SUPPORTING DATABASE APPLICATIONS AS A SERVICE

ZHOU YUAN

Bachelor of Engineering

East China Normal University, China

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2010

Acknowledgement

I would like to express my deep and sincere gratitude to my supervisor, Prof. Ooi Beng Chin. I am grateful for his patient and invaluable support. His wide knowledge and his conscientious attitude of working set me a good example. His understanding and guidance have provided a good basis of my thesis. I would like to thank Hui mei, Jiang Dawei and Li Guoliang. I really appreciate the help they gave me during the work. Their enthusiasm in research have encouraged me a lot.

I also wish to thank my co-workers in the Database Lab who deserve my warmest thanks for our many discussions and their friendship. They are Chen Yueguo, Yang Xiaoyan, Zhang Zhenjie, Chen Su, Wu Sai, Vohoang Tam, Liu Xuan, Zhang Meihui,Lin Yuting, etc. I really enjoyed the pleasant stay with these brilliant people.

Finally, I would like to thank my parents for their endless love and support.

CONTENTS

Summary

With the shift in outsourcing the management and maintenance of database applications, multi-tenancy has become one of the most active and exciting research areas. 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 is the scalability issue, namely the ability to serve an increasing number of tenants without significant query performance degradation. In this thesis, various solutions will be investigated to address this problem. First, three potential architectures are examined to give a good insight into the design of multi-tenant database system. They are *Independent Database and Independent Database Instances* (IDII), *Independent Tables and Shared Database Instances* (ITSI), and *Shared Table and Shared Database Instances* (STSI). All these approaches have some fundamental limitations in supporting multi-tenant database systems, which motivate us to develop an entirely new architecture to effectively and efficiently resolve the problem.

Based on the study of the previous work, we found that a promising way to

handle the scalability issue is to consolidate tuples from different tenants into the same shared tables (STSI). But this approach introduces two problems: 1. the shared tables are too sparse; 2. indexing on shared tables is not effective. In this thesis, we examine these two problems and develop efficient approaches for them.

In particular, we design a multi-tenant databases 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). The former uses a bitmap string to store and retrieve data, while the latter adopts a multi-separated indexing method to improve the query efficiency. M-Store is efficient and flexible because: 1) it does not store NULLs from unused attributes in the shared tables. 2) it only indexes each tenant's own data on frequent accessed attributes. Cost model and experimental studies demonstrate that the proposed approach is a promising multi-tenancy storage and indexing scheme which can be easily integrated into the existing database management systems.

In summary, this thesis proposes techniques of data storage and query processing for Multi-tenant database systems. Through an extensive performance study, the proposed solutions are shown to be efficient and easy to implement, and should be helpful for the subsequent research.

LIST OF FIGURES

CHAPTER 1

Introduction

To reduce the burden of deploying and maintaining software and hardware infrastructures, there is an increasing interest in the use of third-party services, which provide computation power, data storage, and network service to the businesses. This kind of application is called Software as a Service (SaaS) [37, 49, 67]. In contrast to the traditional on-premise software, SaaS shifts the ownership of the software from customers to the external service provider, which results in the reallocation of the responsibility for the infrastructures and professional services.

Generally Speaking, there are three key attributes that determine the maturity of SaaS, which are scalability, multi-tenant efficiency, and configurability. According to Microsoft MSDN[4], SaaS application maturity can be classified into four levels in terms of these attributes.

1. *Ad Hoc/Custom.*

   At this level, each customer has its own customized version of the hosted application, and runs its own instance of the application on the host's servers.

Software at this maturity level is very similar to the traditional client-server application, therefore it requires least development effort and operating costs to migrate those on-premise software to the SaaS model.

2. *Configurable.*

   At the second level, service provider hosts separate instance of the application for each customer. Different from Level 1, all the instances use the same code implementation here, and the vendor provides detailed configuration options to satisfy the customers' needs. This approach greatly reduces the maintenance cost of SaaS application, however it will require more re-architecting than at the first level.

3. *Configurable, Multi-Tenant-Efficient.*

   At the third level of maturity, service provider maintains a single instance for multiple customers. This approach eliminates the need to provide server space for multiple instances, and enables more efficient use of computing resources. The main disadvantage of this method is the scalability problem: with the number of customers increasing, it is difficult for the database management system to scale up well.

4. *Scalable, Configurable, Multi-Tenant-Efficient.*

   Based on the characteristics of the above three maturity levels, the fourth level requires the system to provide the scalability feature. At this level, service provider hosts multiple customers on a load-balanced farm of identical instances, the scalability can be achieved in that the number of servers and instances on the back end can be increased or decreased as necessary to match demand.

Based on the consideration of four maturity levels, in order to host database-

driven applications as SaaS in cost-efficient manner, service providers can design and build a Multi-tenant Database System[13]. In this system, 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 some standard method, e.g., Web Service. All the maintenance costs are transferred from the tenant to the service provider. Fig.1.1 shows the high level overview of Multi-tenant Database System. This system sharply contrasts to the traditional *in-host* database system in which a tenant purchases a data center and applications and operates them itself. Applications of Multi-Tenant Database System include Customer Relationship Management(CRM), Human Capital Management(HCM), Supplier Relationship Management(SRM), and Business Intelligence (BI).

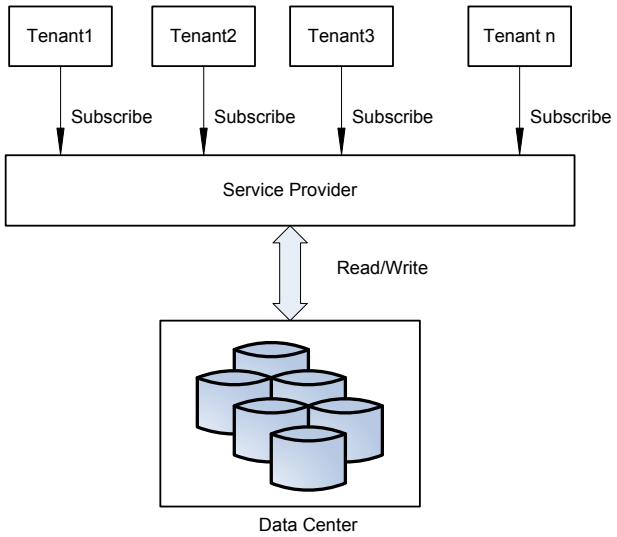

Figure 1.1: The high-level overview of "Multi-tenant Database System"

Intuitively speaking, Multi-tenant database systems have advantages in the following aspects. A database service provider has the advantage of expertise consol-

idation, making database management significantly more affordable for organizations with less experience, resources or trained manpower, such as small companies or individuals. Even for bigger organizations that can afford the traditional approach of buying the necessary hardware, deploying database products, setting up network connectivity, and hiring professionals to run the system, the option is also becoming increasingly expensive and impractical as databases become larger and more complex, and the corresponding queries are increasingly complicated.

One of the most important value of multi-tenancy is that it can help a service provider catch "*long tail* " markets [4]. Multi-tenant database systems save not only capital expenditures but also operational costs such as cost for people and power. 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 the traditional and possibly more expensive on-premise solutions. As revealed in [4, 11], 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 [72].

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 [55, 68].

1.1 Motivation

In this thesis, we argue that the *scalability* issue, which refers to as the ability to serve an increasing number of tenants without significant query performance degradation, deserves more attention in the building of a multi-tenant database system. The reason is simple. The core value of multi-tenancy is to catch the long tail. This is achieved by consolidating data from tenants to the hosted database to reduce the per-tenant service cost. Therefore, 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 practices show that consolidating too much data from different tenants will definitely degrade query performance [30]. 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.

As we mentioned above, multi-tenancy is one of the key attributes that determine the SaaS application maturity. To make the SaaS applications configurable and multi-tenant-efficient, there are three approaches to build 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., a MySQL or DB2 database processes to serve different tenants. The tenant stores and queries data in its dedicated database. This approach makes it easy for tenants to extend the applications to meet their individual needs, and restoring tenants' data from backups in the event of failure is relatively simple. It also offers good data isolation and security. However,

in IDII, the scalability is rather poor since running independent database instances wastes memory and CPU cycles. 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 build 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. Compared to IDII, ITSI is relatively easy to implement and in the meantime, it offers a moderate degree of logical data isolation. ITSI removes the huge maintenance cost incurred by IDII. But 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. 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 [30]. Finally, a significant drawback of ITSI is that tenant data is very difficult to restore in case of system failure. With the independent table solution, restoring database need to overwriting all tenants' data in this database even if many of them have no data loss.

- The third approach is Shared Tables and Shared Database Instances (STSI). Using STSI, tenants not only share database instance but also share tables. The tenants store their tuples to the shared tables by appending each tu-

ple with a `TenantID`, that indicates which tenant the tuple belongs to, and setting unused attributes to NULL. Queries are reformulated to take into account `TenantID` so that correct answers can be found. Details of STSI will be presented in the subsequent chapters. Compared to the above two approaches, STSI can achieve the best scalability since the number of tables is determined by the base schema and therefore is independent of the number of the 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, causing 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 affect 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 thesis, a novel multi-tenant database system, *M-Store*, is implemented. *M-Store* is built as a storage engine for MySQL to provide storage and indexing service for multiple tenants. *M-Store* adopts STSI approach to achieve excellent scalability. To overcome the drawback of STSI, two techniques are proposed. 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 database. BIT format greatly reduces the overhead of storing NULLs in the

shared table. 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 belonging to that tenant.

1.2 Contribution

This thesis examines the scalability issues in multi-tenant database system. The main contributions are summarized as follows:

- A novel multi-tenancy storage technique *BIT* is proposed. *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 [28] and interpreted fields [32], *BIT* does not introduce overhead for NULLs compression and tuples reconstruction.

- To improve the query performance, Multi-Separated Index (*MSI*) scheme is introduced. 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.

- Based on the cost analysis of proposed *BIT* and *MSI* techniques, a scalable and configurable multi-tenant database system, *M-Store*, is developed. The

M-Store system is a pluggable storage engine for MySQL which offers storage and indexing services for multi-tenant databases. *M-Store* adopts *BIT* and *MSI* techniques. The implementation of *M-Store* shows that the proposed techniques in this thesis are ready for use and can be easily grafted into an existing database management system.

- Extensive experimental study of the proposed approaches is carried out in multi-tenant environment. Three parts of experiments examine the different aspects of system scalability. The results show that the *M-Store* system is a highly scalable multi-tenant database system, and the proposed *BIT* and *MSI* solutions are promising multi-tenancy storage and indexing schemes.

Overall, our proposed approaches provide an effective and efficient framework for the scalability issue in multi-tenant database system, since they greatly improve the performance of query processing in the event of serving a huge amount of tenants, and significantly reduce the expenditure of data storage.

1.3 Organization of Thesis

The rest of the thesis is organized as follows:

- Chapter 2 introduces the related work and reviews the existing storage and query processing methods.

- Chapter 3 outlines the multi-tenant database system and discusses three possible solutions: Independent Databases and Independent Database Instances(IDII), Independent Tables and Shared Datbase Instances(ITSI) and Shared Tables and Shared Database Instances(STSI).

- Chapter 4 presents the proposed Multi-tenant database system: M-Store. Two techniques are applied in this model: Bitmap Interpreted Tuple Format and Multi-Separated Indexing Scheme. Cost model is given to analyze the efficiency of the proposed techniques.

- Chapter 5 empirically evaluates the scalability of the M-Store system. Experimental results indicate that the proposed approaches can significantly reduce the disk space usage and improves index lookup speed, thus provide a highly scalable solution to the application of multi-tenant database system.

- Chapter 6 concludes the work in this thesis with a summary of our main findings. We also discuss some limitations and indicate directions for future work.

CHAPTER 2

Literature Review

There have been research works for designing a system which provides database as a service. NetDB2[49] offers mechanisms for organizations to create and access their databases at the host site managed by the third party service provider. PNUTS[19, 40], a hosted data serving platform which is designed for various Yahoo!'s web applications, focuses on providing low latency for concurrent requests by the use of massive servers. SHAROES[67], a system which delivers raw storage as a service over a network, focuses on delivering a secure raw storage service without consideration on the data model and indexing. Bigtable[38], a structured data storage infrastructure for Google's products, employs a sorted data map with uninterpreted strings to provide storage services to different applications. Other systems such as Amazon S3[1], SimpleDB[2] and Microsoft's CloudDB[5] all provide such outsourcing services.

Although the service provider expects to provide highly scalable, reliable, fast and inexpensive data services, outsourcing database as a service poses great challenges on both data storage and query processing in many aspects. One of the main problems is the sparse data sets. A sparse data set typically consists of hundreds or even thousands of different attributes, while most of the records are filled with

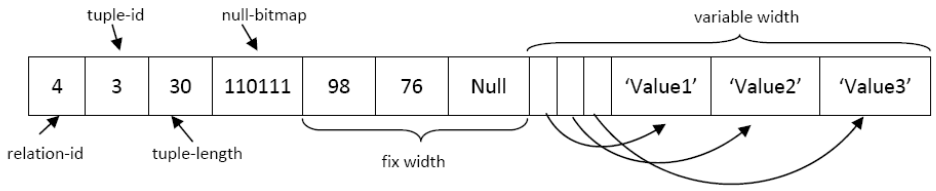

Figure 2.1: Positional Storage Format

non-null values in a small fraction of attributes. Sparse data can arise from many sources, including e-commerce applications[6, 28], medical information systems[36], distributed systems[63, 64] and even information extraction systems[27], therefore providing efficient support for such sparse data has become an important research problem. This chapter will review approaches developed for handling sparse data, including data storage methods as well as techniques for query construction and evaluation over sparse tables.

2.1 Row Oriented Storage

2.1.1 Positional Storage Format

Most commercial RDBMS adopt a positional storage format [48, 61] for their records. The positional storage format defines a tuple in the following way (Figure 2.1): the layout of the tuple begins with a tuple header, which stores the relation-id, tuple-id, and the tuple length. Next is the null-bitmap, indicating the fields with null values. Following the null-bitmap field is the fixed width data, whose storage space are pre-allocated by the system, regardless of the null values. Finally, there is an array of variable width offsets which point to and precede the variable width data. The system catalog maintains the mapping from attribute name to value within a tuple by recording the order of the attributes in the tuple.

This approach is effective for dense data and enables fast access to the values of the attributes. But it faces with a big challenge when handling the sparse data

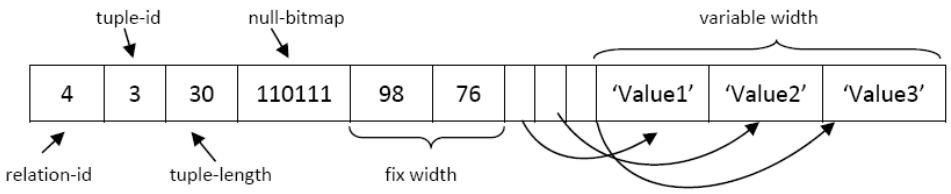

Figure 2.2: PostgreSQL Bitmap-Only Format

sets. In the positional storage format, a null value for a fixed-width attribute takes one bit in the null-bitmap and the full size of the attribute; a null value for variable-width attribute takes a bit in null-bitmap as well as a pointer in the record header. Therefore, the large amount of null values in the sparse data sets occupy and waste vast valuable storage space.

2.1.2 PsotgreSQL Bitmap-Only Format

The storage strategy for PostgreSQL is the bitmap-only format[14]. The tuple header in this storage layout contains the same information as the positional storage format. It also has a null-bit map field which indicates the null fields. Different from traditional positional format, bitmap-only format does not pre-allocate the space for the null values (Figure 2.2).

This method attempts to save the space by eliminating the pre-allocated space for the null attributes. However, the retrieval of a value for bitmap-only format is complex. To retrieve a non-null attribute, it is necessary to know the data-lengths of all non-null fields in the prior n-1 attributes of the record, as well as the information from the system catalog containing the information on the length of non-null attributes and use the aggregate of their sizes to locate the position.

