A FLEXIBLE AND SCALABLE DATA MODEL FOR MULTI-TENANT DATABASES FOR SOFTWARE AS A SERVICE

INDRANI GORTI

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

April 2013

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: GORTI INDRANI

Entitled: A FLEXIBLE AND SCALABLE DATA MODEL FOR MULTI-TENANT DATABASES FOR SOFTWARE AS A SERVICE

and submitted in partial fulfillment of the requirements for the degree of

MASTER IN COMPUTER SCIENCE

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

DR. JOUMARD BRIGITTE Chair

DR. GOSWAMI DHRUBAJYOTI Examiner

DR. OLGA ORMANDJIEVA Examiner

DR. SHIRI NEMATOLLAH & DR. RADHAKRISHNAN THIRUVENGADAM Supervisor

Approved by _____
Chair of Department or Graduate Program Director

_____
Dean of Faculty

Date April 30, 2013

ABSTRACT

A FLEXIBLE AND SCALABLE DATA MODEL FOR MULTI-TENANT DATABASES FOR SOFTWARE AS A SERVICE

INDRANI GORTI

We study existing data models for multi-tenant architectures for Software as a Service (SaaS) in the context of cloud computing, and compare them in terms of space utilization, query formulation and processing. We propose a new model called, the *Index table layout (ITL),* which stores data column-wise, and enjoys a simple mapping process between schemas of original data of tenants and the cloud schema, thus making the model flexible. We conducted experiments to evaluate the performance of the ITL model, for which we used three types of queries, compiled from the literature as well as new queries. The results of our numerous experiments show significant improvement over existing models.

Acknowledgements

I am deeply indebted to my supervisor Dr. Shiri for having accepted me as his student. I thank him for his teaching, valuable advices, financial support in the course of my Master's program. I sincerely express my gratitude for his motivation and constant help throughout and in correcting thesis, technical reports.

I would also like to thank my co-supervisor Dr. Radhakrishnan Thiruvengadam for his comprehensive guidance, for providing financial support and his valuable suggestions throughout the course.

I would like to thank the Department of Computer Science and Software Engineering, Concordia University for providing a dynamic environment for research. I would also like to thank my fellow graduate students for being helpful in discussions and very amicable. I would like to thank my friends and acquaintances to make my stay in Montreal a very comfortable and memorable one.

Last but not the least, I would like to thank my parents, to have always given me moral strength, and for having confidence in me.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

SaaS (Software as a Service) is one of the many types of public cloud computing available. Cloud computing is the use of both hardware and software over a network entrusting remote services with a user's data, software, and computation. In Cloud computing, end users access cloud-based applications through a web browser or a light-weight desktop or mobile app while the business software and user's data are stored on cloud servers at a remote location. Cloud computing allows enterprises to get their applications up and running faster, with improved manageability and less maintenance cost, and enables IT to more rapidly adjust resources to meet fluctuating and unpredictable business demands.

In the business model of SaaS, users over the wide area network are provided access to application softwares and databases. SaaS which is sometimes referred to as "on-demand software", is a software delivery/provision model in which software and associated data are centrally hosted on the cloud, typically accessed through a thin client using a web browser over the Internet. The usage of the software hosted on the cloud is usually priced by the customers on a pay-per-use basis.

SaaS has become a common delivery model for many business applications, including accounting, customer relationship management (CRM), collaboration, management information systems (MIS), enterprise resource planning (ERP),

invoicing, human resource management (HRM), content management (CM), and service desk management. SaaS has been incorporated into the strategy of leading enterprise software companies. One of the main selling points for these companies has been the potential to reduce IT support costs by outsourcing hardware and software maintenance and support to the SaaS provider.

Examples of SaaS include Google Apps, Microsoft Office 365, Onlive, GT Nexus, Marketo, and TradeCard.

1.1 Hosted Services and Multi-tenancy

SaaS is a hosted service architecture. In such an architecture, a service provider develops an application and hosts it. Customers access the application over the Internet using web browsers or web server clients. With continued advancements in the Internet technology, hosted services have become popular for a wide variety of enterprise applications, including sales management applications, marketing, support, human resources, planning, manufacturing, inventory, financials, purchasing, etc. When compared to traditional on-premise solutions, hosted services, such as Google file systems and Hotmail, have surely appeared to reduce the total cost of ownership of an application by aggregating customers together and leveraging economy of scale. Hosted services have shown to be very useful to support day to day business and growth of small to medium size businesses, which would have otherwise been very expensive to consider.

Multi-tenancy, is viewed as an optimization approach to store data for hosted services where the data from multiple customers is consolidated onto the same operational system.

This approach which was pioneered by *salesforce.com* [39], is exceptionally treated as a new paradigm and business model due to the fact that companies do not really have to purchase and maintain their own infrastructure but instead get the services embodied by software from a third party [24]. In multi-tenancy, the tenants are customers sharing similarities in some way or the other. When these "tenants" share the same application and database resource, it allows full use of economy of scale and allows tenants to configure the application to fit their needs as if it runs on their dedicated environment [12]. A tenant can thus be thought of as a "usage profile", and multi-tenancy refers to an approach to storing data of many such similar tenants in a database in a way that supports the growing demands of the tenants (in terms of number of tenants and the size of data).

1.2 Multi-Tenant vs. Multi-User

Hosted services can be classified as either multi-user or multi-tenant applications. A multi-user application uses a multi-instance approach [5]. This means each tenant gets his/her own instance of the application (and possibly also of the database). With the increase in popularity of virtualization and cloud computing, there is a need to cater businesses with many tenants. Multi–instance approach is easy to realize from development perspective and is well suited if the number of tenants is likely to remain low. Otherwise, it results in increased maintenance cost, incurred due to effort of deploying updates to numerous instances of the application [5]. This paved way for the need of a multi-tenant system with single instance of software running on the server.

In a multi-tenant application, on the other hand, each tenant has the possibility of configuring the application to his/her needs. Hence, two tenants may be using the same

building blocks in their configuration; however the appearance or workflow of the application they have may be different.

1.3 Challenges in Multi-tenancy

Multi-tenancy has a number of challenges and even though these challenges exist for single tenant software, they appear in different forms and are sometimes complex. These challenges are explained in a broad sense as follows.

1. Performance: Because multiple tenants share the same resources, the hardware usage in general is high. If one tenant consumes more resources, the performance of other tenants will be compromised. This may lead to inefficient utilization of resources and is therefore not desirable in a pure multi-tenant system.

2. Scalability: Because all tenants share the same application and data store, scalability is an issue. In a multi-tenant situation we cannot assume that the number of tenants will remain the same or that the tenant does not require more than one application and database server. There may be constraints such as the requirement to place for growing data and speed up typical database queries.

3. Security: A security breach can result in exposure of data to other tenants. Hence data protection is very important for the success of SaaS.

4. Zero-Downtime: Addition of new tenants and adapting to changing requirements require constant growth and evolution of a multi-tenant system. Adaptations,

however should not indulge in services provided to other tenants and the server of the system should not go down.

1.4 Architectural Approach to Multi-tenancy

In order to address the challenges discussed above, the traditional architecture of a traditional three-tier web application has been adapted, which essentially comprises of the authentication layer, the configuration layer and the database layer.

Authentication: The purpose of this layer is to identify each tenant. This is done by generating a ticket once a tenant logs in.

Configuration: To make an application multi-tenant capable, it is necessary to allow at least the following types of configurations:

Layout: Allows the use of tenant specific themes.

General configuration: Allows configurations of database settings, encryption key settings, and personal profile settings.

File I/O: Allows specification of tenant specific paths, which can be used for report generations, etc.

Workflow: Allows configuration of tenant specific workflows.

Database: There is a requirement for a layer between business logic and the application's database pool. The main difference between a single- and a multi-tenant application is the greater focus on data management in the former and isolation in the latter.

1.5 Characteristics of Multi-Tenancy

The key characteristics of multi-tenancy are as follows:

Hardware Resource Sharing: The concept of multi-tenancy comes in different flavors, and depending on which flavor is implemented, the utilization of the underlying hardware can be maximized. The following variants of (semi-)multi-tenancy can be distinguished [12, 27]:

- Shared application, separate database.
- Shared application, shared database, separate table
- Shared application, shared database, shared table (pure multi-tenancy)

Throughout the thesis, we will assume the pure multi-tenancy, as the other two have performance issues when a large number of tenants are placed on the same server [12, 35]

High degree of configurability: In a multi-tenant setup, all tenants share the same application instance. Hence, the main requirement is to configure and customize the application to a tenant's need. As it is undesirable to deploy different instances of a multi-tenant application, version support should be an integral part of a multi-tenant setup.

Shared application and Database Instance: A single-tenant application may have running instances and they could all be different from each other because of

customization. In multi-tenancy, however, these differences do not exist as application is run-time configurable.

There are a number of challenging issues related to the SaaS architecture, including performance, scalability, security, zero-downtime, and maintenance. The following work discusses into the database aspects of multi-tenant cloud architecture. In a multi-tenant application, it is essential to have data isolation since all the tenants use the same database instance and they should be able to access their own data. The traditional DBMSs are not capable of this [21].

Hence an SaaS architecture should support functionalities such as:

- Creating new tenants in the database
- Query adaptation
- Load Balancing

As pointed out in [12, 38], since the Shared database, shared table approach is best suited for pure multi-tenancy, in this thesis we consider this approach and propose a flexible data model for SaaS and show its effectiveness in terms of space utilization and query processing.

1.6 Research Motivation

A number of data models of the shared database, shared table variant have been discussed in the literature survey and are presented in Chapter 2. We observed that the existing data models used for realizing multi-tenancy have the following limitations:

1. Storage of null values leading to sparsity.
2. Larger meta-data than the actual data for storage.
3. Separate tables for representing data based on data type.
4. Addressing the issue of insert-only scenarios.
5. No support for dynamically changing data. This includes, heavily and non-heavily utilized data, change in the number of tenants, addition of number of attributes and change in common attributes among tenants.
6. Complicated mapping process between private table layout and the cloud data model layout.
7. Inflexibility of the data model.
8. Many join operations that are expensive.

The need to improve the situation is the motivation for our work in this thesis, and to propose an efficient and flexible data model. By "flexible" we mean a data model that has a simple mapping stage between the user data model and the cloud data model. By "efficient", we mean being able to give reasonably good query performance. The proposed model is flexible and stores the data of multiple tenants in a single table of a database in a column-store feature. In this regard, we studied the Chunk table layout [6], as the most recently developed model, and observed that it had some limitations. We also observed that Chunk table layout [6], was developed with the assumption that the data can be partitioned into heavily and non-heavily utilized data. In the context of SaaS, the utilization pattern cannot be predicted, and this assumption is not realistic.

As discussed previously, tenants are customers with similar usage profiles. We proposed that we could partition data based on similar attributes that the tenants share and further

extended for improved efficiency and flexibility. We show that in terms of query processing, ITL supports a variety of queries which we compiled and categorized into different types in this thesis, based on the ability to query.

1.7 Objectives of the thesis

The objectives of the present thesis are:

- To develop a new data model, called herein as the Index Table Layout (ITL), which makes use of the column-store approach based on the Decomposition model described in [13].
- To perform a comprehensive set of experiments for different query types, to evaluate the performance of ITL. In this regard, a set of queries will be compiled.
- To compare the ITL model with the Chunk table layout using the compiled set of queries.
- To compare the ITL model with the Multiple Sparse Table [9].

1.8 Outline of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we provide an overview of the relevant literature on data storage in SaaS. Chapter 3 provides the background on the Chunk table layout, its evolution, its drawbacks, and possible immediate improvements. In Chapter 4, we present the Index table layout approach proposed in this thesis. We also present the results of our experiments to study the performance of the ITL approach and analyze the results. Chapter 5 compares ITL with Multiple sparse table approach [9] and shows the advantages of our approach. Concluding remarks and a list of possible

directions for further development are presented in Chapter 6. A description of the data we created and used in our experiments is provided as an appendix.

Chapter 2

Background and Related Work

In this chapter we review existing approaches and techniques of modeling and data sharing in the context of SaaS. We discuss the advantages and disadvantages of each approach and explain why Shared Table approach has been considered a base model in the related literature for existing models. We review the existing data models thoroughly in this chapter and motivate the need for a new model, which we propose in this thesis. We use the terms "data models" and "layouts" interchangeably in this thesis.

2.1 Approaches for Sharing Storage

Aulbach and Jacobs [21] discuss the three approaches for sharing storage, introduced below. Efficient storage of data helps in scaling, since multiple tenants share the same application and database instance.

Shared Machine: In the Shared machine approach, each customer gets his/her own database process and multiple customers share the same hardware. Limitations of this approach include no memory pooling, no scalability beyond certain users, and use of shared sockets among customers per server.

Shared Process: The Shared process is relatively better because each customer gets his/her table and multiple customers share the same database process. Since there is only one database in this approach, the customers can share connection pools. All connection pools must be associated with one principal who can access everything and both security and management of resource contention must be handled at the application layer. This paved way to a Shared table approach.

Shared Table: In Shared table approach, data from many tenants are stored in the same tables. A *Tenant Id* column is present to identify the tenant, i.e., the owner of each row. To allow customers to extend the base schema, each table is given a fixed set of additional generic columns. This approach is clearly the best at pooling resources. Its ability to scale up however is limited by the number of rows the database can hold, which is still far better than shared process approach. Administrative operations can be executed in bulk simply by executing queries that range over the *TenantId* column. Figure 2 illustrates the Shared table storage of the three private tables Account $_{17}$, Account $_{35}$ and Account $_{42}$ shown in Figure 1.

