DE GRUYTER OPEN

BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **16**, No 3
Sofia • 2016 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.1515/cait-2016-0036

Isolated Storage of Multi-Tenant Data Based on Shared Schema

Lida Zou, Qingzhong Li, Lanju Kong

School of Computer Science and Technology, Shandong University, Jinan, Shandong, 250101 China
Emails: zoulida@163.com 1qz@sdu.edu.cn k1j@sdu.edu.cn

Abstract: *Multi-tenant data management is an important part of supporting efficient operation of software as a service application. Multi-tenant data use shared schema to reduce resource usage cost. However, massive data of different tenants are stored in the same schema, which causes useless data of other tenants to be read when a tenant just need access its own disk data. In this paper we focus on disk storage method of multi-tenant data based on shared schema to address the above low efficiency of data access. According to isolation requirement of multi-tenant data, we store a tenant's data in some contiguous disk blocks. The experimental results illustrate that query efficiency in range query and join query is 1.5-2 times the existing storage method, and no indexing query efficiency improves 10-70 times.*

Keywords: *Multi-tenant database, disk storage, shared schema, SaaS.*

1. Introduction

Along with continuous expansion of Software as a Service (SaaS) [1] application, multi-tenant database becomes an important foundation for quick development and efficient operation of SaaS application. Multi-tenant data storage is the base of multi-tenant data access. As for storage scheme of multi-tenant database, Frederick Chong [2] proposes three methods including separate database, shared database separate schema and Shared Database Shared Schema (SDSS).

With the development of cloud computing, multi-tenant databases are deployed on many cloud nodes with SDSS [3, 5, 8]. The persistent storage medium of data processing nodes are mostly low-cost disks. According to characteristics of disk data access, access efficiency will improve if data in database are stored in contiguous disk blocks [4, 9]. Graefe and Shapiro [10] conclude that the more the valid data in one disk block, the higher the access efficiency, since database access task can be finished just by inputting/outputting a few disk blocks.

Most of multi-tenant data deployed in the clouds are stored on these low-cost disks [5]. The storage method of SDSS stores data of all tenants in the same schema table. When data are persisted to disk, data tables become huge and a disk block contains multiple tenants' data. This gives rise to the following problems when accessing multi-tenant data.

1. Query efficiency is low. Tenant data access has characteristics of isolation, i.e., tenant just accesses its own data. Since a query needs buffer memory and disk to swap data, data volume and times of I/O is critical for query performance. Non-isolated stored data need read more disk blocks to finish one query processing from a tenant.

2. Query optimization cannot be done for multi-tenant database. To do query optimization, data volume of each data object and their used number of disk blocks need counting. However, non-isolated storage schema stores data of different tenants into the same disk block, which messes up relational algebra logic and makes it hard to do statistics.

3. Efficiency of data analysis is low. When doing analysis, data filtering by no indexing attributes often exists and it need traverse the whole data table. Since multi-tenant table in SDSS is huge, do analysis of multi-tenant data is time-consuming.

To address the above problems, we present a novel isolated storage method for multi-tenant data. It makes continuous disk blocks as a storage area and each area is allocated to data objects of one tenant. This ensures that one disk block just stores data from one tenant. Since different tenants have different data volume and data volume increases gradually with time, storage space for each tenant increases gradually. This way adapts to the demands of different tenants for storage space and achieves efficient usage of disk space. Last our experiments show that query efficiency of range query and join query is 1.5-2 times InnoDB storage engine [6, 7], and no indexing query efficiency improves 10-70 times.

2. Multi-tenant data storage structure

In this section we introduce multi-tenant data storage process in shared schema and discuss the problems of disk storage for multi-tenant data. For clear description, some definitions are given as follows:

The organization or user who rents multi-tenant application is called *tenant* and whose subscript is denoted as t to distinguish different tenants. A data record to describe a business object of tenant is called *tuple* and whose subscript is denoted as to distinguish different tuples. *Data object* is a collection of tuples with the same database schema. A tenant may have many data objects. We use to denote the subscript of data object belonging to a tenant. The concrete location of disk where a tuple is stored is called *physical address* and denoted as *p*. Multi-tenant database can directly locate and obtain tuple data by physical address. In operating system, disk space is used after disk partition. The data in database are stored in one disk partition. Assume the start address of one disk partition is SA and the relative

address of a tuple in the disk partition is r, the physical address of the tuple is $p = SA + r$.

