RC24426 (C0711-037) November 19, 2007
Computer Science

IBM Research Report

Data Integration and Composite Business Services, Part 3, Building a Multi-Tenant Data Tier with with Access Control and Security

Chang Jie Guo
IBM Research Division
China Research Laboratory
Building 19, Zhouguancun Software Park
8 Dongbeiwang West Road, Haidian District
Beijing, 100094
P.R.China

Mary Taylor
IBM Software Group

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Data Integration and Composite Business Services, Part 3, Building a Multi-Tenant Data Tier with Access Control and Security

November, 2007

Authors:
Chang Jie Guo
Mary Taylor

This article explores how the capabilities of DB2 address the issues of data architecture and data security in a Software as a Service (SaaS) implementation. The primary concern of a SaaS service consumer is data security. The primary concern for a SaaS service provider is a data architecture that provides cost effective data administration and maintenance while addressing the service consumer's concern at the same time.

DB2 Viper provides XML storage capabilities that simplify the configurability and extensibility of the data. DB2 Viper also provides row level security authorization. DB2 v8 and higher provides partitioning and backup technologies that address scalability and maintenance concerns. In this article, we will show how these DB2 features can be exploited to build a multi-tenant data architecture. This data architecture will address the primary concerns of both SaaS service consumers and service providers.

Introduction

There are varying degrees of data isolation for a SaaS application that range from an isolated environment to a totally shared environment. Implementations along this spectrum include:

- Totally isolated: separate databases per tenant
- Partially shared: shared database, separate schema
- Totally shared : same database, same schema

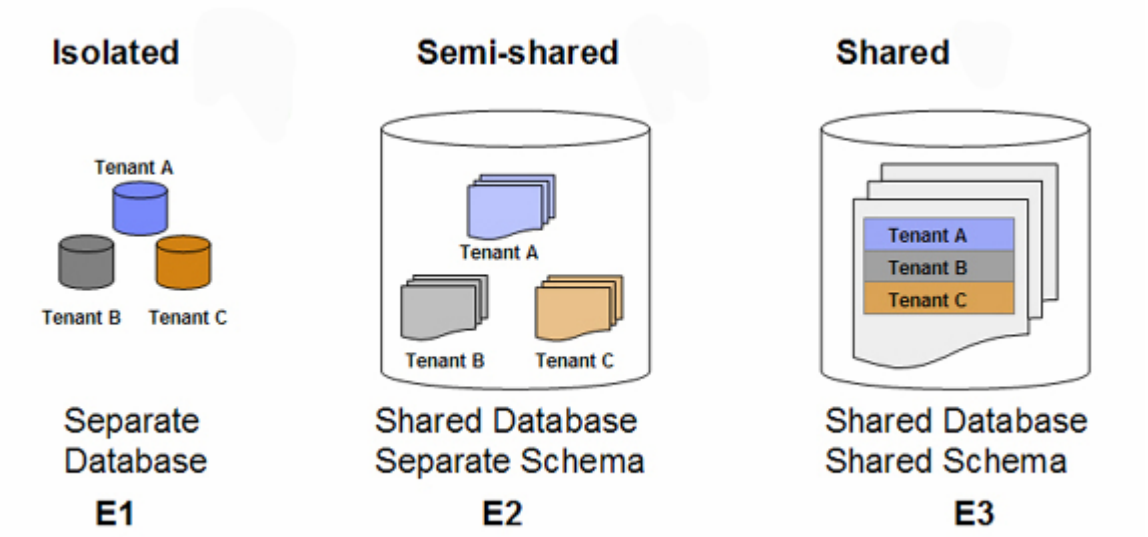

Figure 1 Isolated vs Shared Data Environments

Figure 1 above compares the spectrum of isolation options in terms of ease of configurability.

- On the left we show an isolated environment called E1. Each tenant has their own database. E1 offers the most flexibility in terms of configuration as the tables are designed with custom columns to support the single tenant.
- In the middle we show a shared database with separate schemas called E2. E2 offers a lesser degree of data isolation than E1. E2 offers the same ease of configurability as E1 by using custom columns.
- On the right we show a shared schema environment called E3. This approach has typically been viewed as the most difficult to configure, requiring the use of either name/value pair columns or pre-allocated fields to satisfy unique tenant requirements.