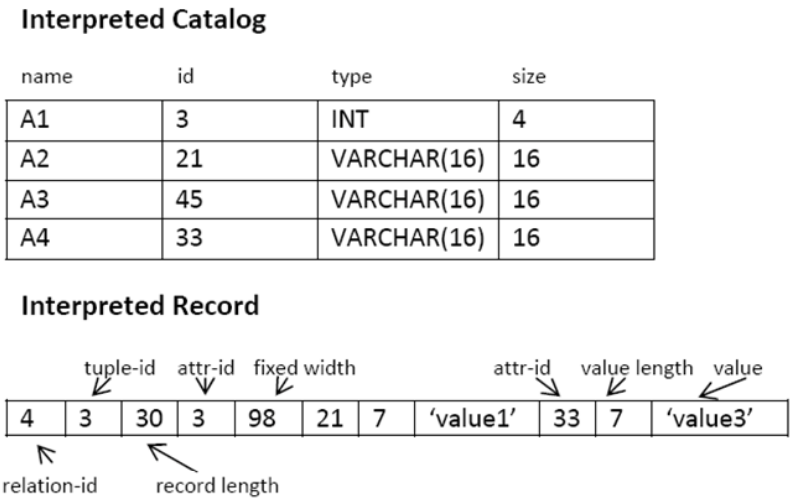

Figure 2.3: Interpreted record layout and corresponding catalog information (taken from [32])

2.1.3 Interpreted Storage Format

Interpreted storage format was introduced in [32] to avoid the problem of storing nulls in sparse datasets. To interpret a tuple, the system maintains an interpreted catalog, which records each attribute's name, id , type, and attribute size. For each tuple, it starts from storing the relational-id, tuple-id, and record length. For each non-null attribute, the tuple contains its attribute-id, length, and value. For any attribute appearing in the interpreted catalog but not in the tuple, it is straightforward to know that they have the null value. Figure 2.3 shows a representative interpreted record layout and the corresponding catalog information.

By using the interpreted format, sparse datasets with a large number of null values can be stored in a much more compact manner. Given the condition that some attributes are sparse while others are dense, it is appropriate to use positional approach to store the dense attributes in a horizontal table. Then interpreted storage format can be applied to store the sparse attributes.

The interpreted format can also be viewed as an optimization of the vertical

storage approach[28]. Both of the formats store the "attribute, value" pairs, but interpreted layout differs from vertical storage in the following aspects. First, in interpreted format, all the pairs are viewed as a single object so there is no need to combine them with a tuple id or reconstruct the tuple during query evaluation. Second, the attributes are collected as one object, while the entity is a set of independent tuples in the vertical schema. Third, the interpreted catalog records the attribute names, whereas in the vertical format these names must be managed by the application. We will review details of vertical storage format in the subsequent section.

The disadvantage of interpreted schema is the complexity of retrieving values from attributes in the tuple, which means the nth attribute can only be found by scanning the whole tuple rather than jumping to it directly using the pre-compiled position information from the system catalog. This kind of value extraction is a potential expensive operation and reduces the system performance.

2.2 Column Oriented Storage

An alternative approach to row stores is column oriented storage format [20, 23], in which each attribute in a database table is stored separately, i.e., column-by-column. Recent years a number of column-oriented commercial products has been introduced, including MonetDB [12], Vertica [18], Sybase [57], and C-Store [70], etc. In this section, we review approaches developed for column storage format and explore the tradeoffs between row-store and column-store.

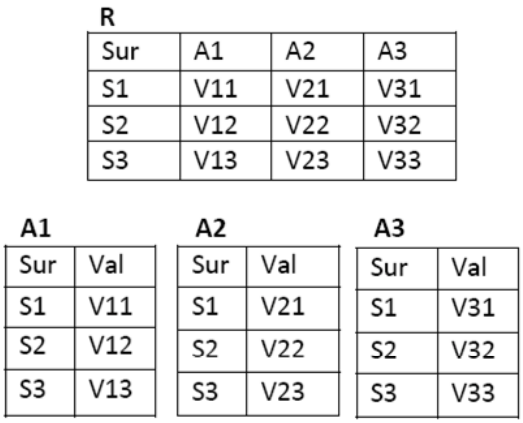

Figure 2.4: Decomposition Storage Model (taken from [41])

2.2.1 Decomposition Storage Format

One column based storage format for sparse data sets is Decomposed Storage Model (DSM) [41, 54]. In this approach, system decomposes the horizontal tables into many 2-ary relations, one for each column in the relation (Figure 2.4). In this way, DSM vertically decouples the logical and physical storage of entities. On advantage of DSM is that this method can reduce the overhead of space saving by eliminating null values in the horizontal table. Comparisons of DSM with horizontal storage over dense data have shown DSM to be more efficient for queries that use a small number of attributes. However, while there are applications that store data in a large number of tables, having thousands of decomposed tables makes the system harder to manage and maintain. In addition, DSM suffers from the expensive cost of reconstructing the fragments of the horizontal table when there are requests for several attributes.

DSM has been implemented in the Monet System [33] and been used in some commercial database products such as DB2[66]. Other decomposition storage approaches include creating one separate table for each category, creating one table for common attributes and per category separate tables for non-common attributes,

Horizontal (H)

Oid	A1	A2	A3
1	a	b	⊥
2	⊥	c	d
3	⊥	⊥	a
4	b	⊥	d

Vertical (V)

Oid	*Key*	*Val*
1	A1	a
1	A2	b
2	A2	c
2	A3	d
3	A3	a
4	A1	b
4	A3	d

Figure 2.5: Vertical Storage Format (taken from [28])

as well as the solution for storing XML data [45].

2.2.2 Vertical Storage Format

Similar to the decomposition storage format, R.Agrawal et.al[28] proposes a 3-ary vertical scheme to store the sparse tuples. In this vertical scheme, the pairs of attributes and non-null values of the sparse tuples are stored in the vertical table which contains the information on object-id, attribute name, and their values. For example, if the horizontal schema is $H(A1, A2, ..., An)$, the schema of the corresponding vertical format will be $H_v(Oid, Key, Val)$. A tuple $(V1, V2, ..., Vn)$ can be mapped into multiple rows in the vertical table: $(Oid, A1, V1)$, $(Oid, A2, V2)$, ..., (Oid, An, Vn). Figure 2.5 illustrates a simple horizontal and vertical table representation.

The difference between the vertical storage format and DSM is that, similar to the horizontal representation, the vertical representation takes only one table to store all data, whereas the binary representation in DSM splits the table into as many tables as the number of attributes. When there is a spare data set, managing thousands of tables becomes a bottleneck for data management. Another advantage of the vertical schema stems from the fact that vertical schema is efficient for schema evolution, while DSM incurs additional costs on adding and deleting a table. The

disadvantage of vertical schema is that no effective support is available to data typing because all the values are stored as VARCHARs in the *Val* field.

One major problem of such vertical schema is that simple queries over the horizontal schema are usually cumbersome. Figure 2.6 gives an example of the differences between the equivalent horizontal and vertical queries. Notice that simple projection and selection queries over a horizontal table are transformed into complex self join queries in order to match the predicate. More complicated condition happens when some of the database users expect the results of queries to be returned in standard horizontal form, while others prefer vertical format without so many null values. Therefore RDBMS is supposed to undertake extra processing to convert the tuples from one storage schema to another equivalent one, namely Vertical-to-Horizontal (V2H) Translation and Horizontal-to-Vertical (H2V) Translation[28].

V2H Translation

There are two main approaches to V2H translation, left-outer-join (LOJ)[28] and PIVOT [42]. LOJ takes a vertical view of the data and constructs an equivalent horizontal table by projecting each attribute separately from a vertical table and then joining all of the columns to construct a horizontal table. By using the *oid* in the vertical row, the join operation groups all the attributes spreading over multiple vertical tuples.

The formal description of V2H operation Ω(V) can be defined as[28]:

$$\Omega^k(V) = [\pi_{oid}(V)] \rtimes [\rtimes_{i=1}^{k} \pi_{oid,val}(\sigma_{key='Ai'}(V))]$$

Left outer join is key to constructing a horizontal row, since it not only returns tuples that match the predicate but also returns any non-matching rows as null values. Here is a simple example for the V2H transformation which converts a

Select	Project

Horizontal
 	
`SELECT *`
`FROM H`
`WHERE A1='v1'`
`AND A2='v2'`
`AND ...`
`AND Aj='vj'`
	Horizontal
	
	`SELECT A1, ..., Ak`
	`FROM H`

Vertical
 	
`SELECT *`
`FROM V`
`WHERE oid IN`
`(SELECT V1.oid`
`FROM V V1, ..., V Vj`
`WHERE (V1.attr = 'A1'`
`AND V1.value = 'v1')`
`AND ...`
`AND (Vj.attr = 'Aj'`
`AND Vj.value = 'vj')`
`AND (V1.oid = V2.oid`
`AND V1.oid = V3.oid`
`AND ...`
`AND V1.oid = Vj.oid))`
	Vertical
	
	`SELECT *`
	`FROM V`
	`WHERE attr = 'A1'`
	`OR attr = 'A2'`
	`OR ...`
	`OR attr = 'Ak'`

Figure 2.6: Select and project queries for horizontal and vertical (taken from [32])

vertical table into a corresponding horizontal one with two columns C1 and C2 using LOJ.

```
SELECT C1, C2
FROM
    (SELECT DISTINCT oid FROM V) AS t0
     LEFT OUTER JOIN
    (SELECT oid,val AS C1
     FROM V WHERE attr = 'C1') AS t1
     ON t0.oid = t1.oid
     LEFT OUTER JOIN
    (SELECT oid,val AS C2
     FROM V WHERE attr = 'C2') AS t2
```

```
    ON t0.oid = t2.oid
```

PIVOT[42] is an alternative to LOJ for V2H translation. In PIVOT, group-by and aggregation operations are used to produce horizontal tuples. For example, a PIVOT operator that produces a three column horizontal table *H(oid,C1,C2)* from a vertical schema is:

```
SELECT oid,
     MAX(CASE WHERE attr='C1' THEN val ELSE null) as C1,
     MAX(CASE WHERE attr='C2' THEN val Else null) as C2,
FROM V
GROUP BY oid
```

To handle the data collisions (two values map to the same location), the above PIVOT syntax uses the aggregate function (MAX()). Another possible solution is pre-defining a special constraint. Both approaches can preclude duplicates in the schema map. For missing values, PIVOT can use null values to satisfy this condition.

H2V Translation

In case that some applications prefer to handle results in a vertical format rather than the wide horizontal results with many null values, H2V operation[28] is proposed as the inverse of V2H, which translates a horizontal table with the schema (*Oid,A1,...,An*) into a vertical table (*Oid,Key,Val*). It is defined as the union of the projections of each attribute in a horizontal table. The formal description of V2H operation $\mho$(H) can be written as:

$$\mho^k(H) = [\cup_{i=1}^{k} \pi_{Oid,`Ai',Ai}(\sigma_{Ai \neq `\perp'}(H))] \cup [\cup_{i=1}^{k} \pi_{Oid,`Ai',Ai}(\sigma_{\wedge_{i=1}^{k} Ai=`\perp'}(H))]$$

The second term on the right hand side is the special case when a horizontal tuple has null values in all of the non-*Oid* columns. This operation is also referred to as UNPIVOT operator [42], which works inversely of what PIVOT operator does. H2V is useful when the user wants to hold the vertical result from the queries. Here is an example of a two column H2V translation:

```
SELECT oid,'A1',A1 FROM H WHERE A1 is not null
UNION ALL
SELECT oid,'A2',A2 FROM H WHERE A2 is not null
```

2.2.3 C-Store

In contrast to the most current database management systems (write-optimized), C-Store [70] is a read-optimized relational DBMS which keeps the data in a column storage format. At the top level of C-Store there is a small Writable Store (WS) component, which is designed to support high performance insertions and updates. Then there is a larger component, namely Read-optimized Store (RS), that is used to support very large amounts of information and optimized for read operations. Figure 2.7 shows the architecture of C-Store.

In C-Store, both RS and WS are column stores, therefore any segment of any projection is broken into its constituent columns, and each column is stored in order of sort key for the projection. Columns in RS are compressed using encoding schemes, where the encoding of column depends on its ordering and the proportion of distinct values it contains. Join indexes must also be used to connect the various projections anchored at the same table. Finally, there is a tuple mover, responsible for the movement of batched records from WS to RS by a merge-out process (MOP).

C-Store outperforms traditional row store databases in the following aspects: It stores each column of a relation separately and scans only a small fraction of

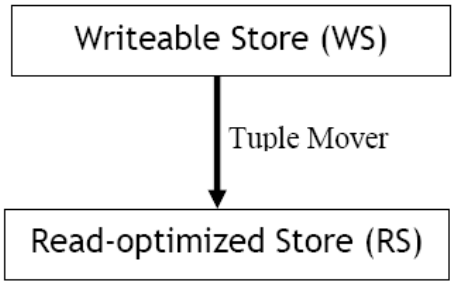

Figure 2.7: The architecture of C-Store (taken from [70])

columns that are relevant to the query. In addition, it packs column values into blocks and uses a combination of sorting and value compression techniques. All of the above features make C-Store greatly reduce disk storage requirements and dramatically improve the query performance.

2.2.4 Emulate Column database in Row Oriented DBMS

There are mainly three different approaches that are used to emulate a column-database design in a row oriented DBMS: The first method is *Vertical Partitioning*[15, 54]. This approach employs the method of decomposed storage format which is previously introduced. It creates one physical table for each column in the logical schema. The table contains two columns, storing the value of the column in the logical schema and the value of the 'position column' respectively. Queries are revised by performing joins on the position attribute. The major drawback of this method is that it requires the position attribute to be stored in each column, and row-store normally stores a relatively large header on each tuple, which wastes storage space and disk bandwidth. To alleviate this problem, Halverson et al.[50] proposed an optimization called "super tuples", which avoids duplicating header information and batches many tuples together in a block. The second approach is *index-only plans*, which stores tuples using a standard row-based design, but

adds a unclustered B^+-tree index on every column of every table. By creating a collection of indices that cover all of columns used in a query, it is possible for the database system to answer a query without going to the underlying tables. But the problem of this plan is that it may ask for some slow index scan if a column has no predicate on the index. This problem can be solved by creating the index with composite keys. The third approach is to build a set of *materialized views* for every query flight in the workload, where the optimal view for a given flight has only the columns needed to answer queries in that flight. More details on it will be provided in next section on query optimization.

2.2.5 Trade-Offs between Column-Store and Row-Store

Abadi concludes the trade-offs between column-stores and row-stores in [20]. There are several advantages for column-store. First, it improves the storage bandwidth utilization[54]. Only the attributes which are accessed by the query need to be read from the disk, whereas in row store, all surrounding attributes are also fetched. Second, column store utilizes the cache locality[29]. A cache line tends to contain irrelevant surrounding attributes in the row store, which wastes cache space. Third, it exploits code pipelining[33, 34]. The attribute data can be iterated directly without indirection through a tuple interface, resulting in high efficiency. Finally, it facilitates better data compression[24].

On the other hand, there are also some drawbacks existing in column store. It worsens the disk seek time since multiple columns are read in parallel. It also incurs higher costs on tuple reconstruction as well as insertion query. It is inefficient to transform the value from multiple columns into a row store tuple. When an insertion query is executed, the system has to update every attribute stored in the distinct locations, resulting in expensive costs.

2.3 Query Construction over Sparse Data

The main challenge on querying over sparse data is that the oversized number of attributes makes it difficult for the users to find the correct attribute. For example, there are about 5000 attributes in CNET[6] data sets, we cannot expect the user to specify the exact attribute, unless the users can remember all the attribute names, which are fairly infeasible. Even when some drop-down lists are provided for the users to select the desirable attributes, it is still difficult for them to locate the right one among thousands of selections. The use of keyword search for querying a structured database [46, 52, 56] is a nature solution because the users do not need to specify the attribute names, but its imprecise semantics is problematic when the keyword appears in multiple columns or rows, and it is inapplicable when users require range queries and aggregates. In such cases, the results of keyword search may contain many extraneous objects.

To alleviate this problem, E.Chu et al.[39] proposed a fuzzy attribute method: F_SQL, allowing users to make guesses about the names of attributes they want, and trying to find the matching attributes in the schema by using a name-based schema-matching technique[60]. For SQL query, the system replaces the fuzzy attributes with the matching attributes and re-execute the revised query. When there are several possible matches to a single fuzzy attribute, the system can either pick up the matching with the highest similarity score, or return all the matches exceeding some similarity threshold, whose query results can then be merged to get the final result. However, these two approaches may raise the problem when either the system chooses the incorrect attributes or the results deteriorate for low attribute selection precision. To improve the effectiveness of F_SQL, another method F_KS was introduced, which combines keyword search with fuzzy attributes. In this method, the system runs keyword search on the data value of fuzzy attributes

and performs name matching between fuzzy attributes and keyword search results. F_KS has advantages over F_SQL on the point that it matches the fuzzy attribute with only a number of attributes that contain the keyword. Moreover, it also improves the quality of the keyword search. But F_KS is less efficient, since an expensive keyword query is run first, and it does not apply for range queries.