The storage process of multi-tenant data consists of three steps, just as shown in Fig. 1. We explain it as follows.

1. It uses standard SQL to store tenant data into logical tenant view. Tenants think they exclusively use storage and processing resources and they access tenant view just like common database. Thus tenant view has data tables belonging to different tenants and database schemas of these tables are different.

2. It stores different tables of tenant view into universal table. Tenant identifier and data object identifier are added in the front of each tuple to mark its belonging tenant and data object. When querying certain tenant data, query rewriting technology is used [3, 11], i.e., it first selects the data belonging to target tenant and target data object and then executes the business query for the data.

3. It stores the tuples in universal table to disk through mapping address. Given the tuple identifier, it gets physical address of the tuple using address mapping method and stores it.

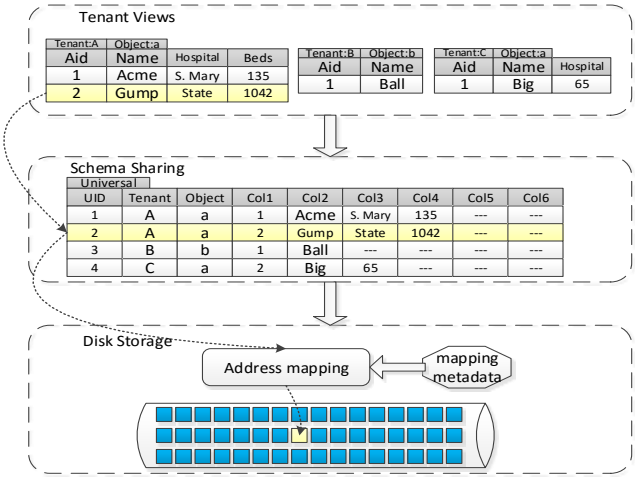

Fig. 1. Multi-tenant data storage process

The tuple data of different tenants are cross-generated. Since multi-tenant data storage schema based on universal table cannot ensure that the data belonging to one tenant are stored contiguously, a disk block may store the data belonging to many tenants. A tenant only accesses its own data, which causes useless data of other tenants to be read and reduces query efficiency and access efficiency.

An intuitive means is to reserve storage space in universal table for each tenant in advance and achieve the isolated storage of tenant data. For example, the tuples marked by $\delta \in [0, 99]$ store the data of tenant A and the tuples marked by $\delta \in [100, 199]$ store the data of tenant B. Since the isolation is implemented logically, this method is called Logically Isolated Storage Approach (LISA). However, the data volumes of different tenants are different and the number of tenants gradually increases as well. This method of reserving storage space causes

low space usage efficiency. In Section 5, LISA is chosen as one of comparison objects to help validate our designed storage approach.

From the above, it is urgent to design a disk storage approach of multi-tenant data based on SDSS to address the problem of low access efficiency caused by non-isolated storage. The proposed disk storage approach should achieve isolated and contiguous storage of tenant data to improve both space usage efficiency and data access efficiency.

3. Isolated storage of multi-tenant data

In this section we store tenant data to disk in isolated way. We design a storage approach where tenant storage space gradually increase with data volume and each disk block just stores the tuples of a data object belonging to one tenant. For convenient description, we first assume that each tenant has only one data object and then extend to the scenario of multiple data objects.

3.1. Formal definition of storage unit in disk space

Given a disk partition, multi-tenant data are stored in the partition. For the convenience of managing disk storage space, some definitions on disk storage unit are given as follows:

Definition 1. The minimum unit of swapping data between buffer memory and disk is called **block**. In a disk partition, the storage space consists of several blocks. The blocks are numbered sequentially. Assuming b is the subscript of block, the size of the block is denoted as ω_b.