This article will focus on the third implementation, a shared database and shared schema in which a given database table can include records from multiple tenants stored in any order. In this approach we will use a Tenant ID column in every table to associate each row with the appropriate tenant. We will discuss how DB2 capabilities address the extensibility, scalability, maintenance and security concerns in this type of environment.

Overview of Approach

There are a number of considerations when implementing a multi-tenant application regardless of whether you are implementing in an isolated or shared environment. Security, scalability and performance, maintenance and extensibility are all primary concerns of both the SaaS provider and subscriber.

1. **Data Security**
   Data security refers to the manner in which you ensure that each tenant only has access to their own data. In an isolated environment, each tenant has their own database, and tenant access is administered at the database level. But as you move towards a shared environment where multiple tenant's data resides in either the same database or the same tables, then you need to consider additional security mechanisms such as row level access to data, or tenant views on the data. In this article we'll talk about the row level access control that DB2 provides.

2. **Scalability and Performance**
   In multi-tenant scenarios, the shared database architecture design should be scaled up and out in a cost-effective manner. As the number of tenants increase incrementally, the system performance should remain constant. Furthermore, the architecture should provide acceptable isolation capabilities to prevent performance, availability and management interference among tenants. This article will describe some thoughts on this aspect by leveraging some of DB2 Viper's new features.

3. **Maintenance**
   Shared infrastructure also introduces maintenance issues, especially regarding the tenant-oriented backup and restore. The traditional DBMS level support only focuses on some aspects of this capability, such as database and table-space maintenance, and doesn't consider the new dimension of tenant. This article will address this topic and give a practical approach.

4. **Extensibility**
   The fourth topic to be covered in this article is database extensibility in a shared environment. One of the benefits of implementing an isolated data tier, or even a shared data tier with separate schema, is that database changes for one tenant can be made directly to the database tables without consideration of how those changes impact other tenants. The perceived drawback to implementing a shared schema data tier is the complexity of implementing tenant data customization. What we'll describe in this article how the use of DB2 Viper's pureXML simplifies this configurability so that its ease of use is comparable to the use of custom columns in an isolated data tier.

1. Data Security

In a multi-tenant context, the security isolation mechanism is very important in preventing the data of one tenant from being accessed illegally by other tenants. Considering the shared schema/tables pattern, there are generally two kinds of access control isolation approaches, implemented from the application and the DBMS level respectively.

1.1 Application Level Data Access Control Isolation

When one tenant tries to access shared tables, a common database account is delegated to handle this database access request. The delegated account is shared by all customers of the tenant and owns the privileges to access all data for a particular tenant in the shared tables. The delegated account is unique amongst the different tenants. A sub-clause like "*where tenant_id = xyz*" needs to be inserted into the SQL statement, to filter out data records not belonging to the current tenant. For example, for an application query such as "*Select name, address, phone from customerData*", the query would need to be re-written as "*Select name, address, phone from customerData where tenant_id='xyz'*".

1.1.1 Limitations of Application level data access control for multi-tenancy

Although easy to implement, application level access control has some potential security risks. For example, SQL injection, which is a technique that exploits a security vulnerability occurring in the database layer of an application, may occur when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. In the multi-tenant context, a well-designed user input may bypass the sub-clause used to filter out other tenants' data. A typical example is as follows:

Original SQL Statement:

*SELECT * FROM SALES_ORDER WHERE TENANTID = 'xyz' AND SOID = '" + Order_Id + "'*

If the "*Order_Id*" variable is crafted in a specific way by a malicious user, the SQL statement may do more than the code author intended. For example, setting the "*Order_Id*" variable as:

123' or '0'='0

Then the new SQL statement will become:

*SELECT * FROM SALES_ORDER WHERE TENANTID = 'xyz' AND SOID = '123' or '0'='0'*

Obviously, in this case, all tenants' orders residing in the shared table will be accessed illegally.

