AND tr.table_row_id IN (SELECT distinct tr.table_row_id FROM **table_row** tr WHERE tr.tenant_id = 1000 AND tr.db_table_id = 16 AND tr.table_column_id = 50 AND (cast(value as numeric) > '9000')) ORDER BY 3,4 LIMIT 800 OFFSET 0;
Q8-2	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 = 16 AND t**r.table_column_id in (47,48,49)** AND tr.table_row_id IN (SELECT distinct tr.table_row_id FROM **table_row** tr WHERE tr.tenant_id = 1000 AND tr.db_table_id = 16 AND tr.table_column_id = 50 AND (cast(value as numeric) > '9000')) ORDER BY 3,4 LIMIT 800 OFFSET 0 ;
Q8-3	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 = 16 AND tr.table_row_id IN (SELECT distinct tr.table_row_id FROM **table_index** tr WHERE tr.tenant_id = 1000 AND tr.db_table_id = 16 AND tr.table_column_id = 50 AND (cast(row_value as numeric) > '9000')) ORDER BY 3,4 LIMIT 800 OFFSET 0;

8.4 SUMMARY

A multi-tenant access control service called EETACS is proposed in this chapter. This service allows each tenant in a multi-tenant database to have several users with different types of access grants to access the tenant's data. The concept of retrieving data from the multi-tenant database is slightly different from the single-tenant database that does not differentiate between the data of different tenants' users. While, the data of the multi-tenant database should be partitioned to differentiate between the data that is owned by each single tenant, and by accessing columns and rows granted to the tenants' users based on a number of groups or roles assigned to them. In this chapter, the data architecture of the EET access control tables and the EET access grants were proposed. Two types of experiments were carried out to verify the practicability of applying the proposed access table columns and rows grants on the EETPS. The first experiment verified that the cost of executing a query for a user who can access some numbers of columns of a VET is less than the cost of executing the same query for a user who can access all the VET columns. The second experiment verified that the cost of executing a query for a user who can access a percentage of a VET rows is less than the cost of executing the same query for a user who can access all of the VET rows. This chapter overcomes the multi-tenant access control issues, including (1) isolating the tenants' data and the tenants' user data in EET multi-tenant database by dividing it into partitions; (2) optimizing the query execution by considering the number of rows or columns that are accessed by a tenant's user, and generating a query structure different from the structure of retrieving all the table rows, or all the tables column; (3) fulfilling different access controls requirements for multiple tenants who are using a single database schema.

CHAPTER 9

CONCLUSIONS AND FUTURE RESEARCH

This chapter summarizes the achievements of this thesis and presents possible future research and work directions.

9.1 CONCLUSIONS

This study aims to address the challenges and issues of multi-tenant database that accommodates data for multiple tenants in one single database schema. As identified in Chapter 1, such a multi-tenant schema should be highly secured, optimized, configurable, and extendable during the runtime execution to fulfil the applications' requirements of different tenants. These multi-tenant database schema capabilities are not supported by traditional RDBMS, and to enable RDBMS to support such capabilities is a significant challenge. This study proposed a novel configurable multi-tenant database schema design called Elastic Extension Tables (EET), and developed a prototype framework to implement EET schema in a RDBMS and simplify the data access of software applications in general and multi-tenant SaaS and Big Data applications in particular. This study has made significant contributions from both theoretical and practical aspects in the area of multi-tenant database, as detailed below:

- Proposing EET single multi-tenant database schema that consists of CTT, ET, and VET. Multiple tenants use this schema, which allows each tenant to create his own virtual database schema that includes the required number of tables and columns, rows, virtual database relationships with any of CTTs or VETs, and to assign suitable data types and constraints for columns during multi-tenant application run-time execution. It has a flexible way of creating database schemas for multiple tenants, by extending a business domain database that is based on a RDBMS, or by creating a tenant's business domain database from the scratch. It improves the multi-tenant database performance by avoiding NULL values, assigning primary keys to unique virtual columns, providing indexes to virtual table columns, and storing BLOB and CLOB data types in separate designated tables. Moreover, it allows allows tenants to choose from three database models: Multi-tenant Relational Database, Integrated Multi-tenant Relational Database and Virtual Relational Database, and Virtual Relational Database. This capability is not implemented in any of the existing multi-tenant database schema designs yet. In using this capability, the service provider can accommodate a huge number of virtual tables beside the physical tables of the business domain data that he offers, by allowing a large number of tenants to create virtual tables and populate these tables with their data in the eight ETs of EET. Furthermore, it allows to store different data types of Big Data including structured, semi-structured, and unstructured data that are collected from various online sources of information. The Big Data 4Vs that stated in Chapter 2 are fulfilled in EET multi-tenant schema as follows. *Firstly*, EET can accommodate a large volume of data for multiple tenants in a single schema, and the volume of the data is reduced by avoiding storing any tenants' redundant data in EET. *Secondly*, the velocity and the effectiveness of accessing data from EET verified in the experiments that carried out in this thesis. *Thirdly*, EET multi-tenant schema allows to store various data types as stated above. *Fourthly*, the EET stores consistence data; because the structure of its physical (CTTs) and virtual (VETs) are relational

