Abstract—Software-as-a-Service (SaaS) is now a hot issue in both academia and business. Multi-tenant data storage as a fundamental component of a SaaS system draws much attention. The need of scalability and multi-tenant support in SaaS makes traditional RDBMS unappealing and calls for a better data storage solution. In this paper, we categorize some proposed solutions and evaluate the performance of a Native XML Database (NXD) solution implemented on Berkeley DB XML. Experimental results show that the opportunity of applying NXD in SaaS has not yet arrived.

Index Terms—Software-as-a-Service, Multi-Tenancy, Native XML Database

I. INTRODUCTION

Software-as-a-Service (SaaS) has brought several new challenges to Relational Database Management System (RDBMS), which are widely used in most kinds of software and information system[1]. Although it’s believed that the ideal multi-tenant data storage for SaaS has not appeared yet[2], people peruse researches from different aspects and it’s necessary to compare and evaluate those proposed solutions.

II. MULTI-TENANT DATA STORAGE

The number of tenants sharing a same SaaS software can scale from dozens to thousands or even more, these tenants may have their particular needs. These scenarios bring two major challenges to traditional relational database: whether the database can afford the increase of both data and request accompanied with the growth of tenants and how can the database meet the particular needs of one tenant efficiently and safely without affecting the others. Many solutions aim at solving these problems have been proposed.

A. RDBMS-based Solution

Most solutions focus on mapping relational data stored in RDBMS to logical view of each tenant so as to fulfill the needs of SaaS Software. These solutions can be divided into four categories according to how they deal with the relationship between the shared data and customized data, the former is unified among all users while the latter is not[2][3].

1) Private Table Layout: These approaches, including Independent Database and Independent Instance (IDII) and Independent Tables and Shared Instance (ITSI), separate all tenants’ data in their private tables (or database instance).

2) Shared Table Layout: These approaches store all tenants’ data in shared tables, where the tuples are stored with a additional tenantid attribute. Tenant-customized data are appended to the tuples by approaches like pre-allocated attributes[4] and XML attribute[5]. Since the increasing tenant number only means scaling of data size, layouts of this kind have good performance on scalability. However, it also suffers several problems, the approach of pre-allocated attributes brings massive NULL values into database[6], and dealing with XML may influence query performance[2].

3) Extension Table Layout: These approaches are hybrid of the last two: using shared base table to store shared data of all tenants and separated extension tables to store tenant-customized data. Base tables and extension tables are connected by rowid. Data model of a extension table may be relational or Key-Value pair based[3], the flaw of scalability in private table layout still exists in the former[4] while the needs of additional metadata and complicated data reconstruction seems unavoidable in the latter.

4) Pivot Table Layout: These approaches split data rows into single data items, each of which is stored in a tall narrow table along with an identifier for its column. The identifier is critical in reconstructing data. Since DDL are not needed in any customizations, these approaches have high scalability. However, they suffer from complexity of data reconstruction and redundancy caused by introducing identifiers. The above four categories and their variants take great part of the whole researches in this area. BechMann et al. (2006)
proposed storing sparse columns with Key-Value pairs [6]. M. Hui et al. (2009) proposed an approach named Bitmap Interpreted Tuple (BIT) which append tenants information to table structure to avoid storing redundant NULL values in shared table layout[7]. Foping et al. (2009) proposed applying XML data in extension table layout[8]. Aulbach et al. (2008) proposed an improved solution based on pivot table layout[4]. Meanwhile, some researchers do work of multi-tenant data storage from other aspects like NoSQL and Native XML Database (NXD)[9][13][10].

B. NXD-based Solution

After Bourret (2003) giving the primitive definition of Native XML Database (NXD)[11], it draws much attention. NXD deal with XML natively rather than transform it to relational data model or object-oriented model, which avoids the cost of performance during model transformation. It can efficiently support the features of XML documents like self-describing, semi-structure and orderliness; it also support XML query language like XQuery rather than transform it into SQL or OQL(Object Query Language)[12]. Inspired by the experience of enabling XML support in relational database, W. J. Xu et al. (2010) put forth the idea of taking Native XML Database as multi-tenant storage[13], but its performance in multi-tenant scenario has not been sufficiently evaluated yet. In this paper, we try to show it by a series of experiments, which basically follow the methods of [2].

III. EXPERIMENTS OF NXD AS MULTI-TENANT STORAGE

Since the comparison made by Aulbach et al.(2009) had comprehensively covered all other solutions[2], we only evaluate the performance of NXD solution proposed in [13]. In the experiments, we implement the solution on Berkeley DB XML (BDB XML), and as a comparison, we implement shared table layout solution based on DB2 pureXML technology.

A. Experimental Settings

Since there is no standard data set for this task, we generate our own multi-tenant data set from table customer in TPC-H database[14]. In DB2 solution, we append attribute tenantid for identifying tenants and an additional XML column for storing customized data; in BDB XML solution, we implement a XML schema equivalent to shared table layout. There are 300,000 records in the data set shared by 100 tenants, the number of customized attribute of each tenant are chosen randomly in range [0,10]. A typical record in the two solutions are shown in Fig.1.

