Supporting Database Applications as a Service

Mei Hui #1, Dawei Jiang #1, Guoliang Li *2, Yuan Zhou #1,

#School of Computing
National University of Singapore, Singapore
1{huimei, jiangdw, yuanzhou}@comp.nus.edu.sg

*Department of Computer Science and Technology
Tsinghua University, China
2liguoliang@tsinghua.edu.cn

***Abstract*— Multi-tenant data management is a form of Software as a Service (SaaS), whereby a third party service provider hosts databases as a service and provides its customers with seamless mechanisms to create, store and access their databases at the host site. One of the main problems in such a system, as we shall discuss in this paper, is scalability, namely the ability to serve an increasing number of tenants without too much query performance degradation. A promising way to handle the scalability issue is to consolidate tuples from different tenants into the same shared tables. However, this approach introduces two problems: 1) The shared tables are too sparse. 2) Indexing on shared tables is not effective.**

To resolve the problems, we propose a multi-tenant database system called M-Store, which provides storage and indexing services for multi-tenants. To improve the scalability of the system, we develop two techniques in M-Store: Bitmap Interpreted Tuple (BIT) and Multi-Separated Index (MSI). BIT is efficient in that it does not store NULLs from unused attributes in the shared tables and MSI provides flexibility since it only indexes each tenant's own data on frequently accessed attributes. We extended MySQL based on our proposed design and conducted extensive experiments. The experimental results show that our proposed approach is a promising multi-tenancy storage and indexing scheme which can be easily integrated into existing DBMS.

I. Introduction

Interest in Multi-tenant Database Systems has been increasing in recent years[15], [16], [19], [6]. The multi-tenant database system adopts the *Software as a Service* (SaaS) model, whereby a service provider hosts a data center and a configurable base schema designed for a specific business application, e.g., Customer Relationship Management (CRM) and delivers data management services to a number of businesses. Each business, called a tenant, subscribes to the service by configuring the base schema and loading data to the data center and interacts with the service through a standard method, e.g., Web Service. The ownership of database applications and the maintenance costs are thus transferred from the tenant to the service provider. Figure 1 shows the high level overview of outsourcing databases as a service. This model sharply contrasts with the traditional *on-premise* model whereby a tenant buys a data center and applications and operates them itself. Applications of multi-tenant database include Customer Relationship Management(CRM), Human Capital Management(HCM), Supplier Relationship Management(SRM) and Business Intelligence (BI).

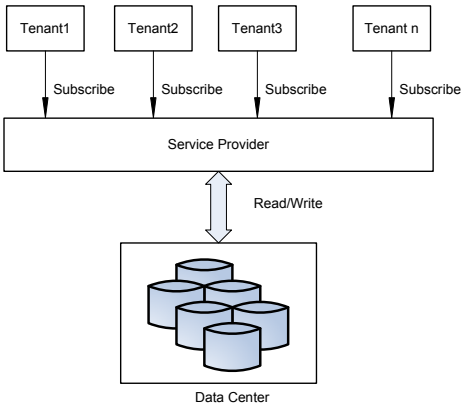


Fig. 1. High-level overview of generalized "outsourced databases as services"

The value of multi-tenancy is that it can help a service provider catch "*long tail*" markets [1]. By consolidating applications and their associated data to a centrally-hosted data center, the service provider amortizes the cost of hardware, software and professional services to an amount of tenants it serves and therefore significantly reduces per-tenant service subscription fee by use of the economy of scale. This per-tenant subscription fee reduction brings the service provider entirely new potential customers in long tail markets that are typically not targeted by traditional and possibly more expensive on-premise solutions. As revealed in [1], [5], access to long tail customers will open up a huge amount of revenue. In terms of IDC's estimation, the market of SaaS will reach $14.5 billion in 2011 [14].

In addition to the great impact that it can have on the software industry, providing database as a service also opens up several research problems to the database community, including security, contention for shared resources, and extensibility. These problems are well understood and have been discussed in recent works [16], [19].

In this paper, we argue that the *scalability* issue, which refers to the ability to serve an increasing number of tenants without too much query performance degradation, should deserve more concern in the building of a multi-tenant database system. The reason is simple. The core value of multi-tenancy

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.82

is to catch long tail. This is achieved by consolidating data from tenants to the hosted database to reduce the per-tenant service cost. So, the service provider must ensure that the database system is built to scale up well so that the per-tenant subscription fee may continue to fall when more and more tenants are taken on board. Unfortunately, recent practice shows that consolidating too much data from different tenants will definitely degrade query performance [9]. If performance degradation is not tolerated, the tenant may not be willing to subscribe to the service. Therefore, the problem is to develop effective and efficient architecture and techniques to maximize scalability while guaranteeing that performance degradation is within tolerable bounds.

Basically, there are three approaches to building a multi-tenant database system. The first approach is Independent Databases and Independent Database Instances (IDII). In IDII, the service provider runs independent database instances, e.g., MySQL or DB2 database processes to serve different tenants. The tenant stores and queries data in its dedicated database. Although this approach offers good data isolation and security, scalability is rather poor since running independent database instances wastes memory and CPU cycles. As an example, the initialization of a new MySQL instance consumes 30M memory or so. Furthermore, maintenance cost is huge. Managing different database instances requires the service provider to configure parameters such as TCP/IP port and disk quote for each database instance.

The second approach to building a multi-tenant database is Independent Tables and Shared Database Instances (ITSI). In ITSI, only one database instance is running and the instance is shared among all tenants. Each tenant stores tuples in its private tables whose schema is configured from the base schema. All the private tables are finally stored in the shared database. ITSI removes the huge maintenance cost incurred by IDII. However, the number of private tables grows linearly with the number of tenants. Therefore, its scalability is limited by the number of tables that the database system can handle, which is itself dependent on the available memory. For example, MySQL 5.1.26 allocates 9K memory for metadata of each table. Thus, 100,000 tables occupy 900M memory. Furthermore, memory buffers are allocated in a per-table manner, and therefore buffer space contention often occurs among the tables. A recent work reports significant performance degradation on a blade server when the number of tables rises beyond 50,000 [9].

The third approach is Shared Tables and Shared Database Instances (STSI). Using STSI, tenants not only share database instances but also tables. The tenants store their tuples to the shared tables by appending each tuple with a `TenentID`, that indicates which tenant the tuple belongs to, and setting unused attributes to NULL. Queries are reformulated to take into account `TenentID` so that correct answers can be found. Details of STSI will be presented in the following sections. STSI can achieve the best scalability since the number of tables is determined by the base schema and is therefore independent of the number of tenants. However, it introduces two problems: 1) The shared tables are too sparse. In order to make the base schema general, the service provider typically covers each possible attribute that the tenant may use, rendering the base schema has a huge number of attributes. On the other hand, for a specific tenant, only a small subset of attributes is actually used. Therefore, too many NULLs are stored in the shared table. These NULLs waste disk space and harm query performance. 2) Indexing on the shared tables is not effective. This is because each tenant has its own configured attributes and access patterns. It is unlikely that all the tenants need to index on the same column. Indexing the tuples of all the tenants is unnecessary in many cases.