structures. Further, this multi-tenant schema is secure because it has an access control data architecture that permits each tenant in a multi-tenant database to have a number of users with different types of grants to access the tenant's table columns and rows. To evaluate the performance of EET, this study compared the performance of inserting, updating, retrieving, and deleting data from EET and UTSM by using a generated test dataset, which is similarly populated in both schemas. The experimental study result shows an improvement when retrieving, updating and deleting data from EET over the UTSM. Especially when retrieving data from EET, it is much faster than UTSM. However, the execution time of inserting rows in EET is slightly slower than UTSM. As reviewed and concluded in the Literature Review (Chapter 2), the Universal Table that is used in UTSM, is considered the optimal schema design for multi-tenant applications. Thus, the UTSM was chosen to compare its performance with EET performance. Accordingly, this experimental study makes EET the optimal schema for implementing multi-tenant databases and multi-tenant SaaS and Big Data applications.

- Proposing conceptual framework architecture and developing it based on the EET multi-tenant schema. This framework simplifies and speeds up the development of multi-tenant database applications. It allows database service providers to create a single database application that supports multiple tenants on the same software and hardware infrastructure. Moreover, it overcomes multi-tenant database challenges from technical and business perspectives and reduces the TCO from the tenants' perspective, by avoiding them from writing SQL queries and backend data management code. Alternatively, they access the APIs of this framework that manage the tenant's data and retrieve simple and complex queries. While, from the database service provider perspective, the EET reduces the ongoing operational costs, through providing a database self-service that configures and manages the tenants' data by the tenants themselves, rather than the database service provider. This database solution is suitable to be used by the

tenants' developers, to store and access the tenants' data from the cloud to build their applications, or integrate this data with other applications or online data sources without spending much time and efforts on managing their database. Consequently, the database layer that the EET framework provides can be used as a base to build software applications in general and SaaS and Big Data applications in particular. In the service layer of this framework prototype, four types of services were developed, which provide functions that allow tenants to access, manage, and retrieve their data by calling the functions of the services that are listed below:

1) A multi-tenant data management service called EETSHS, which allows tenants to create VETs and create VETs' columns, rows, relationships, primary keys, indexes, and other columns constraints. In addition, tenants can create CTTs' rows, and database relationships between CTTs and VETs, whereas the rest of the CTT database operations, including creating CTTs, CTTs' columns, database relationships between two CTTs, primary keys, indexes, and other columns constraints are managed from a traditional RDBMS instead of EETSHS. This service ensures a high level of multi-tenant data quality, configurability, consistency, accessibility, and manageability. In this study, three algorithms that manage CTTs and VETs rows were developed, and several experiments were performed using these algorithms to measure the feasibility and the effectiveness of managing data using this service that based on EET. The experimental results show that the query execution time of inserting and updating rows in the tenants' CTTs is slightly faster than in the tenants' VETs. The increase in the query execution time of VET is not significant compared to the benefits that this service brings to SaaS and Big Data applications, in addition, these data operations are not equally significant as when retrieving data from EETPS. The experimental results of deleting rows from the tenants' CTTs are approximately four times slower than from the tenants' VETs. This increase

in the query execution time occurs in CTTs that are the traditional physical tables of EET, due to the process of deleting a CTT row is more complicated than VET. In general, these experimental results make this service and EET schema a suitable candidate for the management of multi-tenant data.