1.2 DBMS Level Access Control Isolation via LBAC

LBAC (Label-Based Access Control) is a new security feature provided by the DB2 Viper release. It lets you decide exactly who has read and write access to individual rows and individual columns, and thus greatly increases the control you have over who can access your data. LBAC controls access to table objects through the use of security labels

and security policies. Security labels identify criteria that are used to decide who should have access to particular pieces of data in the database. Security labels can be associated to specific rows and columns in the database, and are granted to users to allow them to access that data. Security policies describe the criteria that will be used to decide who has access to what data. Users attempting to access an object must have its security label granted to them. When there's a match, access is permitted; without a match, access is denied.

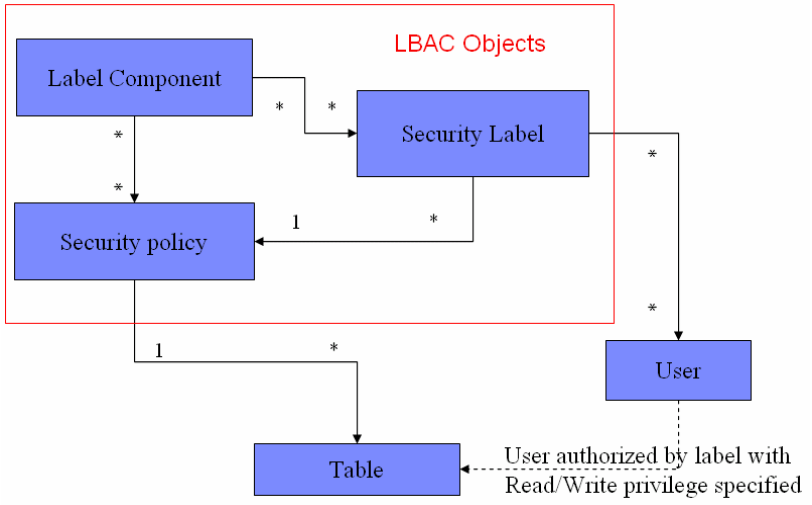

Figure 2 Object Relationships of LBAC

Figure 2 shows the basic object relationship in LBAC. There are three types of security labels which are associated and granted to three different database objects respectively – row, column and user. A security label is further composed of one or more security label components. The protected table should include a column of type *DB2SECUIRTYLABEL* that holds the security label and is used to attach the security policy to the table. Users will be granted access to the appropriate security labels to access the protected table in runtime. For the details of LBAC, please refer to the paper *DB2 V9 Administration Guide: Implementation* & *DB2 Label-Based Access Control: A Practical Guide.*

In the remainder of this section, we will focus on how to apply LBAC in a multi-tenant scenario to protect each tenant's data from illegal access, while preserving the flexibility of allowing multiple tenants access to data stored in the shared tables restrictively.

To avoid the potential SQL injection introduced previously, we need to set up a dedicated database account for each tenant. This means that data access requests from different tenants for those shared tables will be handled using different database accounts. In this way we can take advantage of the LBAC mechanism to guarantee the DBMS level security access control isolation amongst the tenants.

Figure 3 demonstrates a basic approach to supporting a multi-tenant scenario. An LBAC rule set is a predefined set of rules that are used when comparing security labels. When the values of a two security labels are being compared, one or more of the rules in the rule set will be used to determine if one value blocks another. There is a single rule set

provided in DB2 Viper called DB2LBACRULES. Within this rule set there are 16 pre-built security label components and each has 64 independent elements. A security policy named "*MTSecurityPolicy_Sales_Order*" is created for the shared table "*SALES_ORDER*", with a set of security labels inside. Each security label includes one element selected from one of the 16 label components. To guarantee tenant access control isolation, the same element shouldn't be referenced by more than one label.