In this paper, we propose a multi-tenant database system called M-store. We build M-store as a storage engine for MySQL to provide storage and indexing service for multiple tenants. M-store adopts STSI to achieve excellent scalability. To overcome the drawback of STSI, we develop two techniques. The first one is Bitmap Interpreted Tuple (BIT). Using BIT, only values from configured attributes are stored in the shared table. NULLs from unused attributes are not stored. Furthermore, a bitmap catalog which describes which attributes are used and which are not is created and shared by tuples from the same tenant. That bitmap catalog is also used to reconstruct the tuple when the tuple is read from the disk. In BIT, the overhead for compressing NULLs in unused attributes is near 0. Moreover, the BIT scheme does not undermine the performance of retrieving a particular attribute in the compressed tuple. To solve the indexing problem, we propose the Multi-Separated Index (MSI) scheme. Using MSI, we do not build an index on the same attribute for all the tenants. Instead, we build a separate index for each tenant. If an attribute is configured and frequently accessed by a tenant, an individual index is built on that attribute for the tuples that belong to that tenant.

Our contributions are as follows:

- We propose a novel multi-tenancy storage technique BIT. BIT is efficient in that it does not store NULLs from unused attributes in shared tables. Unlike alternative sparse table storage techniques such as vertical schema [8] and interpreted fields [10], BIT does not introduce overhead for NULLs compression and tuples reconstruction.
- We propose the MSI indexing scheme. To the best of our knowledge, this is the first indexing scheme on shared multi-tenant tables. MSI indexes data in a per-tenant manner. Each tenant only indexes its own data on frequent accessed attributes. Unused and infrequent accessed attributes are not indexed at all. Therefore, MSI provides good flexibility and efficiency for a multi-tenant database.
- We implemented BIT and MSI in M-store. M-store is a pluggable storage engine for MySQL which offers storage and indexing services for multi-tenant databases. By doing this, we show that our proposed techniques are ready for use and can be easily grafted into an existing database management system.
- We conducted extensive experiments to evaluate the ef-

ficiency and effectiveness of our proposed techniques. The results show that our approach is a promising multi-tenancy storage and indexing scheme.

The paper is organized as follows. Section II outlines the multi-tenant database system and discusses three possible solutions. Section III describes the proposed M-store. Section IV empirically evaluates the efficiency of M-store. Section V discusses related work, and is followed by conclusions and future work in Section VI.

II. The Multi-tenant Database System

A. Problem Settings

To provide database as a service, the service provider maintains a base configurable schema S which models an enterprise application such as CRM and ERP. The base schema $S = \{T_1, \ldots, T_n\}$ consists of a set of tables. Each table T_i models an entity in the business, e.g., `Employee`, and consists of C compulsory attributes and G configurable attributes.

To subscribe to the service, a tenant configures the base schema by choosing the tables to be used and in each table the configurable attributes that are appropriate for the application. Compulsory attributes must be chosen. The service provider may also provide certain extensibility to the tenants by allowing them to add some attributes if the attributes are not in the base schema. However, if the base schema is designed properly, this case does not occur often. We do not consider the extensibility issue in this paper. Discussion on that problem may be found in [9].

After configuration, the main problem is to store and index tuples in terms of the configured schema produced by the tenants. There are three approaches to building a multi-tenant database.

B. Independent Databases and Independent Database Instances (IDII)

The first approach to implementing a multi-tenant database is Independent Databases and Independent Instances (IDII). In this approach, tenants only share hardware (data center). The service provider runs independent database instances to serve independent tenants. Each tenant creates its own database and stores tuples there by interacting with its dedicated database instance. Figure 2 illustrates the architecture of IDII.

IDII is the simplest approach to implementing a multi-tenant database. It is entirely built on top of a current DBMS without any extension and it provides good data isolation and security. However, IDII introduces huge maintenance cost. To manage a variety of database instances, the service provider needs to perform a lot of configuration work. For example, to run a new MySQL instance, the DBA should provide a separate configuration file to indicate the data directory, network parameters, performance tuning parameters, access control list, etc. The DBA also needs to allocate disk space, memory, and network bandwidth for a new instance. Furthermore, the scalability of IDII is rather poor. The number of database instances grows linearly with the number of tenants. 1,000 tenants cause 1,000 database instances. The database instance is a heavy weight OS process. The startup of a modern database instance consumes tens of megabytes memory. Also, scheduling too many database processes introduces huge overhead to the operating system.

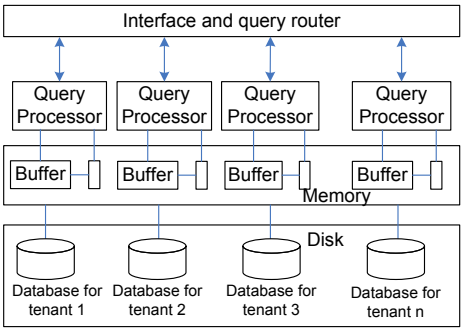


Fig. 2. Architecture of IDII

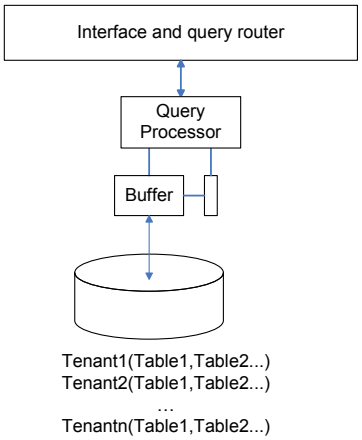


Fig. 3. Architecture of ITSI

C. Independent Tables and Shared Database Instances (ITSI)

The second multi-tenancy architecture is Independent Tables and Shared Instances (ITSI). In this approach, tenants not only share hardware but also database instances. The service provider maintains a large shared database and serves all tenants. Each tenant loads its tuples to its own private tables configured from the base schema and stores the private tables into the shared database. Each private table name is appended a `TenantID` to indicate who owns the table. As an example, tenant 1's private employee table reads $\texttt{Employee}_1$. Queries are also reformulated to recognize the modified table names so that correct answers can be returned. Typically, this reformulation is performed by a query router. Figure 3 depicts the architecture of ITSI. Table I shows the private `Employee` tables layout of three tenants.