2) A multi-tenant proxy service called EETPS, which integrates, generates, and executes the tenants' queries by using a codebase solution that converts multi-tenant queries into traditional database queries and execute them in a RDBMS. EETPS has three objectives. Firstly, it allows tenants to choose from the three database models of EET. Secondly, it allows each single tenant to extend his database schema, by extending a business domain database schema that based on a traditional RDBMS during the application's runtime execution. Thirdly, it avoids efforts of writing SQL queries and backend data management code by utilizing the service functions that execute simple and complex queries including join operations, filtering on multiple properties, and filtering of data based on subqueries results. In this study, five algorithms for five functions of EETPS were developed, and five experiments were carried out to verify the effectiveness of EETPS algorithms. These experiments were classified according to the complexity of the queries that are used in the experiments, which compare the response time of retrieving data from CTTs, VETs, and both CTTs and VETs. The result of these experiments shows that most of the experiments that performed by calling functions from the EETPS to retrieve data from VET and CTT-and-VET outperform the performance of CTT (traditional physical tables). These results verify the practicability and the effectiveness of using EETPS and EET multi-tenant database schema and its three database models.

3) A multi-tenant query optimizer service called EETQOS, which estimates the cost of different query execution plans to determine the optimal query execution plan. Three types of methods are used in this service to optimize the data retrieval from EET multi-tenant schema, including the query access

control, index selection, and table row selection. Such a query optimizer service reduces the query execution time of EETPS that is accessing data from EET multi-tenant schema. In this study, six types of experiments were carried out to verify the practicability of implementing the EETQOS on the EETPS. These experiments conclude that the cost of executing a query from a VET when using all the indexes of that VET is high in comparison with the cost of executing the same query without using any index. However, the least query execution cost was recorded when a custom index with a percentage of a VET indexes is used to retrieve table rows from the same VET. Moreover, these experiments confirm the effectiveness of using the data structure of EET multi-tenant schema that is the base storage of EETPS and EETQOS, and the benefits of EETQOS query execution plans for EETPS and EET multi-tenant schema.

4) A multi-tenant access control service called EETACS, which allows each tenant in a multi-tenant database to have several users with different types of access grants to access the tenant's data. The concept of retrieving data from the multi-tenant database is different from the single-tenant database that does not differentiate between the data of different tenants' users. While, the data of the multi-tenant database is partitioned by differentiating between the data that is owned by each single tenant, and by accessing columns and rows granted to the tenants' users based on a number of groups or roles assigned to them. In this study, the data architecture of the EET access control, and the EET access grants were proposed. Moreover, six access control algorithms were developed, and two types of experiments were carried out to verify the practicability of applying the proposed access table columns and rows grants on the EETPS. The first experiment verifies that the cost of executing a query for a user who can access some numbers of columns of a VET is less than the cost of executing the same query for a user who can access all the VET columns. The second experiment verifies that the cost of executing a query

for a user who can access a percentage of a VET rows is less than the cost of executing the same query for a user who can access all of the VET rows.

In conclusion, this study has achieved its objectives that stated in Chapter 1. The theoretical development and practical advancements of EET multi-tenant schema and EET framework architecture create a great opportunity to develop a multi-tenant framework that can be used as an intermediate database layer between RDBMS and software applications in general, and multi-tenant applications in particular. This database layer avoids tenants from writing SQL queries and backend data management code, alternatively, they call functions from EET framework to manage and access their data.

9.2 FUTURE RESEARCH

Given the time constraints of this study, the scope of the system development was limited to how to store, access, manage, retrieve, and optimize each single tenant's data using EET multi-tenant schema and EET framework prototype. The EET framework prototype can be extended by adding other artefacts to its architecture. In future research, a large number of future research directions and work directions can be considered for this study. Five of these future research directions and two of the future work directions are listed below:

Future research directions:

(i) In this thesis, the EET multi-tenant schema and EET framework prototype developed to serve multiple tenants in one instance application. Such an application should be scalable and highly available to accommodate a huge number of data for a large number of tenants. However, the focus of this thesis was on developing and evaluating the main data operations of EET multi-tenant schema using EET framework on one server because before considering scale-up or scale-out this solution to optimize or evaluate its performance, the performance should be examined in one server.

Therefore, the scalability of EET multi-tenant schema using EET framework is one of the future research directions of this study.

