Adaptive Database Schema Design for Multi-Tenant Data Management

Jiacai Ni, Guoliang Li, Lijun Wang, Jianhua Feng, Jun Zhang, Lei Li

Abstract—Multi-Tenant data management is a major application of Software as a Service (SaaS). For example, many companies want to outsource their data to a third party which hosts a multi-tenant database system to provide the data management service. The multi-tenant database system requires to have high performance, low space requirement and excellent scalability. One big challenge is to devise a high-quality database schema. Independent Tables Shared Instances (ITSI) and Shared Tables Shared Instances (STSI) are two state-of-the-art approaches to design the schema. However, they suffer from some limitations. ITSI has poor scalability since it needs to maintain large numbers of tables. STSI achieves good scalability at the expense of poor performance and high space overhead. Thus it calls for an effective schema design method to address these problems.

In this paper, we propose an adaptive database schema design method for multi-tenant applications. We trade-off ITSI and STSI and find a balance between them to achieve good scalability and high performance with low space requirement. To this end, we identify the *important attributes* and use them to generate an *appropriate* number of *base tables*. For the remaining attributes, we construct *supplementary tables*. We discuss how to use the kernel matrix to determine the number of the base tables, apply graph-partitioning algorithms to construct the base tables and evaluate the *importance* of attributes using the well-known PageRank algorithm. We propose a cost-based model to adaptively generate the base tables and supplementary tables. Our method has the following advantages. First, our method achieves high scalability. Second, our method achieves high performance and can trade-off the performance and space requirement. Third, our method can be easily applied to existing databases (e.g., MySQL) with minor revisions. Fourth, our method can adapt to any schemas and query workloads including both OLAP and OLTP applications. Experimental results on both real and synthetic datasets show that our method achieves high performance and good scalability with low space requirement, and outperforms state-of-the-art methods.

Index Terms—SaaS, Multi-Tenant, Adaptive Schema Design

1 Introduction

Software as a service (SaaS) has attracted significant attention recently and become a common software delivery model for many business applications[1]. SaaS has been adopted by many leading enterprize Internet or software companies, e.g., Google, Amazon, Oracle, Microsoft, SAP, IBM and Salesforce.

Multi-tenant data management is a major application of SaaS. For example, many companies want to outsource their data to a third party which hosts a multi-tenant database system to provide the data management service. Each company is called a tenant. Tenants manage and query their data just as the traditional databases on their local computers, by posing SQL queries to the database system.

- *Jiacai Ni, Guoliang Li, and Jianhua Feng are with Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing 100084, China. E-mail: njc10@mails.tsinghua.edu.cn; {liguoliang, fengjh}@tsinghua.edu.cn*
- *Lijun Wang is with Network Research Center, Tsinghua University, Beijing 100084, China. E-mail: wanglijun@cernet.edu.cn*
- *Jun Zhang is with SINA Weibo.com platform, Beijing 100080, China. E-mail: jiangsukid@gmail.com*
- *Lei Li is with IBM China Development Lab, Beijing 100085, China. E-mail: leili@cn.ibm.com*

1. http://en.wikipedia.org/wiki/Software_as_a_service

In most traditional enterprises, databases are deployed on dedicated database servers. Usually these servers are not fully utilized during much of the time. It is reported in traces from almost 200 production servers from different organizations, an average CPU utilization is less than 4 percent [18]. The extreme low utilization can be solved by consolidating multiple databases on one or fewer machines, reducing hardware and operational costs. The multi-tenant data management system amortizes the cost of hardware, software and professional services to a large number of tenants and thus significantly reduces per-tenant cost by increasing the scale. As the service provider wants to support as many tenants as possible, the multi-tenant database system requires to have excellent performance, low space requirement and good scalability. One big challenge is to devise a high-quality database schema, which is a very important factor in multi-tenant databases.

To our best knowledge, Independent Tables Shared Instances (ITSI) and Shared Tables Shared Instances (STSI) are two state-of-the-art approaches to design the schema. However, they suffer from some limitations. ITSI has poor scalability since it needs to maintain large numbers of tables (see Section 2.2.2). STSI achieves good scalability at the expense of poor performance and high space overhead (see Section 2.2.3). Thus it calls for an effective schema design method to address these problems.

In this paper, we propose an adaptive database schema design method for multi-tenant applications. We trade-off

Digital Object Indentifier 10.1109/TKDE.2013.94

ITSI and STSI and find a balance between the two methods to achieve good scalability and high performance with low space requirement. To this end, we identify the *important attributes* and use them to generate several *base tables*. For each of other attributes, we construct *supplementary tables*. We discuss how to use the kernel matrix to determine the number of the base tables, apply the graph-partitioning algorithm to construct the base tables and evaluate the *importance* of attributes using the well-known PageRank algorithm. We develop a cost-based model to adaptively generate the *base tables* and *supplementary tables*. To summarize, we make the following contributions.

- We propose an adaptive database schema design method for multi-tenant applications. Our method has the following advantages. First, our method achieves high scalability as the number of tables depends on the number of attributes shared by tenants, instead of the number of tenants. Second, our method can trade-off the performance and space requirement. Third, our method can be easily applied to existing databases (e.g., MySQL) with minor revisions. Fourth, our method can adapt to any schemas and query workloads including both OLAP and OLTP applications.
- We propose to use the kernel matrix to determine the number of the base tables and adopt the graph-partitioning algorithm to construct the base tables.
- We analyze the importance of attributes using the well-known PageRank algorithm and discuss how to identify the *important* attributes.
- We propose a cost-based model to automatically generate high-quality database schema.
- Experimental results on both real and synthetic datasets show that our method achieves high performance and good scalability with low space requirement and outperforms state-of-the-art methods.

The rest of this paper is organized as follows. In Section 2, we formulate the problem of multi-tenant database schema design and introduce three major approaches. Section 3 introduces the basic idea of our method. In Section 4 and Section 5 we analyze how to build base tables. Experimental results are provided in Section 6. We review related work in Section 7. In Section 8 we conclude the paper.

2 PRELIMINARIES

In this section, we first formulate the multi-tenant database schema design problem in Section 2.1 and then review existing studies in Section 2.2.

2.1 Problem Formulation

In multi-tenant data management applications, tenants outsource their data to the service provider which devises multi-tenant databases to manage the multi-tenant data. Tenants pose SQL queries to the system which returns the corresponding answers. Next we formalize the problem.

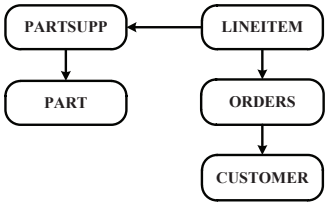


Fig. 1. Five Tables used in TPC-H Benchmark

TABLE 1
Source Tables

(a) CUSTOMER Table for Tenant 1

C_CUSTKEY	C_NAME	C_ADDRESS	C_SALARY	C_NATION
0004	Jim	New York	8000	Germany
0015	Mike	Shanghai	7000	France

(b) CUSTOMER Table for Tenant 4

C_CUSTKEY	C_NAME	C_ADDRESS	C_SALARY
0009	Mary	London	6500
0047	Kate	Paris	7500

(c) CUSTOMER Table for Tenant 9

C_CUSTKEY	C_NAME	C_ADDRESS	C_AGE
0123	Lucy	Tokyo	28

Each tenant outsources a database with a set of tables $\{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n\}$. Each table is called a *source table* . To provide each tenant with fast responses, the multi-tenant database requires to achieve high performance. To serve large numbers of tenants, the multi-tenant database needs to have excellent scalability and low space requirement. To achieve these goals, one big challenge of multi-tenant databases is to devise high-quality schemas. It means the service provider needs to redesign the database schema to efficiently manage the data. The redesigned tables in the multi-tenant databases are called *physical tables* . In this paper we study how to adaptively design high-quality physical tables.

For example, we choose five core tables from TPC-H[1] as the source tables, as illustrated in Figure 1. Here for simplicity, we only use table `CUSTOMER` as our running example as illustrated in Table 1. In our example, we only show three tenants.

2.2 State-of-the-art Studies

In this section, we review existing multi-tenant database schema design methods and discuss their limitations.

2.2.1 Independent Databases and Independent Database Instances (IDII)

The first approach to designing a multi-tenant database is Independent Databases and Independent Instances (IDII). In this method, hardware will be shared by different tenants. The service provider will create one specific database to serve each tenant. It means each tenant can store source tables in its own database.