ITSI provides better scalability than IDII and also reduces the huge maintenance cost of managing different database instances. However, the number of private tables in the shared database grows linearly with the number of tenants. Therefore, the scalability of ITSI is limited by the number of tables that

TABLE I
ITSI PRIVATE TABLE LAYOUT

(a) Private Table of Tanent3

ENo	EName	EAge
053	Jerry	35
089	Jacky	28

(b) Private Table of Tanent21

ENo	EName	EPhone	EOffice
023	Mary	98674520	Shanghai
077	Ball	22753408	Singapore

(c) Private Table of Tanent33

ENo	EName	EAge	ESalary	EOffice
131	Big	40	8000	London
088	Tom	36	6500	Tokyo

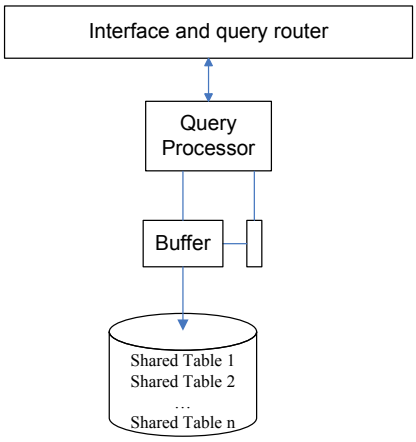


Fig. 4. Architecture of STSI

a database system can handle and is actually dependent on the available memory. A state-of-the-art machine such as a blade server can support up to 50,000 tables.

D. Shared Tables and Shared Database Instances (STSI)

The third multi-tenancy architecture is Shared Tables and Shared Database Instances (STSI). In STSI, tenants not only share a database but also tables. In the service setup phase, the service provider initializes the shared database by creating empty source tables according to the base schema. Each source table, called a Shared Table (ST), is then shared among the tenants. Each tenant stores its tuples in ST by appending each tuple with a `TenantID` attribute and setting unused attributes to NULLs. Table II shows the layout of a shared `Employee` table which stores tuples from three tenants.

To retrieve tuples from ST, a query router is used to reformulate queries to take `TenentID` into account. Figure 4 illustrates the architecture of STSI. As a query transformation example, to retrieve tuples in the `Employee` table, the source query issued by tenant 17 is as follows:

```
SELECT Name FROM Employee
```

The transformed query is:

```
SELECT Name From Employee WHERE TenentID='17'
```

Using STSI, the service provider only maintains a single database instance. The maintenance cost is therefore greatly reduced. Moreover, the number of tables in the database is determined by the base schema and is therefore independent of the number of tenants. Compared with IDII and ITSI, STSI provides the best scalability since its scalability is no longer limited by the available memory.

However, STSI introduces two performance issues. First, consolidating tuples from different tenants into the same ST causes that ST to store too many NULLs. The schema of ST is usually very wide, typically including hundreds of attributes. This is because the service provider wants to cover every possible attribute that the tenants may configure. For example, the service provider often offers many phone number attributes such as mobile phone number, office phone number and home phone number to meet the variety needs of tenants in storing contact information. For a tenant, it is less likely, if not impossible, that all the configurable attributes will be used. In typical cases, only a small subset of attributes is actually chosen. Thus, many NULLs are produced. Although commercial databases handle NULLs fairly efficiently, many studies have shown that if the table is too sparse, the disk space wastage and performance degradation are not negligible [10]. Second, fine-grained support for indexing is impossible. The tenant must build an index on the same attribute altogether or none of them can do it. This *all-or-nothing* limitation is not acceptable in many cases since different tenants have different indexing requirements.

So, the overall conclusion for STSI is that if the storage and indexing problems can be solved properly, it is a promising method for multi-tenant databases because of its excellent scalability.

III. M-STORE

This section presents our proposed multi-tenant system, M-store. M-store achieves excellent scalability by following the STSI approach and consolidating tuples of different tenants into the same shared tables. To overcome the drawback of STSI, M-store adopts the proposed Bitmap Interpreted Tuple (BIT) storage technique and Multi-Separated Indexing (MSI) scheme. We present them in the following subsections.

A. Bitmap Interpreted Tuple Format

One of the problems introduced by STSI is that storing tuples in a large wide shared table produces a number of NULLs. These NULLs waste disk bandwidth and undermine the efficiency of query processing. Existing methods that deal with the sparse table such as Vertical Schema and Interpreted Format either introduce much overhead in tuple reconstruction or prevent the storage system from optimizing random accesses to locate a given attribute. To the best of our knowledge, none of them is optimized for multi-tenant databases.

TABLE II
STSI SHARED TABLE LAYOUT

TID	ENo	EName	EAge	EPhone	ESalary	EOffice
Tenant 3	053	Jerry	35	NULL	NULL	NULL
Tenant 3	089	Jacky	28	NULL	NULL	NULL
Tenant21	023	Mary	NULL	98674520	NULL	Shanghai
Tenant21	077	Ball	NULL	22753408	NULL	Singapore
Tenant33	131	Big	40	NULL	8000	London
Tenant33	088	Tom	36	NULL	6500	Tokyo

One of the properties of a multi-tenant database is that tuples will have the same physical storage layout if they come from the same tenant. For example, if a tenant configures the first two attributes of the shared table T and leaves out the rest of the other two attributes, then all the tuples from that tenant will have a layout where the first two attributes have values and the last two attributes are NULLs. Based on this observation, we develop a Bitmap Interpreted Tuple Format (BIT) technique to efficiently store and retrieve tuples for multi-tenants without storing NULLs from unused attributes.

Our approach comprises two steps. First, a bitmap string is constructed for each tenant that decodes which attributes are used and which are not. Second, tuples are stored and retrieved based on the bitmap string of each tenant. We describe each step below.

In the first step, each tenant configures a table from the base schema by issuing a `CREATE CONFIGURE TABLE` statement, which is actually an extension of the standard `CREATE TABLE` statement. As an example, tenant 17 configures an employee table as shown below. We ignore the data type declaration in the base schema for simplicity.

```
CREATE CONFIGURE TABLE
Employee(ENo,EName)
FROM BASE
Employee(ENo, EName, EPhone, EPost);
```

Next, a bitmap string is constructed in terms of the table configuration statement. The length of the bitmap string is equal to the number of attributes in the base source table and positions corresponding to used and unused attributes are set to 1 and 0 respectively. In the above example, the bitmap string for tenant 17's employee example is 1100. The bitmap string is thereafter stored somewhere for later use. In our implementation, bitmap strings of tenants are stored with the table catalog information of the shared source table. When the shared table (ST) is opened, the table catalog information and bitmap strings are loaded into the memory together. This *in-memory* strategy is possible in that even when the base source table has 100 attributes, loading bitmap strings for 1000 tenants only incurs about 12KB memory overhead, which is rather negligible. Figure 5 demonstrates our implementation on MySQL. We extend the MySQL's table catalog file, i.e., .frm file associated with the table created in MySQL, and append the bitmap strings of the tenants at the end of the file immediately following the original table catalog information part.

In the second step, tuples are stored and retrieved according to the bitmap strings. When a tenant performs a tuple insertion, NULLs in attributes whose positions in the bitmap string are marked as 0 are removed. The rest of the attributes in the inserted tuple are compacted as a new tuple and finally stored in the shared table. The physical layout of the new compacted tuple is the same as the row-store layout used in most current commercial database systems. It begins with a tuple head which includes tuple-id and tuple length. Next is null-bitmap and values in each attribute. Fixed-width attributes are stored directly. Variable-width attributes are stored as length-value pairs. The null-bitmap decodes which fields in the configured attributes are null. We should not confuse the nulls in the configured attributes with NULLs in unused attributes. The nulls in configured attributes means the values are missing while the NULLs produced by unused attributes indicate the attributes are not configured by the tenant. Figure 6 summarizes the tuple insertion process.