(ii) The EETPS functions of the EET framework prototype have the capabilities of retrieving data from CTTs or VETs by using a number of query options that stated in Chapter 6, one of the future research directions of this study is to add GROUP BY and ORDER BY query options to EETPS. Another future research direction of EETPS is to improve the join operation functions to handle more than two tables, instead of only handling two tables.

(iii) Optimizing the EETQOS by adding more methods to determine the optimal query execution plans, and caching the frequently used queries effectively to speed up the EETQOS processing time, and reduce the consumption of the EET database resources. Subsequently, these two enhancements of EETQOS will optimize the query execution time of EETPS, as long as it consumes EETQOS to optimize its query execution time.

(iv) In this study, the empirical tests are conducted on the structured data, but the empirical tests of semi-structured, and unstructured data are out of the scope of this thesis. For two reasons, first, as reviewed in chapter 2, storing and retrieving data in XML files (semi-structured data) has the highest response time between the reviewed seven multi-tenant database schema designs. therefore, the semi-structured data can be stored in EET, but it is not recommended to be used as a storage for multiple tenants. Second, there are many techniques for storing and retrieving different data types of Big Data. Comparing all of these techniques with EET is hard, complex, and time-consuming task that is hard to be achieved during the time frame and size of one PhD thesis. Thus, comparing EET with other data types and other techniques is one of the future research directions of this study.

(v) As concluded in the Literature Review chapter, the Universal Table that is used in UTSM, is considered the optimal schema for multi-tenant applications. Therefore, this study measured the feasibility and effectiveness of EET by comparing it with UTSM. However, comparing EET with other existing multi-tenant database schema designs that are based on RDBMS can be considered as a future research direction of this study.

Future work directions (practical implementation work):

(i) The APIs artefacts were introduced in the EET framework architecture to show a complete scenario on how EET framework works. The implementation of these artefacts is not part of the objective of this study, and it is out of this thesis scope. Nevertheless, it is one of the future work directions of this study.

(ii) Introducing two new artefacts to the EET framework architecture, including (1) database management user interface, which allows each single tenant to create, manage, organize, and administrate his data. This user interface can consume the functions of the EETSHS through using the EET Data Management API to perform the data management tasks. (2) access control user interface, which allows each tenant to manage his users and grant them permissions to access the tenant's data. This user interface can consume the functions of EETACS through using the EET Data Retrieval API to perform the access control tasks. These two artefacts can be considered as future work directions for this study.

ABBREVIATIONS

Abbreviations	Descriptions
ASP	Application Service Provider
BLOB	Binary Large Object
CCVS	Covert CTT to VET Structure
CLOB	Character Large Object
CPVR	Creating Physical and Virtual Rows
CRM	Customer Relationship Management
CTT	Common Tenant Tables
CTTDAO	Common Tenant Tables Data Access Object
DAC	Discretionary Access Control
DAO	Data Access Object
DPVR	Deleting Physical and Virtual Rows
EET	Elastic Extension Tables
EETACS	Elastic Extension Tables Access Control Service
EETDAO	Elastic Extension Tables Data Access Object
EETPS	Elastic Extension Tables Proxy Service
EETQOS	Elastic Extension Tables Query Optimizer Service
EETSHS	Elastic Extension Tables Schema Handler Service
ET	Extension Tables
GTRQ	Get Table Row Query
HR	Human Resources
IaaS	Infrastructure as a Service
ICT	Information and Communications Technology
IT	Information Technology
LBAC	Label-Based Access
LJQ	Left Join Query
MAC	Mandatory Access Control
NoSQL	Not Only SQL
ORM	Object Relational Mapping
OTMQ	One To Many Query
PaaS	Platform as a Service

QoS	Quality of Service
RBAC	Role Based Access Control
RCAC	Row and Column Access Control
RDBMS	Relational Database Management Systems
SaaS	Software as a Service
SQL	Structured Query Language
SRA	Store Rows in Array
STQ	Single Table Query
TCO	Total Cost of Ownership
TTQ	Targeted Tables Query
IQ	Individual Query
UDD	Universal Data Dictionary
UPVR	Updating Physical and Virtual Rows
UQ	Union Query
URI	Uniform Resource Identifier
UTSM	Universal Table Schema Mapping
VET	Virtual Extension Tables
XML	Extensible Markup Language

BIBLIOGRAPHY

Agrawal, D., Das, S. & Abbadi, A.E. 2010, 'Big data and cloud computing: new wine or just new bottles?',*Proceedings of the VLDB Endowment*, vol. 3, no.2, pp. 1647-1648

Agrawal, D., Das, S. & Abbadi, A.E. 2012, 'Data management in the cloud: challenges and opportunities', *Synthesis Lectures on Data Management*, vol. 4, no. 6, pp. 1-138.