Obviously, IDII is easy to be deployed and can be built directly on top of any current database. IDII has good data isolation and security. However, the maintenance cost of IDII is very expensive. To manage different database instances, the service provider needs to do much configuration work including some files and parameters. In addition,

for each instance the system will allocate a number of memory. For example, the initialization of a new MySQL instance will consume 30M memory[24]. Furthermore, in the IDII approach, the number of databases in the server grows in proportion to the number of tenants. For example, 5,000 tenants will involve 5,000 database instances. Thus, the maintenance cost for the service provider is extremely large and it means the scalability of IDII is very poor.

2.2.2 Independent Tables and Shared Database Instances (ITSI)

The second multi-tenant schema design method is Independent Tables and Shared Instances (ITSI). In this approach, tenants share not only the hardware but also the database instance. The service provider uses the same database instance to store all source tables from multiple tenants. ITSI maintains *private tables* for each tenant. As source tables from different tenants may create tables with the same name, we need to add a tenant ID to distinguish the tables from different tenants. Thus, submitted queries can be translated using the tenant ID so as to return correct answers to the corresponding tenant. In our running example, ITSI creates a shared database for all the tenants. For each tenant, we use its source tables as its private tables.

As ITSI manages a single database instance instead of multiple database instances, ITSI can reduce the maintenance cost compared with IDII. Thus ITSI provides better scalability than IDII. However, the number of private tables in the shared database grows in proportion to the number of tenants. Then the scalability of ITSI is limited by the number of tables that a database system can well support. In database a number of buffer pages need to be allocated for the meta-data of each table. When the number of tables is large the memory cost will increase obviously. Contending the left memory among many tenants will become the bottleneck. The performance on a server begins to degrade when the number of tables is over 50,000[12]. So the scalability of ITSI is still the limitation for the multi-tenant database to provide service for a large number of tenants.

2.2.3 Shared Tables and Shared Database Instances (STSI)

The third multi-tenant database schema design method is Shared Tables and Shared Instances (STSI). In STSI, different tenants share not only database instance but also tables. The service provider uses big tables to manage the multi-tenant data. The attributes of the big table are the union of attributes of all tenants. All the tuples from each tenant will be consolidated into the big table. In order to distinguish the tuples from different tenants, we add a column tenant ID (TID). Notice that some tenants may have no attributes in the big table (the attributes may be from other tenants), and thus we have to set `NULLs` in the big table for these attributes from such tenants. For example, in our running example, STSI builds a big table (Table 2).

Using STSI, the service provider can reduce the maintenance cost significantly since the total number of big tables is determined only by the maximal number of source tables from different tenants. Thus STSI has better scalability than IDII and ITSI. However, STSI has some limitations. First, if some attributes are shared by a small number of tenants, it will involve large numbers of `NULLs` and wastes large amounts of space. Second, even if many databases can deal with the `NULLs` efficiently, Hui et al. [24] revealed the performance is obviously degraded when database tables become too sparse. Some column oriented features such as the indexes or integrity constraints can not be efficiently used to improve the performance. In addition, the number of indexes that can be built on one MySQL table has limitation[2]. Thus when the number of attributes becomes large we can not fully use the index technique and performance will be dramatically degraded.

As all of state-of-the-art methods have various limitations, in the paper we propose an adaptive method to design database schemas to address these limitations.

3 Our Adapt Method

In this section, we first introduce the overview of ADAPT method in Section 3.1. In Section 3.2 we will present the framework of our adaptive method. Then we will show the advantages of ADAPT method in Section 3.3.

3.1 Overview

The STSI approach achieves high scalability at the expense of involving large numbers of `NULLs` by consolidating different tenants into one table. It will waste much unnecessary space and degrade the performance. Compared with STSI, the scalability of ITSI is poor since the number of tables will increase in proportion to the number of tenants. To address these problems, in this paper we trade-off the two methods and find a balance between the two methods so as to achieve good scalability, high performance, and low space requirement.

Basic Idea: In real applications, many outsourced tables from different tenants are topic-related, and the number of the attributes configured by all the tenants is not large. Based on this observation, we build the physical tables from the attribute level instead of the tenant level. We first extract the *highly important attributes* from the tenants and build several *base tables* using such *important* attributes. Then for the remaining unimportant attributes, we build *supplementary tables* for each of them, like column-based tables. To this end, we propose an adaptive method (called ADAPT) to build base tables and supplementary tables based on database schemas of different tenants and query workloads. Obviously in ADAPT the number of tables is much smaller than the total number of distinct attributes and will not grow in proportion to the number of tenants.

Next we discuss how to design the base tables and supplementary tables in general. For ease of presentation, we first introduce several concepts.

Definition 1 (Common Attributes): An attribute from the source tables is called a common attribute if it is a highly

TABLE 2
CUSTOMER Table Layout in STSI

TID	C_CUSTKEY	C_NAME	C_ADDRESS	C_SALARY	C_AGE	C_NATION
1	0004	Jim	New York	8000	NULL	Germany
1	0015	Mike	Shanghai	7000	NULL	France
4	0009	Mary	London	6500	NULL	NULL
4	0047	Kate	Paris	7500	NULL	NULL
9	0123	Lucy	Tokyo	NULL	28	NULL

shared attribute, i.e., the ratio of the number of tenants containing the attribute to the total number of tenants is not smaller than a given threshold τ.

As the common attributes are highly shared by multiple tenants, we can say they are very important. For example, in Table 2, supposing $\tau = 0.8$, as C_NAME and C_ADDRESS are shared by all the tenants 1, 4 and 9, they are the common attributes.

Definition 2 (Star Attributes): An attribute from the source tables is called a star attribute if it is a primary key and it may be referenced by some other source tables.

As the star attributes will be used by multiple tables, they will lead to low performance since they may involve many costly join operations. Thus we can say these star attributes are very important. In table CUSTOMER the attribute C_CUSTKEY is the primary key and is referenced by table ORDERS as illustrated Figure 1, then it is the star attribute.

Definition 3 (Uncommon Attributes): An attribute is called an uncommon attribute if it is neither a common attribute nor a star attribute.

Besides the common attributes and the star attributes, some of the uncommon attributes are also important. We use an example to show our idea. For example, consider the following SQL query from a tenant.

```
SELECT C_CUSTKEY,C_NAME,C_SALARY
FROM CUSTOMER
ORDER BY C_CUSTKEY;
```

If C_SALARY and the common attribute C_NAME are from different physical tables, there will be a join operation. Because C_SALARY occurs only in the SELECT clause and the tuples containing C_SALARY account for 80 percent of the total, thus the join cost is large. On the contrary, if C_SALARY is inserted into the base table, the costly join operation will be avoided. Moreover it will not involve huge space as only one NULL is produced. Thus attribute C_SALARY is very *expensive* in terms of query processing but very *light* in terms of space, and we call it the *dependent attribute* . We will formally define the concept and discuss how to identify the dependent attributes in Section 5.

For example, in the given workload, Table 3 lists the number of occurrence times of those *expensive* attributes in the costly operation. These costly operations are caused by plenty of join operations between the *expensive* attribute and base tables or even with the aggregate function. In Table 3 we also record the number of increasing NULLs if this expensive attribute is inserted into the corresponding base table. Meanwhile, we need to record the corresponding base table number for each *expensive* attribute. We can take the attribute C_SALARY as a dependent attribute.

TABLE 3
Expensive Attribute List

ATTR_NAME	BTable_NO	OCC_TIME	NULL_NUM
C_SALARY	1	10	1
C_AGE	1	5	4
C_NATION	1	2	2

3.2 Adaptive Schema Design

In this section, we propose an adaptive schema design method to construct the base tables and supplementary tables. For simplicity, common attributes, star attributes and the dependant attributes are all called *important attributes*. First, we use all important attributes to construct several base tables. Then, for the remaining unimportant attributes usually shared by a few tenants, we build supplementary tables for each of such attributes.

Notice that similar to STSI, for both the base tables and supplementary tables we need to add a column - Tenant ID (TID) to distinguish which tenants own the tuple. In addition, we also add the column of the primary key of the source table to join the base tables and supplementary tables when necessary so as to reconstruct the source tables. Based on ADAPT we maintain the attribute mapping between logical source tables and physical tables. The physical tables refer to base tables and supplementary tables. A tenant submits a query to the multi-tenant database based on its own source tables, the system uses the mapping and TID to locate the tuples in the physical tables and answers are returned according to the physical tables. The translation is very simple, the response to each query is very fast.