To retrieve specific attributes in the tuple, the bitmap string is also used. If all the configured attributes are of a fixed width, the offset of each attribute can be efficiently computed by counting the number of ones before the position of that attribute in the bitmap string. In our implementation, if the tuple is of a fixed width, the offset of each attribute is computed when the bitmap string is loaded into memory. If a variable-width attribute is involved, calculation of the offset of attribute A_n requires addition of data-lengths of the prior $n-1$ attributes. The algorithm is the same one used in commercial database systems, and we shall ignore the details here.

Compared with the alternative sparse table storage techniques such as Interpreted Format and Vertical Schema, our proposed technique BIT is specifically designed for supporting multi-tenant databases. To store tuples from different tenants in a wide base table, we only maintain a per-tenant bitmap string whose length is fixed by the number of attributes in the base schema. Therefore, the overhead for storing NULLs in unused attributes per tuple is near 0. None of the aforementioned techniques can give such a guarantee. Furthermore, if the configured table is of a fixed width, our approach does not degrade the performance of random access on the attributes. This is a performance gain that cannot be achieved with the Interpreted Format.

B. Multi-Separated Indexing

In a multi-tenant database, the shared table stores tuples from a number of tenants. The data volume is huge. Therefore,

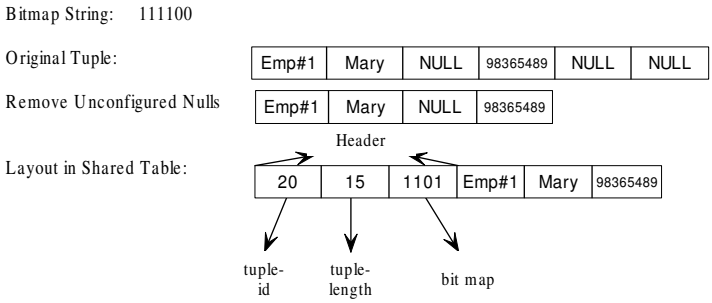


Fig. 6. Process of Tuple Insertion in ST

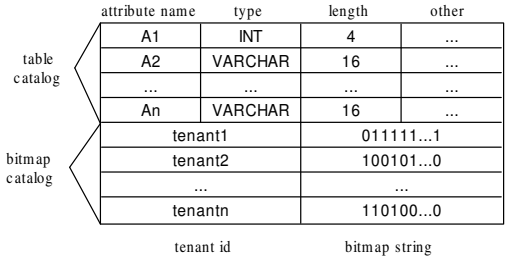


Fig. 5. Catalog of BIT

it is of vital importance to develop an efficient indexing technique to retrieve tuples in the shared table.

In principle, one can build a big B^+-tree on a given attribute of the shared table to index tuples from all the tenants. We call this approach the *Big Index* (BI). The BI approach has the advantage that the index is shared among all tenants. As a result, the memory buffers for index pages may be efficiently utilized, especially for selection and range queries. In these queries, the search path starts from the root to leaves. Buffering the top index pages (pages towards the root) in the memory will reduce the number of disk I/Os when multiple tenants concurrently search the index. Unfortunately, index scan is fairly inefficient. To step through its own keys, a common operation for aggregate and join queries, a tenant needs to scan the whole index file which is very time consuming because the index has keys of all tenants. Furthermore, the BI approach fails to provide sufficient flexibility for a multi-tenant database. Although tenants consolidate their tuples into the same shared tables, different tenants configure different attributes and will have different access patterns. It is therefore impossible to support a fine-grained index.

Instead of using one BI for each ST, in M-store, we propose another indexing scheme called *Multi-Separated Index* (MSI). Instead of building an index for all tenants, we build an index for each tenant. If a hundred tenants want to index tuples on an attribute, one hundred individual indexes are built for these tenants. At first glance, MSI may not be efficient since the number of indexes grows linearly with the number of tenants and too many indexes may contend for the memory buffer, thus slowing down the query. However, as we have mentioned, different tenants configure different attributes and have different access patterns on those attributes. Therefore, given a particular attribute, only a fraction of tenants will build an index on that attribute. So, in real applications, MSI does not make the number of indexes explosive.

Compared to BI, MSI has several advantages. First, MSI is flexible. Each tenant indexes its own tuples on the fly. We do not enforce that all tenants have to build index on the same attribute or none of them can do it. Second, index scan in the MSI approach is efficient. To perform an index scan, each tenant needs to scan only its own index file. This is unlike BI, where all the tenants share the same index, causing a tenant to scan the whole index even if the tenant only wants to retrieve a small subset of keys that belong to it in the index.

At this point, we must note that MSI is different from view indexing [17], [2]. View is dynamic and content based – a tuple that is indexed is dropped when its index attribute value does not satisfy the view. On the contrary, the number of tuples indexed by an MSI index for a tenant over an attribute does not change with respect to changes to attribute values. MSI is also different from partial indexing scheme which builds indexes on less than a complete column [20]. MSI builds an index on a complete column. If a tenant builds an index on a column, all tuples belonging to that tenant will be indexed no matter whether the tuples qualify a partial filtering condition or not. That is, each MSI behaves like a conventional index, but over a subset of tuples that belong to a given tenant.

IV. Experiments

In this section, we empirically evaluate the efficiency and scalability of M-store through two main sets of experiments. In the first set of experiments, we consider the scalability of the storage module in M-store by measuring disk space usage as the number of tenants increases. In the second set, we evaluate the query performance of M-store through executing three kinds of query workloads, namely simple queries, analysis queries, and update queries. The original STSI is used as the baseline in the experiments.

A. Benchmarking

It is of vital importance to use an appropriate benchmark to evaluate multi-tenant database systems. Unfortunately, to the best of our knowledge, there is no standard benchmark for this task. Traditional benchmarks such as TPC-C [3] and TPC-H [4] are not suitable for benchmarking multi-tenant database

systems. TPC-C and TPC-H are basically designed for single-tenant database systems, and they lack an important feature that a multi-tenant database must have, namely, the ability for allowing the database schema to be configurable for different tenants. Therefore, we develop our own DaaS (Database as a Service) benchmark by following the general rules of TPC-C and TPC-H.

Our DaaS benchmark comprises three modules: a configurable database base schema, a query workload generator, and a worker. We describe them below.