ALzain, M.A. & Pardede, E. 2011, 'Using multi shares for ensuring privacy in Database-as-a-Service', *The 44th Hawaii International Conference on System Sciences*, IEEE, pp. 1-9.

Ambrose, W., Dagland, N. & Athley, S. 2010, 'Cloud computing: Security risks, SLA, and trust', Bachelor thesis, Jönköping University, Sweden.

Arnold, D., Diniro, S., Lee, V., Musker, S., & Woods, J. A. 2012, 'Row and column access control', *Unleashing DB2 10 for Linux, UNIX, and Windows*, vol. 10, no. 1, pp.65-86.

Aulbach, S. 2011, 'Schema flexibility and data sharing in multi-Tenant databases'. PhD thesis, Technical University of Munich, Germany.

Aulbach, S., Grust, T., Jacobs, D., Kemper, A. & Rittinger, J. 2008, 'Multi-tenant databases for software as a service: Schema-mapping techniques', *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ACM, Vancouver, Canada, pp. 1195-1206.

Aulbach, S., Jacobs, D., Kemper, A. & Seibold, M. 2009, 'A comparison of flexible schemas for software as a service', *Proceedings of the 2009 ACM SIGMOD international conference on Management of data*, ACM, Rhode Island, USA, pp. 881-888.

Bezemer, C.P., Zaidman, A., Platzbeecker, B., Hurkmans, T. & t Hart, A. 2010, 'Enabling multi-tenancy: An industrial experience report', *Software Maintenance (ICSM), 2010 IEEE International Conference on*, IEEE, Timisoara, Romania, pp. 1-8.

Bobrowski, S. 2011, 'Optimal multitenant designs for cloud apps',*2011 IEEE 4th International Conference on Cloud Computing*, IEEE, Washington, USA, pp. 654-659.

Brian, O. Brunschwiler, T., Christ, H., Falsafi, B., Fischer, M., Grivas, S. G., Giovanoli, C., Gisi, R. E., Gutmann, R., Kaiserswerth, M., Kundig, M., Leinen, S., Muller, W., Oesch, D., Redli, M., Rey, D., Riedl, R., Schar, A., Spichiger, A., Widmer, U., Wiggins, A., Zollinger, M. & Kaiserswerth, M.2012,'Cloud Computing', White Paper SATW.

Brodersen, K., Thomas M.R., Matthew S.M., Mingte J.C., & Anil A. 2004, *Database access method and system for user role defined access*, US Patent 6732100.

Burno, F. 2006, 'Exeuting an IP Protection Strategy in a SaaS Environment', *Contract Management*,EBSCO, vol. 46, no. 7, pp. 14-16.

Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J. & Brandic, I. 2009, 'Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility', *Future Generation computer systems*, vol. 25, no. 6, pp. 599-616.

Carolan, J., Gaede, S., Baty, J., Brunette, G., Licht, A., Remmell, J., Tucker, L. & Weise, J. 2009, 'Introduction to cloud computing architecture', White Paper, 1st edn, Sun Micro Systems Inc.

Cattell, R. 2011, 'Scalable SQL and NoSQL Data Stores', *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12-27.

Chamberlin, D.D. & Boyce, R.F. 1974, 'SEQUEL: a structured English query language', *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, ACM, Ann Arbor, USA, pp. 249-264.

Chang, V., Walters, R.J. & Wills, G. 2013, 'The development that leads to the Cloud Computing Business Framework', *International Journal of Information Management*, vol. 33, no. 3, pp. 524-538.

Chengtong, L., Qing, L., Zhou, L., Junjie, P., Wu, Z. & Tingting, W. 2010, 'PaaS: A revolution for information technology platforms', *Educational and Network Technology (ICENT 2010)*, IEEE, Qinhuangdao, China, pp. 346-349.