Figure 2 shows the architecture of our ADAPT method and Table 4 describes the table layouts in our running example. For simplicity, in this example we build only one base table and two supplementary tables. In the real complicated applications, the number of the base tables can be more than one.

In Section 3.1 we have known that the common attributes, the star attributes and the dependent attributes are all *important* attributes. Among them, we can easily get the star attributes based on primary keys. However, for the common attributes and dependent attributes, the selection decisions are not that easy to make.

First, for the common attributes, given different τ we have different numbers of common attributes. Under different criterion we can group them into different clusters

TABLE 4
CUSTOMER Table layout in ADAPT

(a) Base Table

TID	C_CUSTKEY	C_NAME	C_ADDRESS	C_SALARY
1	0004	Jim	New York	8000
1	0015	Mike	Shanghai	7000
4	0009	Mary	London	6500
4	0047	Kate	Paris	7500
9	0123	Lucy	Tokyo	NULL

(b) Supplementary Tables

C_AGE Table

TID	C_CUSTKEY	C_AGE
9	0123	28

C_NATION Table

TID	C_CUSTKEY	C_NATION
1	0004	Germany
1	0015	France

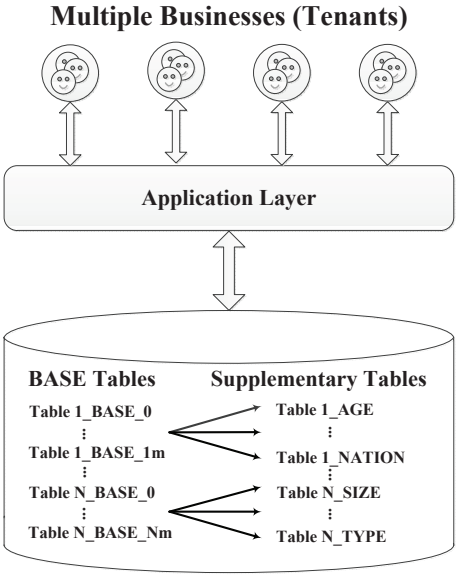


Fig. 2. Architecture of ADAPT

and generate different base tables. This will directly affect the number and the quality of base tables, thus selecting and clustering the common attributes *adaptively* is significant.

Second, the dependant attributes reply on the common attributes, the database schema and query workloads and they also need to be selected *adaptively*.

To sum up, the adaptive features can be categorized into two parts in our ADAPT method. One is the selection and clustering for the common attributes, and the other is the selection for the dependent attributes.

Then the processes of building the base tables can be summarized as the following steps:

1) Determine a proper τ, extract the common attributes and group them into an appropriate number of base tables. In Section 4 we will introduce how to *adaptively* select and cluster the common attributes.
2) Put the star attributes into the base tables.
3) Select the dependent attributes based on the common attributes, database schema and the query workloads and put them into the base tables. In Section 5 we will discuss how to select the expensive but light dependent attributes *adaptively*.

3.3 Advantages of ADAPT Method

Besides high scalability, ADAPT has good performance as well as low space overhead with adequate flexibility and high availability. The main reasons are as follows.

High Scalability: The service provider usually serves for the topic-related tenants, the total number of attributes from multiple tenants is not too large. Then the number of tables does not grow linearly and the scalability is excellent.

High Performance: STSI will degrade the performance as it involves many NULLs and uses more I/Os to retrieve specific tuples. ADAPT reduces the number of I/Os. In addition the indexes in our method are very efficient and thus ADAPT achieves high performance.

Low Space Requirement: In STSI, the attributes from all the tenants are put into the big table without considering *importance*. The big table is very sparse and involves plenty of NULLs. Instead, ADAPT distinguishes the importance of different attributes and only adds the important attributes into the base table. Thus ADAPT do not produce many NULLs and the smaller indexes also save space.

Adequate Flexibility and High Availability: In the real application, the physical schema is not changed often. If there are obvious changes of the attribute configuration or query workloads we will alter the schema. Furthermore, ADAPT does provide high availability and adequate flexibility when altering the schema. First, there are a number of base tables and the scale of which is not large, thus the changes on these base tables are not costly. For the remaining unimportant attributes, it is more flexible because the cost of adding new or deleting old attributes is much smaller compared with the extreme wide tables.

4 COMMON ATTRIBUTE SELECTION AND CLUSTERING

Based on the definition, the common attributes rely on the threshold τ. If we use a too large threshold, there will be a small number of the common attributes and thus we cannot make fully use of the significance of the common attributes. Thus we need to determine an appropriate threshold. The value of τ may vary under different applications and can be determined based on the concrete datasets. After defining τ, we obtain the common attributes. Since the base tables are constructed mainly according to the common attributes. Thus before organizing the common attributes we have some following issues related to the base tables to consider.

- The base tables should not be too wide (large number of attributes) in order to ensure the performance.
- The base tables should be as dense as possible without too many NULL values in order to obtain both the high performance and low space requirement.

- The number of the base tables should not be too many in order to achieve the high scalability in our ADAPT method.

Therefore, we need to use the common attributes to build an *appropriate* number of *dense* base tables with *proper* width first. However, it is difficult to precisely define all the constraints above. Further, tuples can be added or deleted and attributes can be altered in the real application. Therefore, we can design an approximate solution to construct the base tables periodically. To this end, we formulate the problem into an *attribute clustering problem* in which the attributes refer to the common attributes.

To solve the clustering problem, we adopt the well-known graph partition algorithm and apply it to our attribute clustering problem. In the clustering problem it is significant to know how to cluster and how to determine the number of clusters. In Section 4.1 we introduce how to use the graph partition algorithm to cluster the common attributes into different base tables. In Section 4.2, we discuss the method to determine the number of base tables.

4.1 Common Attributes Clustering

Given the common attributes, we can model them as an undirected weighted graph. For ease of presentation and better understanding, we first introduce some definitions.

Definition 4 (Attribute Undirected Weighted Graph): In the graph, each node represents one common attribute. Each pair of nodes is connected by a weighted edge. The weight will be defined later.

In order to represent the attribute node and to compute the weight of each pair of nodes, the attribute node is defined as an M-dimension vector which records the configuration of different tenants for this attribute.

Definition 5 (Attribute Node Vector): The attribute node vector V_x for attribute node x is an M-dimension vector. The dimension M is the number of the tenants who use the source table the attribute x belongs to. The *ith* dimension is set 1 if the *ith* tenant choose to use the attribute; otherwise it is set 0.

Here for the simplicity, we decrease the complexity of the definition for the attribute node vector. Because for one source table different tenants will have different numbers of tuples. Thus in the real application we can set the *ith* bit value of each attribute vector according to the number of tuples of the *ith* tenant.

Now we use an example to illustrate our methods, e.g., V_1 represents the first common attribute from table CUSTOMER as follows:

$$V_1 = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

This attribute vector means that there are 6 tenants using the table CUSTOMER. The first, the third, the fifth and the sixth tenants configure to use the first attribute.

Accordingly, in table CUSTOMER, e.g., after we define τ there are totally 4 common attributes configured by all the 6 tenants, then the common attributes of table CUSTOMER can be represented as the matrix $M_{customer}$ below:

$$M_{customer} = \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

After we define the graph and the nodes we next discuss how to compute the weight of the edge. We use the similarity between each pair of two attribute node vectors and then set it as the weight. The Jaccard coefficient[3] is a classic algorithm to compare the similarity of two objects and is adopted in our method.

Definition 6 (Weight of Edge): The weight of edge $W_{(x,y)}$ between two nodes V_x and V_y is defined as the Jaccard coefficient between the two attribute node vectors V_x and V_y:

$$Jaccard(V_x, V_y) = \frac{|V_x \bigcap V_y|}{|V_x \bigcup V_y|} \quad (1)$$

The Jaccard coefficient value ranges between zero and one. When the two attributes are configured by two different groups of tenants, the value is zero and this means if the two attributes are put together there are no cooccurrence values in the same row. This corresponds to the worst case. However, when two attributes are shared by the same group of tenants and the number of tuples for each tenant equals, there are no NULL values in the same row and this is the most desirable case. Although the latter case hardly appears in the real application, we try to group the common attributes approximate to the extreme desirable design.

After modeling the common attributes into the graph, then we use the k-way clustering solution implemented in CLUTO [4] to solve the attribute clustering problem. In the k-way clustering solution, based on the attribute node vectors, it constructs the nearest-neighbor graph and then uses the min-cut graph partition method to find well-separated clusters.