We follow the logical database design of TPC-H to generate the configurable database base schema. Our benchmark database comprises four tables. These tables are chosen out of eight tables from the TPC-H database. They are: `lineitem`, `orders`, `part` and `customer`. For each table, we extend the number of attributes to 100 by appending attributes to the original table schema, one of which is `tid` (tenant ID) that denotes the tuple owner. The data type of extended attributes, exclude `tid` whose data type is integer, is string. The first two or three attributes in each table are marked as compulsory attributes that each tenant must choose. The rest of the other attributes are marked as configurable. The simplified `customer` table schema is given below for illustration purpose. In this example, `tid`, `c_custkey`, and `c_name` are compulsory attributes. The rest attributes, i.e., `c_col1`, `c_col2`, and `c_col3`, are configurable.

```
customer(
    tid, c_custkey, c_name,
    c_col1, c_col2, c_col3
)
```

We develop a tool called SGEN to generate private schemas for each tenant. SGEN is configured by three parameters, i.e., the number of tenants N_t, the average number of configured attributes μ, and the derivation σ. To generate private schemas for tenant T_i, SGEN randomly selects n_i configurable attributes in each table from the database and collects the chosen attributes to form the final private database schema. The number n_i is a random number chosen from normal distribution $\mathcal{N}(\mu, \sigma)$. The process is repeated N_t times to generate private table schemas for N_t tenants.

To populate the database, we use MDBGEN for data generation. MDBGEN is essentially an extension of DBGEN tool equipped with TPC-H. It actually uses the same code of DBGEN to generate value for each attribute. The only difference is that MDBGEN generates data for each tenant by taking into account the private schema of that tenant. The values in the extended attributes are generated by random v-string algorithm used in DBGEN. The values in unused attributes are outputted as NULLs.

Following TPC-C and TPC-H, we design and implement a query workload generator to generate the query sets for benchmark. Our query generator can generate three kinds of query workloads:

- **Simple Query:** Randomly select a set of attributes of tenants according to a simple filtering condition. An example of such a query is as follows. Note that we have associated the predicate on `tid` to locate the correct answers for the given tenant when we generate queries.

```
SELECT c_custkey, c_name
FROM customer
WHERE customer.tid=100 AND
      customer.c_custkey>9000;
```

- **Analysis Query:** Run reporting queries which perform join, aggregation, and/or grouping on the shared tables of the tenants. An example is given as below.

```
SELECT max(o_orderkey)
FROM orders, lineitem
WHERE lineitem.tid=100 AND
      orders.tid=100 AND
      o_orderkey=l_orderkey;
```

- **Update Query:** Insert and delete tuples of tenants to the shared tables.

The last module in our benchmark is worker. It is conceptually equivalent to the driver in the TPC-H benchmark. The worker submits queries to the multi-tenant database system under test and measures and reports the execution time of those queries. We run worker and the multi-tenant database system in a "client/server" configuration. We place the worker and the database system in different machines interconnected by a network. The worker is written in Java and interacts with the database system through standard JDBC interface. It is designed to simulate concurrent accesses to the database system from multiple tenants. The worker achieves this by stimulating N_s database sessions to the database system concurrently. Each database session is dedicated to perform query streams of a fixed number of tenants.

B. Experimental Settings

We present the experimental settings in this section. We first present settings for benchmark databases generation. Then, we present hardware and software settings.

We generate private database schemas for tenants by running SGEN. For each tenant, we generate 4 sets of schemas by setting μ to 5, 10, 20, 30 respectively and fixing $\sigma = 2$. We finally generate 4 groups of schemas for 100, 400, 700, 1,000 tenants. These schemas are then used for evaluating the scalability of storage and query processing under different schema variabilities.

We run MDBGEN to generate data for benchmark databases according to the resulting private schemas. For each tenant, we generate 5,000 tuples for `lineitem`, 2,000 tuples for `orders`, 1,500 tuples for `customer` and `part`. The raw data disk space of 1,000 tenants is 9.23 GB under the setting of $\mu = 30$.

For the worker, we set the number of concurrent database sessions $N_s = 50$. We also set each database session to serve equal number of tenants. So, to perform query streams from 100 tenants, each database session is dedicated to run queries from unique 2 tenants.

The worker and the database system are run on two different machines. These two machines are connected by 1.0Gb/s LAN. Each machine is equipped with a Intel Core2 Duo CPU and 4GB memory.

We evaluate two kinds of multi-tenant database systems. One is STSI, and the other is M-store. We implement the STSI on top of MySQL 5.1.26. We choose MyISAM as the underline storage engine for the storage and indexing components of STSI. MyISAM is a known and proven as a popular storage engine for highly scalable Web applications and is the default storage engine of MySQL. BI is used as the indexing scheme of STSI.

We implement M-store as a custom plug-in storage engine of MySQL so that the two systems, i.e., STSI and M-store, can be compared under the same database server. To create a shared `customer` table in MySQL with M-store engine, one can issue following statement. We use MSI indexing scheme in M-store.

```
CREATE TABLE customer(
    tid, c_custkey, c_name,
    c_col1, c_col2, c_col3
) engine=mstore;
```

As for the server performance tuning parameters such as block size, memory buffer size, we use the default settings of MySQL.

Following the guideline of TPC-H benchmark, the experiment is conducted as an execution of the load test followed by the performance test. In the load test, we populate the database with generated data and study the scalability of the storage module, measured by the disk usage, as the number of tenants increases under different schema variability settings. In the performance test, we evaluate the query performance with three kinds of query workloads.

C. Scalability of Storage

Figure 7 depicts the disk space usage of M-store and STSI in different settings. It can be clearly seen that M-store outperforms STSI in terms of storage requirement in all the experiments. For a very sparse setting, i.e., Figure 7(a), M-store only uses 30% storage space of STSI to store the same number of tuples. The reason is as follows. In this setting, the average number of attributes each tenant configures is 5, i.e., $\mu = 5$, but the total number of attributes in the base table is 100. Therefore, STSI consumes large disk space to store NULLs in the tuples. M-store, on the other hand, does not store these NULLs. So it uses little disk space. As schema sparsity decreases, the space usage gap between M-store and STSI decreases gradually. However, even if 30% attributes are used on average, i.e., Figure 7(d), M-store still outperforms STSI by reducing 20% disk space.

D. Performance of Simple Queries

In this experiment, we test the query performance of M-store and STSI under simple query workloads. Particularly, we are interested in the performance comparison of indexing techniques employed in M-store and STSI. Simple queries are typical OLTP workloads. It is well known that appropriate indexing technique can speedup such queries.

To conduct the experiment, we built indexes on the DaaS benchmark databases. We first built a primary index on `tid` of each shared table. This index is built for efficient table scan for a specific tenant, namely speedup query that retrieve all tuples belonged to that tenant. Then, we built indexes on seven randomly selected attributes. Each index is a compound index of that attribute and tenant ID. For example, the index we built on `customer` is `tid` and `c_custkey`. For STSI, all tuples in the shared table are indexed no matter whether the tuple owner configured attributes in the compound index. For M-store, different tenant has different index structures. If a tenant does not configure an attribute in a compound index, the index structure belonged to that tenant is empty, namely no tuples from that tenant will be indexed.