In addition to F_SQL and F_KS, there is a complementary query-building technique[39], which tries to build an attribute directory or browsing-based interface on the hidden schema and helps the user to exploit appropriate attributes for writing structured queries. This approach is especially valuable for users without any idea about the schema or specific query.

2.4 Query Optimization over Sparse Data

2.4.1 Query Optimization over Row-Store

Wide sparse tables pose great challenges to query evaluation and optimization. Scans must process hundreds or even thousands of attributes in addition to the specified attributes in the query. Index is also a problem since the probability of having an index on a randomly chosen attribute in a query is very low. E.Chu et al.[39] exploits these problems with a *Sparse B-tree Index*, which maps only the non-null values to the object identifiers. The size of a sparse index is proportional to the number of rows that have a non-null value for that attribute. Therefore, it incurs much lower storage overhead and maintenance cost. To improve the efficiency of index construction, a bulk-loading technique called *scan-per-group* are adopted. This bulk loading method scans the table once per group of m indexes. This algorithm divides the buffer pool into m sections, each scan of table creates m indexes. By this way, the I/O cost and fetching cost are significantly reduced.

Besides creating sparse index, data partition is another option to avoid the complete scan for the entire sparse table. Using vertical partition is more efficient because there are fewer attributes to process. To achieve good partition quality, [39] suggests a *hidden schema* method, which automatically discovers groups of co-occurring attributes that have non-null values in the sparse table. This hidden schema is inferred via attribute clustering, where the Jaccard coefficient is used to measure the strength of co-occurrence between attributes and k-NN clustering algorithm is used to create the hidden schema. With this hidden schema, the table can be vertically partitioned into a couple of materialized views so that we can scan these views instead of the original table. As the partitions are relatively dense and narrow, storage overhead and query efficiency are both improved. Similar work is done by Edmonds et al. [43], which describes a scalable algorithm on finding empty rectangles in 2-dimensional data sets. With all null rows are omitted, the sparse table can achieve both vertical and horizontal partitioning and the cost of storage is greatly reduced.

Based on the concept of vertical partition, another query optimization approach was proposed in [50], which utilizes a "super tuple" to avoid duplicating per-tuple header information and batch tuples together in a block. This approach turns out to reduce the overheads of the vertically partitioned scheme and make a row store database competitive with a column store.

2.4.2 Query Optimization Over Column-Store

In this section, four common methods of optimization in column oriented database systems are reviewed. First is *Compression*[24]. Column store returns the data sets with low information entropy which can improve both the effectiveness and the efficiency of compression algorithm. In addition, compression is able to improve

the query performance, by reducing disk space and I/O. The second approach for query optimization is the *Late Materialization*[26, 34, 73]. Compared to the *early materialization* which constructs tuples from relevant attributes before query execution, most recent column-store systems choose to keep data in columns as late as possible in a query plan, and operate directly on these columns. Therefore, intermediate 'position' lists are constructed in order to match up corresponding operations performed on different columns. This list of positions can be represented as a simple array, which is a bit string or as a set of ranges on the positions. These position representations are then intersected to create a single position list and applied on value extraction. The third approach is *Block Iteration*[73]. In order to process tuples, row stores first iterates through each tuple, extracts the needed attributes form these tuples through a tuple representation interface[47]. In contrast to the row-store method, in all column stores the blocks of values from the same column are set to an operator in a single function call. The fourth approach is *Invisible Join*[25]. This approach can be used in column-oriented databases for foreign-key/primary-key joins on star schema style tables. It is also a late materialized join, but minimizes the position values that need to be extracted. By rewriting the joins into predicates on the foreign key columns, this approach can achieve great improvement on query performance.

2.5 Summary

Software as a Service(SaaS) brings great challenges to the database research. One of the main problems is the sparse data sets generated by consolidating different tenants' data on the host site. The sparse data sets typically have two characteristics: 1) large number of attributes 2)most objects have non-null values for only

a small number of attributes. These features pose challenges on both data storage and query processing. In this chapter we reviewed approaches developed for handling the sparse data, including data storage methods as well as techniques for query construction and evaluation over sparse tables. For data storage, several row-oriented methods were introduced, including *positional storage layout*, *bitmap-only storage* and *interpreted storage format*. Column-oriented storage is an alternative approach to row stores, which stores attributes from a table separately. The typical column-storage format includes *decompositions storage format* and *vertical storage format*. We can also emulate column oriented storage from Row-stores. For query construction, fuzzy attribute methods *F_SQL* and *F_KS* were reviewed to help the user find the matching attributes in the sparse schema. For query optimization, we introduced two row-oriented optimization methods: *Sparse B-tree Index* and *Hidden Schema*) and several column-oriented optimization techniques: *Compression*,*Late Materialization*, *Block Iteration* and *Invisible Join*.

CHAPTER 3

The Multi-tenant Database System

In this chapter, we describe the basic problems of multi-tenant database systems. There are three possible architectures to build a multi-tenant database, which are Independent Database and Independent Database Instances (IDII), Independent Tables and Shared Instances (ITSI), and Shared Tables and Shared Database Instances (STSI). All these approaches aim to provide high quality services for multiple tenants in terms of query performance and system scalability, but all of them have some pros and cons.

3.1 Description of Problem

To provide database as a service, the service provider maintains a base configurable schema S which models an enterprise application like 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 that are required. For each table, compulsory attributes are requisite

for the application and thus cannot be altered or dropped; configurable attributes are optional so that tenants can determine whether to choose or not. The service provider may also provide certain extensibility to the tenants by allowing them to add some attributes if such necessary attributes are not available in the base schema. However, if the base schema is designed properly, this case does not often occur. Based on the above configuration, tenants load their data into the remote databases and access it through an online query interface provided by the service provider. The network layer is assumed to be secured by mechanisms such as SSL/IPSec, and the service provider should guarantee the correctness of the services in accordance with privacy legislations.

In the above scenario, the main problem is how to store and index tuples in terms of the configured schema produced by the tenants. Generally speaking, there are three potential approaches to building multi-tenant databases.

3.2 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. For example, given three tenants and their tables as illustrated in Table 3.1, IDII needs to create three database instances and provides each tenant with an independent database service.

To implement IDII, for each tenant T_i with private relation R_i, we maintain its data as a set of tables $\{T_iR_1, T_iR_2, ..., T_iR_n\}$ within its private database instance.

Table 3.1: Private Data of Different Tenants

(a) Private Table of Tanent1

ENo	EName	EAge
053	Jerry	35
089	Jacky	28

(b) Private Table of Tanent2

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

(c) Private Table of Tanent3

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

Each tenants can only access its own databases and different instances are independent. Figure 3.1 illustrates the architecture of IDII. The advantage of IDII is obvious in that all the data, memory and services are independent, and the provider can set different parameters for different tenants and tune the performance for each application; thus query processing is optimized with respect to each application/query issued for each instance. In addition, IDII makes it easy for tenants to extend the applications to meet their individual needs, and restoring tenants' data from backups in the event of failure is relatively simple. Furthermore, IDII is entirely built on top of current DBMS without any extension and thus naturally guarantees perfect data isolation and security. However, IDII involves the following problems:

1. Managing a variety of database instances introduces huge maintenance cost. Service provider needs to do much configuration work for each instance. For example, to run a new MySQL instance, the DBA should provide a separate

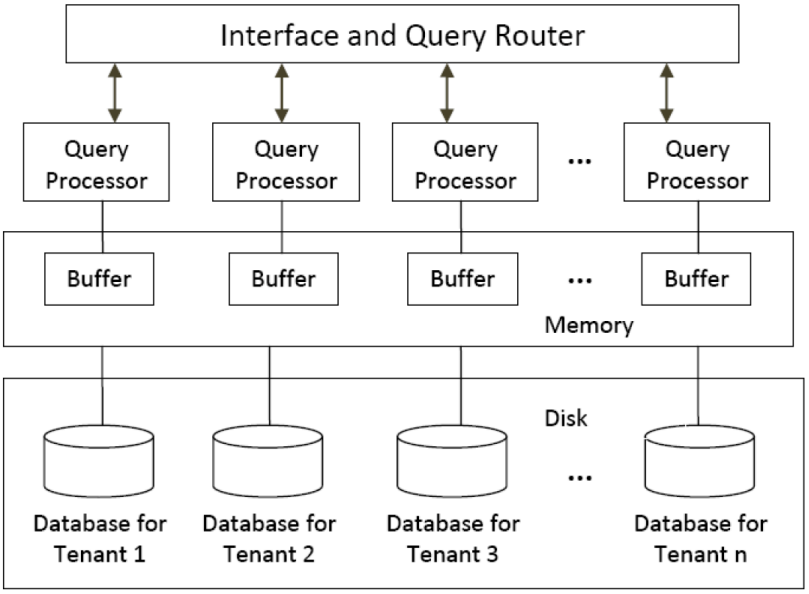

Figure 3.1: The architecture of IDII

configuration file to indicate the data directory, network parameters, performance tuning parameters, access control list etc. The DBA also needs to allocate disk space and network bandwidth for the new instance. Therefore it is impractical for the provider to maintain many heterogenous database services as it needs a lot of manpower to manage many processes, and the economy of scale may be greatly reduced.

2. Buffer/memory has to be allocated for each instance, and once in operation, it is costly to dynamically increase/decrease buffer size, and the same applies for other tuning parameters.

3. The scalability of the system, defined as the ability to handle an increasing amounts of tenants in a effective manner, is rather poor as the system cannot cut cost with the increase in the number of applications.

3.3 Independent Tables and Shared Database Instances (ITSI)

For memory/buffer sharing in database services, this section describes another multi-tenanct architecture, Independent Tables and Shared Instances (ITSI). In this approach, the tenants not only share hardware but also share 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 table in the shared database instance. The private tables between different tenants are independent.

The details of ITSI architecture are described as follows: In contrast to IDII, the system contains only one shared database, as well as shared query processor and buffer. The shared database stores data as sets of tables from all tenants $\{\{T_1R_1,...,T_1R_n\},\{T_2R_1,...T_2R_n\},...,\{T_mR_1,...,T_mR_n\}\}$, where T_iR_i stands for the private table of tenant T_i with relation R_i. In case of duplicate table name from different tenants, the name of each private table is appended a *TenantID* to indicate who owns the table. As an example, tenant 1's private *Employee* table reads $Employee_1$. Queries are also reformulated to recognize the modified table names so that correct answers can be returned. For instance, to retrieve tuples from *Employee* table, the source query issued by tenant 17 is as follows.

```
SELECT Name,Age,Phone FROM Employee
```

The transformed query is:

```
SELECT Name,Age,Phone FROM Employee_1
```

Typically, this reformulation is performed by a query router on top of the system. Figure 3.2 depicts the architecture of ITSI. The main components of ITSI include:

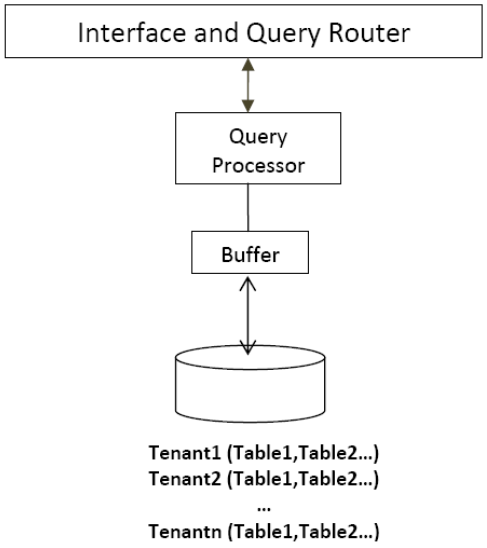

Figure 3.2: The architecture of ITSI

- **User Interface and query router:** This receives user queries and transforms the query from multiple single-tenant logical schemas to the multi-tenant physical schema in the database.

- **Query Processor:** This executes the queries transformed from the **query router** and processes the queries in the DBMS.

- **Independent Tables and Shared Database:** This keeps tenants data in their individual table layout but stores all tables in a shared database.

Unlike in IDII, the query processor, database instance and cache buffer of all the services in different tenants are shared in ITSI. Data of the same tenant are shared but the data between different tenants are independent. The advantage of this method is obvious in that there is no need for multiple database instances, which means a much lower cost especially when a large number of tenants are being handled. Thus ITSI provides much better scalability than IDII, and reduces the huge maintenance cost for managing different database instances.

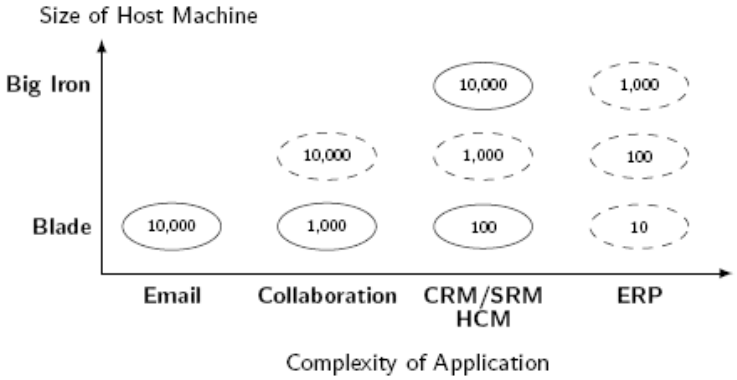

Figure 3.3: Number of Tenants per Database (Solid circles denote existing applications, dashed circles denote estimates)

However, ITSI still involves a problem in that each table is stored and optimized independently, and 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 maximum number of tables the database system supports, which is itself depends on the available memory. As an example, IBM DB2 V9.1[7] allocates 4KB of memory for each table, so 100,000 tables consume 400MB of memory up front. Figure 3.3 (taken from S.Aulbach's paper [30]) also illustrates that the shared database can support a limited amount of tenants and the scalability is extremely low when the application is complex (Blade server is estimated to support only 10 tenants for ERP application). In addition, buffer pool pages are allocated for each table so there is great competition for the cache space. Concurrent operations (especially long running queries) from multi-tenants to the shared database would introduce high contention for shared resources[3]. Finally, ITSI encounters a problem in that tenant data is very difficult to restore in case of system failure. Restoring database need to overwrite all tenants' data even if some tables do not experience data loss.

3.4 Shared Tables and Shared Database Instances (STSI)

Jeffrey D. Ullman et al. [44, 58] proposed the universal relation table to simulate the effect of the representative instance. The universal relation model aims at achieving complete access-path independence in relational databases by relieving the user of the need for logical navigation among relations. The essential idea of the universal relation model is that access paths are embedded in attribute names. Thus, attribute names must play unique "roles". Furthermore, it assumes that for every set of attributes, there is a basic relationship that the user has in mind. The user's queries refer to these basic relationships rather than the underlying database. More recently, Google proposed BigTable[38] for effectively organizing its data. Many projects at Google store data in BigTable, including web indexing, Google Earth[8], and Google Finance[9]. These applications place very different demands on BigTable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, BigTable has successfully provided a flexible, high-performance solution for all of these Google products. Based on the concept of BigTable, Bei Yu et al.[59] proposed a universal generic table for storing and sharing information of all types of domains, which is demonstrated to be a flexible structure placing no restriction on data units. Inspired by the idea of the universal relation model, Shared Tables and Shared Database Instances (STSI) is proposed as the third possible multi-tenant architecture to address the problem.