4.2 Determine the Number of Base Tables

In the clustering solution we need to determine the number of the k in advance. In our ADAPT method, in order to make the base tables achieve excellent performance with low space overhead and promising scalability, we need to determine a practical number of base tables. The best scalability means to construct only one base table with most NULLs and poor performance. No NULLs means to have the largest number(equals to the number of the common attributes)of base tables with poor scalability and low performance. Because the common attributes may usually occur in the real workload, then the number of join operations may increase sharply. Intuitively, the practical choice of k needs to strike a balance between one single cluster and the maximum number of clusters. In ADAPT method, the practical number means to build an appropriate number of base tables with proper table width, few NULLs

in which the performance, space and scalability can all be close to perfect in the database system.

If the number of common attributes is quite small, we can enumerate the number of clusters and determine an appropriate value. In this case, the enumeration cost is not very large. However, when the number of the common attributes is very large, the enumeration method is rather expensive. Next we will briefly discuss several classic methods to determine the practical number of clusters.

In [5], it introduces several classic methods to determine the number. Given N objects, the *Rule of thumb* is usually chosen as $\sqrt{N/2}$ which does not consider the characteristics of the data. The *Information Theoretic Approach* or *Cross-validation* all need the enumeration work. Another representative method to determine the number of cluster is to use the *elbow*. In the *elbow* method, we can plot the variance against the number of clusters. At one point the variance is altered sharply, we call it the *elbow* point and this *elbow* point can be set as the practical number of the clusters. The naive method to find the *elbow* point is to use the enumeration and plot the corresponding data. The cost is rather high in the complicated application. Next we introduce an efficient method to analyze the attribute configuration of different tenants beforehand and finally determine the practical number of clusters.

Honarkhah et al. [23] introduces the kernel matrix analysis method in order to find the practical number of clusters without enumerating each number. In the kernel matrix analysis method, firstly, it uses a kernel function to map the original data into a new feature space which can still maintain the original similarity between different objects. Second, we build a kernel matrix. Finally, we analyze the structure of the kernel matrix, plot the elbow point and determine the number of clusters. Now we can give a general overview of this kernel matrix method.

(1) Attribute Node Vector Mapping. In Section 4.1 we have modeled each attribute as an attribute node vector. Based on this expression, firstly we can use a kernel function to map the original attribute node vector into a new feature space. This transformation can still maintain the similarity of each pair of common attributes. Usually, we choose to use the Gaussian radial basis function as the kernel function which is the same as the choice in [23].

$$k\left(V_x, V_y\right) = exp\left(-\frac{\left\|V_x - V_y\right\|^2}{2\sigma^2}\right) \tag{2}$$

When using the kernel function, we need to define its bandwidth σ. Honarkhah et al. [23] has mentioned that a very large bandwidth will not have the good ability to differentiate the data. Because the data are almost the same. Conversely, the kernel with a very small bandwidth does not understand the relationship between different data, because different data in this case almost has no similarity at all. Lampert et al. [27] provides a heuristic method based on the input data. For the Gaussian kernel it is transformed as in Equation 3:

$$k\left(V_x, V_y\right) = exp\left(-\gamma\ d^2(V_x, V_y)\right) \tag{3}$$

where d is the distance function between the input data, e.g, the Euclidean Distance. One rule of setting σ is computed below:

$$\frac{1}{\gamma} \approx median_{x,y=1,\ldots,N} d(V_x, V_y) \tag{4}$$

(2) Construct the Kernel Matrix. After defining the kernel function, we can transform the original attribute node vectors into the new feature space. Now we can construct a kernel matrix. The kernel matrix K contains all the information necessary to represent the original attribute vectors. The kernel matrix K is defined as follows:

$$K = \begin{pmatrix} k(V_1, V_1) & k(V_1, V_2) & \cdots & k(V_1, V_N) \\ k(V_2, V_1) & k(V_2, V_2) & \cdots & k(V_2, V_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(V_N, V_1) & k(V_N, V_2) & \cdots & k(V_N, V_N) \end{pmatrix} \tag{5}$$

(3) Find the Elbow in the Kernel Matrix. Now we can analyze the structure of the kernel matrix. We use the eigenvalue decomposition method to decompose the kernel matrix based on the equation 6.

$$K = U\Lambda U^T \tag{6}$$

In equation 6, Λ is the diagonal matrix, the *ith* columns of the matrix U are the eigenvectors u_i for the corresponding eigenvalues λ_i. Further, we follow the equation 7 introduced in [23]

$$1_N^T K 1_N = \sum_{i=1}^{N} \lambda_i \{1_N^T u_i\}^2 \tag{7}$$

In equation 7, 1_N is an N×1 dimensional vector whose values are all $1/N$. Then we can produce a plot $\log(\lambda_i\{1_N^T u_i\}^2)$ against i, find the elbow point and set it as the practical number of cluster.

5 Dependent Attribute Selection

In this section we discuss how to identify the dependent attributes and put them into base tables. A straightforward method adds *expensive* attributes into the base tables. However, if large numbers of *expensive* attributes are inserted into the base tables, it will produce lots of `NULLs` and also degrade the performance of other operations. Thus we want to identify the more *expensive* and much *lighter* attributes and add them into the base tables.

There are three challenges to identify dependent attributes. The first one is to formally determine *expensiveness* of an uncommon attribute. The second challenge is how to define *weight* for the uncommon attribute and then the light is easy to understand. The third one is to adaptively select the *expensive* and *light* uncommon attributes, especially when the number of attributes is large and the

workload is complicated. In Section 5.1, we address the first challenge. The other two challenges will be addressed in Section 5.2 and Section 5.3 respectively .

5.1 Expensiveness of Attribute

The expensiveness of an uncommon attribute refers to the query processing. Thus we need to evaluate the *expensiveness* of different operations. The traditional database system usually consists of 4 basic operations, *selection*, *insertion*, *update* and *deletion*. Now we will analyze each operation cost in detail.

5.1.1 Selection Operation

Selection is the most common operation in daily workloads. We can divide *selection* operation into 3 major categories based on the selected attributes. The first is the query without *uncommon attributes* in both `SELECT` and `WHERE` clauses. The second is the query involving uncommon attributes in `WHERE` clause. The third is the query involving uncommon attributes only in `SELECT` clause.

(1) Query without Uncommon Attributes: For example, a tenant just wants to know some general information about its source tables. It will submit a query:

```
SELECT C_NAME FROM CUSTOMER
WHERE C_ADDRESS = 'New York';
```

where `C_NAME` and `C_ADDRESS` are both *common attributes* coming from the base table. This query does not involve uncommon attributes. The number of tuples in the base table is much smaller than that in STSI method. Even if the common attributes may evolve different base tables it can be still quite fast. First, the number of the base tables is not many. Further, the number of columns in the base table is much smaller. Thus we can use fewer I/Os to locate and return the corresponding tuples. Obviously, for this kind of selection, ADAPT method is much faster than STSI.

(2) Query with Uncommon Attributes in the WHERE Clause: For instance, one tenant wants to know the result which satisfies its specific condition, and it may submit a query:

```
SELECT C_NAME FROM C_CUSTOMER
WHERE C_NATION = 'France';
```

where `C_NATION` is an uncommon attribute. In this type of queries, the `WHERE` clause contains selection operations on uncommon attributes. In this case, ADAPT may involve more join operations than STSI. However, ADAPT still outperforms STSI method for the following reasons: First, the uncommon attributes will be shared by several tenants, thus the number of involved tuples of the uncommon attribute is small. And in this kind of workload the queries have already specified these uncommon attributes with selection in the `WHERE` clause. The intermediate results can be further reduced. Second, we usually build efficient indexes on the frequently queried attributes. Thus even if the number of join operations is larger, the indexes can still ensure the fast query performance.

(3) Query with Uncommon Attributes only in the SELECT Clause : A tenant want to query some information in an organized way it will submit a query:

```
SELECT C_CUSTKEY,C_NAME,C_SALARY
FROM CUSTOMER
ORDER BY C_CUSTKEY;
```

where `C_SALARY` is an uncommon attribute. Different from the former two cases, this kind of query is costly. First, there are no filter conditions on these uncommon attributes in the `WHERE` clause, then the indexes cannot be efficiently used. If the average number of the attributes for each source table is low, the join operation cost is not large. However, when the number is higher and many of the uncommon attributes occur in the workload only in the `SELECT` clause, the performance will be degraded. In the typical OLAP application, the tenant may submit many aggregate function to the query. Thus these uncommon attributes which are often queried and only occur in the `SELECT` clause produce large costs. These operations are very *expensive* and those uncommon attributes occur only in the `SELECT` clause in this kind of operation meet the requirements *expensive* of the dependent attributes.