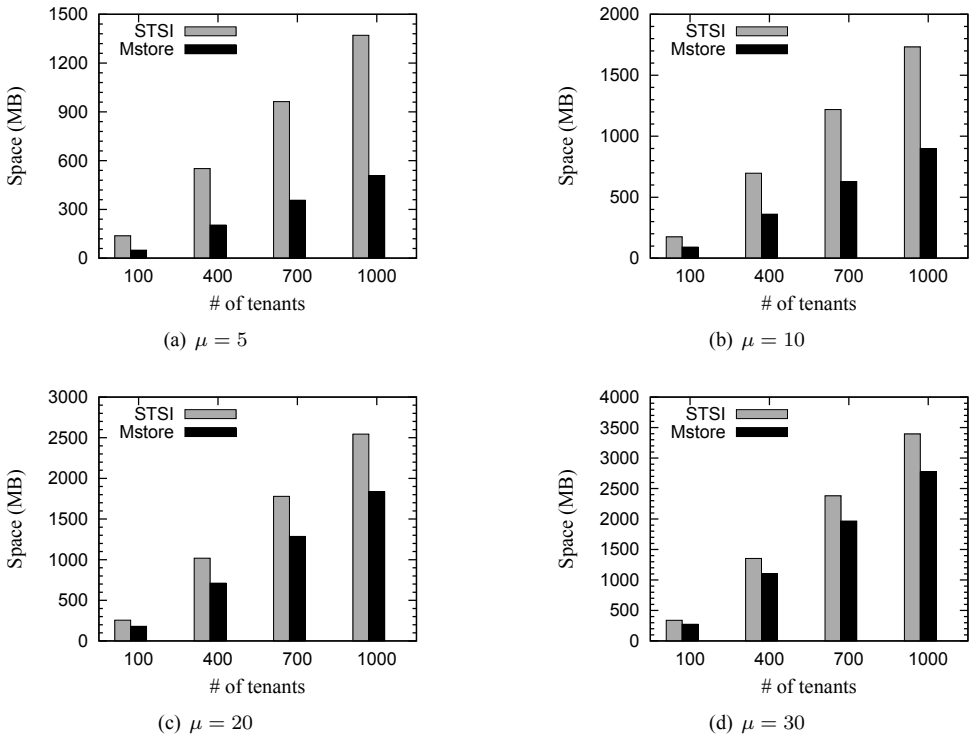
The query processing of STSI is simple. The query optimizer just chooses an appropriate index, if available, and then processes the query. The MySQL query optimizer performs a fairly good job in this task. We use it for STSI directly.

For M-store, query processing is a little more complicated. In M-store, even though the index is built on the shared table, different tenants do not share the same indexing structure. Instead, they have their own indexing structure. So, it is important to route the queries to the correct indexing structures. The MySQL query optimizer is not able to handle this new task since it has not been designed for multi-tenant databases. In our implementation, we use a workaround to solve the problem, and we illustrate how it works with a concrete example below.

Suppose we want to build an index I on `(tid,c_custkey)` of the shared `customer` table. We first issue a `CREATE INDEX` statement on `customer` as we do for STSI. This statement creates an empty index tree T on the given attributes. We actually insert a key, say (20, 12300), to the index. Instead of inserting that key to T, we insert the key to T_{20} which is an index tree built for tenant 20. During query processing, if the query contains filter conditions on `(tid,c_custkey)`, we manually reformulate the query to force the query optimizer to use the index I even if it is empty. This is done by adding a `FORCE` hint in the original select statement. More information may be found in the MySQL manual, and we omit the details here. When the query is actually executed, MySQL will pass the name of index tree, i.e., I and the search key to the index search module of M-store. We then extract the tenant ID information from the search key and route the query to the proper index tree. This workaround helps save development work on modifying the MySQL query optimizer and it is sufficient for evaluating the idea. In the next version of M-store, we will modify the MySQL query optimizer to select the right index automatically and remove the workaround.

We finally use the workload generator to generate 5,000 simple queries and assign these queries to tenants. Figure 8 plots the average response time collected by the worker module. The experiment is repeated six times to reduce the software and hardware impact. For each run, we restart the database server machine and client machine so that memory buffers are flushed.

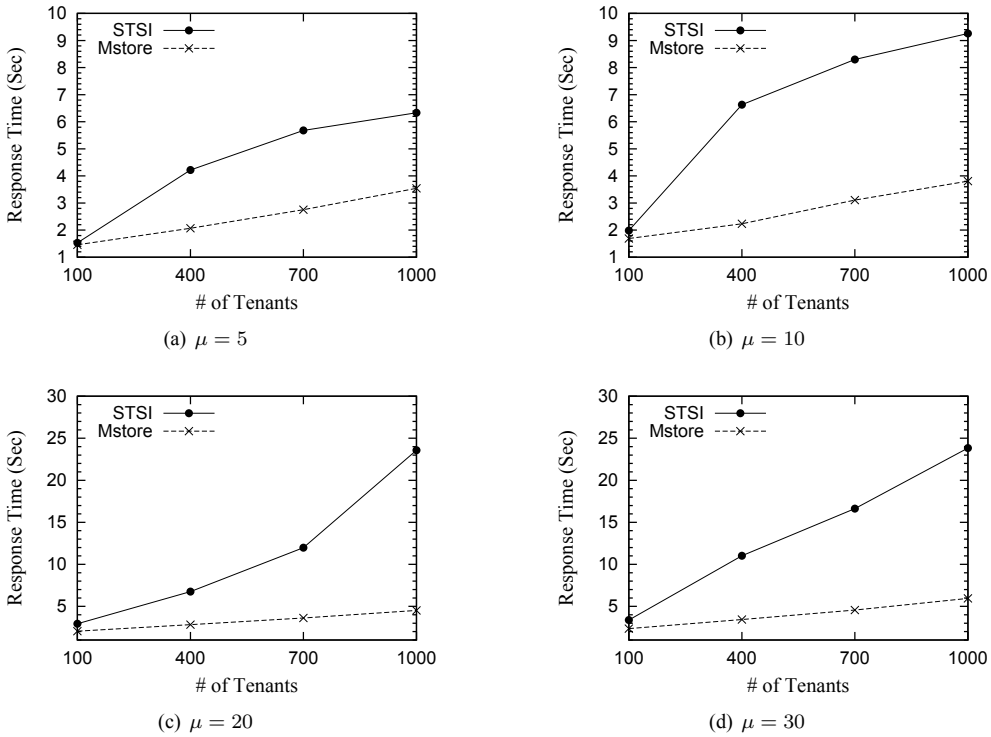
From Figure 8, we can see that the scalability of M-store