Account 17			
Aid	Name	Hospital	Number of Beds
1	Acme	St. Mary	135
2	Gump	State	1042

$ F F R X Q W	
$ L G	1 D P H
1	Ball

$ F F R X Q W		
$ L G	1 D P H	' H D O H U V
1	Big	65

Figure 1: Private Table Layout

7 H Q D Q W1,DGP	H + R V S	L1WXDPOE % H G V		' H D O H U V
17	Acme	St. Mary	135		
17	Gump	State	1042		
35	Ball				
42	Big	Gump			65

Figure 2: Shared Table approach

The shared table approach suffers from a number of limitations, including:

- The major difficulty is that the queries intended for a single customer have to contend with data from all customers which compromises query optimization
- File on disks have intermingled data from multiple customers. Hence, migration requires executing queries against the operating system. Since customer's data are spread across many disk blocks, migration results in decreased performance to access data.
- Use of generic columns is feasible only when the database has a compact representation for sparse tables. If typing of generic columns has been abandoned, it is difficult to implement column-oriented features such as indexes and integrity constraints.
- Row-level access privileges should be assigned to different rows in the same table. This is because each row comprises of data belonging to one tenant.

2.2 Schema Mapping Techniques

This section describes the basic schema mapping techniques for multi-tenancy. Schema mapping refers to the mapping of data from the tenant's data to the data of the desired data model. Figure 3 illustrate a running example. It shows various layouts for account tables of 3 tenants with Id's 17, 35 and 42. Tenant 17 has extension for health care industry while tenant 42 has an extension for the automotive industry.

Private Table Layout

The basic method to support extensibility is to simply maintain private table layout [4]. Since there is a table for each tenant, this layout is suitable for small number of tenants that can produce sufficient load to fully utilize the host machine. Private table layout is illustrated in Figure 1.

Extension Table layout

This layout is based on the decomposed storage model proposed in [13] which splits up a table of n columns into n tables of 2 columns that are joined through surrogate values. This layout is adopted by column-oriented databases [27, 29]. The basic layout and the private table layouts can be combined by splitting off the extensions into separate tables. Extension tables and base tables must be given a *Tenant* column as multiple tenants use the same extension. A column "*Row*" must also be added so the logical source tables may be constructed again. This vertical expansion of table helps improve the performance of analytics [28] and RDF data [1].

$Account_{Ext}$			
Tenant	Row	Aid	Name
17	0	1	Acme
17	1	2	Gump
35	0	1	Ball
42	0	1	Big

Healthcare Account			
Tenant	Row	Aid	Name
17	0	St. Mary	135
17	1	State	1042

Automotive Account		
Tenant	Row	Dealers
42	0	65

Figure 3: Extension Table Layout

Universal Table Layout

A Universal Table Layout is a generic data model with a Tenant Column, a Table column and generic columns. The n^{th} column of the logical source table is the n^{th} data-column of the Universal Table. This allows tenants to extend the same table in different ways. The table however has many null values.

Universal							
Tenant	Table	Col1	Col2	Col3	Col4	Col5	Col6
17	0	1	Acme	St. Mary	135	-	-
17	0	2	Gump	State	1042	-	-
17	1	1	Ball	-	-	-	-
17	2	1	Big	65	-	-	-

Figure 4: Universal Table Layout

Pivot Table

A Pivot table is another generic data model. A Pivot table has a column *Col* in addition to columns *Table, Row* and *Tenant*. Hence this representation has multiple Pivot Tables for different data types of data columns. The meta-data here, occupies much more space than the actual data. Also n-column logical table requires (n-1) aligning joins. This layout eliminates handling of null values but increases the query processing time.

Agrawal et al. [3] compared the performance of Pivot Tables (Vertical Tables) with the conventional horizontal tables. They also compare performance of selection, projection, and join operations, and conclude that the Pivot table layout performs better because it allows selecting columns to be read in. In most of the scenarios studied in [24, 26] it has been shown that vertical representation is better.

Beckman et al. [7] propose a technique for handling sparse data sets using a Pivot Table Layout. In comparison to explicit storage of meta-data columns, they chose an

"intrusive" approach which manages the additional runtime operations in the database kernel. Cunningham et al. [14] present an "intrusive" technique for supporting general-purpose pivot and unpivot operations.

Pivot $_{int}$

Tenant	Table	Col	Row	Integer
17	0	0	0	1
17	0	3	0	135
17	0	0	1	2
17	0	3	1	1042
35	1	0	0	1
42	2	0	0	1
42	2	2	0	65

Pivot $_{str}$

Tenant	Table	Col	Row	String
17	0	1	0	Acme
17	0	2	0	St. Mary
17	0	1	1	Gump
17	0	2	1	State
35	1	1	0	Ball
42	2	1	0	Big

Figure 5: Pivot Table Layout

Chunk Table Layout

Chunk Table is a data structure proposed in [6]. A table is partitioned into groups of columns, each of which is assigned a chunk ID and mapped into an appropriate Chunk Table. A Chunk Table is like a Pivot Table except that it has a set of data columns of various types, with and without indexes, and the Col column is replaced by a Chunk column. It works well when the base data can be partitioned into "meaningful" subsets. Here "meaningful" refers to any criteria for partitioning data. In [6], the authors have two chunks. One for heavily utilized data and the other one for non-heavily utilized data. In comparison to Pivot Tables, chunk table layout reduces the ratio of stored meta-data to actual data as well as the overhead for reconstructing the logical source tables. In comparison to Universal Tables, this approach provides a well-defined way of adding indexes, breaking up overly-wide columns, and supports typing. By varying the width of the Chunk Tables, it is possible to find a middle ground among these approaches. This provides flexibility but comes at the price of a more complex query-transformation layer. Since this model is more suited when the base data can be partitioned into heavily and non-heavily utilized content, this paves way for the need of a more generic that does not take into consideration the utilization pattern.

Chunk int|str					
Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	1	Acme
17	0	1	0	135	St. Mary
17	0	0	1	2	Gump
17	0	1	1	1042	State
35	1	0	0	1	Ball
42	2	0	0	1	Big
42	2	1	0	65	-

Figure 6: Chunk Table Layout

Chunk Folding

Chunk Folding [6] technique combines Extension and Chunk Tables. It partitions the logical source tables vertically into chunks that are folded together into different physical multi-tenant tables which are joined as needed. The database's "meta-data" content is

divided among application-specific conventional tables and a large fixed set of Chunk Tables.

For example, Figure 7 illustrates a case where base Accounts are stored in a conventional table and all extensions are placed in a single Chunk Table. In contrast to generic structures that use only a small, fixed number of tables, Chunk Folding attempts to exploit the database's entire meta-data content in as effective a way as possible. Good performance is obtained by mapping the most heavily utilized parts of the logical schemas into the conventional tables and the remaining parts into Chunk Tables that match their structure as closely as possible.

Chunk Row

Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	135	St.Mary
17	0	0	1	1042	State
42	2	0	0	65	-

Account Row

Tenant	Row	Aid	Name
17	0	1	Acme
17	1	2	Gump
35	0	1	Ball
42	0	1	Big

Figure 7: Chunk Folding

Multi-tenant Shared Table

The shared table layout introduced in [18] aims at separating the common data from tenant specific information. The idea behind this approach is extending a relational database management system to support the concept of tenants at the database layer so that the database engine can actually bind a tenant's request with him/her and thereby selecting the appropriate storage area. The other goal in Multi-tenant shared table is that, only one schema instance is used per application. The authors in [18] state that the practical aspects of this idea remain to be seen. In our work, we implemented the Multi-tenant Shared table and also compared it with the Index table layout. The results of the comparison are given in Chapter 4. Multi-tenant shared table is a model that uses object-oriented technique.

Hybrid Schema Sharing Technique

Foping et al. [15] proposed to split the common content table shared by each tenant with the extension table containing additional information. The common content is stored in an XML document which in turn is stored in a PostgreSQL table and queried with XPath 1.0. This approach suffers from a few deficiencies. The validity of the XML document stored in the database cannot be checked against a schema as it is generated at run time. The security aspect of this design is another drawback since small alterations in queries can give away the information to malicious users [15].

The Multiple Sparse Tables Approach

The authors in [9] discuss the differences between the multi-tenant sparsity and the traditional sparsity. Since there is large number of columns when storing data in a shared table, we cannot predict the needs of tenants. The authors argue that in order to evolve a frequently altering table that is flexible, it is better to divide the tenants based on the requirements, i.e., number of columns. Hence a group of tenants may require 20 columns, some 50, some 100, and so on. This also, reduces large number of nulls which result in better space utilization and query performance. In [9], the authors also consider schema nulls and value nulls. Schema nulls indicate that the tenant does not customize columns. Value nulls indicate that the tenant customizes columns but the values are real nulls. In a Multiple Sparse table approach, tenants consume the columns from left to right and hence it is left-intensive [9].

Figure 8 illustrates Single sparse table layout. In this layout, all data will be put into Single sparse table. The table has some fixed number of attributes. This results in many nulls as values are entered only in certain attributes. The Multiple sparse table is evolved to improve the Single sparse table. It can be noted that in a Multiple sparse table approach, we get to choose the most suitable table among the N Multiple sparse tables; each having a fixed number of columns. As mentioned earlier, we divide the tenants based on the requirements of number of columns. An efficient such split strategy is developed based on speculation of tenants' demands. In the experiments in [9], three schemas used have 10, 50 and 100 columns. When customizations of one tenant occur, we should determine which table instance is appropriate by considering the number of columns customized and the metadata of sparse table.