The time transferring data from a disk block to buffer memory is constant [12]. When executing a query, query efficiency is high if the data stored in a block are all required. On the contrary, when there are a few related data and massive irrelevant data in a block, query efficiency reduces largely. For tenant data stored in universal table, i.e., a block may store data of multiple tenants, since there is no cross-tenant query, query efficiency will obviously be low.

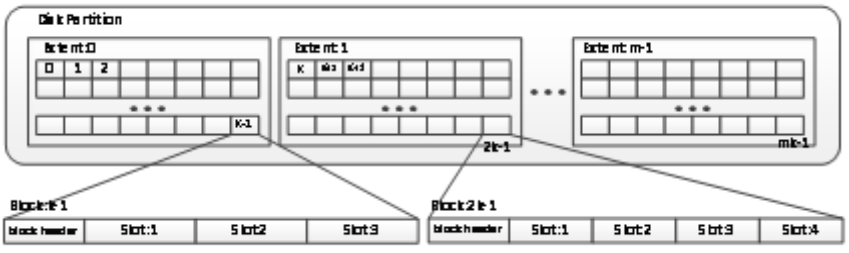

Fig. 2. Disk storage unit

Definition 2. Several blocks make an **extent**. The number of blocks in an extent is denoted k. Each extent has the same size. In Fig. 2, blocks make an extent. A disk partition has m extents, and then *mk* blocks. Assuming *e* is the subscript of extent, the size of extent is denoted as ω_e and $\omega_e = k\omega_b$.

The time reading data from disk consists of seek time and data transfer time. Data access efficiency will increase if once seeking can read more contiguous blocks [9]. Thus tenant data are stored to an extent, which is composed of contiguous blocks.

Definition 3. In a disk block, the remaining space except block head is divided into several smaller storage spaces. These further divided spaces is called **slot**. Assume the subscript of slot is denoted as s.

A data tuple of a tenant is stored to a slot. It is different from the disk storage method of universal table, because universal table has to reserve space for null value. In the paper we set slot size according to the size of tuple. Assuming that a block stores data with the same database schema, the size of tuple is the same and then the size of slot is the same as well. As shown in Fig. 2, the $(k-1)$-th and the $(2k-1)$-th block can respectively be divided into three slots and four slots according to the demands of tenant tuples for storage space. For tenant t, the slot size of data object o is denoted as $\omega_{t,o,s}$, and the size of block head is denoted as ω_h.

3.2. Single data object

To describe conveniently the process of address mapping, we first assume each tenant in multi-tenant database has single data object and different tenants have different database schema.

In multi-tenant database, data volumes of different tenants are heterogeneous and the volume increases gradually. Moreover, the number of tenants ascends dynamically. Multi-tenant database needs to provide data storage service for newly arrived tenants. In this case, in order to avoid the waster of storage space, we need to allocate disk space for tenants according to their data volume.

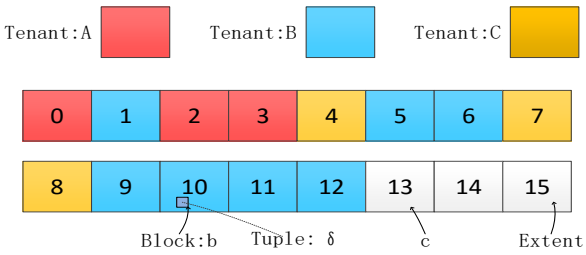

Fig. 3. Multi-tenant storage space allocation in disk partition

We propose an Isolated Storage Approach (ISA) based on SDSS, which allocates storage space for tenants in an incremental way. The storage space of a tenant consists of several contiguous extents. When the storage space of a tenant is insufficient, other contiguous extents are assigned and the allocated size doubles the last allocated one. The allocation details are as follows.

1. For a tenant, when it first stores data, a disk extent is assigned.

2. With the increase of tenant data, the storage space is insufficient. The storage space for the tenant needs to be allocated more space. The number of

allocated extents doubles the last time and the extents are adjacent. The allocated storage space is called Contiguous Extent Space (CES). When we allocate CES for the tenant the *j*-th time, the number of allocated extents is 2^{j-1}.