(a) $\mu = 5$ (b) $\mu = 10$

(c) $\mu = 20$ (d) $\mu = 30$

Fig. 7. Disk space used by STSI and M-store



(a) $\mu = 5$ (b) $\mu = 10$

(c) $\mu = 20$ (d) $\mu = 30$

Fig. 8. Results for simple queries performance

is quite good. In spite of schema variability, the simple query performance of M-store does not degrade much as the number

of tenants grows from 100 to 1,000. On the other hand, the query performance of STSI degrades sharply when the number of tenants increases. M-store outperforms STSI for two reasons. First, compared to STSI, M-store uses less space to store the same number of tuples. That is to say, given a query, M-store uses fewer disk I/Os to load the answer tuples of that query to memory than STSI. This feature particularly saves times when the database server performs a table scan to retrieve all tuples belonging to a given tenant. Second, in STSI, all the tenants share the same B^+-tree index that is built on the attributes of the shared table. As more and more tenants load tuples into the database, more and more keys are inserted into the shared B^+-tree. Thus, the B^+-tree becomes taller and taller. The index lookup becomes inefficient since every index lookup goes from the root to the leaves. In contrast to STSI, M-store builds a separate index for each tenant. The height of each index tree is entirely determined by the number of tuples belonging to that tenant. Thus, index lookup performance does not suffer from the increasing number of tenants. However, too many index trees increase the number of random disk I/Os when the database server loads index pages to memory. This is why the response times of M-store also increase as more and more tenants issue queries. In our experiments, this performance penalty is not great. We will use a larger benchmark in the future and study whether this argument still holds with hundreds of thousands of tenants.

E. Performance of Analysis Queries

In this experiment, we run 2,000 analysis queries on M-store and STSI. These queries are generated from 4 reporting query templates. Each query template contains join, aggregation and grouping on the shared tables. The attribute in join condition and aggregation is chosen from the indexed columns. Other settings are the same with simple query experiments.

Figure 9 shows the average execution time of the benchmark analysis queries. It can be seen from the figure that there is a clear performance gap between M-store and STSI. Further analysis on query plans reveals the cause of the performance gap. MySQL only supports the nested loop join algorithm. To perform join query $R \bowtie S$, MySQL uses the `tid` index to retrieve tuples in R belonging to the given tenant and then uses another index to join tuples with S by index lookup. Compared to STSI, M-store is more efficient in performing table scan and index lookup since disk space is smaller and index trees are shorter. So, M-store outperforms STSI.

F. Performance of Updates

In the final experiment, we randomly generate 1,000 insertions and 1,000 deletions using a workload generator and evaluate the update performance of M-store and STSI.

Figure 10 depicts the average execution time. In all settings, M-store performs very well. On average, M-store performs three to five folds faster than STSI when 1,000 tenants concurrently insert and delete tuples from the database. This is mainly because M-store adopts the MSI indexing scheme. By building a separate index for each tenant, there is no need to assign and release locks between tenants when they concurrently insert and delete tuples from the database. Therefore, the overhead for concurrent control management is minimized. Furthermore, the disk I/O cost of each insertion and deletion is mainly determined by the number of tuples belonging to the tenant. The cost is not sensitive to number of tenants. Therefore, as we can see in Figure 10, the average execution time of M-store does not grow much when the number of tenants increases. In contrast, STSI follows the single shared indexing scheme. As the number of tenants grows, more overhead on concurrent management is introduced. Moreover, for a single insertion and deletion, the disk I/O cost also grows with the number of tenants. So, the performance gap between M-store and STSI becomes larger and larger in Figure 10.

V. Related Work

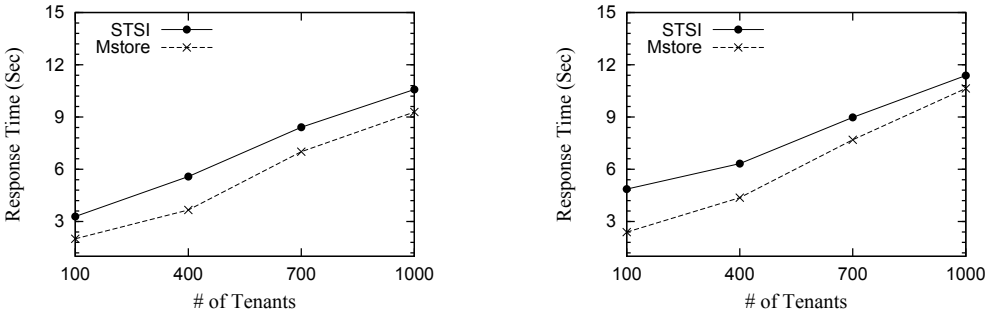
The work that are related to ours can be classified into two categories: a) outsourcing database as a service, and b) extending relational DBMS to support sparse datasets.

Research in category a) focuses on designing a system which provides database as a service. In [15], a system called NetDB2 was proposed to provide mechanisms for organizations to create and access their databases at the host site managed by a third party service provider. This work focuses on solving data security issues. In [19] and [16], further study was carried out on the assurance and security issues in query execution and indexing. These works are complementary to ours in that they focus on security issues while we study the scalability issue.

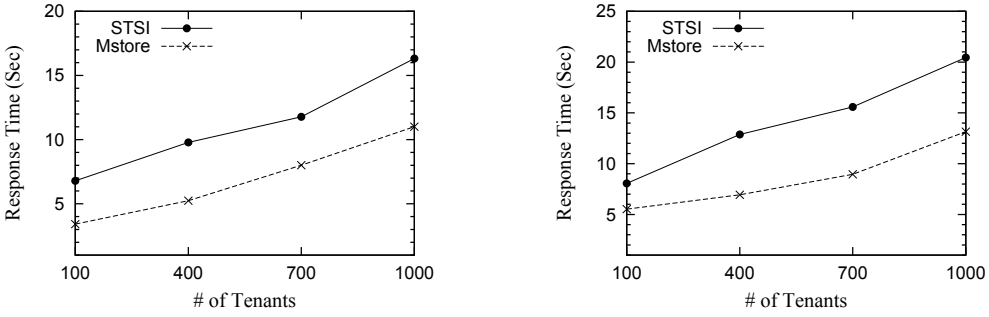
In [9], the authors present schema-mapping techniques for multi-tenant databases. This work is closely related to ours, as it also presents ITSI and STSI architectures for multi-tenant databases. Our work was independently started in the late 2007, and the fundamental difference between their proposal and ours is the motivation. They focus on an extension of STSI that provides extensibility for multi-tenant databases. Particularly, they propose a chunk folding technique which uses chunk tables to store data in extension attributes. Our proposal, on the other hand, focuses on improving the storage scalability and query performance of STSI by designing new storage and indexing schemes. In the future work, we are interested in whether our storage scheme can be enhanced to support extensibility but without the performance overhead of tuple reconstruction incurred by chunk folding.

In [21] and [13], the authors propose PNUTS, a hosted data serving platform which is designed for various Yahoo's web applications. The proposal focuses on providing low latency for concurrent requests including updates and queries by use of massive servers. They do not consider scalability or the maximum applications that each server can support.