5.1.2 Insertion Operation

In ADAPT method, when a tenant inserts a tuple to a source table, this insertion can be divided into several smaller insertions to corresponding physical tables: both the base tables and the supplementary tables.

During the process of insertion, our method can be much faster. Firstly, ADAPT can directly insert the tuples into the base tables and other supplementary tables. Unlike STSI, ADAPT does not need to insert many unnecessary `NULL`s and this will save some time.

Meanwhile, the size of the indexes in the supplementary tables is pretty small compared with those in STSI. Thus the maintenance cost on indexes in ADAPT is much smaller and it will further save much time.

5.1.3 Update Operation

In the common OLTP application, the update operation is one typical operation. In the real workload, the number of columns to be updated is one or not many. The attributes occurring in the `WHERE` clause are often the star attributes which are in the base tables. Thus the number of join operations is quite limited, and the performance for the update operation is quite excellent and much faster than that in STSI.

5.1.4 Deletion Operation

In ADAPT, when a tenant submits a delete operation, it will also be divided into several deletions and this is similar to the insert operation. Unlike the insertion performance priority over STSI, the deletion in ADAPT may not have excellent performance. This is because we can not directly delete the tuples, and we have to locate the tuples which satisfies the “delete” requirements first. Thus we have to do

more select operations before the deletion. It is also the fact that we can not ensure all the attributes will be indexed, thus the selection operation for the tuples without indexes is much more costly.

In the well-designed applications, the number of delete operations is rare. So the delete operations may be costly but not *expensive* in reality.

To sum up, only the selection operation that query the uncommon attributes but does not specify the condition in the `WHERE` clause are *expensive*. Thus we can define the expensive attributes.

Definition 7 (Expensive Attributes): Expensive Attributes are those uncommon attributes that appear only in the `SELECT` clause and have no specific conditions in the `WHERE` clause.

Thus we can easily identify the *expensive* attributes from query workloads.

Further, if we want to know which uncommon attribute is more expensive. Then we can define the cost for each expensive attribute.

Definition 8 (Cost of Expensive Attribute): The cost of expensive attribute is the occurrence times of the expensive attribute in the costly operation.

Now we can conclude that if the cost of one expensive attribute is larger it is more expensive.

5.2 Weight of Attribute

For ease of understanding which expensive attribute is lighter, we firstly introduce a concept to define *weight* and then *light* is easy to understand.

Definition 9 (Weight of Attribute): The weight w of an expensive attribute is the number of increasing `NULLs` if it is inserted into the corresponding base table.

Suppose there are $Null_{num}$ `NULLs` values in the $base_i$ table. If one expensive uncommon attribute is inserted into the $base_i$ table, the number of the `NULLs` values is increased to $Null'_{num}$, then the weight for this uncommon attribute is $w = Null'_{num} - Null_{num}$.

If the weight of one expensive uncommon attribute is quite small compared with others, we can say this expensive uncommon attribute is very light.

5.3 Dependent Attribute Selection

We have made it clear that which uncommon attribute is more expensive and which one is lighter. Next we will discuss how to choose the *dependent attributes* for a single source table and multiple source tables in Section 5.3.1 and Section 5.3.2 respectively.

5.3.1 Dependent Attribute Selection for One Source Table

We study that if the service provider has a space budget $\mathcal{B}$, we want to use the budget to find the dependent attributes and achieve the best performance. Usually, we can use the number of `NULLs` to denote the $\mathcal{B}$. For example, recall expensive attributes maintained in Table 3. If the number of `NULLs` $\mathcal{B}$ permitted by the service provider is set 2, the attribute `C_SALARY` is a dependent attribute.

Consider a source table, a budget $\mathcal{B}$ and a query workload. Suppose there are m expensive attributes $e_1, \ldots, e_m$ in the workload. Let v_i denote e_i's cost and w_i denote e_i's weight. We want to maximize

$$\sum_{i=1}^{m} v_i \cdot x_i$$

subject to

$$\sum_{i=1}^{m} w_i \cdot x_i \leq \mathcal{B}, x_i \in \{0, 1\}.$$

Suppose $X = x_1, \ldots, x_m$ is the best solution of the above problem. If $x_i = 1$, attribute e_i is a dependent attribute. Now we formally define this concept.

Definition 10 (Dependent Attributes for One Source Table): An attribute e_i is a dependent attribute if $x_i = 1$ in the best solution $X = x_1, \ldots, x_m$.

Obviously the problem to find dependent attributes is easily proved to be an NP-complete by an induction from the well-known 0-1 Knapsack problem. To address this problem, we use the well-known dynamic programming algorithm to reduce the maximum of costly operations under $\mathcal{B}$. During the process, we will record the selection of different expensive attributes. Based on these information, we can easily get dependent attributes. Due to space constraints, we omit the details of the proof and the algorithm.

5.3.2 Dependent Attribute Selection for Multiple Source Tables

The tenant usually has multiple source tables. We extend our method to support multiple tables. In the multi-tenant database, the utilization of different source tables may vary from each other. For instance, the primary key of a table $\mathcal{A}$ is also the foreign key referenced by some other tables. It means that in the real workload, the attributes from table $\mathcal{A}$ will have a higher possibility to be queried especially in join operations. Thus when evaluating the importance of different expensive attributes, we should consider the importance of different source tables first. Then we can further similarly follow Section 5.3.1 to choose the dependent attributes.

Firstly, we need to determine the importance of different source tables. We can model different source tables as a graph, where each source table is a node and the directed edge is built relying on the foreign key of the source tables. Next we formally define the source table graph.

Definition 11 (Source Table Graph): The source tables can be modelled as a graph $G = (V, E)$ where V is a set of nodes which are the source tables $\{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n\}$ and E is a set of edges whose directions are based on the foreign key between the source tables.

When evaluating the importance of different source tables in the graph it is easy to combine the case with the web

pages in the internet. In the latter application, the PageRank and HITS algorithms are two classic algorithms. The HITS algorithm is one topic specific algorithm on evaluating the web pages, while the PageRank is an effective but general algorithm. Thus we can simply choose the PageRank algorithm to evaluate the importance of different source tables.

Based on the graph and PageRank we compute the PageRank weights $PR_{\mathcal{T}_i}$ for the source table $\mathcal{T}_i$. In this step, the weight $PR_{\mathcal{T}_i}$ is treated as the schema level. Now these weights do not consider the number of tuples for each source table $\mathcal{T}_i$, denoted by $Tnum_{\mathcal{T}_i}$. In our method, if the number of tuples in the source table is large, the cost of the join operation is large. If we avoid this operation, we can achieve better performance. Thus finally we set $Tnum_{\mathcal{T}_i} \cdot PR_{\mathcal{T}_i}$ as the importance for each source table $\mathcal{T}_i$.

Consider multiple source tables, a budget $\mathcal{B}$ and a query workload. Suppose there are m expensive attributes $e_1, \ldots, e_m$ in the workload. Let v_i denote e_i's cost and w_i denote e_i's weight, s_i denote the importance of the source table that e_i belongs to and it can be computed as $Tnum \cdot PR$. We want to maximize

$$\sum_{i=1}^{m} v_i \cdot s_i \cdot x_i$$

subject to

$$\sum_{i=1}^{m} w_i \cdot x_i \leq \mathcal{B}, x_i \in \{0, 1\}.$$

Suppose $X = x_1, \ldots, x_m$ is the best solution of the above problem. If $x_i = 1$, attribute e_i is a dependent attribute attribute.

Definition 12: (Dependent Attributes for Multiple Source Tables) An attribute e_i is a dependent attribute if $x_i = 1$ in the best solution $X = x_1, \ldots, x_m$.

Similarly, we use dynamic programming algorithm to address the problem for multiple tables. As the workload changes, the dependent attributes vary. For different tenants, they may also change. With our proposed methods we can dynamically select the dependent attributes and design adaptive schema for different tenants under different workloads. Note that we adopt a schema-level method, and our method can be efficient to support dynamical changes.

6 Experimental Study

In this section, we evaluate our proposed techniques and compare with state-of-the-art methods.