3. The extents are allocated in order. To facilitate the allocation of CES for newly arrived tenants, the cursor c is used to record next assignable extent. In Fig. 3, each colour represents a tenant and there are three tenants who are assigned storage space. The CES allocated to each tenant gradually increases. For example, blue tenant B is assigned extent three times and the three CESs are extent 1, extents 5, 6, and extents from 9 up to 12. The current c value is 13.

4. In order to compute physical address of a tuple, the starting extent number of the assigned CES needs recording. All the starting extent numbers of CES assigned to tenant t are recorded in a collection, which is denoted as $x_t = \{x_j \mid j \in 0, 1, 2, \cdots, n\}$, where x_j is the starting extent number of the *j*-th time assigned CES. When a new CES is allocated to tenant t, a new record is added to the collection x_t. For example, the collection x_B of tenant *B* is denoted as $x_B = \{1, 5, 9\}$.

The time complexity of space allocation is $O(1)$.

When tenant t has a single data object o, the physical address of its tuple δ is computed as follows.

Step 1. Compute how many tuples in data object o can each block store, i.e., the number of slots that a block is divided into, which is denoted as $n_{t,o,b}$. It is calculated as $n_{t,o,b} = (\omega_b - \omega_h) / \omega_{t,o,s}$. Then the number of slots in an extent is $n_{t,o,e} = kn_{t,o,b}$.

Step 2. Compute the sequence number of CES where tuple δ is stored, i.e., $j = \lfloor \log(\delta / n_{t,o,e} + 1) \rfloor$. Then we know tuple δ should be stored in the CES whose starting number is x_j. In Fig. 3, tuple δ is stored in the third CES of tenant *B*.

Step 3. Compute block offset in the *j*-th extent of tuple, i.e., which block of the extent space with starting number x_j tuple δ should be stored. The block number is denoted as $\Delta = \lfloor (\delta - (2^j - 1)n_{t,o,e}) / n_{t,o,b} \rfloor$. In Fig. 3, the block offset of block b in the third blue extent space needs computing for tuple.

Step 4. So far the sequence number of block b for tuple is obtained by $b = x_j k + \Delta$.

Step 5. Compute slot number where tuple is stored as $s = \delta \% n_{t,o,b}$, where the sign % denotes the modulus operator and tries to obtain the remainder of integer division.

Step 6. Thus the physical address of tuple in disk is

$$p = \mathrm{SA} + b\omega_b + \omega_h + \omega_{t,o,s} s . \quad (1)$$

We next give a case study of computing the physical address of tuple $\delta = 401$ for tenant *B*. Assuming $k = 10$, $\omega_b = 11$, $\omega_h = 1$, $\omega_{t,o,s} = 1$, we get

$n_{t,o,b} = (11-1)/1 = 10$ and $n_{t,o,e} = 10 n_{t,o,b} = 100$. Go to Step 2 and $j = \lfloor \log(401/100+1) \rfloor = 2$ stands. From Step 3 we obtain $\Delta = \lfloor (401-(2^2-1)\times 100)/10 \rfloor = 10$. Seen from Fig. 3 we know $x_3 = 9$, and $b = 9\times 10+10 = 100$ holds. $s = 401\%10 = 1$ is computed in Step 5. The start address is set as SA = 10,000, we compute $p = 10,000+100\times 11+1+1\times 1 = 11,102$. Therefore, the physical address of tuple δ is 11,102.

There is no loop during computing the physical address of a tuple, thus its time complexity is $O(1)$.

When reading tuple δ, its physical address can easily be obtained. However, when the newly added tuple δ is stored to disk, it is required to first determine whether there is free allocated space for current tenant t. If there is no space, a new extent space needs allocating. Next we give the process of new tuple δ stored to disk for tenant t.

Step 1. First determine where there is free storage space for current tenant t. If $\delta > 2^{|X_t|} n_{t,o,e} - 1$ stands, new storage space needs to be allocated to tenant *t*; else go to Step 3.

Step 2. Allocate the (*j* + 1)-th CES to tenant *t*. The CES have 2^j extents. The starting extent address is $x_{j+1} = c$ and $X_t = X_t \bigcup x_{j+1}$. The cursor *c* is denoted as $c = c + 2^j$.