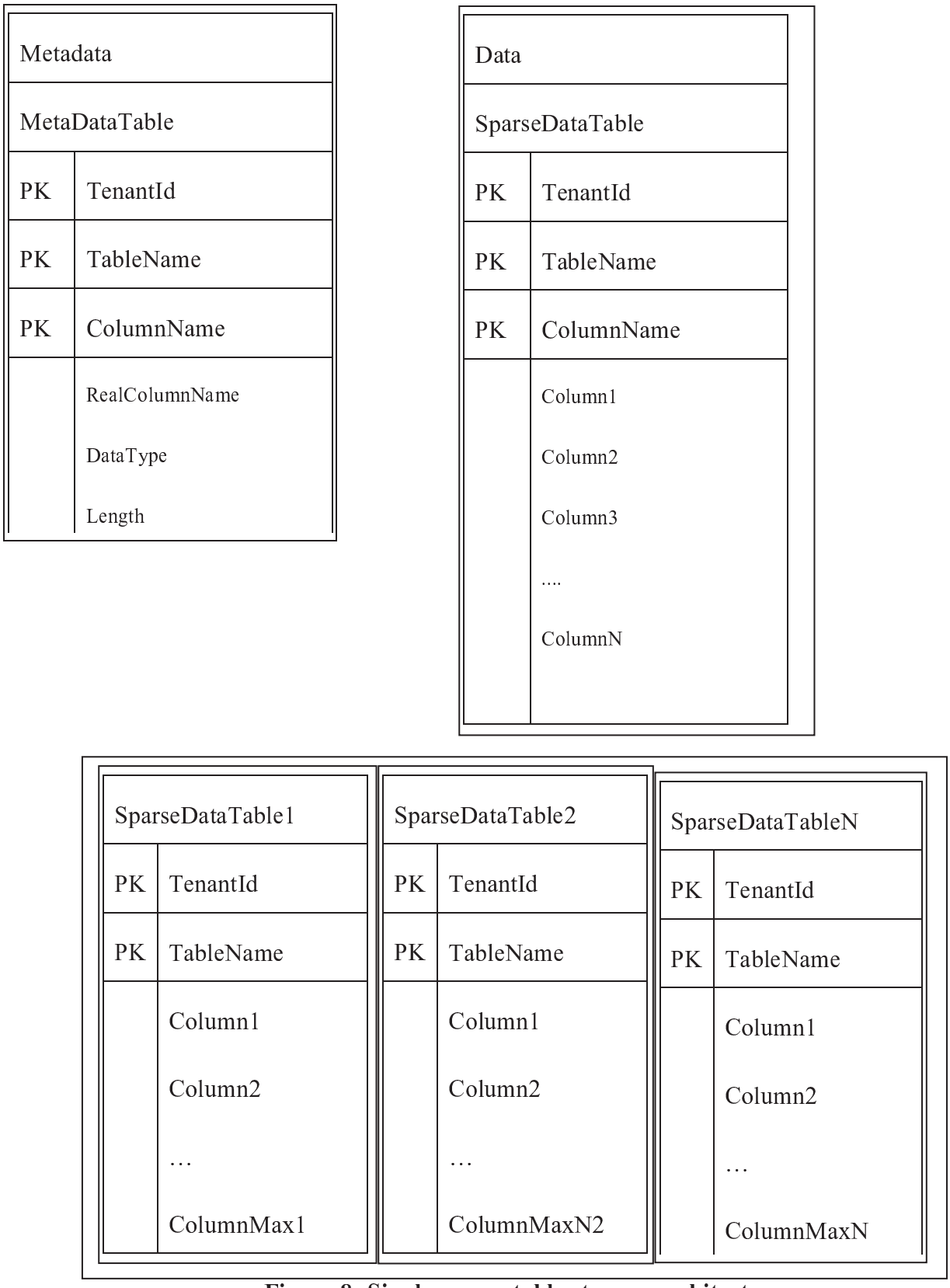

Figure 8: Single sparse table storage architecture

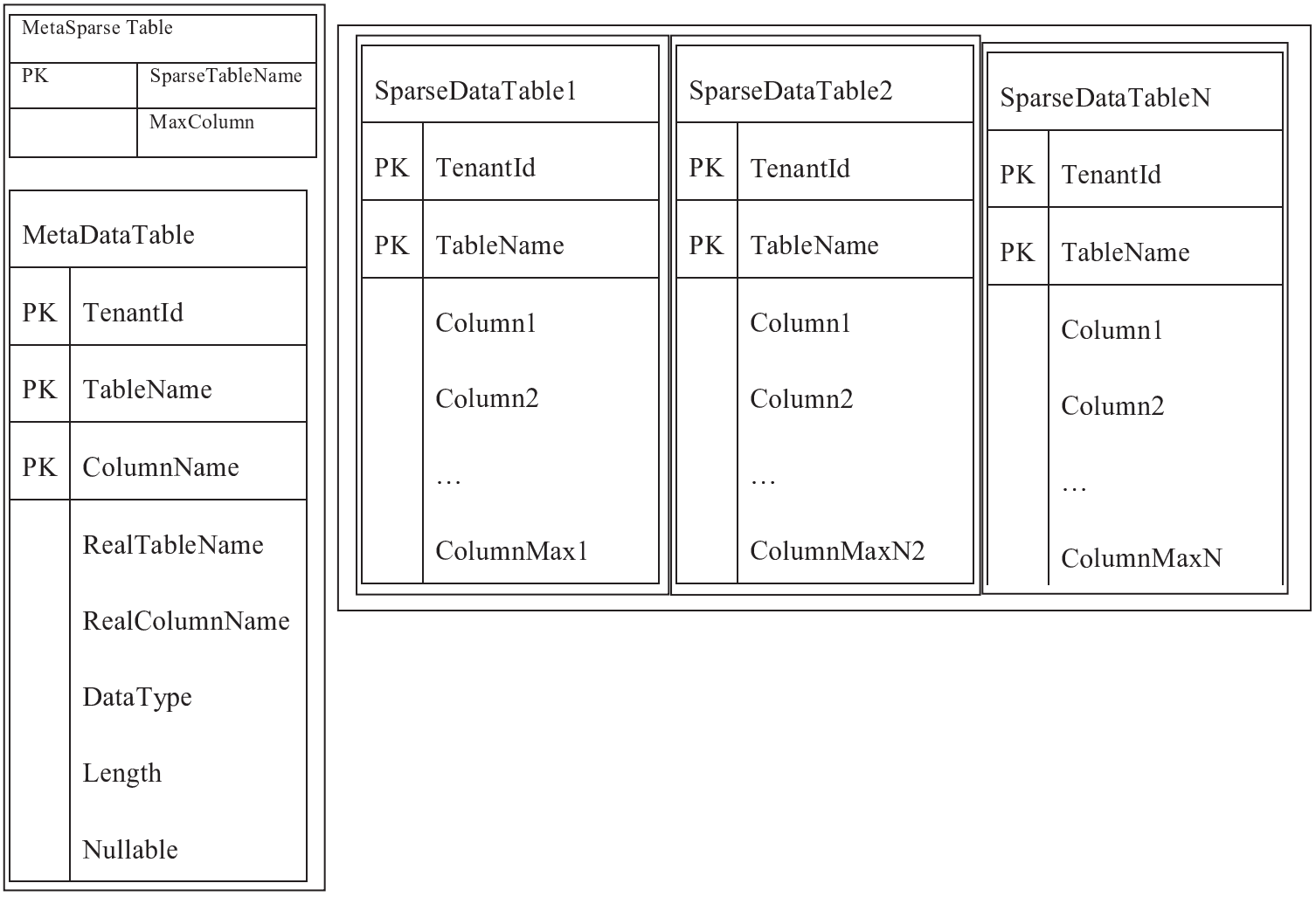

Figure 9: Multiple sparse table storage architecture

Vertical Partitioning

Grund et al. [18] model insert-only data by vertically partitioning the table into two tables, one that is frequently updated and the other that is stagnant. Figure 10 shows the dictionary and document vectors for a single attribute. The dictionary consists of three values: Anton, Beta, and Charles. The belonging offsets, 0, 1, and 2 are not stored with the values but are implicitly available.

Vertical partitioning [19] is applied when data is appended (insert-only scenario), i.e., when update and delete operations are not allowed. This is called "no-overwrite" in implementation of Postgres [31].

One of the disadvantages of the insert-only scenario is the additional memory consumption. Higher the update rate, higher will be the additional memory consumption. When using a time-travel approach [19], for each row that is updated there is a new record that has to be added in the database with the corresponding time-stamps. When using a row-store, this can have negative effects. The old data is mixed with current entries and as a result when querying the databases, all data has to be scanned. On the other hand, when using a column store, the storage overhead can be significantly decreased using compression [13].

Typically column stores use dictionary compression to map variable length content of columns to fixed length data types. In addition, the actual value entries are stored bit-encoded in a document vector that allows efficient offsetting to the requested position.

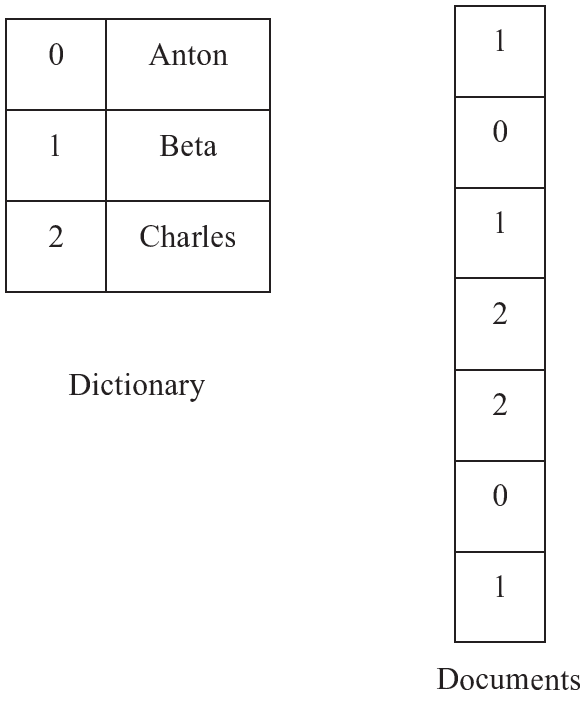

Figure 10: Dictionary Compression for Column Stores

Key	SHKZG	DMBTR	SGTXT	Valid from	Valid to
1	S	20	Reconcile	1	-
2	H	394	New Fok	2	3
2	H	394	Fock	3	4
2	H	394	N.Fock	4	5
2	H	394	Genua	2	-

Figure 11 : Example table with a frequently updated attribute 'SGTXT'

Key	SHKZG	DMBTR	Valid from	Valid to
1	S	20	1	-
2	H	394	2	-

Key	SGTXT	Valid from	Valid to
1	Reconcile	1	-
2	New Fok	2	3
2	Fock	3	4
2	N.Fock	4	5
2	Genua	5	-

Figure 12: Example table with two partitions

In the experiments carried out in [18] memory consumption reduced considerably. As stated in [18], instead of 40% increase in higher memory consumption, this approach increased only by 15%. The vertical partitioning method has its roots in Decomposition Model reported in [13]. Figures 11 and 12 illustrate the vertical partitioning of the table. Here frequently updated attribute is "*SGTXT"*. The dictionary contains the key-value pairs. The document vector is made up of 'keys' of the dictionary.

In [6], the performance of various schema mapping techniques are compared with a wide variety of databases like IBM DB2, HBase, and Microsoft SQL Server. The performance varied depending on the technique and the database used. If a technique had good performance, it suffered from memory consumption. Aulbach et al. [5] argue that an

"ideal" SaaS database system should be based on the Private Table layout. The interleaving of tenants, which occurs in all other mappings, breaks down the natural partitioning of the data. An ideal system forces import and export of a tenant's data, which occurs for backup/restore and migration, and done when querying the operational system [5]. In contrast, each tenant's data in private table is clustered together on disk so it can be independently manipulated. The interleaving of tenant data also complicates access control mechanisms in that it forces them to occur at the row level rather than the table level. According to [5], in such an ideal SaaS database system, the DDL should explicitly support schema extension. The base schema and its extensions should be registered as "templates". There should be multiple tenants and each tenant should be able to select various extensions to the base schema. The data model should not just stamp out a new copy of the schema for each tenant but rather it should maintain the sharing structure. This structure should be used to apply schema alterations to all relevant tenants. In addition, this structure will reduce the amount of meta-data managed by the system. In an ideal SaaS database system, the DDL should support on-line schema evolution [5]. The hard part here is allowing evolution over existing data.

Schiller et al. [28] consider schema inheritance concept tailored to multi tenancy. This is altogether a new model where a tenant context logically assembles all information that describes the tenant's view of the database and yet isolates tenants from each other. Since inheritance is used, the technique eliminates redundancy.

We observe that all the data models that have been studied so far have their own pros and cons and /or suited to some underlying assumptions, e.g., partitioning data into heavily and non-heavily utilized parts; partitioning data into commonly shared and non-shared data, partitioning data into vertical and horizontal patterns, etc. In a dynamically growing system, we require a data model that does not make these assumptions. Hence there is scope for a more flexible and generic data-model using the Shared table approach for data storage in SaaS.

As discussed above, SaaS is still evolving and more work is required before an ideal framework is developed. The main challenge is to balance between the memory consumption and the query processing time.

It is necessary to have a SaaS system that supports schema extension and on-line schema evolution. A database system should distribute data for many tenants across a farm of servers. The distribution should be tenant-aware rather than lumping all tenants into one large data set [6].

The mappings in which the application owns the schema provide only limited support for schema evolution and as such it will be performed poorly. Moreover, they cannot be scaled beyond a certain level. A SaaS database system should have a shared nothing architecture where data is stored on fast local disks. Data should be explicitly replicated and used by the database, rather than a distributed file system, and used to provide high availability [6]. In particular, the database should support multiple communication

patterns for joins, rather than requiring the use of map-reduce. MapReduce is a programming model for processing large data sets used to do distributed computing on clusters of computers. Not to forget scalability, an ideal system must be able to handle more queries for a given set of data.

The vertical storage structures of HBase and BigTable, which were used to implement Pivot Tables, are similar to column stores. They are designed for "narrow" operations over many rows. Such vertical structures may be made more competitive for wide operations by keeping the data in memory, since the cost of reassembling rows is dominated by the time to perform many reads from disk. Advancement in data storage technologies gradually makes main-memory databases more affordable and attractive [16].

It is also evident from the literature that column-wise [18] operations are definitely performing better over the horizontal operations. Hence, better approaches to handle sparsity of a table are required for improved query efficiency and space utilization.

2.3 Chunk Table Layout in Detail

The work in [6] introduces the chunk table and the chunk folding technique. Chunk table is a generic structure proposed on the assumption that the base data can be partitioned into well-known dense subsets. As discussed in the previous chapter, Chunk Folding [6] combines Extension and Chunk Tables. In chunk folding technique, the logical source

tables are vertically partitioned into chunks that are folded together into different physical multi-tenant tables and joined as needed. The number of "meta-data" attributes is divided among application-specific conventional tables and a large fixed set of chunk tables. In contrast to earlier techniques that use only a small, fixed number of tables, Chunk Folding attempts to exploit the entire meta-data budget in as effective a way as possible. Experiments in [6] show transformations needed to produce queries over Chunk Tables by considering the simpler case of Pivot Tables and report on the better performance of the Chunk Folding method.

The performance of chunk tables is improved by mapping the most heavily-utilized parts of the logical schemas into the conventional tables and the remaining parts into Chunk Tables that match their structure as closely as possible. However, this method is useful when the data is partitioned into heavily and non-heavily utilized parts, which may not be always true.

Each of the models above is built from the private table notation. In other words, it is also possible to build the above models from the private table notation. In fact, they are representations of the private tables. However, it might not be possible to convert back these tables to the private tables. This reverse mapping mechanism is helpful especially if one wants to automate the process of converting the information from a source schema to the cloud database.

We now focus on the reverse mapping process for existing schema-mapping techniques. In general, the base tables for the private table layout have fairly less complex schemas. In order to allow the reverse process, we make our comparisons against the Chunk table. Though this kind of process is not always required, it may be inadequate for leading to data ambiguity. When extra column of an existing data type is inserted in the private table, the corresponding update has to be done on the chunk table layout. Figure 13 illustrates this update. As can be seen, the figure includes three private tables, each for a client. Including an extra column in one private table would bring changes in the Chunk table. We illustrate this in the coming figures.

Account 17				
Aid	Name	Hospital	Number of beds	Apt
1	Acme	St. Mary	135	4
2	Gump	State	1042	3

Account 35	
Aid	Name
1	Ball

Account 42		
Aid	Name	Dealer
1	Big	65

Figure 13: Addition of an extra column to the Private Table Account17

If we were to include another column, say Apt, to Account 17 with a data type that already exists (in this case *int*), we should update the chunk table, assuming that it is the schema for the cloud database.

Chunk int|str

Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	1	Acme
17	0	3	0	135	St. Mary
17	0	0	1	2	Gump
17	0	3	1	1042	State
17	0	3	0	4	?
35	1	0	0	1	Ball
42	2	0	0	1	Big
42	2	2	0	65	-

Figure 14: Updated Chunk table with extra column

Since in the Chunk Table Layout we have the data differentiated by their types, we have to place '4' in the 'int' column, the corresponding entries being Tenant-0, Table-0, Row-0. It must be noted that the chunk column of the Chunk Table Layout does not matter in this case as it determines which of the chunks it belongs to; the chunks being determined by how heavily the tables are utilized. Hence, in terms of representing the data of private table to the cloud schema, the columns that are valid are the Tenant, Table, Row, Int1, Str1.

Suppose we insert a new value 4 in column 'Int1'. Then this value can replace the entry 135 (shown in Figure 14). Note that the Str1 value corresponding to 4 is empty. Adding to this, the other three columns Tenant, Table, and Row have the same values. Therefore, it is not possible to construct the Private Table afresh from this model, which is a limitation of the model.