6.1 Benchmark

It is very important to use standard benchmarks to compare the performance of different approaches. TPC-H and TPC-C[6] are two well-known benchmarks. However, they are not well suited to evaluating the multi-tenant database since they can not be configured by different tenants. Therefore, in our experiments we design our MTPC-H benchmark for the multi-tenant database which is fully based on the traditional TPC-H benchmark with minor revisions.

In order to ensure the representativeness of the MTPC-H benchmark and to obviously compare different database schema approaches, we select five core tables depicted in Figure 1. Many transactions happen on these tables. For example, many join operations involve these tables. For each table, we extend the total number of attributes to 100.

Since indexes can enhance the query performance, we build indexes on the attributes involved in the query workload. The index is the compound index of the TID, the key of the base table and that attribute.

The MTPC-H Benchmark we designed for the multi-tenant database consists of 3 components.

Schema Generator: We use the schema generator to produce the schema for each tenant. This generator contains such parameters: the number of tenants *TNum*, the average number of total attributes for each private table is μ, the standard deviation σ. To generate private schema for each tenant, besides the TID attribute, the generator randomly selects μ attributes for each source table for each source table to form the final private database schema. The total number of attributes for each private table satisfies the normal distribution $\mathcal{N}(\mu, \sigma^2)$.

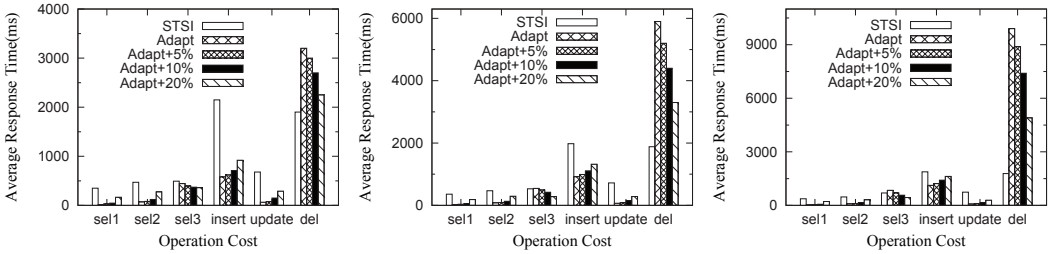
Workload Generator: In order to observe the performance of our proposed method under different workloads, we use the workload generator to produce the workload with configured ratios demanded.

Tenanter: The last module in our benchmark is Tenanter. It is conceptually equivalent to the Driver in the TPC-H benchmark. Each Tenanter submits queries to the multi-tenant database system under test, measures and reports the execution time of those queries. We run Tenanter and the multi-tenant database system in a "client/server" configuration. We place the Tenanter and the database system in different machines interconnected by a network. The Tenanter is written in C++ and each Tenanter acts as a tenant interacting with the database system. Each Tenanter runs its queries in its own thread. Thus multi-threads in our simulation are designed to simulate concurrent accesses to the database system from multiple tenants.

6.2 Experiment Settings

We generate private database schemas for tenants by running schema generator. For each tenant, we generate 3 sets of schemas by setting μ to 10, 15, 20 respectively and fixing σ =2. We finally generate 3 groups of schemas for 200, 500 and 1,000 tenants. For each tenant, we generate 2000 tuples for table `CUSTOMER`, 2000 tuples for table `PARTSUPP`, 1500 tuples for table `PART`, 3000 tuples for table `LINEITEM` and 2000 tuples for table `ORDERS`. The number of tuples for each table is almost in proportion to the standard TPC-H benchmark. We use each Tenanter to run the queries acting as each tenant in its own thread.

The Tenanters and the database system are run on two Windows XP machines with the same configurations. Each machine is equipped with a 3.2GHz Pentium (R) dual-core E5800 cpu and a 4GB of memory. We conduct the experiments using the MySQL database.



(a) $\mu = 10$ (b) $\mu = 15$ (c) $\mu = 20$

Fig. 3. Operation Cost (CUSTOMER Table in MTPC-H)

To compare different methods in a fair way, for each workload we repeat 5 times and obtain the average time. Further, we restart both the database server machine and the client machine to flush the buffers for each experiment.

6.3 Analyzing Different Operations

We evaluate our analysis of different operations in Section 5. We use the MTPC-H benchmark and choose table `CUSTOMER` as the example. Each `CUSTOMER` table has 2000 tuples on average and is shared by 1000 tenants. There are mainly 6 operations. They are queries without uncommon attributes, queries with uncommon attributes in the `WHERE` clause, queries with uncommon attributes only in the `SELECT` clause, insertion, updates and deletion. For simplicity, we call the selection operations *sel1*, *sel2* and *sel3* respectively for short. We compare the operation costs among STSI, our ADAPT method without putting the dependent attributes into the base table(ADAPT) and our ADAPT method using different $\mathcal{B}$, where the $\mathcal{B}$ is set 5 percent(Adapt+5%), 10 percent(Adapt+10%) and 20 percent(Adapt+20%) of the total number of `NULLs` if all the expensive attributes are inserted into the base tables. We conduct different operations on table `CUSTOMER`. Each of the operation cost obtained is the average of 1,000 operations. Figure 3 shows the results.

We see that compared with STSI, *sel1* and *sel2* have much better performance under all of the cases. For *sel3*, when $\mu = 10$, ADAPT without putting the dependent attributes into the base tables can be comparable to STSI since the costly join operation is small. When $\mu = 20$ it is slower than that. This is because the costly join can be more. If we use some $\mathcal{B}$ to put the dependent attributes into the base table, *sel3* improves obviously especially for higher μ. Further, when $\mu = 10$ the $\mathcal{B}$ does not need to be high because the number of costly join is quite small and thus a small $\mathcal{B}$ can ensure the performance. For the insertion, STSI is the slowest because it will need to insert many `NULLs` and maintaining the big index is costly in STSI. Similarly, the update operations in our ADAPT methods with different $\mathcal{B}$ all have obvious priority over STSI. Because the number of attributes to be updated is usually not many and the update operation always specifies the filter condition in the `WHERE` clause. While the delete operation of STSI is the best since other methods will have more deletions and locating the tuples to be deleted will cost some time.

To sum up, except for the deletion, the *sel1*, *sel2* , insertion and update operation in ADAPT with different $\mathcal{B}$ are all more efficient than STSI, even if when μ is larger *sel3* can still achieve obvious priority over STSI with a small $\mathcal{B}$. In the typical OLAP applications, we can add those dependent attributes into the base tables and further improve the performance. Therefore, we can conclude that our adaptive method can well support both the OLTP extensive and OLAP extensive applications.

6.4 Performance under One Source Table

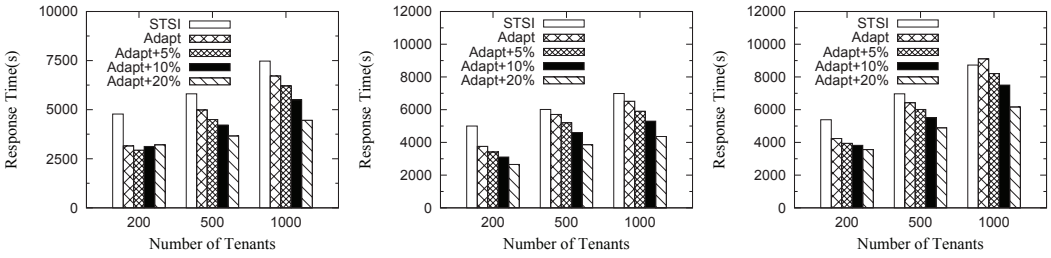
From the experimental results in 6.3, we easily find *sel3* is very costly. Now we set sel3 as the dominate operation and analyze the performance of the whole workload with different operations. In such workload if we can have better performance, our adaptive schema can have excellent performance in other workloads. In this set of experiments, the test workload contains 10,000 operations, 15 percent are *sel1*, 15 percent are *sel2*, 50 percent are *sel3*, 8 percent are insertion, 10 percent are update operation and 2 percent are deletion. Figure 4 illustrates the performance of `CUSTOMER` table shared by different tenants and we also vary the number μ to evaluate the performance.

In our workload, when μ is 20 and `CUSTOMER` table is shared by 1,000 tenants, for *sel3* operation our basic method is slower than STSI and degrades the performance obviously because we do not put the dependent attributes into the base tables. However, when the $\mathcal{B}$ is 5 percent, after we put some dependent attributes into the base table, the improvement for *sel3* is obvious and helps to improve the overall performance. When the number of tenants is 200 and $\mu = 10$, *sel3* is not costly and a small budget can improve the performance which can also save space.