Step 3. Obtain the physical address of tuple according to Equation (1) and store it.

There is no loop in the above steps, therefore the time complexity of storing a new tuple to disk is $O(1)$.

For improving query performance, the collection X_t of tenant resides constantly in the memory when the tenant is active. The space complexity of X_t is $O(|X_t|)$, i.e., the used space is logarithmic function of the number of extents assigned to tenant *t*. To solve the heterogeneity of data volumes for different tenants, our isolated storage approach dynamically allocates space in an incremental way. This method not only improves space usage efficiency for the tenants who have small amounts of data, but also satisfies the demand for storage space with fewer allocation times for the tenants who have large data volume.

3.3. Multiple data objects

In multi-tenant database, a tenant often has multiple data objects. If a block stores data of different data objects, it is inconvenient to do the statistics on database information such as disk block-level sampling. Moreover, the efficiency of range query will reduce because of dispersed storage. Thus we intensively store data of one data object to contiguous disk blocks.

In the scenario of multiple data objects, ISA stores the tuples belonging to a data object to contiguous disk block through twice mapping process. It consists of two steps.

Step 1. It builds a contiguous virtual tenant space for each tenant. For a tenant, tuples of all data objects are mapped to its virtual tenant space.

Step 2. According to the relative location of tuples in virtual tenant space and the distribution of virtual tenant space in disk, it gets the physical address of tuples.

In Fig. 4, there are four data objects for tenant *B*. It allocates isolated storage space for all data objects in virtual tenant space in an incremental way, and then stores tuples to physical disk partition according to its relative location in virtual tenant space, which can achieve isolated storage on the granularity of tenant.

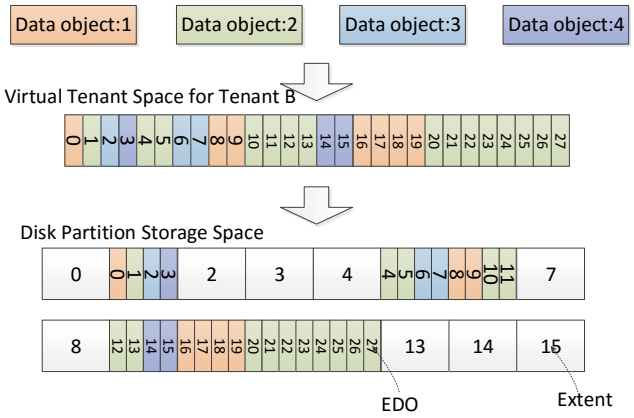

Fig. 4. Mapping data objects in disk storage space

Next we give a detailed introduction on the mapping process. In order to map a tenant's data objects to a virtual tenant space, each extent is divided into smaller area, which is called Extent of Data Object (EDO). In Fig. 4, one extent is divided into four EDOs. A virtual tenant space is composed of several EDOs, and each EDO stores the tuples belonging to a data object. We use d to denote the subscript of EDO and its starting number in each virtual tenant space is 0. EDO is also composed of several blocks. Assuming k' denotes the number of blocks in a EDO and p is the number of EDOs in an extent, the number of blocks in an extent $k = pk'$. For a given tenant t and a data object, the number of tuples that a EDO can store is denoted as $n_{t,o,d}$, and then $n_{t,o,d} = k' n_{t,o,b}$, where $n_{t,o,d}$ is the number of slots in a block.

In virtual tenant space, the storage space allocating steps for data objects are as follows.

Step 1. For a data object o, when data are first stored, an EDO is allocated to store an object data.

Step 2. With the increasing data volume of data object o, storage space needs reallocating when the space is insufficient for data object o. The number of EDOs

each time allocated is twice the one last time and the allocated space is adjacent and contiguous EDO space, which is called contiguous EDO space, i.e., when contiguous EDO space is allocated for the *i*-th time, the number of EDOs is 2^{i-1}.