Chong, F. & Carraro, G. 2006, 'Architecture strategies for catching the long tail', White Paper, MSDN Library, Microsoft Corporation.

Chong, F. 2006, 'Multi-tenancy and virtualization', MSDN Blogs, viewed 27 June 2014,<http://blogs.msdn.com/b/fred_chong/archive/2006/10/23/multi-tenancy-and-virtualization.aspx>.

Chong, F., Carraro, G. & Wolter, R. 2006, 'Multi-tenant data architecture', White Paper, MSDN Library, Microsoft Corporation.

Codd, E.F. 1970, 'A relational model of data for large shared data banks', *Communications of the ACM*, vol. 13, no. 6, pp. 377-387.

Codd, E.F. 1985a, 'Does your DBMS run by the rules?', *Computer World*, vol. 21, p. 11.

Codd, E.F. 1985b, 'Is your DBMS really relational', *Computer World*, vol. 14.

Dash, D., Alagiannis, I., Maier, C. & Ailamaki, A. 2010, 'Caching all plans with just one optimizer call', *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, IEEE, Long Beach, USA, pp. 105-110.

Date, C.J. 1990, 'An introduction to database systems', *Reading, MA: Addison-wesley*, vol. 7.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. & Vogels, W. 2007, 'Dynamo: amazon's highly available key-value store', *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 205-220.

Demchenko, Y., Grosso, P., de Laat, C. & Membrey, P. 2013, 'Addressing big data issues in scientific data infrastructure', *Collaboration Technologies and*

Systems (CTS), 2013 International Conference on, San Diego, USA, pp. 48-55.

Dikaiakos, M.D., Katsaros, D., Mehra, P., Pallis, G. & Vakali, A. 2009, 'Cloud computing: distributed internet computing for IT and scientific research', *Internet Computing*, IEEE, vol. 13, no. 5, pp. 10-13.

Dimovski, D. 2013, 'Database management as a cloud-based service for small and medium organizations', Master thesis, Masaryk University Brno, Czech Republic.

Dobosz, R. 2014, 'An investigation of the impact of big data on bioinformatics software', Master thesis, Trent University, Ontario, Canada.

Domingo, E.J., Nino, J.T., Lemos, A.L., Lemos, M.L., Palacios, R.C. & Berbi, J.M.G. 2010, 'CLOUDIO: A cloud computing-oriented multi-tenant architecture for business information systems', *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, IEEE, Miami, USA, pp. 532-533.

Du, J., Wen, H. & Yang, Z. 2010, 'Research on data layer structure of multi-tenant e-commerce system', *Industrial Engineering and Engineering Management (IE&EM), 2010 IEEE 17Th International Conference on*, IEEE, Xiamen, China, pp. 362-365.

Farahani, M., Sharifnejad, M. & Sharifi, M. 2006, 'An enhanced tuple routing strategy for adaptive processing of continuous queries', *Information and Communication Technologies, 2006. ICTTA'06. 2nd*, vol. 2, pp. 3146-3150.

Ferraiolo, D. F. & Kuhn, D. R. 1992, 'Role-based access controls', *15th National Computer Security Conference*, Baltimore, USA, pp. 554-563.

Fiaidhi, J., Bojanova, I., Zhang, J. & Zhang, L.-J. 2012, 'Enforcing multitenancy for cloud computing environments', *IT professional*, vol. 14, no. 1, pp. 16-18.

Foping, F.S., Dokas, I.M., Feehan, J. & Imran, S. 2009, 'A new hybrid schema-sharing technique for multitenant applications', *Digital Information Management, 2009. ICDIM 2009. Fourth International Conference on*, IEEE, pp. 210-215.

Gorti, I., Shiri, N. & Radhakrishnan, T. 2013, 'A flexible data model for multi-tenant databases for Software as a Service', *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, IEEE, pp. 1059-1066.

Hacigümüş, H., Iyer, B., Li, C. & Mehrotra, S. 2002, 'Executing SQL over encrypted data in the database-service-provider model', *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, ACM, Madison, USA, pp. 216-227.

Heng, L., Dan, Y. & Xiaohong, Z. 2012, 'Survey on multi-tenant data architecture for SaaS', *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 6.

Hoch, F., Kerr, M. & Griffith, A. 2001, 'Software as a service: Strategic backgrounder', White Paper, Software & Information Industry Association (SIIA).