In STSI, the system only provides one query processor, and all the instances and tenants share the same processor. Moreover, unlike ITSI, all the tenants not only share databases but also share tables to manage all the data. STSI differs from the

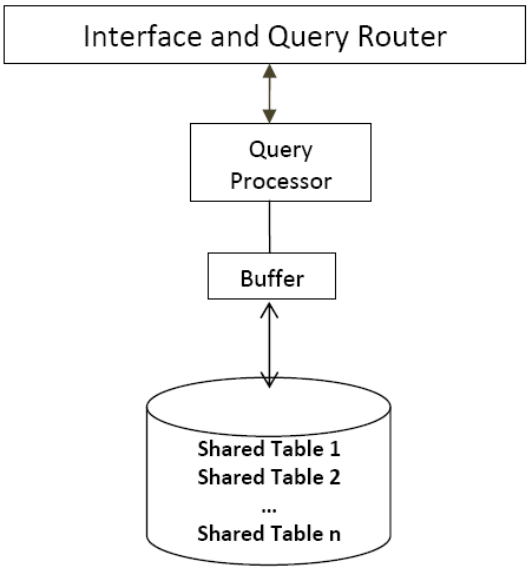

Figure 3.4: The architecture of STSI

universal relation in that the latter is a wide virtual schema which puts all entities and relations in the same logical table, while the former is a schema for physical representation and stores a number of entities that belong to the same entity set in the same table. For example, we would store all employee information in a tenant into a wide table, but we would not put employee, customers and products etc. in the same physical table. Figure.3.4 illustrates the architecture of STSI. From users' point of view, the data owner occupies individual services, sources and so on. From the service provider's point of view, the service provider integrates the data and offers a unified service to all the tenants and database applications.

To implement STSI, 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 tenant identifier *TenantID* and setting unused attributes to NULL. Table 3.2 shows the layout of a shared *Employee* table which stores tuples from three tenants.

Table 3.2: STSI Shared Table Layout

TenantID	ENo	EName	EAge	EPhone	ESalary	EOffice
Tenant 1	053	Jerry	35	NULL	NULL	NULL
Tenant 1	089	Jacky	28	NULL	NULL	NULL
Tenant 2	023	Mary	NULL	98674520	NULL	Shanghai
Tenant 2	077	Ball	NULL	22753408	NULL	Singapore
Tenant 3	131	Big	40	NULL	8000	London
Tenant 3	088	Tom	36	NULL	6500	Tokyo

To differentiate the tenants from each other and to allow query processor to recognize the queries, STSI provides a query router to transform issued queries to the shared table. The system maintains two maps: a map from tenants to *TenantID*s, and another from attributes of tenants to attributes in ST. Thus, we can easily transform queries to their corresponding attributes in ST. As an example, the issued query from tenant 17 to retrieve tuples in *Employee* table can be converted to:

```
SELECT Name,Age,Phone FROM Employee WHERE TenentID='17'
```

Overall, the main components of STSI include:

- **User Interface and Query Router:** This receives user queries and transforms queries to corresponding columns in the Shared Table by using the two maps.
- **Query Processor:** This executes the queries transformed from the **query router** and processes the queries in the Shared Table.
- **Shared Tables and Shared Database:** This stores data from all tenants with a universal storage method and differentiates tenants by adding tenant id attribute to the shared table.

Using STSI, the service provider only maintains a single database instance, therefore the maintenance cost can be greatly reduced. Compared to IDII and ITSI, the number of tables in STSI is determined by the base schema rather than the number of tenants. The advantage of STSI is obvious that everything is pooled, including processes, memory, connections, prepared statements, databases, etc. Thus, STSI approach is believed to be more scalable to a large number of instances/tenants.

However, STSI introduces two performance issues. First, consolidating tuples from different tenants into the same ST causes that ST stores too many NULLs. The schema of ST is usually very wide, typically including hundreds of attributes. For a particular tenant, it is less likely that all the configurable attributes will be used. In typical cases, only a small subset of attributes are actually chosen. Thus, many NULLs are resulted. Although commercial databases handle NULLs fairly efficiently, many works show that if the table is too sparse, the disk space wastage and performance degradation cannot be neglected [32]. Second, ST poses challenges to query evaluation which are not found in queries over narrow, denser tables. Without proper indexing scheme, table scan would become a dominant query evaluation option, but scans over hundreds or thousands of attributes in addition to those required by the query is very costly. Moreover, building and maintaining hundreds or thousands of indexes on a shared table is generally considered infeasible because storage and update costs are extremely high. Therefore, to achieve good system performance, indexing is an important problem that should be considered.

3.5 Summary

As a form of software as a service, multi-tenant database system brings great benefits to organizations by providing seamless mechanisms to create, access and maintain databases at the host site. However, the way of providing high-quality services for multiple tenants becomes a big challenge. To address the problem, we describe three potential multi-tenancy architectures and analyze their features in terms of query performance and system scalability. IDII provides independent services for each tenants, thus query processing is optimized but the cost is huge and scalability is rather poor. ITSI greatly reduces the cost by sharing database instance among tenants but still encounters scalability problem since the performance of system is limited by the number of tables it serves. STSI provides shared tables for all tenants to achieve good scalability but poses a challenge on query processing. Generally speaking, if the storage and indexing problems can be solved properly, STSI is believed to be a promising method for the design of multi-tenant database system.

CHAPTER 4

The M-Store System

4.1 System Overview

The M-Store system defines a framework that supports cost-efficient data storage and querying services to multi-tenant applications. The system accepts data from tenants, stores them at the host site, and provides seamless database services to remote business organizations. Similar to STSI, the framework of M-Store includes one database instance and a number of shared tables. All tenants and applications share the same database instance and stores entities that belong to the same entity set in the same table. For example, the system stores all tenants' customer information into the shared *Customer* table, but it does not put other information such as *Products* in this table. Different from traditional relational database model, the shared table schema in the M-Store system is specifically designed for multi-tenant applications. It contains a set of fixed attributes and a set of configurable attributes. Fix attributes are compulsory and can not be altered or dropped, while configurable attributes are optional for tenants according to their needs. The choice of such shared table model is suitable for multi-tenant database because the number of shared tables is pre-defined and independent of the number of tenants,

bringing benefits to the system scalability and reducing the maintenance cost.

Definition 4.1 *The M-Store shared table schema is an expression of the form R(U), where R is the name of the table, and U is the set of attributes such that* $U = U_F \bigcup U_C$ *and* $U_F \bigcap U_C = \emptyset$. U_F *is the set of fixed attributes that* $U_F = \{tid, A_1, A_2, ..., A_m\}$, *where tid is the tenant identifier.* U_C *is the set of configurable attributes where* $U_C = \{A_{m+1}, A_{m+2}, ..., A_n\}$.

The domains of attributes in U_F and U_C are initially defined by the system. To subscribe to the service, a tenant configures the shared table schema by compulsorily choosing the fixed attributes and selectively choosing configurable attributes that they need. In addition, each tenant is mapped with a *tid* attribute as a tenant identifier. For each tenant T_i, we insert its data into the corresponding columns in the shared table, the unused configurable attributes are set to NULL.

The M-Store system includes a storage manager component to maintain the shared table. It is responsible for storing and indexing data whose volume may grow quickly with the number of tenants increase. As analyzed in Chapter 3, one of the performance issues that STSI encounters is that the sparse ST normally contains a large number of null values and wastes much storage space. To overcome this problem, the M-Store system adopts a *Bitmap Interpreted Tuple* (BIT) storage format as the physical representation and store of the data. Compared to the standard horizontal positional format that STSI uses, the BIT storage format contains additional tenants information which can effectively eliminate null values from the unused attributes. Another drawback of STSI is that there is no efficient indexing scheme for wide and sparse tables, which poses great challenge to query evaluation. In the M-Store system, we develop the *Multi-Separated Index* (MSI) technique, which builds separated index for each tenant instead of one sparse index for all tenants.

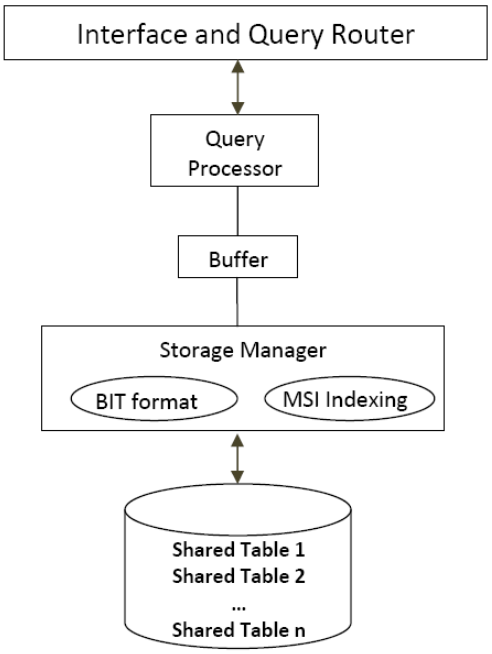

Figure 4.1: The architecture of the M-Store system

The M-Store system also contains a query router to reformulate the queries so that the query processor can recognize data from different tenants. The issued queries are transformed by adding a `RESTRICT ON TENANT` *`tid`* statement. To illustrate, a query from tenant 17 to retrieve tuples in *Employee* table can be converted to:

```
SELECT Name,Age,Phone FROM Employee RESTRICT ON TENANT 17
```

Figure 4.1 illustrates the architecture of the M-Store system.

The M-Store system can be viewed as an optimization of STSI. Both of them maintain only one database instance and shared tables. However, M-Store differs from STSI in three main points. First, STSI stores data with the positional storage format (reviewed in Section 2.1.1), while M-Store adopts a proposed BIT storage format that eliminates null values from unused attributes. Second, STSI builds sparse B-tree indexes on all tenants data but M-Store creates separated indexes for

each tenant with MSI scheme. Third, the query router in STSI reformulates the issued query with a *tid* predicate to differentiate tenants from each other. While M-Store transforms queries by adding a `RESTRICT ON tid` statement, where the *tid* information is an identifer of the tenant that can be used for data storage and query evaluation.

4.2 The Bitmap Interpreted Tuple Format

4.2.1 Overview of BIT 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 work dealing with sparse tables such as Vertical Schema[28] and Interpreted Format[32] either introduce much overhead in tuple reconstruction or prevent the storage system from optimizing random access to locate the given attribute. To the best of our knowledge, none of them is optimized for multi-tenant databases.

One of the properties of a multi-tenant database is that the tuples 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 the layout that the first two attributes have values and the last two attributes are NULLs. Based on this observation, we propose a Bitmap Interpreted Tuple Format (BIT) technique to efficiently store and retrieve tuples for multi-tenants without storing NULLs from unused attributes.

This 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 standard `CREATE TABLE` statement. As an example, tenant 17 configures an *Employee* table as shown below. Note that the data type declaration in the base schema is ignored for simplicity.

```
CREATE CONFIGURE TABLE Employee(ENo,EName,EPhone,ESalary)
FROM BASE Employee(ENo, EName, EAge, EPhone, EDepartment, ESalary,
ENation)
```

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 1101010. 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 is opened, the table catalog information and bitmap strings are loaded into the memory together. This *in-memory* strategy is possible in that even the base source table has 1000 attributes, loading bitmap strings for 1000 tenants only causes about 120KB memory overhead which can be entirely ignored. Figure 4.2 shows a representative BIT catalog information that consists of two parts: table catalog and bitmap catalog. Table catalog contains fields such as attribute name, type and length as in the positional notation. Bitmap catalog starts with a tenant-id then follows bitmap string. When a tenant configures the base schema, the tenant-id and corresponding bitmap information appears in the bitmap catalog, where the

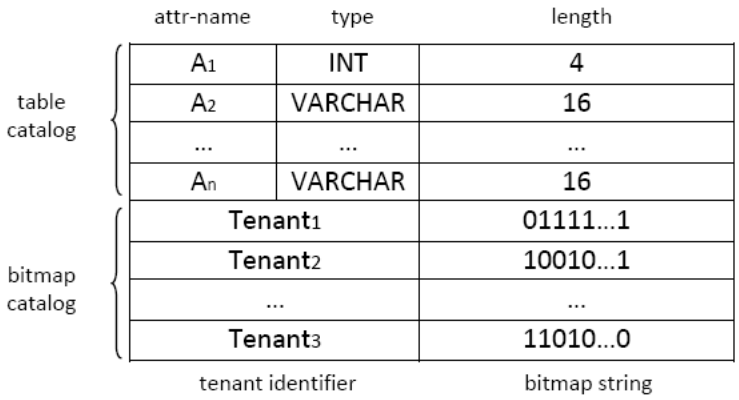

Figure 4.2: The Catalog of BIT

length of bitmap string is the total number of attributes in the base schema, '1' represents configured attributes and '0' denotes unused attributes. In the M-Store system, we extended the MySQL's table catalog file, i.e., `.frm` file associated with the table created in MySQL, and appended 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 the attributes whose positions in the bitmap string are marked as 0 are removed. The rest of other 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 with the row-store layout used in most of the current commercial database systems. It begins with a tuple header 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. Readers should not confuse the nulls in configured attributes with NULLs in unused attributes. The nulls in configured attributes mean the values are missing. While the NULLs produced by the unused attributes indicate that the attributes are not configured by the tenant. Figure 4.3

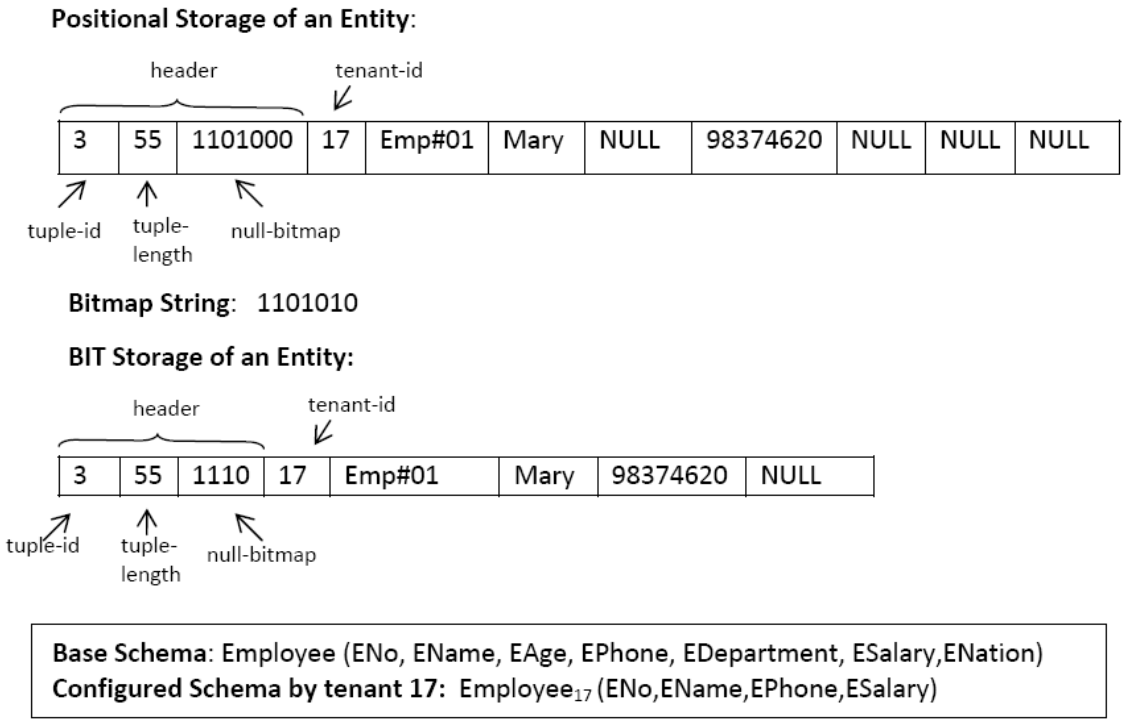

Figure 4.3: The BIT storage layout and it's corresponding positional storage representation

gives an example of the BIT storage layout and it's corresponding positional format representation. In this example, tenant 17 configures its table from the base schema by selectively choosing some attributes (ENo,EName,EPhone,ESalary), a bitmap string is then constructed in terms of table configuration (i.e.,1101010). When a tuple is inserted to the table, attributes whose value in the bitmap string are 0 are removed and the remaining attributes are stored in the table. This approach differs from the traditional positional storage format, in which all attributes in the base schema are stored and unconfigured attributes are set to NULL.

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 fixed-

width, the offset of each attribute is computed when the bitmap string is loaded into memory. If variable-width attribute is involved, calculation of the offset of attribute A_n requires addition of data-lengths of the prior $n-1$ attributes.

BIT format is specifically designed for supporting multi-tenant applications. To store tuples from different tenants in the wide base table, we only maintain a per-tenant bitmap string whose length is fixed by the number of attributes in the base schema. Compared with the traditional positional storage layout used by STSI, BIT format stores nothing for unused attributes, therefore sparse data sets in a horizontal schema can in general be stored much more compactly in the format.