Step 3. EDO is allocated in sequence. To facilitate the allocation of storage space for subsequent data objects, the starting number of EDO of a contiguous EDO space needs recording. For tenant t and data object o, the starting numbers of all the allocated contiguous EDO spaces form a set. The set is denoted as $Y_{t,o} = \{y_i \mid i \in 0, 1, 2, \cdots, n\}$, where y_i is the starting number of the *i*-th time allocated contiguous EDO space. When allocating once contiguous EDO space for data object, a new record is added to the set $Y_{t,o}$. In Fig. 4, the starting number set of data object 2 is $Y_{t,o} = \{1, 4, 10, 20\}$.

Step 4. If virtual tenant space of tenant t runs out, a new tenant storage space needs to be allocated in disk partition using the approach in Section 4.2.

The time complexity of allocating storage space for a data object is $O(1)$.

Then we introduce the computation of physical address for a tuple. For a given tenant *t*, data object o and tuple , the general idea is to first compute the relative block number and slot number of tuple in virtual tenant space, and then compute the actual block number in disk partition. Next we give the concrete steps.

Step 1. In tenant *t*'s virtual tenant space, the number of EDO which tuple is allocated to should be computed, i.e., $i = \lfloor \log(\delta / n_{t,o,d} + 1) \rfloor$.

Step 2. Compute the relative block number in the i th contiguous EDO space which tuple should be stored in, i.e., $\Delta' = \lfloor (\delta - (2^i - 1) n_{t,o,d}) / n_{t,o,b} \rfloor$.

Step 3. Compute the virtual block number of tuple in virtual tenant space, i.e., $b' = y_i k' + \Delta'$.

Step 4. Compute the slot number of tuple in the block, i.e., $s = \delta \% n_{t,o,b}$.

Step 5. Compute which contiguous extent space of disk partition virtual block b' is mapped to, i.e., $j = \lfloor \log(b' / (n_{t,o,e} / n_{t,o,b}) + 1) \rfloor$.

Step 6. Compute the block offset number in the *j*-th contiguous extent space virtual block b' is mapped to, i.e., $\Delta = b' - (2^j - 1)k$.

Step 7. Compute the block number of disk partition the relative block b' is mapped to, i.e., $b = x_j k + \Delta$.

Step 8. Thus we get the physical address of tuple in disk, i.e., $p = \mathrm{SA} + b\omega_b + \omega_h + \omega_{t,o,s} s$.

Taking Fig. 4 as an example, we compute the physical address of tuple $\delta = 201$ for data object 2 of tenant *B*. Assuming $\omega_b = 11, \omega_h = 1, \omega_{t,o,s} = 1$, we get $n_{t,o,b} = (11 - 1) / 1 = 10$. If $k' = 2$ and $p = 4$ stand, we obtain $k = pk' = 8$ and $n_{t,o,d} = 2 \times 10 = 20$. In Step 1 $i = \lfloor \log(201 / 20 + 1) \rfloor = 3$ is computed. In Step 2 $\Delta' = \lfloor (201 - (2^3 - 1) \times 20) / 10 \rfloor = 6$ is obtained. In Step 3 we get $b' = y_i k' + \Delta' = 20 \times 2 + 6 = 46$. In Step 4 we know $s = 201 \% 10 = 1$. Similarly,

$j = \lfloor \log(46/(80/10)+1) \rfloor = 2$ and $\Delta = 46 - (2^2 - 1) \times 8 = 22$ are calculated in Step 5 and Step 6, respectively. From the figure we know that $x_j = 9$ and $b = 9 \times 8 + 22 = 94$ holds. Finally, we set the start address as SA = 10,000 and compute $p = 10,000 + 94 \times 11 + 1 + 1 \times 1 = 11,036$.

There is no loop in the above process, thus its time complexity is $O(1)$.

Our proposed isolated storage approach allocates contiguous storage space for data object. Its advantages are as follows: 1) each disk block only stores data belonging to one data object, which improves hit ratio of query and reduce the times of input/output; 2) based on ISA, we can do the statistics on the distribution of data records in disk blocks, which can provide quantified information for query optimization, and then the optimization model can be developed for tenants; 3) reading contiguous blocks from disk is highly efficient; traversal operation of a data object only need to orderly access several contiguous blocks, which improves traversal efficiency.