To better illustrate the later point, let us consider the following query defined in [6]:

```
SELECT Beds
FROM (SELECT Str1 as Hospital, Int1 as Beds
        FROM Chunkint|str
        WHERE Tenant = 17 AND Table = 0 AND Chunk = 1) AS Account17
            WHERE Hospital= 'State' ;
```

Here, since 'Name' and 'Hospital' are both stored in column 'Str1', the condition Hospital='State' suggests searching for the value 'State' under that column.

The query result would be wrong in case there is another value, say 'State' under the 'Name' column of the Private table notation (i.e., the corresponding entry is also under 'Str1' in the Chunk table). This is a drawback of the model as it is quite possible to have entries with same values under different attributes.

2.4 A Quick fix

The aforementioned limitation of the Chunk Table Layout could be fixed, as follows. A quick fix would be to eliminate the Chunk column altogether and add a column called '*Column*'. This column pertains to the number of columns in the private table in which

the value is placed. Because of this change, there is a considerable change in the schema. Instead of having both columns 'Int1' and 'Str1' in the same column, which may cause ambiguity, we just have one column for each entry in the table. Figure 15 shows the new table, which we call as *Extended Pivot table,* which is in essence is a combination of Pivot Tables [6] and Chunk Tables [6].

Chunk int|str				
Tenant	Table	Col	Row	Value
17	0	0	0	1
17	0	1	0	Acme
17	0	2	0	St. Mary
17	0	3	0	135
17	0	4	0	4
35	1	0	0	1
35	1	1	0	Ball
42	2	0	0	1
42	2	1	0	Big
42	2	2	0	65

Figure 15: Extended Pivot Table obtained by adding column 'Col' to Chunk Table

Hence with the coordinates being `Col`, `Row`, `Table`, and `Tenant`, it is now possible and even to map the values from the Extended Pivot table back to the Private Table.

There is now a one to one correspondence between every value in the Extended Pivot table and the private tables. Note that for every value in the table, there is a meta-data with four coordinate values. The Extended Pivot table is a fairly simple but rich model against which performing queries will be less expensive. The queries are over the column 'Tenant' and comprise of a series of 'AND' operations. The downside of the new model, however, is that it is not space efficient. The space required for storing meta-data of Extended Pivot tables is almost the same as the space taken for the Chunk tables.

2.5 Summary

It is observed in [6] that Chunk table is particularly effective when the base data can be grouped into heavily utilized and non-heavily utilized subsets. This assumption, however, may not always be true. Therefore, we need a new data model that supports SaaS and the model must be flexible and scalable. To be more precise, the notion of scalability refers to having reasonable performance as the size of the data grows. As the mapping process of Chunk Folding is involved, it is also desirable to improve the complicated schema mapping task in the Chunk model to make it flexible.

This is achieved in our proposed, Index table layout which is discussed in the following chapter. The schema mapping in Index table layout is simple and unambiguous. The motivation for Index table layout is based on the Universal table layout, Pivot tables, Chunk table layout and the idea of vertical partitioning. Vertical partitioning and its advantages are presented in section 3.2 while referring to the Decomposition Storage model.

Chapter 3

Proposed Index Table Layout

This chapter presents the proposed *Index table layout,* we proposed to support SaaS. Through extensive experimental study, we show its performance superiority over existing models with respect to space utilization and query processing efficiency.

3.1 The Data Model

The Index table layout, shown in Figure 16, comprises of a base table and a number of supporting tables. The base table contains all the columns that are common to all the individual tenant tables (Private tables) with an additional column, which we call as the *Index*. The tuples in the base table are those relevant to all the tenants. The index column is defined as NOT NULL and UNIQUE, and is incremented automatically with insertion of new tuples. Hence in our example, the Index table layout would have the attributes Index, Tenant, Aid, and Name. Each supporting table has two columns, one for the 'Index' and the other for a column, that is not common to the all the tenants. In other words, if there are n non-common columns among the private tables, then the proposed model would have n supporting tables along with a base table. As a part of representation, only the valid and existing records go into the supporting tables and these records have a matching 'Index' value from the base table. In Figure 16, for instance, when the index value is 2, it means Tenant 17 has a value with 'Aid' being 2,

'Name' being 'Gump', 'Hospital' being 'St. Mary', and the 'number of Beds' being '1042'. Since there are no dealers for 'Gump', we do not use Null or reserve a place for it in the supporting table. This means writing only the values that exist, and hence avoiding sparsely populated tables.

Base Table			
Index	Tenant	Aid	Name
1	17	1	Acme
2	17	2	Gump
3	42	1	Big
4	35	1	Ball

Index	Hospital
1	St. Mary
2	State

Index	Number of Beds
1	135
2	1042

Figure 16: Index table layout

This results in a one to one correspondence between values in the Index Table Layout and the Private Table Layout, hence making it possible to reproduce the private tables from the Index table layout, which is not possible with the chunk table [6]. Furthermore, as will be shown later, processing queries with predicates on columns "Index" and non – common attributes between the base and the supporting tables of the Index table layout is

faster than existing methods. For example, consider the query that asks for the list of all hospital names, which is expressed as a select distinct operation over the supporting table for Hospitals. This would have otherwise been a complex query consisting of many AND operations over multiple conditions on columns `Tenant`, `Row`, `Col`, `Int1`, `Str1` with values, each of which ranging and changing over many figures. In a nutshell, with respect to richness of the model in supporting backward mapping to reconstruct the original private tables and clarity, the Index Table Layout is a more desired and richer than the Chunk tables, Pivot tables, and Extended Pivot tables. The results of our experiments to evaluate the performance presented in Chapter 4 indicate that our proposed model is desired due to its performance advantage.

Addressing queries that are common over all the tenants are obviously queried upon the base table over the tenant id. So the query expressions are not complex. The worst case scenario would be a query that joins all the tables with the joint attribute Index. As we will show, other than this rather unusual query, the Index table layout is superior both in storage utilization and query processing time.

3.2 Decomposition Storage Model

The Index table layout uses features from the Decomposition storage model (DSM) reported in [13]. This section briefly explains the DSM model and its advantageous features over N-ary storage model (NSM). The criteria used in [13] to compare these two models include complexity and generality, storage requirements, update performance, and retrieval performance. Most databases store the data in N-ary storage model for a set of records. This approach stores data as seen in schema of Figure 17 (a). The column

'Sur' refers to surrogate value. 'A1', 'A2' and 'A3' are attributes. The key concept is that in NSM, all the attributes of the schema are stored together. Figure 18(b) illustrates the corresponding Decomposition Storage model for the data that is stored in N-ary Storage model in Figure 18(a). It can be observed that the table is decomposed into several small tables for every attribute A1, A2 and A2.

R	Sur	A1	A2	A3
	S1	V11	V21	V31
	S2	V12	V22	V32
	S3	V13	V23	V33

(a) N-ary Storage Model

A1	Sur	Val
	S1	V11
	S2	V12
	S3	V13

A2	Sur	Val
	S1	V21
	S2	V22
	S3	V23

A3	Sur	Val
	S1	V31
	S2	V32
	S3	V33

(b) Decomposition Storage model

Figure 17 : (a) N-ary Storage Model (b) Decomposition Storage model

The authors state that the DSM offers simplicity. One advantage of simplicity is it has fewer and simpler functions, given fixed development resources and can either further tuned in software or pushed into hardware to improve performance.

Another advantage is that many alternative cases different processing strategies can less often exploited, since the cases are not always recognized A third advantage is reduced user involvement, since less performance tuning is required by users. A fourth advantage of simplicity is reliability [13].

It has been illustrated [13] that the DSM model can support data models which allow multivalued attributes, entities, multiple parent relations, heterogeneous records, directed graphs and a temporal dimension with some simple extensions. The model offers increased physical data independence and availability and offers improved recovery from failure.

It was noted that the space requirement of DSM model is l/2 to 4 times the total storage of NSM [13]. The Chunk table and other data models (discussed in Chapter 2) have high storage requirements due to large number of attributes storing meta-data. The overhead of high storage requirement for DSM-like model (such as our, Index table layout) can be considered instead of Chunk table layout. This is because the Chunk table layout too has large storage requirements for meta-data.

With storage becoming less expensive, the storage requirements are not as critical in most database systems as performance and reliability. The performance for update operations

of the two storage models, depends on the frequency of attribute modifications, record inserts and delete operations.

The retrieval performance of the two storage models depend on the number of attributes involved in retrievals and the size of intermediate and final results. In general, the DSM requires more disk accesses for a large number of retrieval attributes and small intermediate and final results, but otherwise requires fewer disk accesses. This problem can be fixed by using multiple disks, and with the RAM becoming cheaper, more data can be cached at a time [13]. The DSM model allows individual attributes to be cached, which results in better utilization of cache, since not all attributes of a relation have the same frequency. Although more joins are required by the DSM model, each join operation is faster. Similar to the strategy of Reduced Instruction Set Computing (RISC) for CPU design, the design of the DSM model is based on the insight that simplified components (as opposed to complex ones) can yield higher performance.

In [13] the retrieval performance comparison assumed the NSM was highly tuned, so that every constrained or join attribute had an inverted file index. This situation is unlikely in reality. Thus the DSM would have an advantage in an environment where workload characteristics are not static.

The idea of DSM has been adapted with variations by many column store databases like Vertica, MonetDB, Google Big table, etc., and has been very well acknowledged in NoSQL databases (commonly interpreted as not only SQL databases) in the current trend

of cloud databases. We make use of this idea in a slightly modified form in our data model, Index table layout.

In our work, for storing multi-tenant data in a shared table approach, we use a mix of features of Decomposition storage model and normalization while reducing storage requirements and avoiding null values.

3.3 Advantages of Index Table Layout

The advantages of the proposed Index table layout include better space utilization for not needing nulls and fast access of the values in a column. In terms of performance, most heavy operation would be when performing a join operation over all the supporting tables. This, however is less likely to be frequent in SaaS applications as most queries for SaaS application are over individual columns. This is complimented by the use of REST APIs (Representational State Transfer Application Programming Interface) which are light weight and can easily fetch data on to the browser.

Next chapter presents our experimental study of the performance of the Index table layout proposed in our work.

Chapter 4

Experiments and Results

To evaluate the proposed Index table layout model in various aspects and to compare its performance with existing models, we carried out extensive experiments. This chapter describes the experiments and presents the results.

For the experiments, we considered different sizes of tables and different types of queries. We used the MySQL Workbench 5.2 CE as the DBMS running on an Intel Xeon Processor with a 4 GB RAM. In order to simulate the test beds, we first created the schema for the Private tables, and then created the schemas for Chunk and Index table layouts. The Chunk table layout we created is based on the model proposed in [6]. The model assumes that the data is partitioned into heavily and non-heavily utilized data. It considerably reduces meta-data size unlike previous models like Pivot tables. It is easy to control the meta-data storage by varying the width of Chunk tables. This is not possible in Pivot tables as each pivot table has many meta-data columns. On the other hand, this flexibility of Chunk table layout comes at the price of a more complex query-transformation layer. The data is generated by means of scripts written in Perl and Shell and executed in batches for all the layouts so as to maintain and demonstrate the equivalence of data between both layouts.

One of the scripts developed was for generating and populating data for the Chunk table layout. Using this data, we then constructed the Index table and the associated individual tables. Hence, there is a one to one correspondence between the data in the private tables and the data in the index and Chunk table layouts. More details on data and query generation scripts will be provided later.

4.1 The Database Layout

The database layout we considered for performance evaluation is an OLTP (Online Transaction Processing)) schema with one-to-many relationships. Individual users within a business (a tenant) are not modeled, but the same tenant may engage in several simultaneous sessions so data may be concurrently accessed. Every table in the model has a tenant-id column so that it can be shared by multiple tenants.

For fairness of comparison of results, we simulate the database of the same size as in experiments of [6]. Each table contains about 20 columns, one of which is the entity's ID. Every table has a primary index on the entity ID and a unique compound index on the tenant ID and the entity ID.

4.2 Database Operations

In the experiments we perform create, read, and update operations and reporting tasks that simulate the daily activities of individual users. In SaaS an application, normally delete operation is not performed, as data in the cloud data model has intermingled data from many tenants stored in the same table. To facilitate analysis of the results, we have

grouped the queries ranging from light weight operations to heavy ones. The range of light to heavy operations can be seen in queries of Type 1.

The various light and heavy operations are described as follows:

Select Light: Selects all attributes in a single entity set or a small set of entities as if they were to be displayed on an entity detail page in the browser. Most SaaS sytems are framework based and make use of RESTful API's. The database is normally accessed via these API's and are displayed in the browser

Select Heavy: Runs one of five reporting queries that perform aggregation and/or parent-child-rollup.

Insert Light: Inserts a single tuple into the database as if it had been manually entered into the browser.

Insert Heavy: A transaction that inserts hundreds of tuples into the database in a batch as if they had been imported via a Web Service interface.

Update Light: Updates a single entity or a small set of entities as if they had been modified in an edit page in the browser.

The set of entities is specified by a filter condition that relies on a database index.

Update Heavy: Updates hundreds of tuples identified by their ID's on which we have a primary key index.

4.3 Data Generation

We construct a Chunk table that has 10^4 rows and then constructed the corresponding Private and Index tables with the same information content. We then keep increasing the

number of tuples by multiples of ten. We compare the results of the Chunk tables with 10^4, 10^5, 10^6, 10^7, and 10^8 tuples and their corresponding tables presented in our Index table layout. The results of these experiments are presented next.

4.4 Tables Configuration Settings

Appropriate configuration parameters have been set in MySQL (my.ini in Windows environment and my.cnf in Linux) to support tables having large number of tuples. Figures 18 to 25 compare the Index table layout and the Chunk table layout in terms of storage utilization and query performance.

4.5 Storage Utilization

Note that in Index table layout, we only enter the values for existing records in the supporting tables. This way the sparse values present in the 'index' column in the Index table layout ties together the rows in the base table with the related entries in the supporting tables.