BigTable [11] is developed and deployed by Google as a structured data storage infrastructure for different Google's products. The design and implementation of BigTable is different from ours. To scale up the system to thousands of machines and serve as many projects as possible, BigTable employs a simple data model that presents data as a sorted

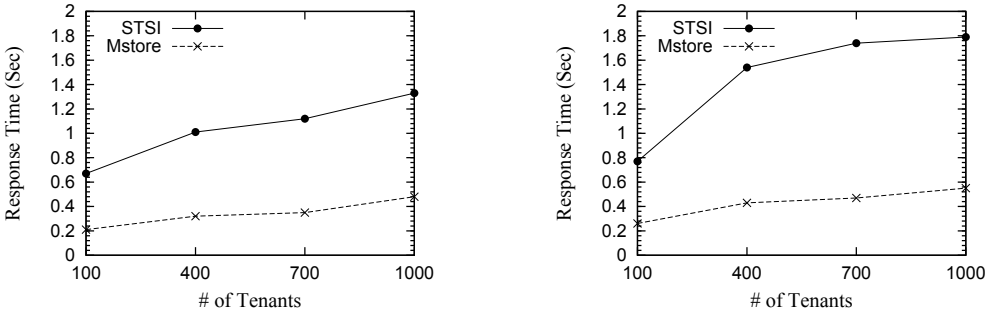


(a) $\mu = 5$ (b) $\mu = 10$

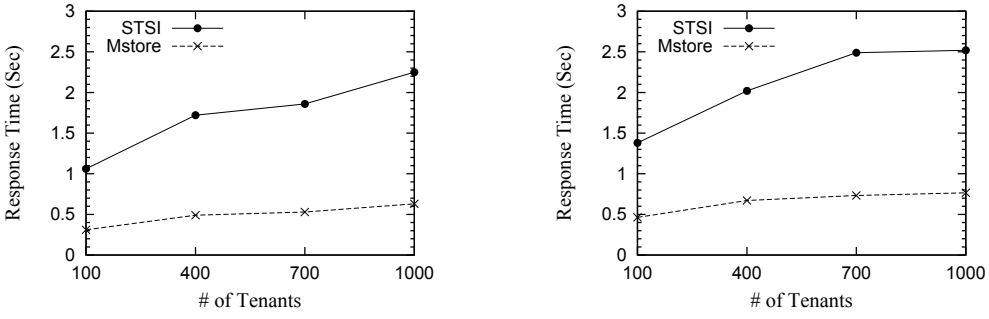


(c) $\mu = 20$ (d) $\mu = 30$

Fig. 9. Results for analysis queries performance



(a) $\mu = 5$ (b) $\mu = 10$



(c) $\mu = 20$ (d) $\mu = 30$

Fig. 10. Results for updates

map in which each value is an uninterpreted string. BigTable enforces the user, by writing specific programs, to store and retrieve those uninterpreted strings. This data model is not suitable to our applications. It burdens a high development cost on the customer and may prevent transferring ownership and maintenance cost of applications from the customer to the service provider. As we argued in this paper, this transfer is an effective way for the service provider to reduce the cost and catch long tail markets.

In [18], the authors present SHAROES, a system which delivers raw storage as a service over a network. The proposal focuses on delivering a secure raw storage service without consideration on the data model and indexing. Our proposal, on the other hand, provides a richer database facilities, including a multi-tenant configurable relational data model and indexing. Amazon's S3 and SimpleDB services, and Microsoft's CloudDB project are the systems which are also related to our work. However, little information about them has been published so far. Therefore, we cannot compare the architecture and techniques of these systems to ours.

Research in category b) focuses on extending relational DBMS to support sparse datasets where relations exhibit many attributes that are NULL for many tuples. These works are related to ours. In our system implementation, we also investigate effective ways to store a large number of NULL values. The work in [10], [12] proposes using interpreted format to store sparse datasets in relational DBMS. A similar method is used in our previous work [22] which stores user contributed tags in a sparse table. Compared to these work, our proposed BIT technique is explicitly designed for supporting multi-tenant databases. Using BIT, the space overhead for storing NULLs in unused attributes is near to 0. This guarantee cannot be achieved by these work. The work in [7] discusses effective techniques for compressing NULL values in a column-oriented database system. While our design is influenced by this work, our system is targeted at developing multi-tenancy database service support on top of traditional column-oriented database systems.

VI. Conclusion and Future Work

In this paper, we have proposed and developed the M-store system which provides storage and indexing service for a multi-tenant database system. The innovative techniques embodied in M-store include:

- A Bitmap Interpreted Tuple storage format which is optimized for multi-tenant configurable shared table layout and does not store NULLs in unused attributes.
- A Multi-Separated Indexing scheme that provides each tenant fine granularity control on index management and efficient index lookup.

Our experiments on a 9.23 GB database benchmark show that Bitmap Interpreted Tuple significantly reduces disk space usage and Multi-Separated Indexing considerably improves index lookup speed as compared to the STSI approach.

In our future work, we intend to extend M-store to support extensibility. In our current implementation, we assume the number of attributes in the base schema is fixed. However, as presented in [9], in certain applications, the service provider may add attributes to the base schema to meet the specific purposes of tenants. We will study whether an extension to M-store can support that requirement. Another direction is query processing. Currently, we manually activate the optimizer to use the right index. As a next step, we will study how to get the optimizer to generate best query plans with Multi-Separated Index automatically.

Acknowledgement

We would like to thank Beng Chin Ooi for his insight and supervision and NUS DBA Eng Koon Sze for his explanation on the features of a commercial system.

References

[1] Architecture strategies for catching the long tail (microsoft) http://msdn2.microsoft.com/en-us/library/aa479069.aspx.
[2] http://www.sqlteam.com/article/indexed-views-in-sql-server-2000.
[3] http://www.tpc.org/tpcc/.
[4] http://www.tpc.org/tpch/.
[5] The long tail http://www.wired.com/wired/archive/12.10/tail.html.
[6] Multi-tenant architecture http://msdn.microsoft.com/en-us/library/aa479086.aspx.
[7] D. J. Abadi. Column stores for wide and sparse data. In *CIDR*, Asilomar, CA, USA, 2007.
[8] 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 Data Bases*, pages 149–158, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
[9] 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.
[10] 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 the 22nd International Conference on Data Engineering*, page 58, Washington, DC, USA, 2006. IEEE Computer Society.
[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
[12] 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*, pages 821–832, New York, NY, USA, 2007. ACM.
[13] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!s hosted data serving platform. In *VLDB'08*, 2008.
[14] E.TenWolde. Worldwide software on demand 2007-2011 forcast: A preliminary look at delivery model performance. In *IDC Report*, number 206204, 2007.
[15] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, 2002.
[16] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, 2006.
[17] N. Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.
[18] A. Singh and L. Liu. SHAROES: A data sharing platform for outsourced enterprise storage environments. In *ICDE'08*, 2008.
[19] R. Sion. Query execution assurance for outsourced databases. In *VLDB*, 2005.
[20] M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.
[21] Sunnyvale. Community systems research at yahoo! *SIGMOD Record*, 36(3):47–54, September 2007.
[22] B. Yu, G. Li, B. Ooi, and L. Zhou. One table stores all: Enabling painless free-and-easy data publishing and sharing. In *CIDR*, 2007.