4. Performance evaluation

In this section, we test the proposed ISA of multi-tenant data. To evaluate our ISA, we use other two approaches as comparison objects. We first introduce their experimental setup, and then compare their efficiency on equality query, range query, join query and no indexing query.

4.1. Experimental setup

The experimental environment is 64-bit Ubuntu OS, 4 processors, 8G memory and 7200r disk. Next we introduce the experimental setup of three approaches, i.e., sequential storage approach, logically isolated storage approach and ISA.

ISA is developed based on the idea in Section 4.3. We use MySQL5.7 as data processing engine and develop storage engine based on Linux Kernel 2.6 [13] and InnoDB. Assuming there are more than 200 tenants and each tenant randomly generates 10^5 to 10^6 tuples, there are about 10^8 tuples altogether.

Sequential Storage Approach (SSA) is to disorderly store multi-tenant data, i.e., data of all tenants are cross-stored together. We build a universal table in MySQL based on SDSS to store multi-tenant data. Each attribute of universal table is set to VARCHAR type and InnoDB is used as storage engine. All the tuples of 200 tenants are stored to universal table in a random order.

Logically Isolated Storage Approach (LISA) isolates tenant data at the logical layer of universal table, which is an intuitive isolation. Before inserting data, each tenant should be allocated a contiguous extent and the values of tuples in the extent are null. When inserting tuples for data object, the tuples need storing to already assigned extent in updated way. For example, it pre-assigns tuples $\delta \in [0, 99]$ to store data of tenant A and tuples $\delta \in [100, 199]$ to store data of tenant *B*. Thus a contiguous extent just stores data of a tenant and achieves the isolated storage. To

make sure the tuples are inserted into accurate location, the corresponding relation between each contiguous extent and tenant data object is recorded.

4.2. Data access performance

In this section, we test the performance on equality query, range query, join query and no indexing query for three approaches.

4.2.1. Equality query

Equality query is to search the tuples that a certain attribute equals some value from data object. Since multi-tenant database is tenant-shared, multiple queries will be processed concurrently. We measure the time cost and throughput when 2-128 tenants concurrently issue an equality query. Its query result is shown in Fig. 5. For equality query, the time costs have little difference for three approaches, since equality query just needs accessing a block to finish the query and the time of reading a block is basically the same. As for throughput, LISA and ISA have higher throughput when the number of tenants is small. That is because tenant data are stored mainly in buffer memory, the hit ratio is high, and then the average query time is short.

4.2.2. Range query and join query

Range query is to obtain a result set given the maximum and minimum of an attribute. About join query, we focus on equal join query of two data objects.

The time cost and throughput of range query and join query are shown as Fig. 6 and Fig. 7. ISA has obvious advantages than SSA in time cost. Since data of a data object is stored to fewer blocks, ISA just needs to read fewer blocks in disk to finish the query. As for throughput, it is similar to equality query. When the number of concurrent tenants is small, the hit ratio of buffer memory is high. Thus ISA has higher throughput for range query and join query. From the figures, we see that the time cost and throughput is similar for LISA and ISA. However, the time cost of LISA is still high, because it has low efficiency of storage space.