When building the Index table layout, the null values which were present in the universal table are not represented. Only non-null values are recorded in the supporting tables. The null values are simply omitted, thus reducing the space usage.

In our experiments, we used both Chunk and Index table layouts for varying number of tuples. Each table has about 20 attributes. Table 1 shows the approximate sizes of tables in MB for tables we used in our experiments with the number of tuples 10^4, 10^5, 10^6, 10^7, and 10^8 tuples. It can be seen that, the space utilization gets 10 times larger as the number of tuples increases 10 times. This is observed for both the Index table layout and

the Chunk table layout. Figure 19 illustrates the approximate space utilization in MB for these two models.

No of tuples	Size in MB of Chunk table layout	Size in MB of Index table layout
10^4	1.5	0.3
10^5	5.78	4
10^6	55.132	34
10^7	600	370
10^8	6200	4075

Table 1: Sizes of Chunk table and Index table layout for various number of tuples

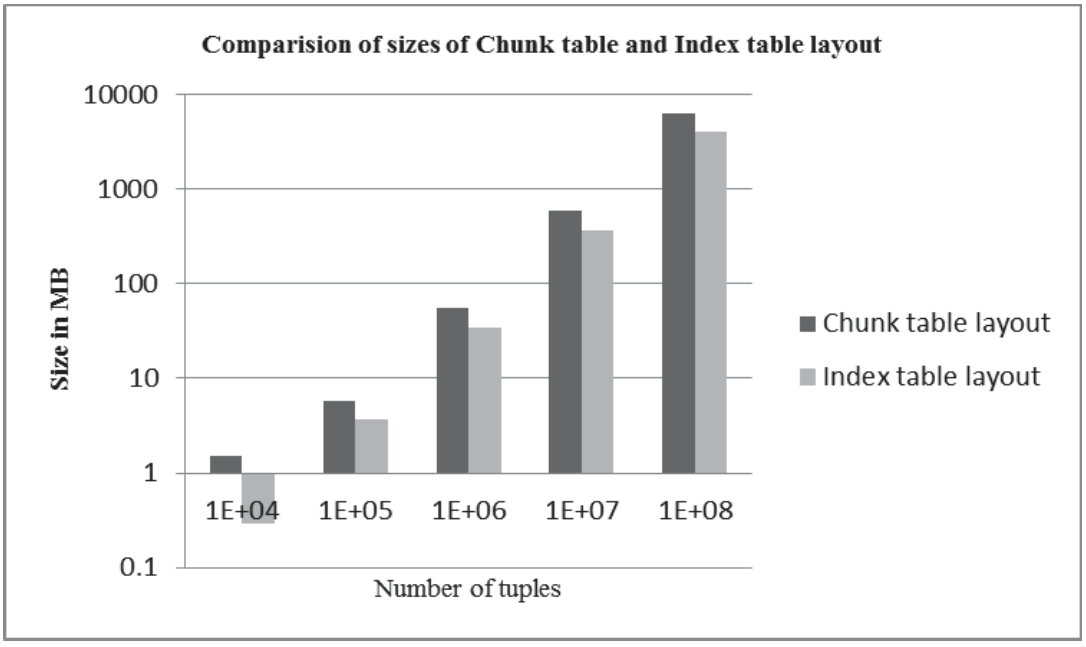

Figure 18: Comparison of sizes of Chunk table layout and Index table layout

4.6 Query Performance

We also studied the performance of query processing against these models. In addition to all the queries used in experiments reported in [6], we also considered and used a number of other types of queries in our performance evaluation. We compared the two models based on expressivity and flexibility with which one can formulate queries, and on efficiency of query processing. As mentioned earlier, the mapping process to build the Chunk table is complex and involved. We also remark that the Chunk table layout can be constructed from the Private table layout, but the reverse is not possible, by which we consider this model to be not expressive enough.

A classification of different types of queries we considered in these set of experiments is as follows, together with an explanation of why such queries were considered.

- Type 0: Queries that run against both Chunk table layout and Index table layout.

We observe that for the Chunk table layout, it is not possible to pose a query that retrieves information for a predicate over two or more columns. This is because such queries require values from the meta-data columns (like `row`, `col`, `table`) and there is an ambiguity in values under different columns. For instance, if there are two attributes of the same data type, the row id at which the corresponding value is stored is ambiguous. Hence, queries that can be posed against the Chunk table, and ours too are simple select queries, called here as Type 0. It should be noted that queries in SaaS are usually over single columns or involve simple predicates in which an attribute is compared with a

given value. As can be seen, the performance of Index table layout for such queries is an order of magnitude better than that of the Chunk table layout. These results are given in Table 2 and are illustrated in Figure 19.

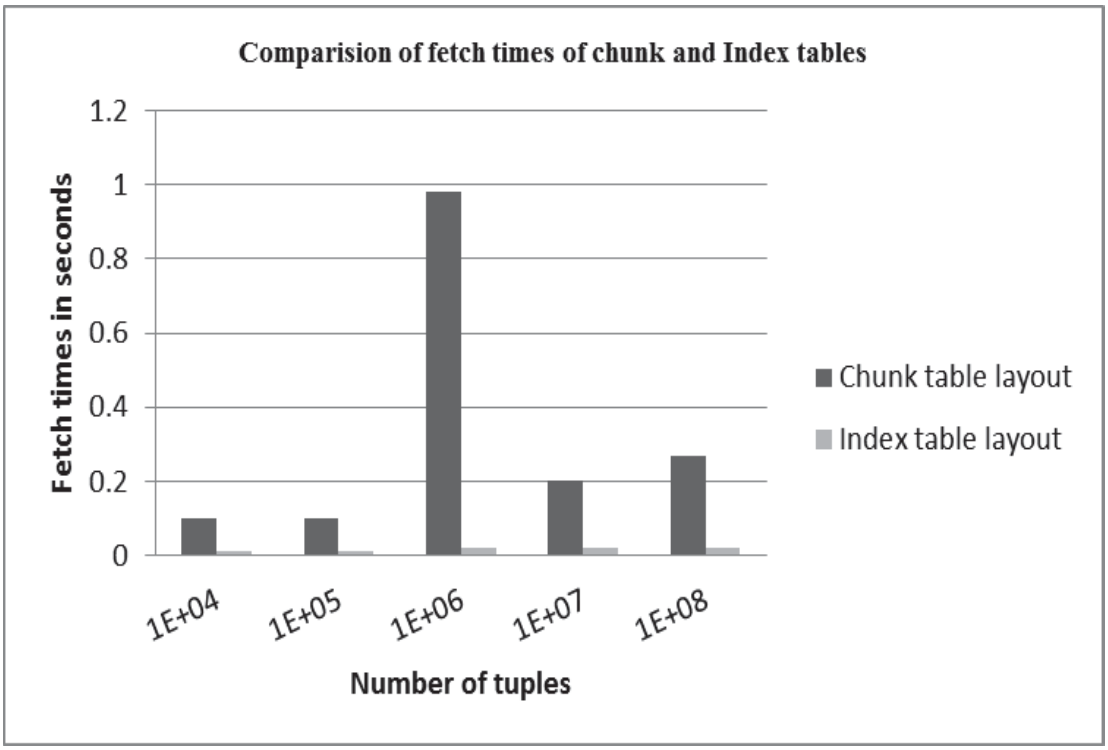

Figure 19: Comparison of execution times of Chunk and Index table layouts for Type 0 queries

- Type 1: These queries are selected from the related literature. For instance, queries considered in [4] were used to compare performance of Chunk table layout using different DBMSs. For this, they considered types of queries for various layouts such as Private and Extension layouts. The queries they used in their experiments and evaluated against different layouts were Select 1, Select 50, Select 1000, Insert 1, Insert 50, Update 1, and Update 100 attributes. Figures 19 to 23 show execution times for the Chunk and Index table layouts for tables with tuples 10^4, 10^5, 10^6, 10^7, 10^8, respectively. It can be observed that for Type 1

queries, the performance of Chunk table and Index table layouts are almost similar. It can be seen, that the query execution times gets approximately 10 times larger as the number of tuples increases 10 times.

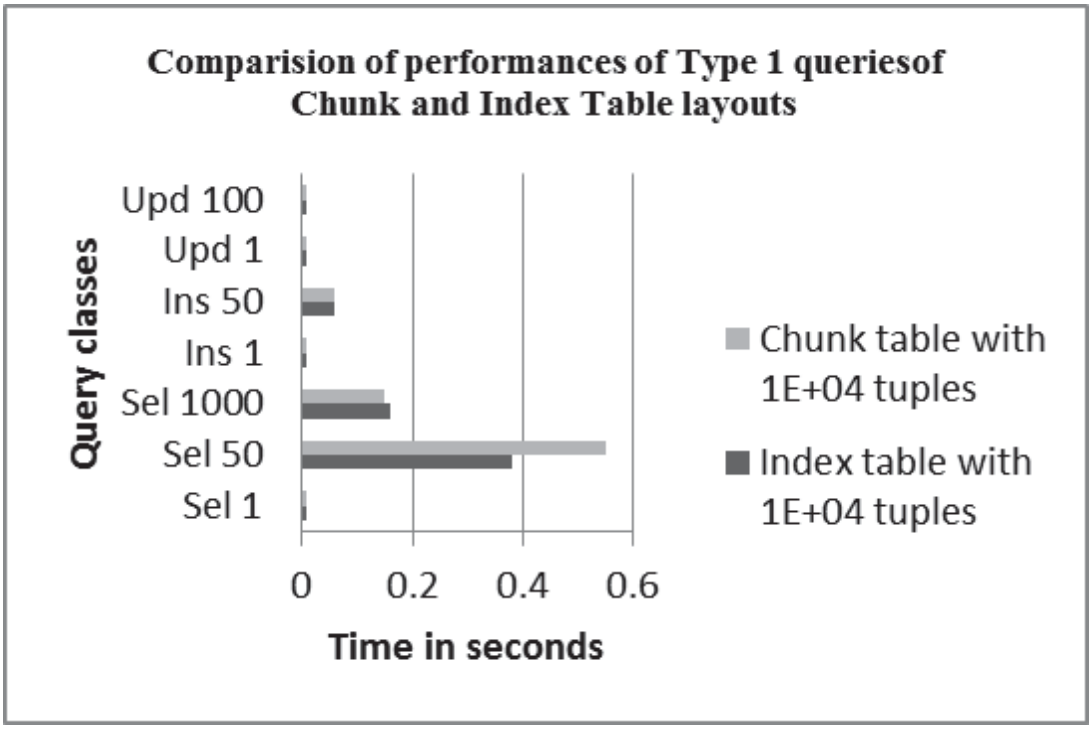

Figure 20: Comparison of execution times of Chunk and Index table layouts for 10^4 tuples and Type 1 queries

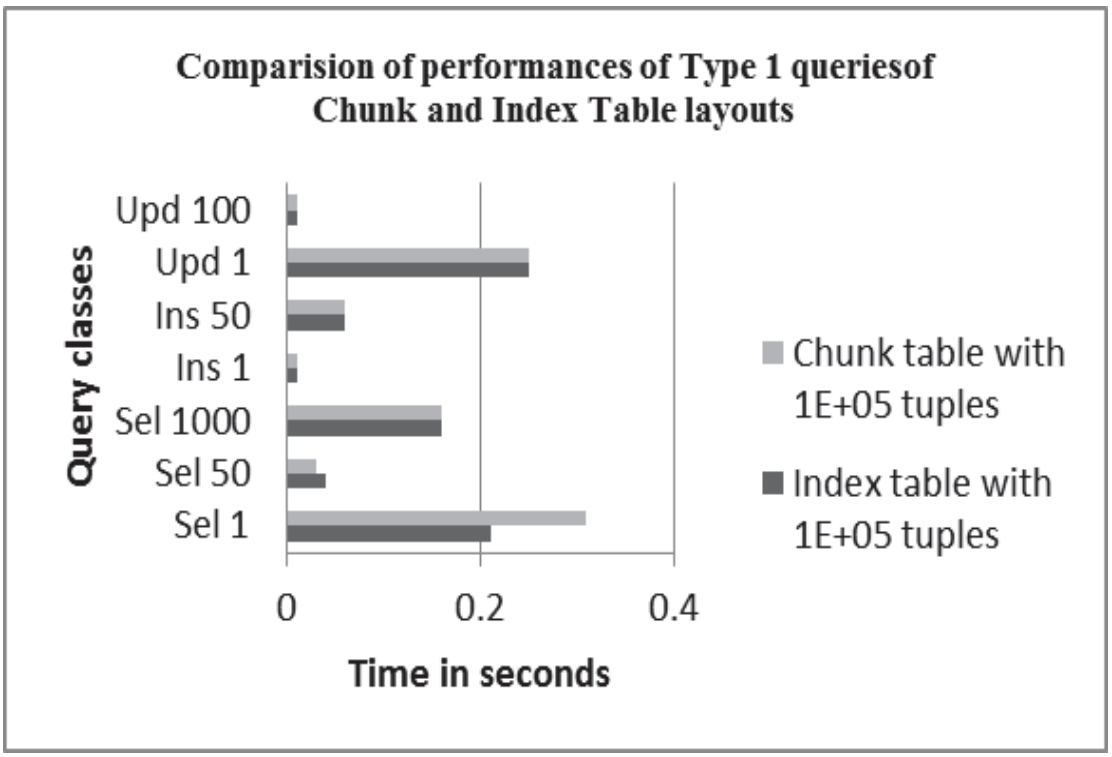

Figure 21: Comparison of execution times of Chunk and Index table layouts for 10^5 tuples and Type 1 queries