4.2.2 Cost of Data Storage

In this section we analyze the cost of data storage in M-Store and STSI. As we mentioned before, the M-Store system adopts the proposed BIT storage format whereas STSI stores data in the positional storage layout[48, 61], which stores NULLs for both unused attributes and configured attributes whose data value is NULL.

Suppose the base configurable schema $R = \{A_0, A_1, \ldots, A_k\}$, where k is the number of attributes in the base table layout. In the M-Store system, a bitmap string is constructed for each tenant which is used to decode the used and unused attributes in the base schema. The length of bitmap string is k and the corresponding value in the bitmap for each attribute A_i is set to '1' or '0'. In the M-Store system, bitmap strings are stored together with the table catalog information and are loaded into memory when the shared table is opened. If there are totally M tenants in the M-Store system, bitmap strings only consume approximately $(M * k)$ bits memory.

According to the bitmap string, the attributes whose value in bitmap are set

to 0 are removed, and the remaining attributes in the tuple are compacted as a new tuple. Let $|L_i|$ be the length of attribute A_i. The overhead of storing a new compact tuple T_{new} is $\sum_{i=1}^{k}(|L_i| * b(T,i))$, where $b(T,i)$ is the bit value of the ith attribute in the bitmap string for tenant T, i.e., the value of $b(T,i)$ is '1' or '0' for configured and unused attributes respectively. Given M tenants, the average tuple length ATL is calculated as:

$$ATL = \frac{\sum_{j=1}^{M} t_{new}}{M} = \frac{1}{M}\sum_{j=1}^{M}\sum_{i=1}^{k}(|L_i| * b(T_j,i)) \tag{4.1}$$

Suppose the size of one disk page is P, the average number of data records that can fit on a page $N_{d,mstore}$ is estimated as:

$$N_{d,mstore} = \left\lceil \frac{P}{ATL} \right\rceil = \left\lceil \frac{P * M}{\sum_{j=1}^{M}\sum_{i=1}^{k}(|L_i| * b(T_j,i))} \right\rceil \tag{4.2}$$

Assume that the total disk space is XGB, the volume of data records that M-Store system can support is:

$$V_{M-Store} = \left\lceil \frac{X}{P} \right\rceil * N_{d,mstore} = \left\lceil \frac{X}{P} \right\rceil \left\lceil \frac{P * M}{\sum_{j=1}^{M}\sum_{i=1}^{k}(|L_i| * b(T_j,i))} \right\rceil \tag{4.3}$$

While in STSI, the shared table is stored in positional storage layout where all unused attributes are set to NULLs and occupy disk space. Given the base schema $\{A_0, A_1, \ldots, A_k\}$ and the length of attribute $|L_i|$, the overhead of storing a tuple is $\sum_{i=1}^{k}|L_i|$. Therefore, with XGB's storage space, the volume of data records that STSI can support is:

$$V_{STSI} = \left\lceil \frac{X}{P} \right\rceil \left\lceil \frac{P}{\sum_{i=1}^{k}|L_i|} \right\rceil \tag{4.4}$$

Table 4.1: Table of Notations

Notation	Description
$R(U)$	M-Store shared table schema
U_F	the set of fixed attributes
U_C	the set of configurable attributes
$\|L_i\|$	the length of attribute A_i
$b(T, i)$	the bit value of the ith attribute in the bitmap string for tenant T
t_{new}	the compact tuple after removing unused attributes
ATL	the average tuple length
$N_{d,mstore}$	the average number of data records that can fit on a page in M-Store
$N_{d,stsi}$	the number of data records that can fit on a page in STSI
V	the volume of data records
N_i	the number of index entries that can fit on a page
F	the average fanout of the B^+-tree
$\|\|r_i\|\|$	the number of data records in tenant T_i
$\sigma_p(T_i)$	the number of records that satisfy the query predicate p in tenant T_i

By adopting BIT technique, M-Store dose not introduce overhead for storing NULLs from unused attributes. With such efficient storage format, I/O cost of data scanning can also be reduced. Equation 4.5 and 4.6 compute the approximate I/O cost of reading Y tuples in M-Store and STSI respectively.

$$C_{M-Store} = \left\lceil \frac{Y}{N_{d,mstore}} \right\rceil = \left\lceil Y \Big/ \left\lceil \frac{P * M}{\sum_{j=1}^{M} \sum_{i=1}^{k} (|L_i| * b(T_j, i))} \right\rceil \right\rceil \tag{4.5}$$

$$C_{STSI} = \left\lceil Y \Big/ \left\lceil \frac{P}{\sum_{i=1}^{k} |L_i|} \right\rceil \right\rceil \tag{4.6}$$

For ease of reading, all of the notations are summarized in table 4.1.

4.3 The Multi-Separated Index

4.3.1 Overview of MSI

Wide, sparse tables pose great challenges to query evaluation. For example, scans must process hundreds or thousands of attributes in addition to those specified in the query. In a multi-tenant database, the shared table stores tuples from a number of tenants, and the data volume is normally huge. When such huge data sets are stored in a single table, it is crucial that we minimize the need to scan the whole table. A common approach to avoid table scans is indexing.

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 *Big Index* (BI). The BI approach has an advantage that the index is shared among all the 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. However, the BI approach incurs problem that by indexing tuples from all tenants, the storage overhead and maintenance cost of such Big Index is very high. In addition, the scan of index file is rather inefficiency. For example, 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 a very time consuming operation because the index has keys of all tenants.

Instead of using BI for each sparse table, in the M-Store system, another indexing technique called *Multi-Separated Index* (MSI) is developed. Instead of building an index for all tenants, we build a separated index for each tenant. If a hundred of tenants want to index tuples on an attribute, one hundred separated 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 which may degrade the query performance. However, in multi-tenant applications the shared table is generally sparse, and given a particular attribute, only a certain number of tenants configures this attribute and have index on it. Therefore, in real applications, MSI does not make the number of indexes explosive.

In the M-Store system, there is a query router component which transforms issued queries with a `RESTRICT ON TENANT tid` statement so that the query processor can recognize data from different tenants. The *tid* information is considered as the tenant identifier which helps the optimizer automatically locate the corresponding separate index. A simple query from tenant 23 is given below as an example. By using MSI, only tenant 23's index file is loaded and scanned.

```
SELECT max(l_partkey), o_orderkey
FROM orders, lineitem
WHERE orders.o_orderkey = lineitem.l_orderkey
RESTRICT ON TENANT 23
```

Compared to BI, MSI has several advantages. First, MSI is flexible. Each tenant indexes its own tuples so that there is no restriction that all the tenants must build index on the same attribute or none of them can do it. Second, scans of index file is efficient. To perform an index scan, each tenant only needs to scan its own index file. This is different from 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. Third, in MSI indexing scheme, for tuple insertion or deletion, a tenant only needs to update its own index file on the configured attributes. Whereas in BI indexing, the whole index shared

by all tenants needs to be updated which is very inefficient.

MSI is a special case of partial indexes[69]. A partial index contains only a subset of tuples in a table. To define this subset, a conditional expression called the predicate of index is used. Only tuples that are evaluated true in this predicate are included in the index. MSI can be viewed as partial index if the predicate condition is $tenant - id$. For instance, we can define a MSI index on an attribute A_m in a shared table $R(tid, A_1, A_2, ..., A_m, ..., A_n)$ in terms of a partial index as follows:

```
CREATE INDEX MSI_index ON R(A_m)
WHERE tid = tenant-id
```

However, in our implementation of the M-Store system, using generic partial indexes to implement MSI is not a good solution, because we need to build many multi-separate indexes for the shared table. In partial indexing scheme, predicate conditions need to be check during index maintenance and query evaluation. When a tuple is inserted or deleted, the system must evaluate the predicate of each index on the shared table to determine if the index needs to be updated. Therefore, in M-Store, for each tuple insertion or deletion, the shared table containing hundreds of sperate indexes will have hundreds of predicate evaluation, which introduce huge overhead. MSI avoids this cost by eliminating the need to evaluate the filter condition and thus the update of index is quite efficient. In such a case, each MSI behaves like a conventional index, but over a subset of tuples that belong to a given tenant.

MSI is also different from view indexing [10, 65]. View is dynamic and content based – a tuple that is indexed is dropped when its indexed attribute value does not satisfy the view. On the contrary, the number of tuples indexed by an MSI indexing for a tenant over an attribute does not change with respect to changes to

attribute values.

4.3.2 Cost of Indexing

In this section, we analyze the cost of indexing in both M-Store and STSI. As introduced in section 4.3.1, the M-Store system adopts the proposed MSI indexing scheme which builds individual index for each tenant. While STSI uses BI indexing to construct a big index for all tenants' data. Let N_i be the number of index entries that can fit on a page, F be the average fanout of the B^+-tree index, $\|r_i\|$ is the number of data records in tenant T_i. In the M-Store system, the cost of navigating B^+-tree's internal nodes to locate first leaf page is calculated as:

$$Cost^1 = \log_F \left(\left\lceil \frac{\|r_i\|}{N_i} \right\rceil \right) \tag{4.7}$$

Assume that the records follow the uniform distribution, $\|\sigma_p(T_i)\|$ denotes the number of data records that satisfy the query predicate p in tenant T_i, the cost of scanning leaf pages to access all qualifying data entries is:

$$Cost^2 = \left\lceil \frac{\|\sigma_p(T_i)\|}{N_i} \right\rceil \tag{4.8}$$

For each data entry, the cost for retrieving data records is:

$$Cost^3 = \left\lceil \frac{\|\sigma_p(T_i)\|}{N_{d,mstore}} \right\rceil \tag{4.9}$$

Where $N_{d,mstore}$ is the average number of data records that can fit on a page (Equation 4.2). Therefore, the total I/O cost of MSI indexing scheme can be calculated as:

$$C_{M-Store} = \log_F \left(\left\lceil \frac{\|r_i\|}{N_i} \right\rceil \right) + \left\lceil \frac{\|\sigma_p(T_i)\|}{N_i} \right\rceil + \left\lceil \frac{\|\sigma_p(T_i)\|}{N_{d,mstore}} \right\rceil \tag{4.10}$$

Compared to the M-Store system, STSI uses *Big Index* to index tuples from all tenants. Suppose that there are M tenants in the system, the I/O cost of BI indexing scheme can be evaluated as:

$$C_{STSI} = \log_F \left\lceil \frac{\sum_{i=1}^{M} \|r_i\|}{N_i} \right\rceil + \left\lceil \frac{\sum_{i=1}^{M} \|\sigma_p(T_i)\|}{N_i} \right\rceil + \left\lceil \frac{\|\sigma_p(T_i)\|}{N_{d,stsi}} \right\rceil \tag{4.11}$$

$N_{d,stsi}$ is the average number of tuples that can fit on a page in STSI.

$$N_{d,stsi} = \left\lceil \frac{P}{\sum_{i=1}^{k} |L_i|} \right\rceil \tag{4.12}$$

All of the notations are summarized in table 4.1.

MSI outperfroms BI in three points: First, MSI indexes smaller number of data records so that the cost of navigating non-leaf nodes is less than BI. Second, the number of results that satisfy the query predicate in BI is much more than MSI since BI is built on all tenants data. Therefore, MSI introduces less overhead in scanning index entries in leaf pages. Third, the M-Store system adopts BIT storage format, where the unused attributes are removed and tuple is compacted as a smaller one. Therefore the cost of retrieving data records in M-Store is significantly reduced.

4.4 Summary

This chapter presents the proposed multi-tenant database system, M-Store. The M-Store system aims to achieve excellent scalability by following 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 format and Multi-separated Indexing (MSI) scheme. As we aim to solve the scalability issue, we analyze the major cost of the M-Store system, in terms of disk space usage and I/O. Based on the cost model and contrastive analysis on STSI, the M-Store system is demonstrated to be capable of supporting multi-tenant applications with less storage and querying overhead.

CHAPTER 5

Experiment Study

In this chapter, we empirically evaluate the efficiency and scalability of the M-Store system. Scalability is defined as the system ability to handle growing amounts of work in a graceful manner [35]. In our experiments, we consider the scalability of M-Store by measuring system throughput as data scale increases. Two sets of experiments are evaluated in terms of different dimensions of data scale: tenant amounts and number of columns in the shared table. In each set of experiment, we evaluate the capability of the proposed BIT storage model and MSI indexing scheme, by measuring disk space usage and system throughput. The original STSI is used as the baseline in the experiments.

5.1 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 [16] and TPC-H [17] 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

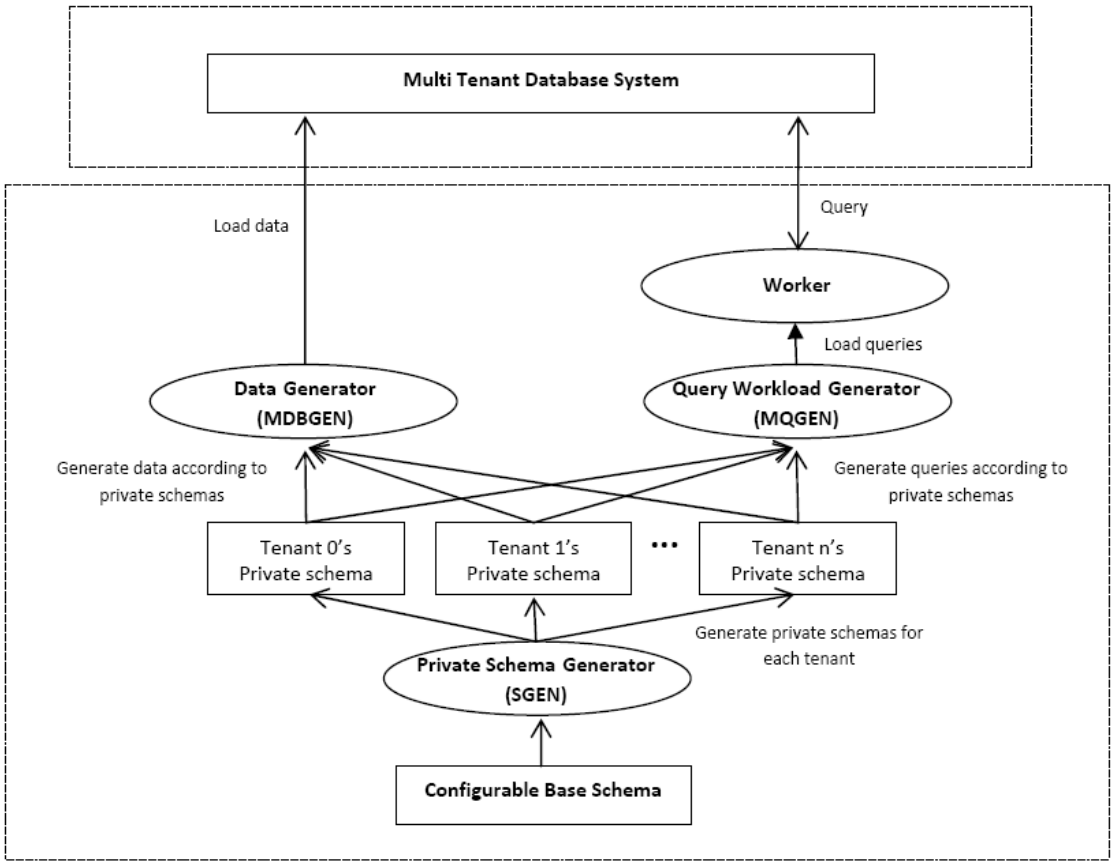

Figure 5.1: The relationship between DaaS benchmark components

important feature that a multi-tenant database must have 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 five modules: a configurable database base schema, a private schema generator, a data generator, a query workload generator, and a worker. Figure 5.1 illustrates the relationship between these components. We will describe the details of them below.

5.1.1 Configurable Base Schema

We follow the logical database design of TPC-H to generate the configurable database base schema. Our benchmark database comprises three tables. These tables are chosen out of eight tables from the TPC-H benchmark. They are: `lineitem`, `orders`, and `customer`. Figure 5.2 illustrates the table relationships in TPC-H. For each table, we extend the number of attributes by including customized attributes to the original table schema, one of which is `tid` (tenant ID) that denotes the tuple owner. The data type of extended attributes, excluding `tid` whose data type is integer, is string. The first few attributes in each table are marked as fixed attributes that each tenant must choose. The remaining attributes are marked as configurable. The simplified `customer` table schema is given below for illustration purpose. In this example, `tid`, `c_custkey`, `c_name`, `c_address` and `c_nation` are fixed attributes. The remaining attributes, i.e., `c_col1`, `c_col2`, and `c_col3`, are configurable.