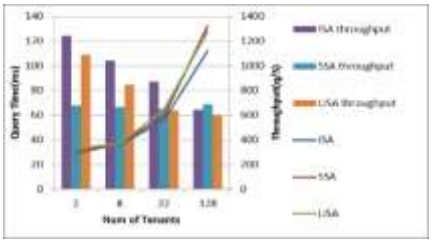

Fig. 5. The comparison of equality query on time and throughput

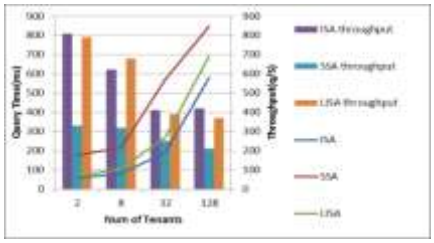

Fig. 6. The comparison of range query on time and throughput

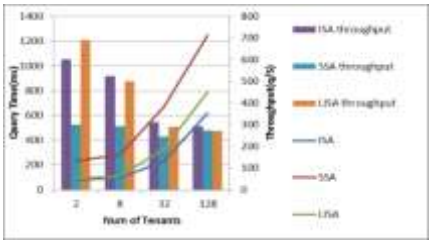

Fig.7. The comparison of join query on time and throughput

4.2.3. No indexing query

In data analysis, query will use various attributes to select data and indexes on some attributes are not built. In the experiment, we measure the performance of three approaches in the case of no indexes. No indexing query uses the query statement of equality query but removes the index. To be fair, we build a composite index for tenant identifier and data object identifier in SSA, which avoids a full table scan. We respectively generate data of 2, 8, 32 tenants and each tenant has tuples. Each time it issues a no indexing query. Fig. 8 shows that query efficiency of ISA is higher than SSA, because the target data object is stored in many contiguous blocks in ISA, which disk access is efficient.

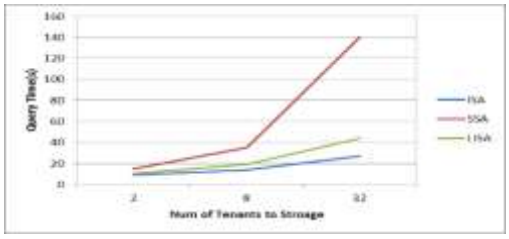

Fig. 8. The comparison of no indexing query on time cost

5. Conclusions

In the paper we propose an isolated storage approach of multi-tenant data based on SDSS. It aims to address the problem that disk blocks store massive irrelative data and tries to adaptively allocate contiguous storage space for tenants. The extensive experiments demonstrate that ISA performs better on range query, join query and no indexing scanning operation. It provides an efficient and effective storage means at the layer of disk for multi-tenant database based on SDSS. Since multi-tenant data are deployed in cloud computing environment, which is low-cost and easily extensible, the method could substantially improve query performance in this scenario.

***Acknowledgments*:** This work is partially supported by the NSFC No 61303085, 61572295; Innovation Method Fund of China No 2015IM010200; SDNFSC No ZR2013FQ014, ZR2014FM031; Science and Technology Development Plan Project of Shandong Province No 2014GGX101047; Shandong Province Independent Innovation Major Special Project No 2015ZDJQ01002, 2015ZDXX0201B03.

References

1. Weissman, C. The Design of the force.com Multitenant Internet Application Development Platform. – In: Proc. of SIGMOD-PODS'09, Providence, RI, United States, 2009, pp. 889-896.
2. Chong, F. Multi-Tenant Data Architecture. **http://msdn.microsoft.com/en-us/library/aa479086.aspx, 2015**
3. Aulbach, S., et al. A Comparison of Flexible Schemas for Software as a Service. – SIGMOD-PODS'09, United States, 2009, pp. 881-888.
4. Garcia-Molina, H. et al. Database System Implementation. NJ, Prentice Hall, Upper Saddle River, 2000. 654 p.
5. Aulbach, S., et al., Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. – SIGMOD'08, Canada, 2008, pp. 1195-1206.
6. Bannon, R., J. D. Ulman, J. Widom. Innodb Concrete Architecture. University of Waterloo, 2002.
7. Reid, M. InnoDB Quick Reference Guide. Packt Publishing, Ltd, 2015.
8. Pathirage, M., et al. A Multi-Tenant Architecture for Business Process Executions. pp. 121-128.
9. Silberschatz, A., et al. Database System Concepts. Vol. **4**. McGraw-Hill, New York, 1997.
10. Graefe, G., L. D. Shapiro. Data Compression and Database Performance. – In: Proc. of 1991 Symposium on Applied Computing, 1991, pp. 22-27.
11. Narasayya, V., et al. Sharing Buffer Pool Memory in Multi-Tenant Relational Database-as-a-Service. – The Very Large Data Base Endowment, 2015, pp. 726-737.
12. Gait, J. The Optical File Cabinet: A Random-Access File System for Write-Once Optical Disks. – Computer, Vol. **21**, 1988, pp. 11-22.
13. Bovet, D. Understanding the Linux Kernel. O'Reilly Media, Inc., 2005.