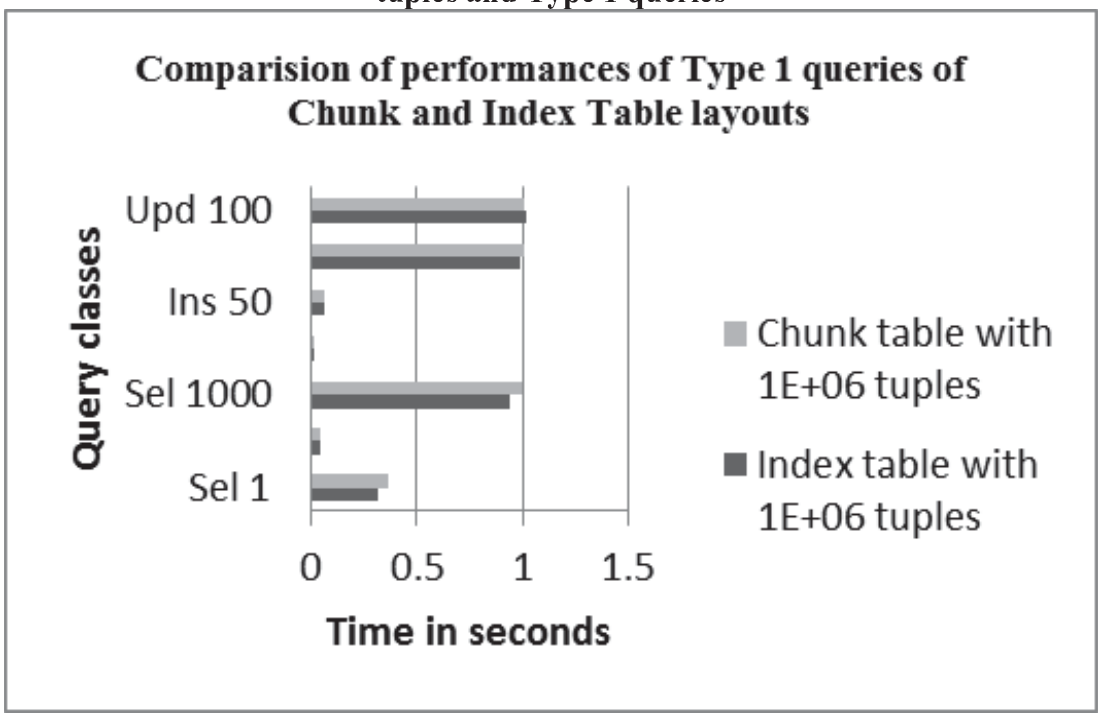

Figure 22: Comparison of execution times of Chunk and Index table layouts for 10^6 tuples and Type 1 queries

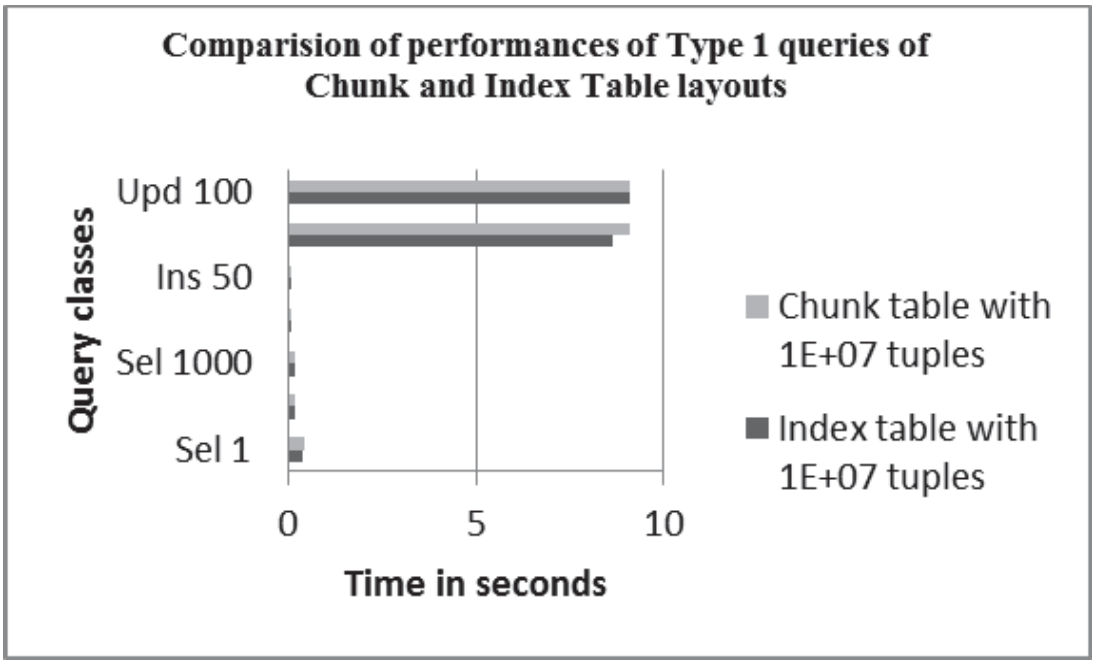

Figure 23: Comparison of execution times of Chunk and Index table layouts for 10^7 tuples and Type 1 queries

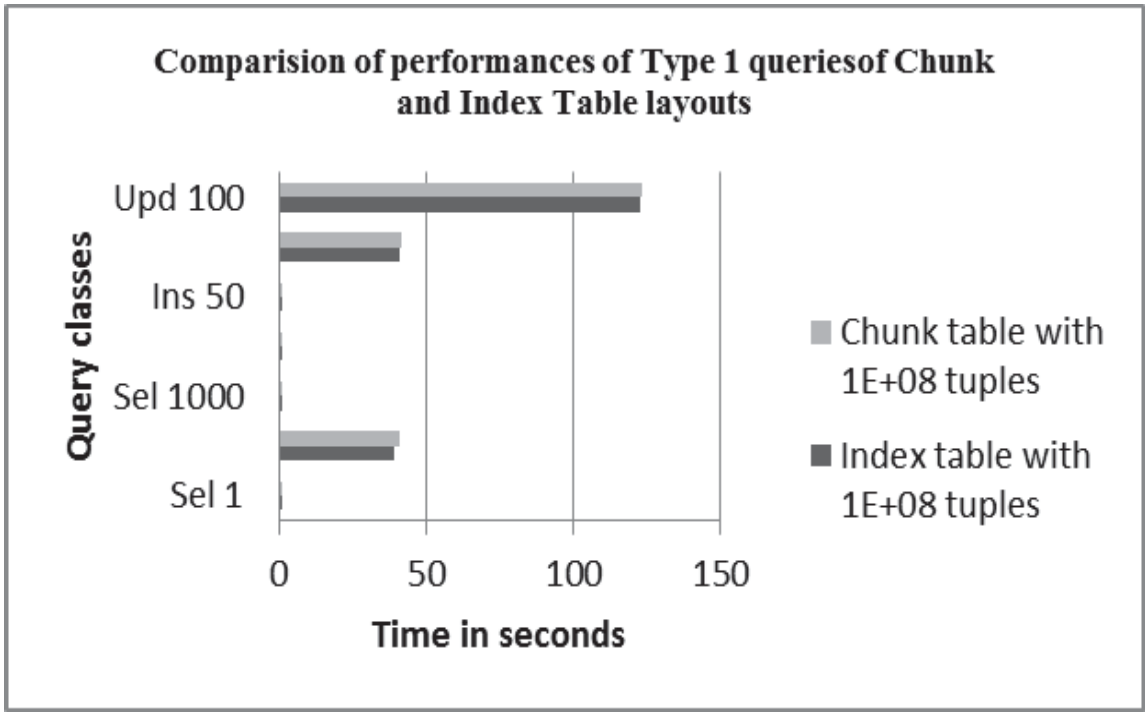

Figure 24: Comparison of execution times of Chunk and Index table layouts for 10^8 tuples and Type 1 queries

- Type 2: Queries that can be posed against the Index table layout, but not the Chunk table layout.

  One limitation of Chunk table layout is that queries including predicates on different columns are not possible to express. This restricts the user/service provider to pose a variety of practical queries. We adapted and a set of such queries from [6] and used in our experiments. Execution times for tables with different number of tuples ranging from 10^4 to 10^8 are presented in Table 8. The same is illustrated in Figure 24.

For Type 2 queries, we considered the following 6 queries in our experiments. This set of queries was compiled from queries mentioned in [6] and [4]. We complement these with some more queries. We have a variety of queries involving aggregation, nested select, join, multiple joins and conjunction.

Query1: Select distinct names of hospitals.

Query for the Index table:

```
SELECT DISTINCT (hospital) FROM ext_hospital;
```

We remark that formulating a corresponding query for the Chunk table layout is not possible, since we query over the column Str1 which can take any value of type String. Having predicates over the columns Chunk, row, table, and tenant here would not help either.

Query 2: Find the number of entries for tenant 17.

Query formulation for the Index table:

```
SELECT COUNT(*) FROM ext_base WHERE tenant=17;
```

A corresponding query for the Chunk table layout would be:

```
SELECT COUNT (*) FROM Chunk_int_str WHERE tenant=17;
```

However, this query does not return the expected result, explained as follows. The number of tuples in the Chunk table grows as the number of columns in Private table grows. In our tables, since each tuple is represented in two columns in the Chunk table, we cannot assume that the number of entries for Tenant 17 would be a multiple of a fixed number (2 in our case).

Query 3: List the hospitals and the number of beds each has.

Query for the Index table:

```
SELECT hospital, number_beds
FROM ext_hospital JOIN ext_beds ON
  ext_hospital.Ind_id = ext_beds.Ind_Id AND
  ext_hospital.hospital = 'State';
```

For the Chunk table layout, this query could be expressed as:

```
SELECT number_beds, hospital
```

```
FROM (SELECT Str1 AS hospital, Int1 AS beds
      FROM Chunk_int_str WHERE Tenant = 17)
WHERE hospital = 'State';
```

Query 4: Find the number of beds for Tenant 17.

For the Index table layout, this query could be formulated as follows:

```
SELECT number_beds
FROM ext_beds, ext_base
WHERE ext_beds.Ind_id = ext_base.Ind_Id;
```

The above query could not be expressed over the Chunk Table layout since selecting any value from the attribute "Int1" would include all values that corresponds to other possible attributes of every tenant.

Query 5: Find details of tenant, aid, name, and hospitals.

For the Index table:

```
SELECT tenant, aid, name, hospital
FROM ext_base JOIN ext_hospital ON
ext_base.Ind_id = ext_hospital.Ind_Id;
```

Query 5 cannot be expressed for the Chunk Table layout.

Query 6: Find details of hospitals and beds for each tenant.

This query for the Index table layout can be expressed as follows, which includes two join operations over the common attributes among the tables, namely ext_base, ext_hospital, and ext_beds.

```
SELECT tenant, aid, name, hospital, number_beds
FROM ext_base, ext_hospital, ext_beds
    WHERE ext_base.Ind_id = ext_hospital.Ind_Id AND
        ext_hospital.Ind_Id = ext_beds.Ind_id;
```

It is not possible to express Query 6 over the Chunk Table layout, since in a Chunk table, data values of the same type are stored in one column, and hence having predicates over Tenant, Chunk, and row in the WHERE clause do not result in retrieving the "right" data.

While queries of types 0, 1, and 2 measure expressivity of the schema, we also compare the Chunk table and Index table layouts based on their space requirements. In the Index table layout, common data among tenants is always stored in the base table. The supporting tables would only include necessary rows. This reduces the space utilization to a large extent. We hence compare the sizes of Chunk table and the Base table of the Index table layout. Figure 25 illustrates the results of this comparison. The data illustrated as graphs are shown as tables in the appendix.

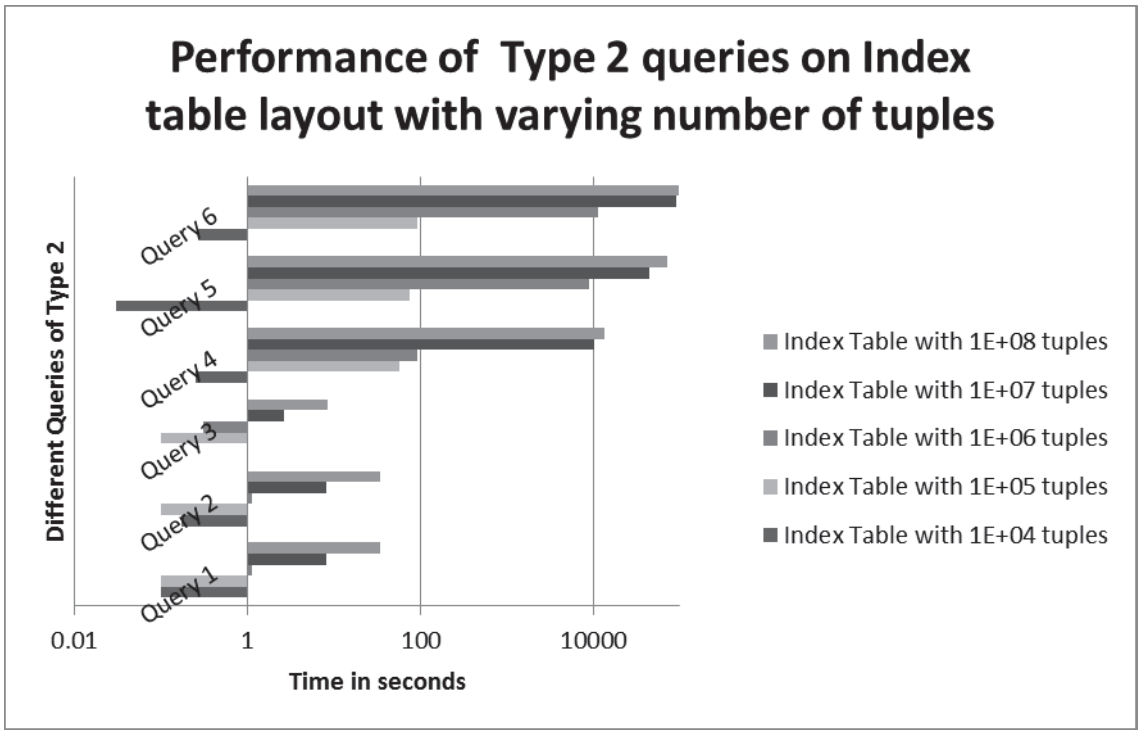

Figure 25: Performances of Type 2 queries over tables with different number of tuples

4.7 Multiple Sparse Table vs. Index Table Layout

In this section we study the Multiple sparse table approach [9] and compare it with our Index table layout. The results of our experiments for evaluating query processing times show that the Index table layout outperforms the Multiple sparse table significantly.