6.5 Performance under Multiple Source Tables

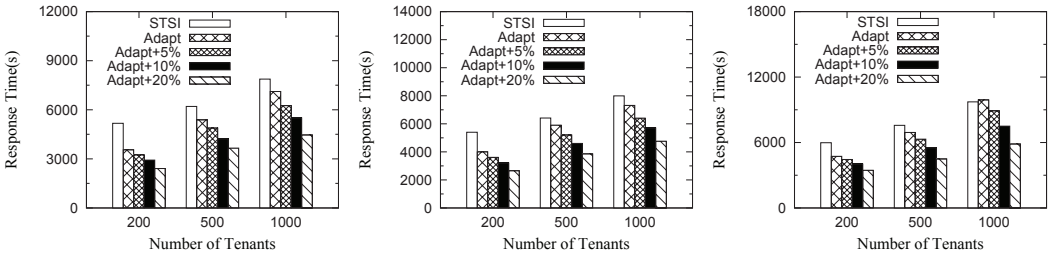
We discuss the performance under multiple tables. We use PageRank algorithm to compute the importance of the five source tables. In the PageRank algorithm we set the damping factor as the classic value 0.85, the initiative PageRank value for each table is 0.2 and the convergence is 0.0001. With these parameters, the PageRank values from the schema level can be computed. Then we can multiply PageRank values by the average number of tuples as the final importance of each source table.

Then we also use the same workload ratio in Section 6.4 and vary μ to evaluate the performance under multiple tables. Of course, the queries in this workload cross multiple source tables and the total number of queries is also 10,000. Figure 5 illustrates the performance under multiple source tables. We see that the improvement of the performance



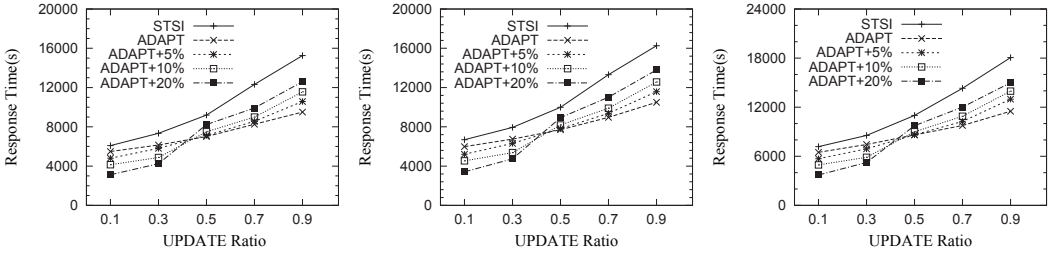
(a) $\mu = 10$ (b) $\mu = 15$ (c) $\mu = 20$

Fig. 4. Performance of One Source Table (CUSTOMER Table in MTPC-H)



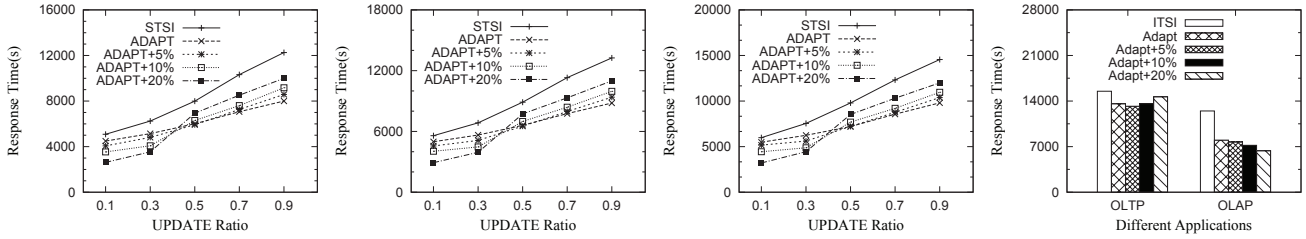
(a) $\mu = 10$ (b) $\mu = 15$ (c) $\mu = 20$

Fig. 5. Performance of Multiple Source Tables(MTPC-H)



(a) $\mu = 10$ (b) $\mu = 15$ (c) $\mu = 20$

Fig. 6. Scalability on Multiple Source Tables(MTPC-C)



(a) $\mu = 10$ (b) $\mu = 15$ (c) $\mu = 20$ (d) 4,000 Tenants, $\mu = 15$

Fig. 7. Scalability on Multiple Source Tables (MMobile)

using ADAPT is quite obvious. The main reason is that the attributes from an important table will have higher probability to occur in the workload, if dependent attributes from this table can be put into the base tables the performance can be enhanced obviously and thus the whole workload has better performance.

6.6 Scalability

Since the TPC-H is a typical ad-hoc decision support benchmark, in order to test the scalability of our adaptive schema approach. We design the MTPC-C benchmark which is almost based on the standard TPC-C benchmark. In addition, we use a schema from a mobile operator and design a MMobile benchmark.

In MTPC-C benchmark, we choose 5 tables `WAREHOUSE`, `DISTRICT`, `CUSTOMER`, `ITEM` and `BLOCK`. The number of tuples for each table are 1500, 1500, 2000, 2000 and 3000 respectively. In addition, because the TPC-C is a typical OLTP benchmark, we vary the percentages of the "UPDATE" operations including insertion, updates and deletions. In addition, all the tables are shared by 1,000 tenants and we also vary the value of μ. Figures 6 shows the results of the performance under this benchmark. Obviously, when the workload is lookup intensive we need to put more dependent attributes into the base tables. However, when the workload is "UPDATE" intensive, we can just adopt a small space budget or even no budget to hold the dependent attributes.

In MMobile benchmark, we use 4 tables and they are `EDW_GSM_CDR_T` with 3000 tuples, `EDW_SMS_CDR_T` with 3000 tuples, `EDW_MDS_CDR_T` with 3000 tuples and `EDW_MCI_CDR_T` with 1500 tuples. Other settings are the

same as MTPC-C benchmark. Figures 7(a),7(b) and 7(c) shows the results.

In the former experiments, we find the STSI method can not be adopted due to the poor performance and high space overhead. While ITSI has poor scalability, eventually, in order to fully test the scalability of our ADAPT method, we increase the number of tenants to 4,000 and compare our ADAPT method with ITSI method. Due to the space limitation, we only use the MMobile benchmark. In this experimental setting, the total number of tables reach 16,000. We design two typical workloads with different UPDATE operations. The first one is called OLTP with 60 percent reads and 40 percent writes. The second one is called OLAP with 90 percent reads and 10 percent writes. The total numbers of the operations for both of the two workloads are all 10,000. Figure 7(d) shows the results. Obviously, when the number of tables is large the performance of ITSI drops dramatically. Predictably, if the application for each tenant becomes more complicated, more tables will be involved. Further in the real application, the number of tenants can be much more, thus the scalability for ITSI method will obviously become the bottleneck.

To sum up, we see that our ADAPT method scales well as the workload changes and can adaptively achieve excellent performance in both OLAP and OLTP applications.

7 Related Work

As Software as a Service(SaaS) has been adopted by a large number of famous software companies. The research on multi-tenancy has attracted extensive attention.There are a number of overviews on the multi-tenant problem in [7], [33], [26], [21], [16], [15], [22], [29].

One significant problem on the multi-tenant database is to design high-quality database schema. Many design methods have been proposed in [35], [34], [31], [24], [12], [13], [25]. In [24], it needs to maintain a large number of indexes for all the tenants and this will increase the chances to contend the resource in memory. Many index pages will be loaded into memory using plenty of random I/Os which is the major disadvantage of disk. And the basic schema needs to be fixed. If one tenant adds a new attribute the cost is large because it will affect the data storage of other tenants. Aulbach et al. [12] focuses on providing extensibility for multi-tenant databases. They introduce chunk tables to store data in extension attributes. However, its table is too big and plenty of costly self-join operations will happen on it and is proved to be slower than the conventional tables in [12]. Similarly, Aulbach et al. [12] also mentioned the extension table layout which is well suited to the customization of the multi-tenant application. Similarly, Schiller et al. [34] also proposes a schema extension method. In these two design, the number of tables still grows linearly with the number of tenants. Aulbach et al. [13] uses the object-oriented thoughts to design the schema which can be extended and evolved but its main purpose is to serve for the main-memory database.