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

5.1.2 SGEN

We develop a tool called SGEN to generate private schemas for each tenant. In addition to the fixed attributes that each tenant must choose, SGEN is mainly responsible for the selection of configurable attributes for each tenant to form the private schema. To generate the independent schema, for each tenant T_i, a configurable column C_j is picked from the configurable attributes in the base schema with a probability p_{ij}. In practice, this probability distribution is not even. A

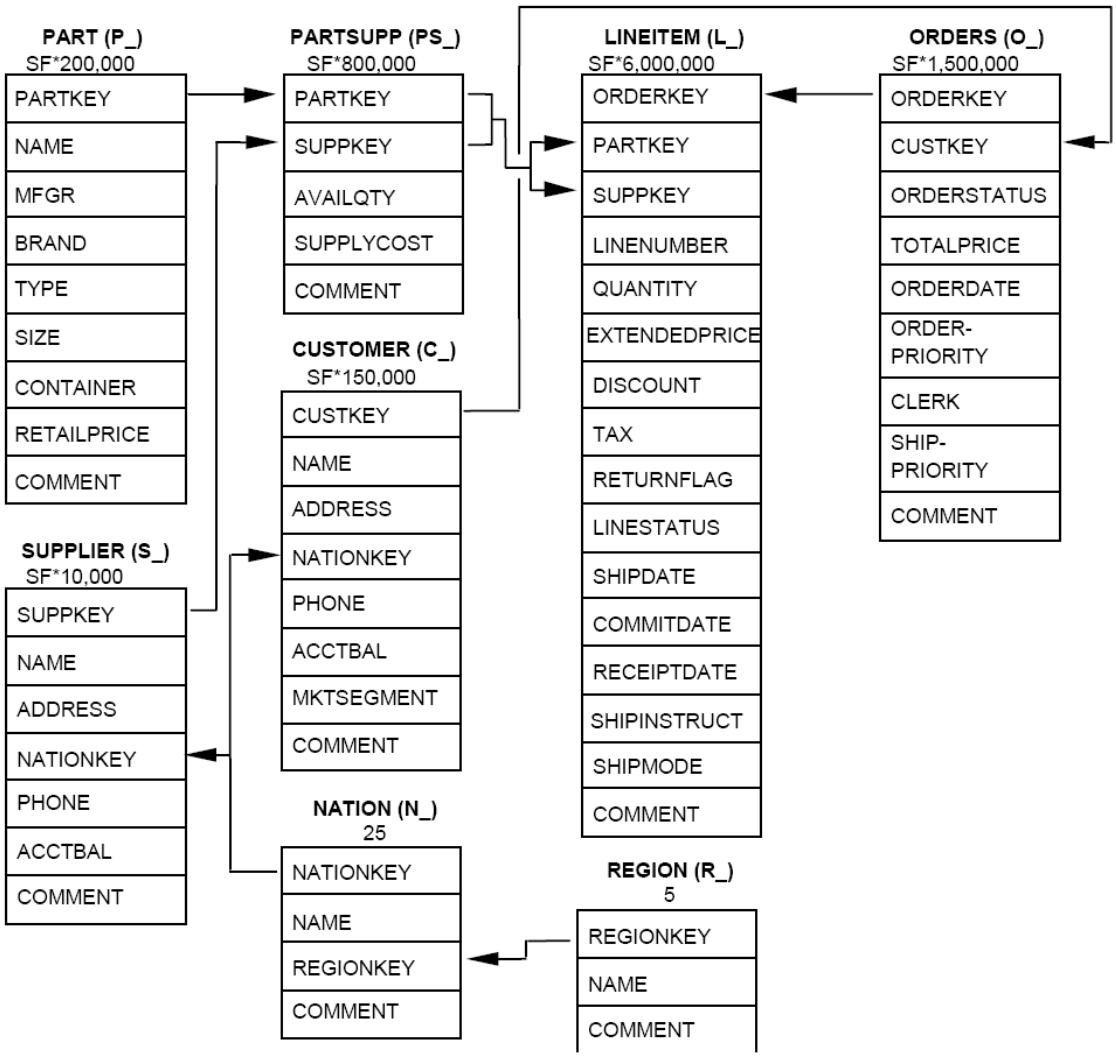

Figure 5.2: Table relations in TPC-H benchmark (taken from [17])

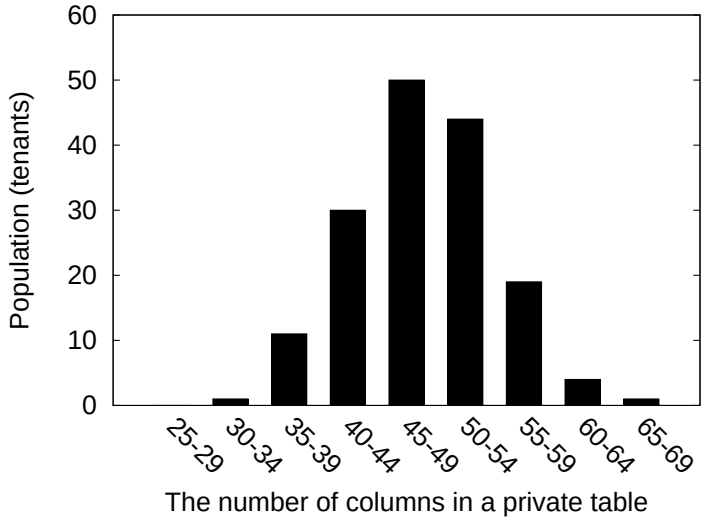

Figure 5.3: Distribution of column amounts. Number of fixed columns = 4; Number of configurable columns = 400; Tenant number = 160; $p_f = 0.5$; $p_i = 0.0918$

small number of attributes in the base schema could be more frequently chosen than other columns. To capture the skewness of the distribution, the configurable columns are divided into two sets, noted as S_f and S_i, indicating the set of frequent and infrequent selected columns. If a column C_j belongs to S_f, the probability it is picked by any tenant is set to p_f, otherwise p_i. In our experiment, a collection of 8 configurable columns are selected to S_f, and others are left in S_i. Namely, the size of the S_f is fixed and the size of S_i varies with the configurable column numbers.

According to this generation method, the number of columns selected by each tenant approximately follow a normal distribution. Let the number of configurable column in the base schema be c, the mean of configurable columns being picked by each tenant is $8p_f + (c-8)p_i$. In our experiment, p_f is fixed to 0.5, and we set the p_i to control the mean of column number. Figure 5.3 illustrates the distribution of column amounts in private table, when generated by the above method. As can be seen in the figure, the distribution well follows a normal distribution. The actual mean is 44.39375, which is closed to the expected value $4+8*0.5+392*0.0918 = 44$.

5.1.3 MDBGEN

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 configurable attributes are generated by random v-string algorithm used in DBGEN. The values in unused attributes are set to NULLs.

5.1.4 MQGEN

Following TPC-C and TPC-H, we design and implement a query workload generator MQGEN 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. In our experiment, simple query is a range query which selects three attributes from the shared table and whose range selection condition has an average selectivity of 0.3 (i.e., the ratio of the number of selected tuples to the number of entire records in the table is 0.3). An example of such a query is as follows. Note that we have `RESTRICT ON TENANT` statement in the query to indicate which tenant does the tuples belong to, and help the optimizer choose the separate index for a given tenant correctly.

```
SELECT c_custkey, c_name, c_nationkey
FROM   customer
WHERE  c_custkey>5000 and c_custkey<9000
RESTRICT ON TENANT 100;
```

- *Analytical Query:* Run reporting queries which perform a projection-join-aggregation operation on the shared tables of the tenants. An example is given as below.

```
SELECT max(o_orderkey), o_orderstatus, o_totalprice,
       l_partkey, l_suppkey, l_linenumber
FROM   orders, lineitem
WHERE  o_orderkey=l_orderkey
RESTRICT ON TENANT 100;
```

  In an analytical query, two tables with foreign/primary key constraints are specified in the *from* clause; six attributes are selected from the joined tables, with three from each table; one join condition is specified in the *where* clause.

- *Update Query:* Insert and delete tuples of tenants to the shared tables. Examples of insertion and deletion are given as follow:

```
INSERT INTO customer (tid, c_custkey)
VALUES (100, 2503404);


DELETE FROM customer
WHERE c_custkey = 2503404
RESTRICT ON TENANT 100;
```

  Insertion query inserts one single tuple. The tenant id and primary key attribute must be specified in the inserted tuple. Deletion query specifies a tuple with its tenant id and primary attribute. In our experiment, insertion and deletion appear pairwise in the query workload.

5.1.5 Worker

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 and throughput of the system. 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.

5.2 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.

Two sets of experiments are examined to evaluate the scalability of the system: the effect of tenants and the effect of column amounts. To conduct these experiments, we first generate private database schema for each tenant by running SGEN. For the first set of experiments, we define the base schema with 400 configurable attributes and the average percentage of non-null attributes (μ) is 10. Therefore, according to SGEN schema generation method, c is set to 400 and $pi = 0.0918$. We finally generate private schemas for 160 tenants. For the second set of experiments, we measure the system scalability with the number of columns increase. We set the base schema with different number of configurable attributes, varying from 300 to 800. The average percentage of non-null attributes is 10 and the number of tenants in the shared table is 80. Table 5.1 shows the settings of pi for private schema

Table 5.1: Settings of p_i under varying column amounts

# of columns (c)	Value of p_i
300	0.089
400	0.0918
500	0.0935
600	0.0945
700	0.0954
800	0.096

Table 5.2: Varying Parameters

Parameter	Varying Range
Number of tenants	20, 40, 60, **80**, 100, 120, 140, 160
Number of columns	300, **400**, 500, 600, 700, 800
μ (% of non-null attributes)	**10**

generation under varying column amounts.

These schemas are then used for evaluating the scalability of STSI and M-Store system. The settings of parameter used in the experiments are summarized in Table 5.2, where default values are shown in bold font.

Next, we run MDBGEN to generate data for benchmark databases according to the resulting private schemas. The data scale of tables for each tenant is illustrated in Table 5.3. Also we run MQGEN to generate query workloads for each private schema. In order to evaluate our system under different types of queries, three kinds of query sets are generated: simple query set, analytical query set, and update query set. For simple and analytical queries, the selection condition is randomly generated according to the data distribution, the selectivity is fixed to 0.3. For update queries, insertion and deletion are generated simultaneously, each insertion is generated together with one deletion. The tuple being inserted and deleted are exactly the same tuple. Therefore, in each deletion, there is at least one tuple being deleted, and the query file can be continually run without significantly impact of

Table 5.3: Data scale of three tables of each tenant

Table name	Number of tuples
lineitem	180,000
customer	4,500
orders	45,000

the data scale. In each query set, MQGEN generates one query file per tenant, which contains 1,000 queries. All queries in a single query set are of the same type.

In our experiments, all fixed attributes and frequently selected attributes (refer to Section 5.1.2) in the shared table are indexed. To evaluate the performance of the proposed indexing technique, queries generated by MQGEN are against those indexed attributes.

For the Worker, we set the number of concurrent database threads to 20. The worker picks queries from each tenant's query file to form a query queue. Each thread runs one query from the query queue for one tenant. The worker finally calculates the throughput and average response time of the system.

The Worker is run on a PC with Intel Core Duo 2.33GHz processor, 4.0GB memory and the database system is run on a windows server with Intel Core Duo 2.99GHz processor, 128GB 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. To speedup the query performance, we build compound BI index on `tid` and other attributes for STSI.

We implement the M-Store system 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. For example, 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_address, c_nation,
    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 system throughput with three kinds of benchmark query workloads.

5.3 Effect of Tenants

In this section, we present the experimental results of M-Store and STSI under different tenant amounts. We evaluate the scalability of both the systems by measuring whether the system has the ability to serve an increasing number of tenants without severe performance degradation. We examine the system scalability in terms of two aspects: the performance of storage and system throughput.

5.3.1 Storage Capability

Figure 5.4 depicts the disk space usage of M-Store and STSI under different tenant amounts. We fix the number of columns in the shared table to 400 and set the

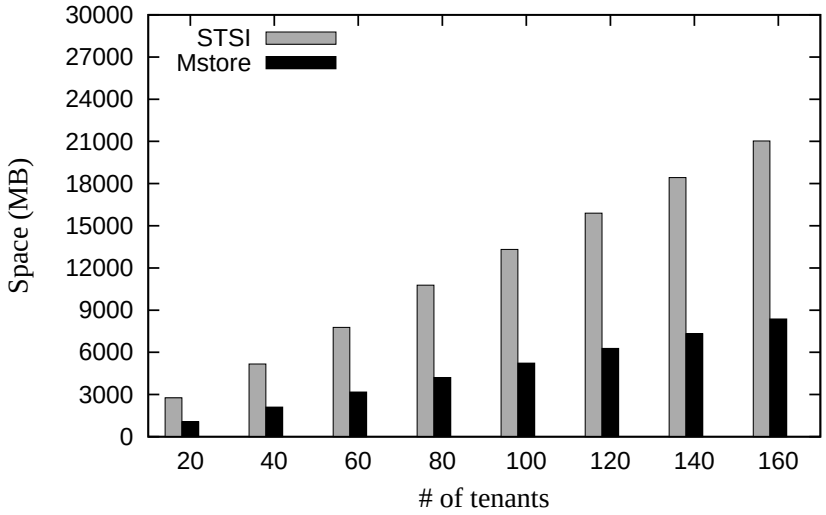

Figure 5.4: Disk space usage with different number of tenants

percentage of non-null attributes to 10. The number of tenants vary from 20 to 160. We randomly generate private schemas for each tenant by SGEN and populate database by MDBGEN, which generates data by taking into account the private schema of each tenant. It can be clearly seen that M-Store outperforms STSI in terms of storage requirement. For a sparse shared table, M-Store only uses about 40% storage space of STSI to store the same number of tuples. With the number of tenants increases, the disk usage of STSI grows explosively, while the M-Store system shows a good scalability. The reason is as follows. In this setting, the average number of attributes each tenant configures is 10% of the total number of column in the shared table. STSI needs to consume large disk space to store Nulls. M-Store, on the other hand, eliminates the overhead of storing these Nulls by adopting BIT storage format. Therefore, it uses much less space to store the data from increasing number of tenants.

5.3.2 Throughput Test

We now investigate the performance of M-Store and STSI on concurrent operations. According to Section 5.2, three sets of query workload are generated by MQGEN: simple query set, analytical query set and update query set. In each set of query MQGEN generates one query file for each tenant, which contains 1000 queries. All queries are against indexed attributes. A multi-thread program (Worker) runs in a single PC to simulate a real multi-tenant environment. We set the number of concurrent database threads to 20, the worker picks queries from each tenant's query file into a query queue. Each thread runs a query from the query queue for one tenant, the results are the average of 20 measurements.

Performance of Simple Query

Figure 5.5 shows the performance of M-Store and STSI under simple query workload. As can be seen, there is a performance gap between M-Store and STSI in terms of system throughput. M-Store's throughput is about twice as high as STSI. Although fluctuations can be seen in the figure, M-Store's performance is not much affected by the number of tenants. The results indicate that M-Store is efficient and scalable under simple queries.