The idea of splitting the data of different tenants adapted in our work was from the Multiple sparse table proposed in [9], which provides also a categorization of tenants based on the attributes required. The two approaches however are different in what they

get split and it is, described as follows: The Index table layout stores the common attributes/columns in the base table and each one of the other attributes in a supporting table. This allows entering values into the supporting tables only if the values exist. The Multiple sparse table, on the other hand, represents the multi-tenant data into multiple sparse tables chosen based on the expected number of attributes required. This kind of categorization provided in multiple sparse tables may not always be helpful as this is not the only criteria to know the tenant's kind. Also we are restricting ourselves to a fixed set of attributes. This does not consider the case where the attributes may be dynamically changing. A typical solution in such a case would be to replicate the existing Multiple sparse table to another multiple sparse table with larger number of attributes. This would lead to data redundancy and duplication. Hence, Multiple Sparse tables cannot be considered as a dynamic and flexible model.

The Index table layout on the other hand is very flexible and scalable. The Index table is scalable in the sense that it can support increasing number of tuples. It can also be observed that as the number of tenants increase, the width of the base table of the Index table still remains similar. This is because in SaaS architecture, the tenants are similar. Hence there are always common attributes between tenants. We use the term "scalable" to address the growing number of tuples. Easy mapping stage between private table layout and the Index table layout makes the later flexible.

The Index table layout accommodates dynamic increase of attributes, implements vertical partitioning in the form of column stores, and reduces nulls significantly. We perform

similar experiments done in [9] to compare the performance of the Multiple sparse table and the Index table layout.

4.7.1. Performance Comparison

We conducted experiments to compare Multiple Sparse table and Index table layout on large tables. For these experiments, we used a desktop computer with Intel Xeon processor, 4 GB RAM. As the DBMS, we used MySQL Workbench 5.2 CE.

To make our results comparable to those reported in [9], we consider 3 schemas in these experiments as follows: Schemas with 10 columns (let us refer to this as Schema5), with 50 columns (called, Schema26), and with 100 columns (called, Schema43). Note that the names of the schemas are named as Schema5, Schema26 and Schema43 as in [9] for consistency purpose. For Single sparse table, all data will be stored in the sparse table with 500 columns. For multiple sparse tables approach, data of Schema5 is stored in a Sparse Table 1 with, say, 30 columns. Similarly, we store data of Schema26 in a Sparse Table 2 with 80 columns, and store the data of Schema43 in a Sparse Table 3 with 200 columns. We then join these Sparse tables and table 'tableJ' which has 50 columns (Column1 to Column 50) and 5000 tuples. We reproduced the same experiments reported in [9] with corresponding data and tables in Index table layout. We perform the join operations for each of the above mentioned three schemas using equivalent Single sparse tables, Multiple sparse tables, and Index table layout.

The query used to perform the join of tables in the Multiple Sparse Table approach is as follows:

```
SELECT * FROM table5 a, table b
        WHERE a.columnName1=b.columnName1 AND a.columnName2=b.columnName2
        AND .. .. AND a.columnName5=b.columnName5 and tenant=17;
```

Similar queries for Schema26 and Schema43 are also performed.

The corresponding query over the Index table layout is as follows:

```
SELECT columnName1, columnName2 ,.., columnName5 FROM ext_base
        WHERE tenant=17;
```

We observed that join attributes for Multiple sparse tables did not require a join operation for the corresponding Index table layout. This is because the corresponding attributes were the attributes in the base table for the Index table layout. Join operation between the base table and one or more of the supporting tables are performed only when some attributes in the query are not in the base table. This situation is application dependent. The Index table layout is basically a flexible schema and the base table can dynamically shrink or grow in size. Hence, the larger the width of the base table, the lesser the join operations. In our experiments, we had the minimum number of join operations to perform.

The results of these experiments are shown in Figures 26, 27 and 28, for Schema5 with 10 columns, Schema26 with 50 columns, and Schema 43 with 100 columns, respectively. For consistency with results in [9], we perform the experiments on tables varying by 5000 tuples. As shown in graphs in [9], in the x-axis scale for Figures 25, 26 and 27, by K tuples, we mean 1000 tuples.

It is observed that for an increase of every 5000 tuples, the execution times of Single sparse table increases in about 2000 milli-seconds. The Multiple sparse table performs relatively better compared to the Single sparse table. The Index table layout for the corresponding data has outperforms the Multiple sparse table approach by about an order of magnitude. It can be observed that the Index table layout fetches data for all join test operations of Schemas, schema5, schema26 and schema43 in nearly10 milli-seconds. The execution times are constant as the operations are over individual columns.

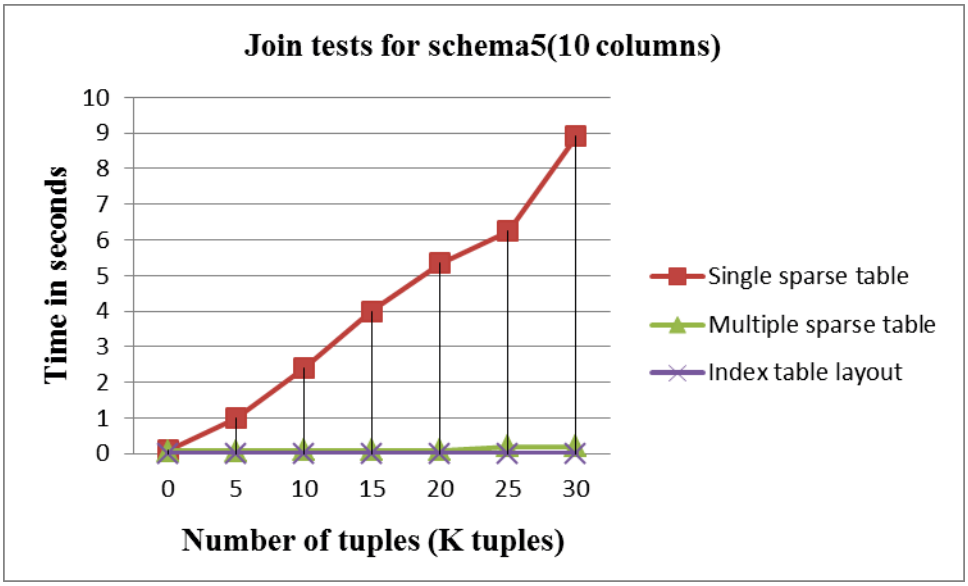

Figure 26: Join tests for schema5

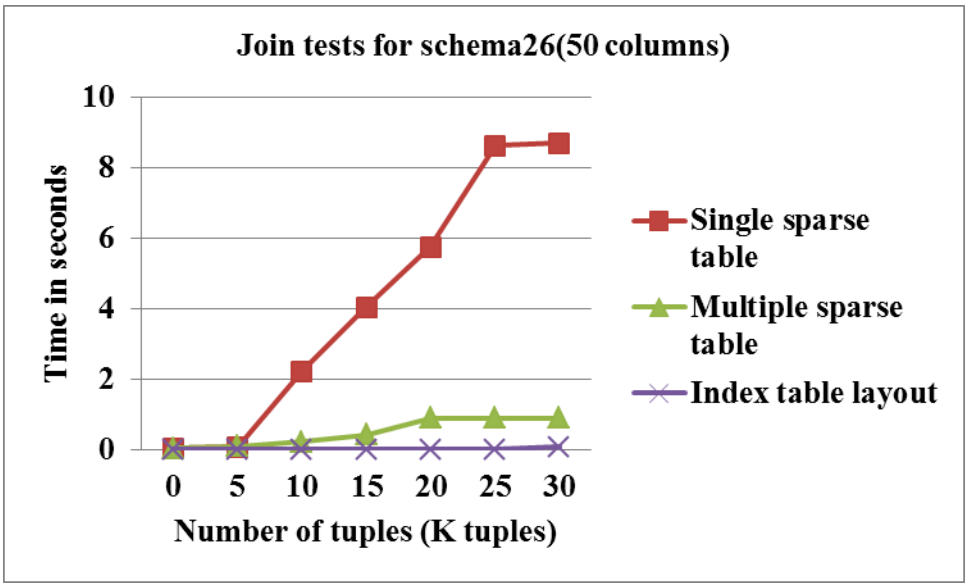

Figure 27: Join tests for schema26

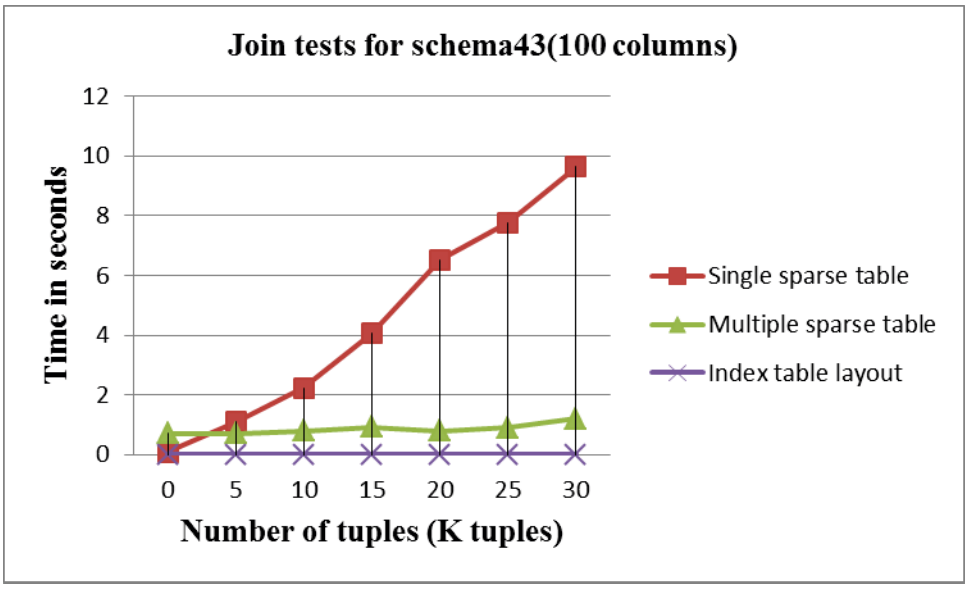

Figure 28: Join tests for schema43

4.8 Summary

We proposed Index table layout, a model for Shared table variation for SaaS that improves over the Chunk model in space utilization as well as query processing time. The queries considered help better highlight the richness of the proposed model in terms of query expressivity.

In terms of storage utilization, our experiments show that the Index table layout takes about two third of the space taken by the Chunk table layout. In terms of query processing times we observe that for queries of Type 0, our proposed model is an order of magnitude faster than the Chunk table layout. We noted similar performance superiority for processing Type 1 queries, which could be posed against both the Chunk table and the Index table layouts. We also compiled a new set of queries as Type 2, which could be supported only by our proposal for being richer in modeling power. Since the Index table layout is a vertically partitioned schema, we can take advantage of column store features in implementation of our model. This advantage is much desired as most of the queries for SaaS are posed on individual columns.

The index column of the base table can be further indexed, sorted, or even hashed if demanded by the application at hand.

In Section 4.7, we pointed out that while the Multiple Sparse table addresses the problem of nulls and performs better than Single sparse table, its performance still suffers from

traditional sparse data processing and management of individual sparse tables. This also results in data redundancy and duplication.

The results of our experiments showed that the Index table layout outperforms the Multiple sparse tables approach in query performance by an order of magnitude. The sparse tables have number of columns to meet the different demands of tenants. This, however, may not always be the right way to categorize tenants. We also note that while the sparse tables have gradient columns, the vertical partitioning technique applied in Index table layout has better flexibility, expressivity, query performance, and space utilization. A disadvantage of Index table layout is the requirement for join operations between base table and supporting tables that are heavy. In SaaS applications, normally the queries are posed on individual columns. Hence it is not very often that many join operations are performed in the same query. In Index table layout, when the multi-tenant data is static, we can also have indexes on joins which speeds up the queries even more.
It should be noted that we use an open source version of MySQL for experiments, as this is an academic experimental setup. The database has been configured to suit very large tables and appropriate tuning parameters have been set to utilize 80% of the RAM capacity for executing queries. We process reasonably large tables with 1 billion records. The free version of MySQL does not support query operations beyond 10^8 tuples for query processing. Number of tuples can be further increased by inserting tuples in batches as opposed to inserting tuples one by one. But while running queries, our desktop computer does not support returning such large results. This is both due to the capacity of RAM as well as the use of free version of MySQL. As is, we make use of a

stable release of MySQL 5.2. In industry, for SaaS, very powerful systems with very high RAM and hard disk capacity are made use with Enterprise editions of DBMSs installed. The Enterprise editions support the setup of a cluster using many servers and support large tables which grow dynamically.

In such case, we expect a better variation in the results of Index table layout to be observed as the number of attributes increase.