Hoogvliet, M. T. 2008, 'SaaS interface design', Bachelor Thesis, Rotterdam University, Holland.

Hudli, A.V., ShiHudli, A.V., Shivaradhya, B. & Hudli, R.V. 2009, 'Level-4 SaaS applications for healthcare industry', *Proceedings of the 2nd Bangalore annual compute conference*, ACM, Bangalore, India.

Indrawan-Santiago, M. 2012, 'Database research: Are we at a crossroad? Reflection on NoSQL', *Network-Based Information Systems (NBiS), 2012 15th International Conference on*, Melbourne, Australia, pp. 45-51.

Jansen, S., Houben, G.-J. & Brinkkemper, S. 2010, 'Customization realization in multi-tenant web applications: Case studies from the library sector', *The 10th International Conference on Web Engineering (ICWE)*, Springer, Vienna, Austria, pp. 445-459.

Ju, J., Wang, Y., Fu, J., Wu, J. & Lin, Z. 2008, 'Research on key technology in SaaS', *Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on*, IEEE, Kuala Lumpur, Malaysia, pp. 384-387.

Kim, G.-H., Trimi, S. & Chung, J.-H. 2014, 'Big-data applications in the government sector', *Communications of the ACM*, vol. 57, no. 3, pp. 78-85.

Kwok, T., Thao, N. & Linh, L. 2008, 'A Software as a Service with multi-tenancy support for an electronic contract management application', *Services Computing, 2008. SCC '0*8. IEEE International Conference on, IEEE, vol. 2, pp. 179-186.

Landy, G.K. 2008, *The IT / digital legal companion: A comprehensive business guide to software, IT, internet, media and IP law*, 1st edn, Syngress, USA.

Lang, B., Foster, I., Siebenlist, F., Ananthakrishnan, R. & Freeman, T. 2009, 'A flexible attribute based access control method for grid computing', *Journal of Grid Computing*, vol. 7, no. 2, pp. 169-180.

Lazarov, V. 2007, 'Comparison of different implementations of multi-Tenant databases', Bachelor thesis, Technical University of Munich, Munich, Germany.

Leavitt, N. 2010, 'Will NoSQL databases live up to their promise?', Computer, vol. 43, no. 2, pp. 12-14

Liao C.-F., Chen, K. & Chen, J.- J. 2012, 'Toward a tenant-aware query rewriting engine for universal table schema-mapping', *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, IEEE, Taipei, Taiwan, pp. 833-838.

Lith, A. & Mattsson, J. 2010, 'Investigating storage solutions for large data', Master thesis, Chalmers University of Technology, Goteborg, Sweden.

Liu, G. 2010, 'Research on independent SaaS platform', *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, IEEE, Chengdu, China, pp. 110-113.

Lopes, J. 2009, 'SaaS (software as a service) – models and infra-structures', Master thesis, Universidade Tecnológica de Lisboa, Portugal.

Louridas, P. 2010, 'Up in the air: Moving your applications to the cloud', *IEEE software*, vol. 27, no. 4, pp. 6-11.

Maier, D. & Ullman, J.D. 1983, 'Maximal objects and the semantics of universal relation databases', *ACM Transactions on Database Systems*, vol. 8, no. 1, pp. 1-14.

Martinez, C.G. 2012, 'Study of resource management for Multitenant Database Systems in Cloud Computing', Master thesis, University of Colorado, Boulder, USA.

Mateljan, V., Cisic, D. & Ogrizovic, D. 2010, 'Cloud database-as-a-service (DaaS) - ROI', *MIPRO, 2010 Proceedings of the 33rd International Convention*, Opatija, Croatia, pp. 1185-1188.

Mathew, R. & Spraetz, R. 2009, 'Test automation on a SaaS platform', *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, Denver, USA, IEEE, pp. 317-325.

Menken, I. & Blokdijk, G. 2009, *Saas and web applications specialist level complete certification kit-software as a service study guide book and online course*, Emereo Pty Ltd, UK.

Mietzner, R., Unger, T., Titze, R. & Leymann, F. 2009a, 'Combining different multi-tenancy patterns in service-oriented applications', *Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International*, IEEE, pp. 131-140.