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, big B^+-tree indexes are built to index tuples from all tenants. Thus, the index lookup becomes inefficient with the number of tenants increase. In contrast to STSI, M-Store builds a separate index for each tenant. The cost of index scan

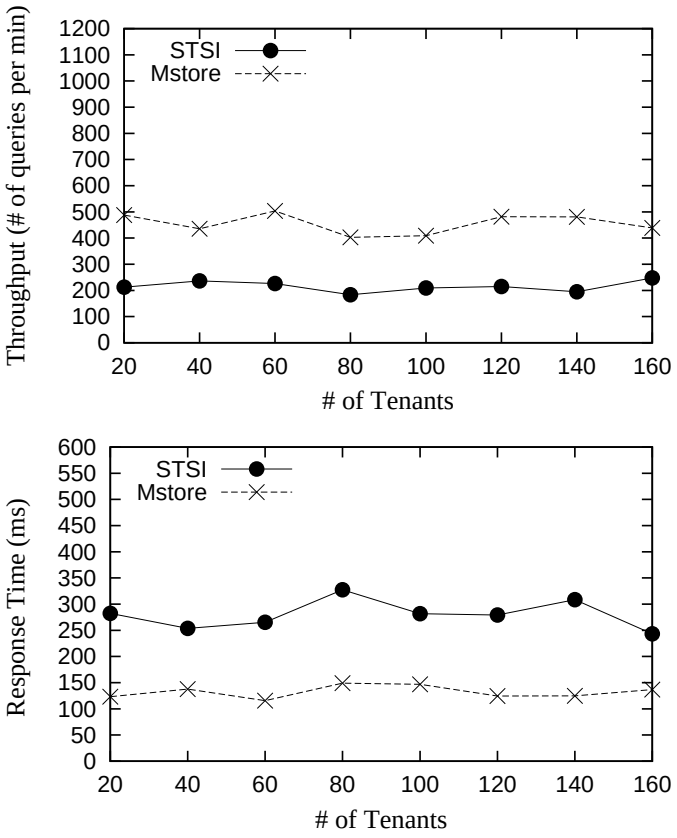

Figure 5.5: Simple Query Performance with Varying Tenant Amounts

is independent of the number of tenants in the system. Therefore, index lookup performance does not suffer much from the increasing number of tenants. Finally, in our experiments, the system performance are affected by the underlying space allocation of data files (e.g., fragments), and the diversity of queries, therefore the results of both the systems show a small fluctuation.

Performance of Analytical Query

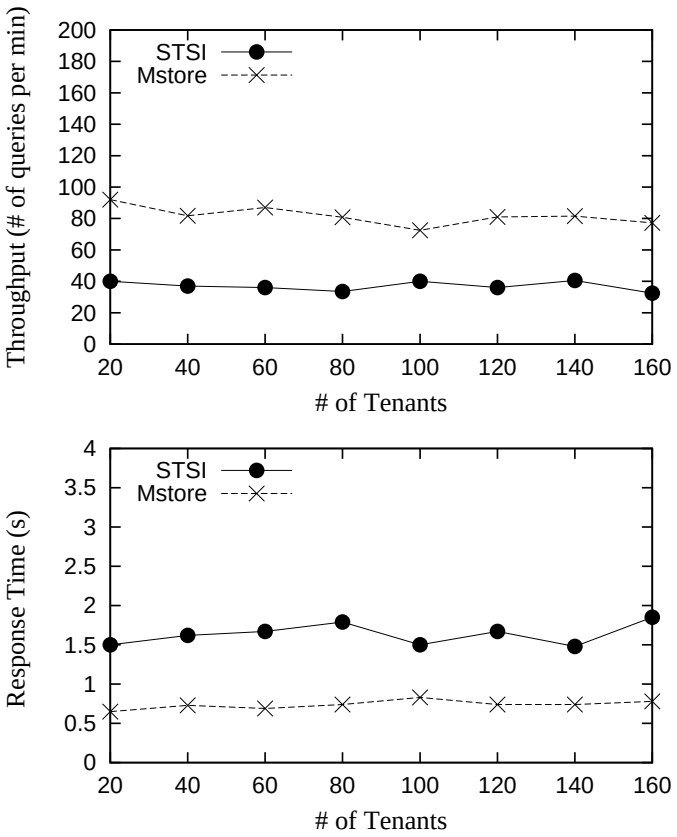

Figure 5.6: Analytical Query Performance with Varying Tenant Amounts

Figure 5.6 shows the throughput and response time under analytical query workload. Analytical queries perform projection-join-aggregation operations. The attributes in the join condition are the indexed attributes from two tables with primary/foreign key constraints. Similar to the results of simple queries in the

previous section, M-Store outperforms STSI and shows a good scalability. The system performance of M-Store are not much affected by the number of tenants.

Analysis on query plans reveals the reason of the performance gap. To perform analytical query $R \bowtie S$, MySQL uses the `tid` index to retrieve tuples in R belonging to the given tenant and then performs an index nested loop join to find the matching tuples in S. Compared to STSI, M-Store is more efficient in performing table scan and index lookup. First, M-Store adopts BIT storage format, which eliminates the overhead for storing Nulls from unused attributes, I/O cost of table scanning can be reduced. Second, MSI indexing technique builds small index for each tenant separately, thus reduces the cost of navigating index trees and scanning qualifying index entries in leaf pages.

Performance of Updates

Figure 5.7 depicts the throughput and response time under update query workload. In general, the performance of M-Store is better than STSI. For both systems, the throughput slightly reduces with the increasing number of tenants. Likewise, the response time increases with the number of tenants.

In our experiments, for update queries, insertion and deletion appear simultaneously. That is, each insertion follows with one deletion. The tuple being inserted and deleted are exactly the same tuple. Therefore, in each deletion, there is at least one tuple being deleted, and the query file can be continually run without significantly impact of the data scales. In such a case, the I/O cost of updates is mainly determined by the cost of navigating B$^+$-tree's internal nodes to locate the leaf page. Since M-Store maintains separated B^+-tree indices and each B^+-tree has a lower height than the BI index used by STSI, M-Store is more efficient in updating the index structure. On the other hand, M-Store adopts BIT storage format which

eliminates the overhead of storing Nulls for unused attributes and reduces the I/O cost of retrieving data records, therefore the throughput of M-Store is higher than STSI. With an increasing number of tenants in the system, the index structure gets larger and the system requires more I/O operation to locate a tuple and process a query, as a result the throughput slightly reduces. Finally, since updates access only a few nodes in the index and can finish very quickly, the system throughput for update queries is much higher than simple and analytical queries.

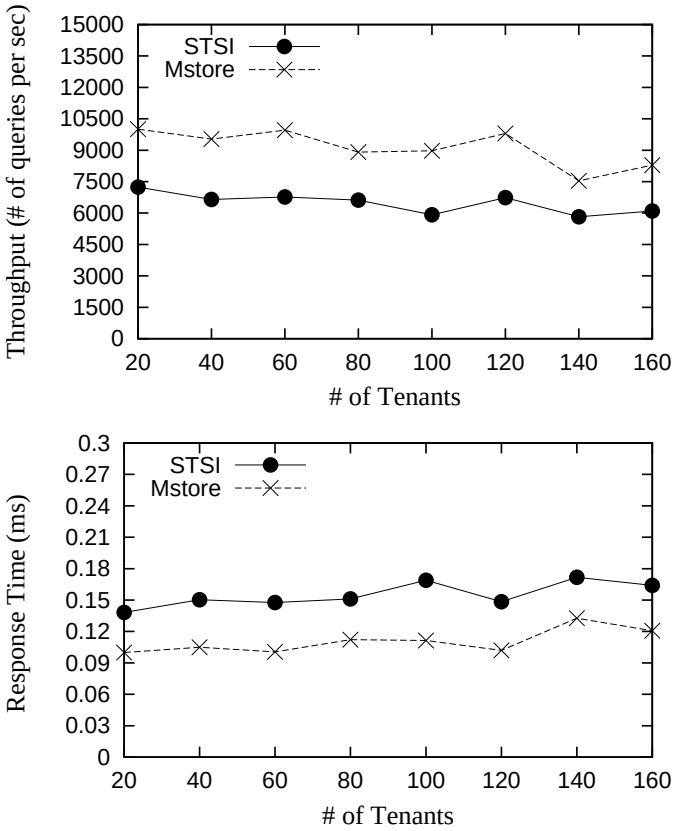

Figure 5.7: Update Query Performance with Varying Tenant Amounts

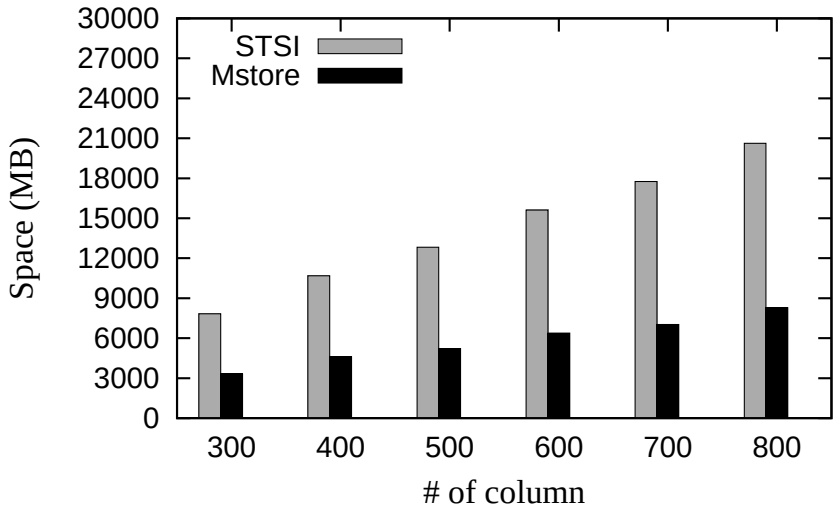

Figure 5.8: Disk space usage with different number of columns

5.4 Effect of Columns

In this section we investigate the scalability of M-Store system and STSI with an increasing number of columns. In multi-tenant database system, there are a large number of tenants and each of them only share a few common attributes, therefore we need to handle the situation that the base schema is very sparse and contains a large amount of configurable attributes. Therefore, it is of vital importance for the system to have the ability to support a sparse table with an increasing number of columns. As in previous experiments (Section 5.3), in this section we evaluate the scalability of M-Store and STSI in two aspects: storage space usage and system performance.

5.4.1 Storage Capability

Figure 5.8 illustrates the disk space usage of M-Store and STSI with the increasing number of columns. In this experiment, the number of columns in the shared table varies from 300 to 800. We use SGEN and MDBGEN to generate private

schemas and tuples for each tenant under different column settings. The shared table contains tuples from 80 tenants and the density of the table is 10%, i.e., the average number of non-null attributes occupies 10% of the entire table. The figure shows that under all column settings, M-Store takes up less storage space than STSI, and the gap between the two systems increases obviously with the number of columns. The reason is that M-Store adopts BIT storage format whereas STSI needs to assign large disk space to store Nulls in the shared table. This result indicates that our proposed M-Store system has the capability to support sparse table with large number of columns, which shows a good scalability in respect to the system storage.

5.4.2 Throughput Test

We now evaluate the effect of columns on the system throughput for both M-Store and STSI. We run MQGEN to generate three sets of queries. For each query set, MQGEN generates one query file with 1000 queries for each tenant. All queries in the same query file are of the same type. All queries are against indexed attributes and have an average selectivity of 0.3. The number of concurrent threads is 20, each thread runs queries from query queue for tenants. The results are the average over 20 measurements.

Figure 5.9 shows the system throughput and response time under simple query workloads when the number of columns in the shared table varies from 300 to 800. For all column settings, M-Store outperforms STSI. There is a slight decreasing for both the systems, however, the decreasing is insignificant and the systems are scalable under increasing number of columns. The reason is that the number of tuples in the system are fixed, index is kept unchanged as the number of columns increasing. BIT storage format mainly contributes to the advantage of the M-Store

system. With the number of columns increasing, STSI encounters a huge overhead of storing Nulls in the shared table, whereas BIT storage format enables M-Store save large space and greatly reduces the I/O cost of retrieving the result records.

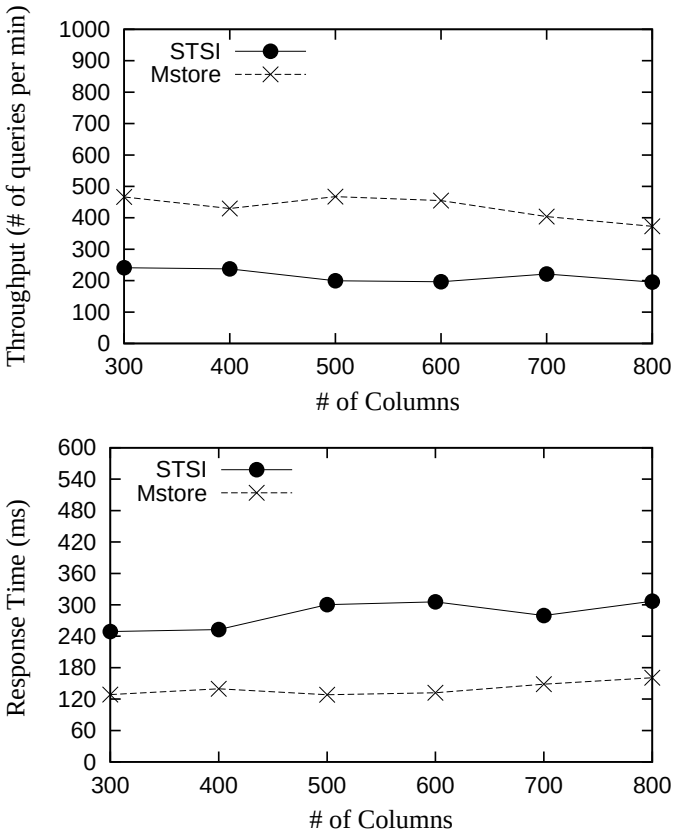

Figure 5.9: Simple Query Performance with Varying Column Amounts

As can be seen from Figure 5.9, the throughput and response time of both STSI and M-Store fluctuate within a certain range. The fluctuation of the throughput is because of the space allocation of data file and the diversity of queries, since the operating system may not store the data file in a sequential space and the queries are randomly generated from three tables with different data scale.

Figure 5.10 shows the system performance under analytical queries. The throughput of both M-Store and STSI suffers a degradation when the shared table contains an increasing number of columns. The reason is that for analytical queries, the I/O

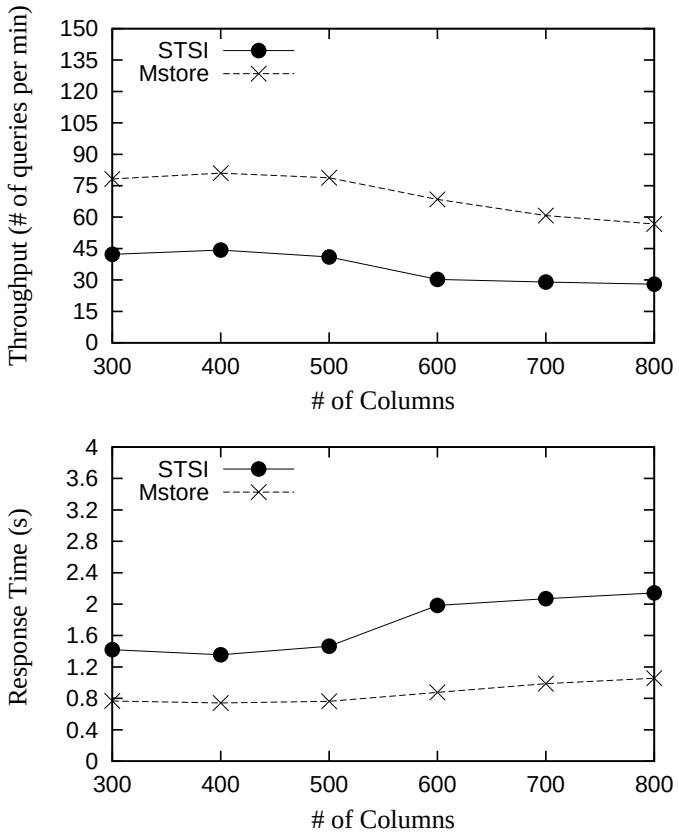

Figure 5.10: Analytical Query Performance with Varying Column Amounts

cost of index nested loop join includes the cost of scanning smaller relation and the cost of index lookup in large relation. With the increasing number of columns, the overhead of tuple scan is increased. However, since M-Store adopts BIT storage format, the I/O cost of scanning and retrieving data is reduced. Therefore the throughput of M-Store is better than STSI.