Chapter 5

Conclusions and Future Work

There has been an increasing amount of research recently in both academia and industry investigating suitable data models for emerging applications in the context of cloud computing under the SaaS paradigm. In this thesis, we studied the various existing data models including the well-known Chunk table layout, and compared their performance in various aspects. We then proposed the *Index Table* layout as a new data model for multi-tenancy. Through numerous experiments, we show its advantages over existing proposals, including the Chunk model layout. The advantages are in better space utilization and query processing efficiency. As a byproduct of this work, we developed and compiled a collection of existing and new queries, used for testing and comparison purposes in our experiments. These queries are categorized into Type 0, Type 1 and Type 2 queries. Type 0 queries are those which can be posed against the Chunk table as well as Index table layouts. Type 1 queries essentially range from "light" to "heavy" data manipulation queries. Type 2 queries are applicable to the Index table layout but not the Chunk table.

The proposed Index table makes use of the advantages of the "flexibility" and "column store" features. It should be noted that in SaaS applications queries are normally posed on individual columns using RESTful APIs. This is possible in Index table layout as

queries are mostly posed on the base table. The other queries are queries with predicates on supporting tables. These queries too, have predicates on individual columns. Hence Index table layout is appropriate to make use of RESTful APIs.

In terms of space utilization, the Index table layout takes about two third of the space required otherwise by the Chunk table layout. The results of our performance evaluation using different classes of the three different types of queries, show that, compared to the Chunk table layout, our Index table layout is more efficient for Type 0 queries by an order of magnitude. For Type 1 queries, which could be posed against both models, our results indicate similar processing times as the Chunk table layout. Type 2 queries, are posed against our proposed index table layout and not Chunk table layout. These queries range from simple select queries to heavy join operations and their performance is similar to that of Type 1. This is possible due to the benefits of the column store feature and vertically partitioned schema.

As the base table in our model becomes wider, the number of supporting tables decrease, resulting in fewer number of joins operations. Hence the Index table layout supports the growing or shrinking in the width (columns) of the base table. In case, the queries posed require data stored in a number of supporting tables, a wider base table can help reduce the number of joins.

With increase in number of tenants, the width of the base table of the Index table still remains about the same. This is because in SaaS architecture, the tenants are similar.

Hence there are always common attributes between tenants and the size of the base table is not affected.

This research work could be further extended in a number of ways. One idea is to build an index on the index column of the base table and use it to speed up the joins with supporting tables. This could be useful when a schema has to be further extended to a data warehouse like application scenario in which there are many CUBE operations on individual columns.

Another avenue is to improve the space requirement of the proposed model through data compression techniques. The column-store design feature of our proposed model leads to significant increase in query processing performance. The evolution of data model for multi-tenancy should be complemented by suitable techniques for tenant analysis and classification. The patterns in the data of different tenants can be studied and a tenant profiling can be done based on the type of attributes two or more tenants share. The size of the base table does not get affected as the number of tenants increase. It is important that a service provider understands the client/tenant and customize the service to his/her needs, resulting in more user satisfaction and improved business. It should be noted that users with different usage profiles will require different base tables. We would like to explore these ideas as future work.

Bibliography

[1] J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. "Scalable Semantic Web Data Management Using Vertical Partitioning." *In Proceedings of the 33rd International Conference on Very Large Data Bases*, University of Vienna, Austria, September 23-27, 2007, pp 411–422.

[2] S. Acharya, P. Carlin, C. A. Galindo-Legaria, K. Kozielczyk, P. Terlecki, and P. Zabback. "Relational Support for Flexible Schema Scenarios." *In Proceedings of Very Large Databases*, 1(2):1289–1300, 2008, Pages 1289-1300.

[3] R. Agrawal, A. Somani, and Y. Xu. "Storage and Querying of E-Commerce Data." *In VLDB '01: Proceedings of the 27th International Conference on Very Large Databases*, San Francisco, CA, USA, 2001, pp 149–158.

[4] S. Aulbach, D. Jacobs, A. Kemper, M. Seibold. "A comparison of flexible schemas for software as a service." *In SIGMOD Conference*, Rhode Island, USA, July 2009, pp 881-888.

[5] S. Aulbach, M. Seibold, D. Jacobs, A. Kemper. "Extensibility and Data Sharing in Evolving Multi-Tenant Databases." *In International Conference for Data Engineering (ICDE) 2011*, Hannover, Germany, April 11-16, 2011, pp 99-110.

[6] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. "Multi-tenant databases for software as a service: schema mapping techniques." *In SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2008, pp 1195–1206.

[7] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. "Extending RDBMSs to Support Sparse Datasets Using an Interpreted Attribute Storage Format." *In Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, Washington, DC, USA, 2006. IEEE Computer Society, pp 58.

[8] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. "Extending RDBMSs To Support Sparse Datasets Using an Interpreted Attribute Storage Format." *In ICDE'06 Proceedings of 22nd International Conference on Data Engineering*, Atlanta, GA, April 3-8, 2006, pp 58.

[9] W. Chen, S. Zhang, L. Kong, "A Multiple Sparse Tables Approach for Multi-tenant Data Storage in SaaS." *In Proceeding of the 2nd International Conference on Industrial and Information Systems*, Dalian, China, Jun 10-11, 2010, pp 413-416.

[10] E. Chu, J. Beckmann, and J. Naughton. "The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets." *In SIGMOD'07, Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, New York, USA, 2007, pp 821-832.

[11] E. Chu, J. Beckmann, Jeffrey Naughton. "The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets." *In SIGMOD '07*, June 11-14, Beijing, China, 2007, pp 821-832.

[12] F. Chong, G. Carraro, and R. Wolter. "Multi-tenant data architecture".http://msdn.microsoft.com/en-us/library/aa479086.aspx, June2006.

[13] G. P. Copeland and S. N. Khoshafian. "A decomposition storage model." *In SIGMOD '85: Proceedings of the ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1985, ACM, pp 268-279.

[14] C. Bezemer, A. Zaidman. "Multi-tenant SaaS Applications: Maintenance Dream or Nightmare". *In Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, Belgium, September 2010, pp 88-92.

[15] Cunningham, G. Graefe, and C. A. Galindo-Legaria. "PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS." *In (e) Proceedings of the Thirtieth International Conference on Very Large Databases*, Toronto, Canada, August 31 - September 3, 2004, pp 998–1009.

[16] F. S. Foping, Ioannis M. Dokas, John Feehan, Syed Imran. "A New Hybrid Schema-Sharing Technique for Multitenant Applications." *In ICDIM'09*, University of Michigan, Ann Arbor, Michigan, November 2009, pp 1-6.

[17] J. Gray. *Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King*. http://research.microsoft.com/~Gray/talks/Flash_Is_Good.ppt, 2006.

[18] M. Grund, M. Schapranow, J. Krueger, J. Schaffner, and A. Bog. "Shared table access pattern analysis for multi-tenant applications." *In AMIGE '08*, Tianjin, China, September 2008, pp 1–5.

[19] M. Stonebraker, L. A. Rowe, and M. Hirohama, "The implementation of postgres." *In IEEE Trans. Knowl. Data Eng.*, vol. 2, no. 1, 1990, pp 125–142

[20] M. Grund, J. Krueger, C. Tinnefeld, A. Zeier. "Vertical Partitioning in Insert-Only Scenarios for Enterprise Applications." *In IE&EM '09*, Hong Kong, China, December 2009, pp 760-765.

[21] T. Grust, M. V. Keulen, and J. Teubner. "Accelerating XPath evaluation in any RDBMS." *In ACM Transactions on Database Systems (TODS)*, Volume 29 Issue 1, March 2004, pp 91-131.

[22] D. Jacobs, S. Aulbach. "Ruminations on Multi-tenant Datases." *In BTW Proceedings*, Volume 103 of LNI.

[23] D. Jacobs. "Data Management in Application Servers." *In Datenbank-Spektrum,* 2004, pp 760.

[24] Jeffrey M. Kaplan. "Saas: Friend or foe?" *In Business Communications Review*, June 2007, pp 48-53.

[25] R. Krishnamurthy, W. Litwin, and W. Kent. "Language features for interoperability of databases with schematic discrepancies." *In Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, May 29-31, 1991, pp 40–49.

[26] T. Kwok and A. Mohindra. "Resource calculations with constraints, and placement of tenants and instances for multi-tenant SaaS applications." *In Proceedings of International Conference on Service Oriented Computing(ICSOC)*, Volume 5364 of LNCS, Springer, 2008, pp 633-648.

[27] P.A. Boncz. "Monet: A Next Generation DBMS Kernel For Query-Intensive Applications." *Ph.D. Thesis, Universiteit van Amsterdam*, Amsterdam, The Netherlands, May 2002.

[28] O. Schiller, B. Schiller, A. Brodt, B. Mitschang. "Native Support of Multitenancy in RDBMS for Software as a Service." *In EDBT 2011*, Uppsala, Sweden, March 22-24, 2011, pp 117- 128.

[29] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. "C-Store: A Column-oriented DBMS." *In Proceedings of the 31st International Conference on Very Large Data Bases*, Trondheim, Norway, August 30- September 2, 2005, pp 553-564.

[30] M Stonebraker. "The Case for Partial Indexes." *In ACM SIGMOD*, Volume 18, Issue 4, December 1989, pp 4-11.

[31] M. Stonebraker, L. A. Rowe, and M. Hirohama. "The implementation of postgres" , *In IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, 1990, pp. 125–142.

[32] Zhi Hu Wang, Chang Jie Guo, Bo Gao, Wei Sun, Zhen Zhang, and Wen Hao An. "A study and performance evaluation of the multi-tenant data tier design patters for service oriented computing." *In Proceedings of the International Conference On e-Business Engineering (ICEBE),* Xi'An, China, 2008, pp 94-101.

[33] Anatomy of MySQL on the GRID: http://blog.mediatemple.net/weblog/2007/01/19/anatomy-of-mysql-on the-grid/

[34] ibm.com: http://www.ibm.com/

[35] mysql.com: http://www.mysql.com/

[36] NetSuite NetFlex: http://www.netsuite.com/portal/products/netflex/main.shtml.

[37] http://www.tpc.org/tpcc/

[38] Salesforce AppExchange.: http://www.salesforce.com/appexchange/about

[39] WebEx: http://www.webex.com/

[40] Zimbra: http://www.zimbra.com/

Appendix

Experimental Data

We conducted experiments using various tables of large sizes. For these experiments, we used an Intel Xeon Processor with a 4 GB RAM and the MySQL Workbench 5.2 CE. We first wrote scripts to randomly generate large data for the Chunk table layout from which we then constructed the individual tables as well as the tables for Index table layout. Hence, there is a one to one correspondence between data in the private tables and the data in the index and Chunk table layouts.

Data Generation

We created a Chunk table with 10^4 tuples and constructed the appropriate Private and Index tables. We then kept increasing the number of tuples by multiples of ten. We compare the results of the Chunk tables with 10^4, 10^5, 10^6, 10^7, and 10^8 tuples and their corresponding tables rewritten to our Index table layout.

The tables below show the results of various experiments.

No. of tuples	Query times of Chunk table layout in seconds	Query times of Index table layout in seconds
10^4	0.1	0.01
10^5	0.1	0.01

10^6	0.98	0.02
10^7	0.2	0.02
10^8	0.27	0.02

Table 2: Comparison of execution times in seconds for query of Type 0

Class	Chunk table with 10^4 tuples	Index table with 10^4 tuples
Sel 1	0.01	0.01
Sel 50	0.55	0.38
Sel 1000	0.15	0.16
Ins 1	0.01	0.01
Ins 50	0.06	0.06
Upd 1	0.01	0.01
Upd 100	0.01	0.01

Table 3: Comparison of execution times in seconds for query of Type 1 over 10^4 tuples

Class	Chunk table with 10^5 tuples	Index table with 10^5 tuples
Sel 1	0.31	0.21
Sel 50	0.03	0.04
Sel 1000	0.16	0.16
Ins 1	0.01	0.01
Ins 50	0.06	0.06
Upd 1	0.25	0.25

Upd 100	0.01	0.01

Table 4: Comparison of execution times in seconds for query of Type 1 over 10^5 tuples

Class	Chunk table with 10^6 tuples	Index table with 10^6 tuples
Sel 1	0.36	0.32
Sel 50	0.04	0.04
Sel 1000	1	0.94
Ins 1	0.01	0.01
Ins 50	0.06	0.06
Upd 1	1.01	0.99
Upd 100	1.01	1.02

Table 5: Comparison of execution times in seconds for query of Type 1 over 10^6 tuples

Class	Chunk table with 10^7 tuples	Index table with 10^7 tuples
Sel 1	0.42	0.37
Sel 50	0.16	0.17
Sel 1000	0.16	0.16
Ins 1	0.01	0.01
Ins 50	0.06	0.06

Upd 1	9.12	8.68
Upd 100	9.12	9.1

Table 6: Comparison of execution times in seconds for query of Type 1 over 10^7 tuples

Class	Chunk table with 10^8 tuples	Index table with 10^8 tuples
Sel 1	0.6	0.6
Sel 50	41	38.67
Sel 1000	0.16	0.17
Ins 1	0.01	0.01
Ins 50	0.06	0.06
Upd 1	41.2	41.06
Upd 100	123.62	122.42

Table 7: Comparison of execution times in seconds for query of Type 1 over 10^8 tuples

Query	Index Table with 10^4 tuples	Index Table with 10^5 tuples	Index Table with 10^6 tuples	Index Table with 10^7 tuples	Index Table with 10^8 tuples
Query 1	0.1	0.1	1.13	8.22	33.93
Query 2	0.17	0.1	1.13	8.22	33.93
Query 3	0	0.1	0.31	2.62	8.59
Query 4	0.25	57.73	93.3	10256	13260.8

Query 5	0.03	76.3	8905.4	44640.1	71640
Query 6	0.27	93.45	11496.5	90565.9	97654.9

Table 8: Performances of different queries of Type 2 over different size tables