Mietzner, R., Metzger, A., Leymann, F. & Pohl, K. 2009b, 'Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications', *Principles of Engineering Service Oriented Systems, 2009. PESOS 2009. ICSE Workshop on*, IEEE, Vancouver, Canada, pp. 18-25.

Mohammed, S. & Fiaidhi, J. 2010, 'The Roadmap for Sharing Electronic Health Records: The Emerging Ubiquity and Cloud Computing Trends', *Future Generation Information Technology*, Springer, Jeju Island, Korea pp. 27-38.

Nitu 2009, 'Configurability in SaaS (software as a service) applications', *Proceedings of the 2nd Annual Conference on India Software Engineering Conference*, ACM, Pune, India, pp. 19-26.

Pritchett, D. 2008, 'Base: An acid alternative', *Queue*, vol. 6, no. 3, pp. 48-55.

Ratametha, T. & Veeragandam, M. 2008, 'CRM: Software as a service versus on-premise-benefits and drawbacks', Master thesis, Lund University, Sweden.

Raza, B., Mateen, A., Sher, M., Awais, M.M. & Hussain, T. 2010, 'Autonomic view of query optimizers in database management systems', *Software Engineering Research, Management and Applications (SERA), 2010 Eighth ACIS International Conference on*, IEEE, pp. 3-8.

Ren, K., Wang, C. & Wang, Q. 2012, 'Security challenges for the public cloud', *IEEE Internet Computing*, vol. 16, no. 1, pp. 69-73.

Sakr, S., Liu, A., Batista, D.M. & Alomari, M. 2011, 'A Survey of large scale data management approaches in cloud environments', *IEEE Communications Surveys & Tutorials*, vol. 13, no. 3, pp. 311-336.

Salesforce, 2013, 'Record-level access: Under the hood', White paper, salesforce.com, inc.

Schiller, O., Schiller, B., Brodt, A. & Mitschang, B. 2011, 'Native support of multi-tenancy in RDBMS for software as a service', *Proceedings of the 14th international conference on extending database technology*, ACM, Uppsala, Sweden, pp. 117-128.

Shao, Q. 2011, 'Towards effective and intelligent multi-tenancy SaaS', PhD Thesis, Arizona State University, USA.

Shuai, Z., Shufen, Z., Xuebin, C. & Xiuzhen, H. 2010, 'Cloud computing research and development trend', *Future Networks, 2010. ICFN '10. Second International Conference on*, Sanya, China, pp. 93-97.

Simmonds, T. 2013, 'Teaching database administration in the world of big data and small budgets', *Friday 5th July 2013 University of Sunderland*, Sunderland, UK, pp. 45-51.

Stonebraker, M. 2010, 'SQL databases v. NoSQL databases', *Communications of the ACM*, vol. 53, no. 4, pp. 10-11.

Takabi, H., Joshi, J.B. & Ahn, G.-J. 2010, 'Security and privacy challenges in cloud computing environments', *IEEE Computer and Reliability Societies*, vol. 8, no. 6, pp. 24-31.

Tweed, R. & James, G. 2010, 'A universal nosql engine, using a tried and tested technology', White Paper, Creative Commons Attribution CC-BY.

Wang, Z.H., Guo, C.J., Gao, B., Sun, W., Zhang, Z. & An, W.H. 2008, 'A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing', *e-Business Engineering, 2008. ICEBE'08. IEEE International Conference on*, IEEE, Xi'an, China, pp. 94-101.

Weissman, C., Dave M., Simon W. & Paul N. 2012, *Multi-tenant database system*, US Patent 8280874.

Weissman, C.D. & Bobrowski, S. 2009, 'The design of the force.com multitenant internet application development platform', *Proceedings of the 2009 ACM SIGMOD international conference on Management of data*, ACM, Rhode Island, USA, pp. 889-896.

Xia, C., Yu, G. & Tang, M. 2009, 'Efficient implement of ORM (object/relational mapping) use in J2EE framework: Hibernate', *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, IEEE, Wuhan, China, pp. 1-3.

Xu, D. 2010, 'Cloud computing: An emerging technology', *Computer Design and Applications (ICCDA), 2010 International Conference on*, vol. 1, IEEE, pp. V1-100-V1-4.

Zhang, L.-J., Zhang, J., Fiaidhi, J. & Chang, J.M. 2010, 'Hot topics in cloud computing', *IT professional*, vol. 12, no. 5, pp. 17-19.