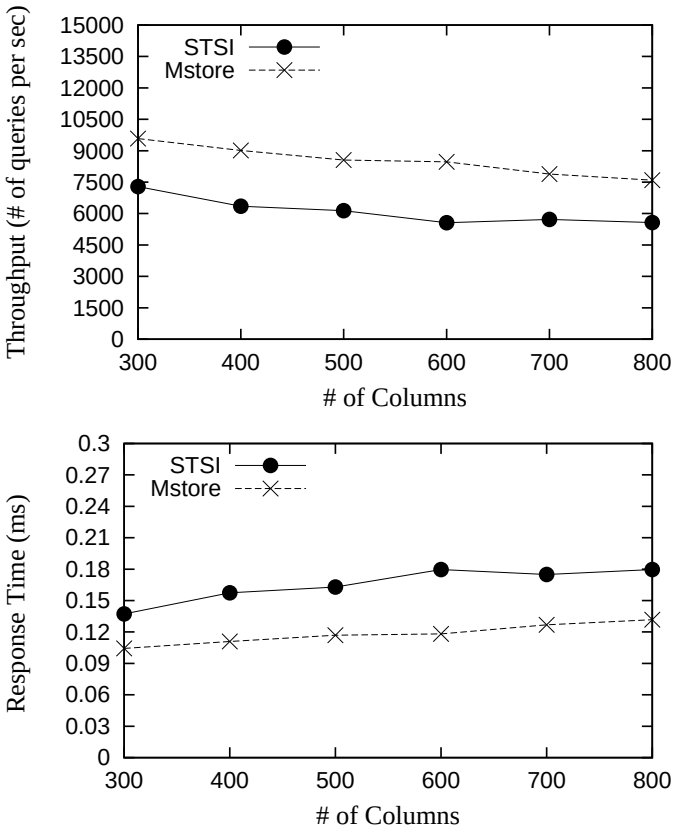

Figure 5.11: Update Query Performance with Varying Column Amounts

Figure 5.11 shows the result of update queries. It can be clearly seen that the throughput of both M-Store and STSI decreases with the number of columns. The reason is that with an increasing number of columns, the overhead of tuple update in the shared table increases correspondingly. For STSI, the system needs to insert Nulls for all unused attributes to the shared table. While for M-Store, BIT storage format is adopted, making the system eliminate the overhead of storing nulls for

unused attributes. Therefore the overall cost of tuple insertion and deletion is greatly reduced.

5.5 Effect of Mix Queries

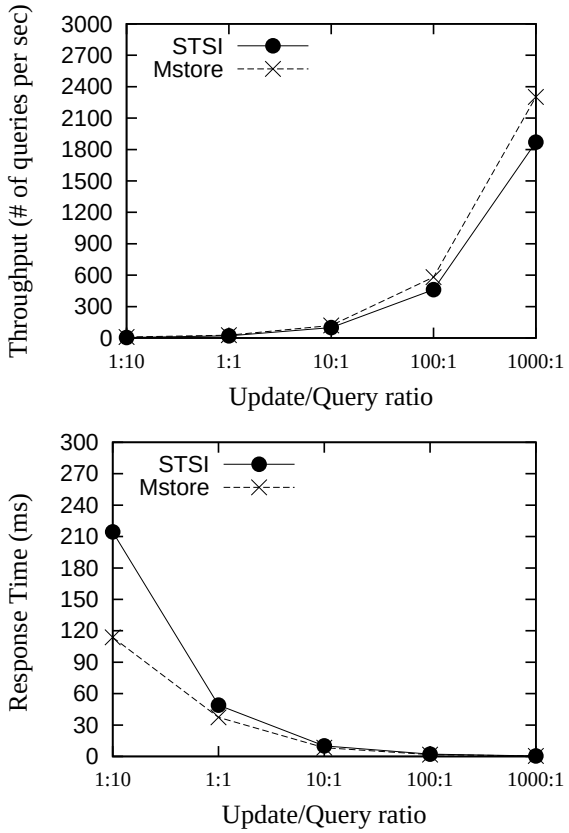

Figure 5.12: System Performance with different Query-Update Ratio

Figure 5.12 shows the throughput and response time with the query-update ratio varying from 10:1 to 1:1000 using 20 threads. The shared table contains 400 columns and stores tuples from 80 tenants. In real multi-tenant database applications, multiple tenants concurrently access the database system, some of them may wish to update their data, while others would like to execute search/query operations. To simulate such scenario, in this experiment we mix simple(range)

query workload and update queries to test the system performance under different query-update ratio.

As shown, the throughput of both the systems increase significantly with more updates and the response time decreases correspondingly. Generally speaking, M-Store and STSI use the same concurrency controlling strategy, where locks are only available on the whole tables and the roots of the indices. That is, the index must be locked as a whole and shared by all threads. In the M-Store implementation, although separated indices are built for tenants, they are considered as one index from the system's view and the lock is on the "root" of the whole index. Therefore, one tenant's updating could block another tenant's updating on the same table. In our experiments, an update query write-locks the table and the root of the index being accessed, all the concurrent requests for reading/writing the table are suspended. On the other hand, the range queries hold read lock on the corresponding index and the tuples being accessed, which do not prevent the other queries. Since update queries are single tuple insertion/deletion, they access only a few nodes in the index and can finish very quickly. Whereas the range queries have to traverse multiple paths and read many leaf pages of the B^+-tree (the selectivity of the queries is 0.3), which takes much longer time than updates. Therefore update workloads contribute more to the system throughput. As a result, when the percentage of updates in the workload increases, the throughput increases and the response time decreases accordingly. As the slow selection query locks all other update on the same table, the throughput is much slower than pure updates, even if only 0.1% of the queries are selections.

Figure 5.13 shows the effect of the number of threads under workload whose query-update ratio is 1:100. The number of threads varies from 4 to 64. The throughput of both the systems first increases and then reduces with increasing

number of threads. As shown in the figure, the throughput reaches the peak with about 32 threads for both the systems. Due to the limitation of computer hardware (e.g. number of CPUs, I/O bandwidth, etc.), the system has a fixed capacity to handle concurrent processes. If the number of active processes does not exceed the limitation, all the processes are executed in parallel, which leads to higher utility of the system resource. As a result, a significant improvement can be seen when the number of concurrent threads increase from 4 to 32. After that, the system hits thrashing point and the throughput starts to decrease. The increasing number of threads bring frequent context switch and compete for the system resources. As a result a slight decrease in the throughput can be seen in the figure when more than 32 concurrent threads run in the system.

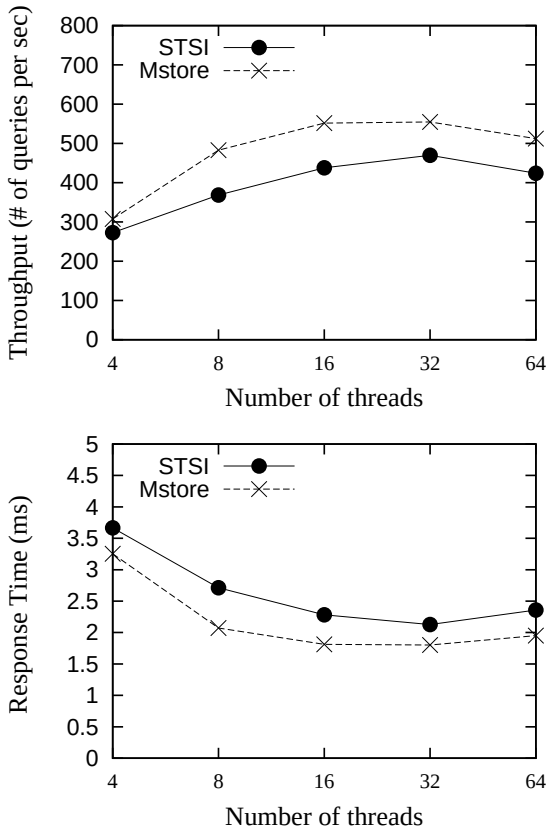

Figure 5.13: System Performance with different number of threads

Since the two systems use the same concurrency controlling strategy, the M-Store system is not given much advantage over MyISAM (used by STSI) on the concurrency. Both the systems have almost the same number of threads concurrently running. The improvement on updates/queries of M-Store results from the big savings on the I/O operation.

In the M-Store system, separated indices are built for tenants. The height of each index depends on the number of tuples that the tenant contains. On the other hand, STSI adopts multi-column index structure to handle the multi-tenant application, which combines the tenant-id and a data column to form a composite key. Queries like $tenant\_id = A$ and $data > B$ can be efficiently handled by this structure. However, this structure requires more I/O operation to locate a single tuple than the M-Store. There are two reasons: firstly, indexing all tenants' data (BI) increases the total tuple number and the height of the tree; secondly, the composite key is longer, and thus each node has lower fan-out. Therefore, M-Store's index (MSI) is more efficient.

The savings on accessing and scanning the data file is also significant. M-store uses BIT storage format, which only takes up 40% storage space to store each tuple, which means fetching/writing a tuple requires 40% I/O of STSI.

In total, with M-Store, less number of I/O operations are required to process an update/query. Almost all updates/queries are I/O bounded. Although both systems allow same number of concurrent queries, M-Store system processes more quires with the same I/O bandwidth, which can achieve a significant improvement in the throughput.

5.6 Summary

This chapter presents the empirical study of the proposed M-Store system. First, we develop DaaS benchmark to evaluate the performance of multi-tenant database system. DaaS benchmark comprises five modules: a configurable base schema, a private schema generator (SGEN), a data generator (MDBGEN), a query workload generator (MQGEN), and a multi-thread client (Worker). With DaaS benchmark, we can set up the experiments and simulate the multi-tenant environment.

Next we empirically evaluate the scalability of the M-Store system. In our experiments, scalability is defined as the system ability to handle growing amounts of data without much performance degradation. We examine the scalability of M-Store and STSI from two aspects: the effect of tenants and the effect of column amounts. For each group of experiments, we evaluate the proposed BIT storage format and MSI indexing by measuring the disk space usage and system throughput. Finally, we test the effect of mix query/updates to the system performance.

By using the BIT storage format and MSI indexing scheme, M-Store outperforms STSI in terms of disk space usage and system throughput in all experiments. The number of tenants does not affect the performance of the M-Store significantly since it builds separated index for each tenant. When the number of columns in the shared table increases, both M-Store and STSI incurs a degradation since the I/O cost of retrieving results increases. The overall results show that our proposed M-Store system is an efficient and scalable multi-tenant database system.

CHAPTER 6

Conclusion

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 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 experimental results 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. M-Store shows a good scalability in handling growing amounts of data.

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 [30], 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. We will study how to get the optimizer to generate best query plans for multi separated indexes in the M-Store system.

BIBLIOGRAPHY

[1] Amazon simple storage service. http://aws.amazon.com/s3/.

[2] Amazon simpledb. http://aws.amazon.com/simpledb/.

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

[4] Architecture strategies for catching the long tail. http://msdn.microsoft.com/en-us/library/aa479069.aspx/.

[5] Clouddb. http://clouddb.com/.

[6] Cnet networks. http://shopper.cnet.com/.

[7] Db2 database for linux,unix, and windows. `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp/`.

[8] Google earth. http://earth.google.com/.

[9] Google finance. http://www.google.com/finance/.

[10] Indexed views in sql server 2000. http://www.sqlteam.com/article/indexed-views-in-sql-server-2000/.

[11] The long tail. http://www.wired.com/wired/archive/12.10/tail.html/.

[12] Monetdb: Query processing at light speed. http://monetdb.cwi.nl//.

[13] Multi-tenant data architecutre. http://msdn.microsoft.com/en-us/library/aa479086.aspx/.

[14] Postgresql. http://www.postgresql.org/.

[15] Sybase iq columnar database. http://www.sybase.com/products/datawarehousing/sybaseiq/

[16] Tpc-c. http://www.tpc.org/tpcc/.

[17] Tpc-h. http://www.tpc.org/tpch/default.asp/.

[18] Vertica-column oriented analytic database. http://www.vertica.com/.

[19] Community systems research at yahoo! *SIGMOD Record*, 36(3):47–54, 2007.

[20] Daniel J. Abadi. Column stores for wide and sparse data. In *CIDR*, pages 292–297, 2007.

[21] Daniel J. Abadi, Adam Marcus 0002, Samuel Madden, and Kate Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *VLDB J.*, 18(2):385–406, 2009.

[22] Daniel J. Abadi, Adam Marcus 0002, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[23] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[24] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*, pages 671–682, 2006.

[25] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD Conference*, pages 967–980, 2008.

[26] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, pages 466–475, 2007.

[27] Eugene Agichtein and Luis Gravano. Querying text databases for efficient information extraction. In *ICDE*, pages 113–124, 2003.

[28] Rakesh Agrawal, Amit Somani, and Yirong Xu. Storage and querying of e-commerce data. In *VLDB*, pages 149–158, 2001.

[29] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[30] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD Conference*, pages 1195–1206, 2008.

[31] Stefan Aulbach, Dean Jacobs, Alfons Kemper, and Michael Seibold. A comparison of flexible schemas for software as a service. In *SIGMOD Conference*, pages 881–888, 2009.

[32] Jennifer L. Beckmann, Alan Halverson, Rajasekar Krishnamurthy, and Jeffrey F. Naughton. Extending rdbmss to support sparse datasets using an interpreted attribute storage format. In *ICDE*, page 58, 2006.

[33] Peter A. Boncz and Martin L. Kersten. Mil primitives for querying a fragmented world. *VLDB J.*, 8(2):101–119, 1999.

[34] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[35] André B. Bondi. Characteristics of scalability and their impact on performance. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, New York, NY, USA, 2000. ACM.

[36] AM Deshpande CA Brandt and et al. Traldb: A web-based clinical study data management system. In *AMIA Annu Symp Proceedings*, pages 334–350, 2003.

[37] K. Selçuk Candan, Wen-Syan Li, Thomas Phan, and Minqi Zhou. Frontiers in information and software as services. In *ICDE*, pages 1761–1768, 2009.

[38] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[39] Eric Chu, Jennifer L. Beckmann, and Jeffrey F. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *SIGMOD Conference*, pages 821–832, 2007.

[40] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[41] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In Shamkant B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 268–279. ACM Press, 1985.

[42] Conor Cunningham, Goetz Graefe, and César A. Galindo-Legaria. Pivot and unpivot: Optimization and execution strategies in an rdbms. In *VLDB*, pages 998–1009, 2004.

[43] Jeff Edmonds, Jarek Gryz, Dongming Liang, and Renée J. Miller. Mining for empty spaces in large data sets. *Theor. Comput. Sci.*, 296(3):435–452, 2003.

[44] Ronald Fagin, Alberto O. Mendelzon, and Jeffrey D. Ullman. A simplified universal relation assumption and its properties. *ACM Trans. Database Syst.*, 7(3):343–360, 1982.

[45] Daniela Florescu, Daniela Florescu, Donald Kossmann, Donald Kossmann, and Projet Rodin. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. Technical report, 1999.

[46] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. Integrating keyword search into xml query processing. In *BDA*, 2000.

[47] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[48] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[49] Hakan Hacigümüs, Sharad Mehrotra, and Balakrishna R. Iyer. Providing database as a service. In *ICDE*, page 29, 2002.

[50] Alan Halverson, Jennifer L. Beckmann, Jeffrey F. Naughton, and David J. Dewitt. A comparison of c-store and row-store in a common framework. Technical report, University of Wisconsin-Madison, 2006.

[51] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, 2006.

[52] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[53] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[54] Setrag Khoshafian, George P. Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*, pages 636–643. IEEE Computer Society, 1987.

[55] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD Conference*, pages 121–132, 2006.

[56] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

[57] Roger MacNicol and Blaine French. Sybase iq multiplex - designed for analytics. In *VLDB*, pages 1227–1230, 2004.

[58] David Maier, Jeffrey D. Ullman, and Moshe Y. Vardi. On the foundations of the universal relation model. *ACM Trans. Database Syst.*, 9(2):283–308, 1984.

[59] Beng Chin Ooi, Bei Yu, and Guoliang Li. One table stores all: Enabling painless free-and-easy data publishing and sharing. In *CIDR*, pages 142–153, 2007.

[60] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[61] Raghu Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 1998.

[62] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. *VLDB J.*, 12(2):89–101, 2003.

[63] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, pages 140–, 1998.

[64] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2(2):129–138, 1999.

[65] Nick Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.

[66] Shepherd S. B. Shi, Ellen Stokes, Debora Byrne, Cindy Fleming Corn, David Bachmann, and Tom Jones. An enterprise directory solution with db2. *IBM Systems Journal*, 39(2):360–, 2000.

[67] Aameek Singh and Ling Liu. Sharoes: A data sharing platform for outsourced enterprise storage environments. In *ICDE*, pages 993–1002, 2008.

[68] Radu Sion. Query execution assurance for outsourced databases. In *VLDB*, pages 601–612, 2005.

[69] Michael Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989.

[70] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[71] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979.

[72] Eric TenWolde. Worldwide software on demand 2007-2011 forcast: A preliminary look at delivery model performance. In *IDC Report*, 2007.

[73] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. Monetdb/x100 - a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.