In addition, since the STSI method has excellent scalability but the table is very sparse, then the method of managing sparse data can be useful and referenced in the multi-tenant database design. The studies [14], [17] have introduced methods to store sparse data in relational DBMS. In [32] it stores user contributed tags in a sparse table. Abadi et al. [10] introduces effective techniques for compressing `NULLs` in a column-oriented database system. Agrawal et al. [11] proposed the three column tuple design, in this method there are many self-join operations, the indexes can not be efficiently used and the isolation is not good either.

Besides schema design, the multi-tenant system involves some other important issues. Curino et al. [18] tries to measure the hardware requirement of the database query workload, predict the resource utilization and does effective consolidation. Lang et al. [28] focuses on how to deploy resources for different tenants with various performance requirements. Muhe et al. [30] consolidates multiple tenants using multi-core servers. In [36] it introduces data privacy problem and data migration is discussed in [19], [20].

In our Adapt method, the base tables and supplementary tables can be directly implemented in the traditional relational database and achieve excellent performance with desirable flexibility and scalability. However, the supplementary tables are similar to the column stores. If we use the real column store to manage the supplementary tables we can further compress the data, save space, improve the query performance and best supports the configuration characteristics of multi-tenant applications. Meanwhile, considering the differences of the base tables and supplementary tables it is appropriate for us to adopt the hybrid database system which both supports the row store and the column store. Oracle Exadata [8] adopts the hybrid columnar compression techniques. Exadata increases the compress ratio so as to both enhance the performance and reduce the space requirement. Aster Data nCluster [9] is a platform that incorporates the column and the row store with massively MapReduce parallel processing techniques.

This paper is a major-value added version of previous work [31]. The significant additions in this extended manuscript can be summarized as follows: First, we propose to use the graph partition and kernel matrix analysis to build an appropriate number of dense base tables. The proposed method makes fully use of the common attributes and improves the overall performance of the system. Second, we add the analysis of the update operation in order to make our new design support both the OLTP and OLAP queries well. Third, we extend our method to support the dependent attributes selection across multiple source tables using the well-known PageRank algorithm. Finally, we construct more experiments to compare with existing methods and test the scalability of our method.

8 Conclusion

In this paper, we have studied the problem of adaptive multi-tenant database schema design. We identified the important attributes, including common attributes, star attributes and dependent attributes. We built several dense base tables using the important attributes. For each of other

attributes, we built supplementary tables. We discussed how to use the kernel matrix to determine the number of base tables, apply the graph partition to building them and evaluate the *importance* of attributes with the well-known PageRank algorithm. We proposed a cost-based model to adaptively generate the schema. Experimental results on both real and synthetic datasets show that our method achieves high performance and good scalability with low space requirement and outperforms state-of-the-art methods.

REFERENCES

[1] http://www.tpc.org/tpch/.
[2] http://dev.mysql.com/doc/refman/5.0/en/features.html.
[3] http://en.wikipedia.org/wiki/Jaccard_coefficient.
[4] http://glaros.dtc.umn.edu/gkhome/views/cluto.
[5] http://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set.
[6] http://www.tpc.org/tpcc/.
[7] http://en.wikipedia.org/wiki/Multitenancy.
[8] http://www.oracle.com/technetwork/middleware/bi-foundation/ehcc-twp-131254.pdf.
[9] http://www.asterdata.com/resources/assets/wp_R_in_nCluster.pdf.
[10] D. J. Abadi. Column stores for wide and sparse data. In *CIDR*, pages 292–297, 2007.
[11] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *VLDB*, pages 149–158, 2001.
[12] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD Conference*, pages 1195–1206, 2008.
[13] S. Aulbach, M. Seibold, D. Jacobs, and A. Kemper. Extensibility and data sharing in evolving multi-tenant databases. In *ICDE*, pages 99–110, 2011.
[14] 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*, page 58, 2006.
[15] C.-P. Bezemer and A. Zaidman. Multi-tenant saas applications: maintenance dream or nightmare? In *EVOL/IWPSE*, pages 88–92, 2010.
[16] H. Cai, B. Reinwald, N. Wang, and C. Guo. Saas multi-tenancy: Framework, technology, and case study. *IJCAC*, 1(1):62–77, 2011.
[17] E. Chu, J. L. Beckmann, and J. F. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *SIGMOD Conference*, pages 821–832, 2007.
[18] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD Conference*, pages 313–324, 2011.
[19] S. Das, S. Nishimura, D. Agrawal, and A. E. Abbadi. Live database migration for elasticity in a multitenant database for cloud platforms. In *Technical Report 2010-09,CS,UCSB*, 2010.
[20] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD Conference*, pages 301–312, 2011.
[21] F. Färber, C. Mathis, D. D. Culp, W. Kleis, and J. Schaffner. An in-memory database system for multi-tenant applications. In *BTW*, pages 650–666, 2011.
[22] K. Haller. Web services from a service provider perspective: tenant management services for multitenant information systems. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–4, 2011.
[23] M. Honarkhah. Stochastic simulation of patterns using distance-based pattern modeling. In *Doctoral Dissertation,Stanford*, 2011.
[24] M. Hui, D. Jiang, G. Li, and Y. Zhou. Supporting database applications as a service. In *ICDE*, pages 832–843, 2009.
[25] D. Jacobs and S. Aulbach. Ruminations on multi-tenant databases. In *BTW*, pages 514–521, 2007.
[26] M. Kapuruge, A. W. Colman, and J. Han. Defining customizable business processes without compromising the maintainability in multi-tenant saas applications. In *IEEE CLOUD*, pages 748–749, 2011.
[27] C. H. Lampert. Kernel methods in computer vision. *Foundations and Trends in Computer Graphics and Vision*, 4(3):193–285, 2009.
[28] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance slos. In *ICDE*, pages 702–713, 2012.
[29] W. Lee and M. Choi. A multi-tenant web application framework for saas. In *IEEE CLOUD*, pages 970–971, 2012.
[30] H. Mühe, A. Kemper, and T. Neumann. The mainframe strikes back: elastic multi-tenancy using main memory database systems on a many-core server. In *EDBT*, pages 578–581, 2012.
[31] J. Ni, G. Li, J. Zhang, L. Li, and J. Feng. Adapt: adaptive database schema design for multi-tenant applications. In *CIKM*, pages 2199–2203, 2012.
[32] B. C. Ooi, B. Yu, and G. Li. One table stores all: Enabling painless free-and-easy data publishing and sharing. In *CIDR*, pages 142–153, 2007.
[33] J. Schaffner, B. Eckart, C. Schwarz, J. Brunnert, D. Jacobs, A. Zeier, and H. Plattner. Simulating multi-tenant olap database clusters. In *BTW*, pages 410–429, 2011.
[34] O. Schiller, B. Schiller, A. Brodt, and B. Mitschang. Native support of multi-tenancy in rdbms for software as a service. In *EDBT*, pages 117–128, 2011.
[35] C. D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *SIGMOD Conference*, pages 889–896, 2009.
[36] K. Zhang, Q. Li, and Y. Shi. Data privacy preservation during schema evolution for multi-tenancy applications in cloud computing. In *WISM (1)*, pages 376–383, 2011.

Jiacai Ni is currently a PhD candidate in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests mainly include multi-tenant data management, schema mapping, and flash-based database.

Guoliang Li received his PhD degree in Computer Science from Tsinghua University, Beijing, China in 2009. He is currently working as an associate professor in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests mainly include data cleaning and integration, spatial databases, and crowdsourcing.

Lijun Wang received his PhD degree in Computer Science from Tsinghua University in 2008. Now he is an assistant professor in the Institute for Network Science and Cyberspace, Tsinghua University. His research interests include Internet routing protocol, information system and health informatics.

Jianhua Feng received his B.S., M.S. and PhD degrees in Computer Science from Tsinghua University. He is currently working as a professor of Department Computer Science in Tsinghua University. His main research interests include native XML database, data mining, and keyword search over structure & semi-structure data.

Jun Zhang currently manages and architects the user/relationships team in SINA Weibo.com platform. He previously worked in the Emerging Technology Institute at IBM China Development Lab. He get Master Degree from Department of Computer Science at Tsinghua University. His working experience focus area is architecting Large Scale Websites, Cloud Computing and Web2.0.

Lei Li is a Chief Integration and Solutions Architect for IBM Information Management China Development Lab. He drives outside-in development leadership on emerging technologies in the information management area and cloud area. He has over sixteen years information management area expertise, including information architecture design, database and Cloud.