In the experiments, we simulate a real multi-tenant scenario by sending query and update requests from many tenants concurrently, and then evaluate the solutions by analysis the response time data captured during those experiments. The requests can be divided into two categories: query requests and update requests.

**Query requests:** choose a set of attributes from database. Examples of such requests are shown as follows (categorized by whether customized attributes are involved):

- query without customized attributes, namely Q1 (SQL, same below):
  ```sql
  SELECT custkey, name FROM customer WHERE tenantid = 10 AND custkey > 9000;
  ```

- query outputting customized attributes, namely Q2:
  ```sql
  SELECT name, xmlquery (' $c/ext/age' passing extension as "c") as age FROM customer WHERE tenantid = 16
  ```

- query using customized attributes as filter condition, namely Q3:
  ```sql
  SELECT name from customer WHERE xmlexists (' $c/ext[email="a@bc.com"]' passing customer.extension as "c") AND tenantid = 68
  ```

**Update requests:** insert records into database, update existing records or delete records from database. Examples are as follows:

- insert records, namely U1:
  ```sql
  INSERT INTO customer (tenantid, custkey, name, extension) VALUES(101,3001, 'J Wang', '<ext>
  <email>w@bc.com</email></ext>')
  ```

- update records, namely U2:
**UPDATE** customer SET extension = 
(xmllist('document ','<ext>
<fax>40877766</fax>
<email>abc@example.com</email>
'</ext>')
WHERE tenantid = 101
AND custkey = 3001
AND xmlexists (
'${c/ext[email="w@bc.com "]',
PASSING customer.extension as "c")

- delete records, namely U3:

**DELETE FROM** customer
WHERE tenantid = 101 AND custkey = 301

In the experiments, we put the multi-tenant database server and clients in "client/server" model. Clients, which are written in Java, are designed to be able to simulate many tenants. Clients interact with server through standard JDBC (for DB2 solution) or BDB XML Java API (for BDB XML solution). Every simulated tenant would submits all the six kinds of requests mentioned above to the server and records the execution time. Both server and four clients in the experiments are run on Dell OPTIPLEX 360 (CPU: E5300 2.6GHz*2, RAM: 2G), all these machines are connected by 100Mb/s LAN. Since BDB XML do not provide any remote access methods, we simulate remote access by file sharing.

**B. Indexes in NXD**

Noticing that the query performance of NXD depends on indexes, we add proper indexes to nodes like tenantid, custkey, email and age in order to optimize the performance as possibly as we can. A comparison of response time to requests before and after adding indexes to BDB XML solution shows that querying performance improves significantly (almost ten times faster than before) but meanwhile updating performance slightly drops after adding indexes, which we think is caused by the existence of indexes. The data in this comparison are shown in Fig.2.

**C. Experimental Results**

According to real scenarios of SaaS system, We divide the experiments into three groups by the number of clients and the number of simulated tenants per client: single client with single tenant, single client with multi tenants, multi client with multi tenants. Each group performs four rounds of experiments, in each round every tenant submits 600 requests(100 each type). We collect average response time as the indicator of evaluation. The comparison between the two solutions are shown in Fig.3.

When dealing with query requests without customized data(Q1), DB2 solution performs much more better then BDB XML, actually it’s almost ten times faster. The advantage of the former probably attributes to the technology accumulation of relational databases. But when the query results contain customized data (Q2, in this case, XML content), relational database has to perform model transformation, which makes it much slower than native XML database. What shock us is when processing queries that use customized attributes as filter condition (Q3), DB2 prevails again. It indicates that the XML query engine and query optimization mechanism of DB2 is more efficient. When it comes to updating queries, BDB XML performs better in single tenant scenario but worse in multi-tenant scenario, except for insert operation. In addition, we notice that the chance of encountering dead lock when updating or deleting records in BDB XML get increased as the number of concurrent tenants grows. In group 3, about 5% of overall transactions have to be re-submit.

As we see it, NXD systems such as BDB XML may show their natural advantage when processing tenant-customized data (in this case, XML), but compared with commercial relational database systems, NXD systems still suffer from many flaws: The performance of processing complex query is not good enough, current access control mechanism can’t sustain concurrent accesses very well, lack of remote data access module, etc. All these flaws reveal that the opportunity of applying native XML database in SaaS has not yet arrived.

**IV. Conclusion**

In this paper, we discuss several kinds of multi-tenant data storage solutions in SaaS model and conduct a series of performance experiments of Native XML Database solution, using shared table layout solution implemented on IBM DB2 as a comparison. Our experiments show that currently there are still several problems should be solved before we can safely and efficiently applying NXD system in SaaS. The technology accumulation of RDBMS in business makes the
solutions based on relational database more efficient, especially when facing concurrent accesses. However, the flexibility of data storage in NXD gives it natural advantage of processing tenant-customized data (confirmed by our experiment of Q2), if it can make more progress in aspects like query executing optimization, concurrent access control and remote access, it is possible to apply it in SaaS.

REFERENCES