When a tenant is on-boarding, the operator may simply select one un-used label and grant it to the database account of the tenant. For example, in the scenario displayed in figure 3 below, the following commands could have been issued by the database administrator when the TenantA, TenantB and TenantX were on-boarded:

"GRANT SECURITY LABEL MTSecurityPolicy_Sales.0001 to USER TenantA for all access"
"GRANT SECURITY LABEL MTSecurityPolicy_Sales.0002 to USER TenantB for read access"
"GRANT SECURITY LABEL MTSecurityPolicy_Sales.1024 to USER TenantX for delete access"

When using LBAC, SQL injection will never result in security issues amongst tenants because the database manager is the one controlling cross-tenant data access at the DBMS level rather than having the security controlled at the application layer.

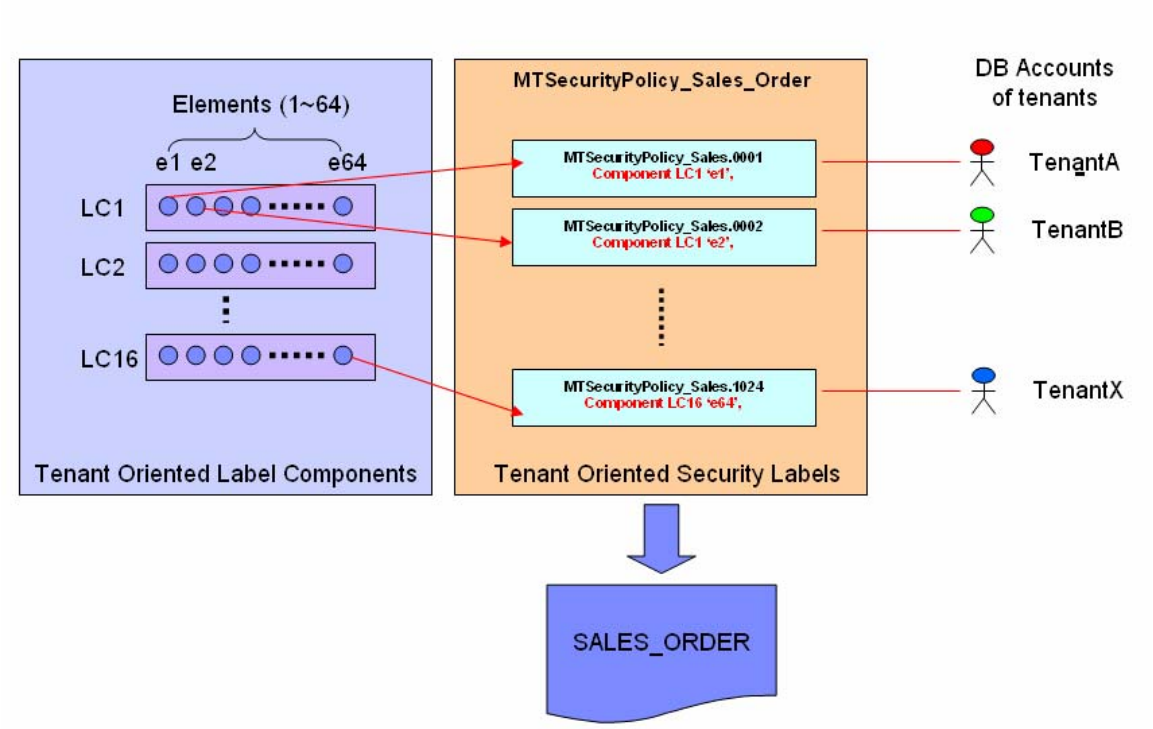

Figure 3 A Simple Multi-tenant Support LBAC Mechanism

1.2.1 Limitations of DB2 LBAC for multi-tenancy

In the current version of LBAC provided in DB2 Viper, no more than 16 security label components can be specified for a security policy, and each security label component can contain no more than 64 elements. Therefore, if each security label uses one element to

isolate each tenant as shown in Figure 3, the maximum number of tenants that can be supported will be 1024 (16*64).

To get around this limitation, we may choose more than one element as "union keys" to compose the tenant isolated security labels. Certainly, we will still need to guarantee that there are no two labels owning the same set of elements. As shown in Figure 4, if each security label includes two elements, the maximum number of tenants supported will reach 523,776.

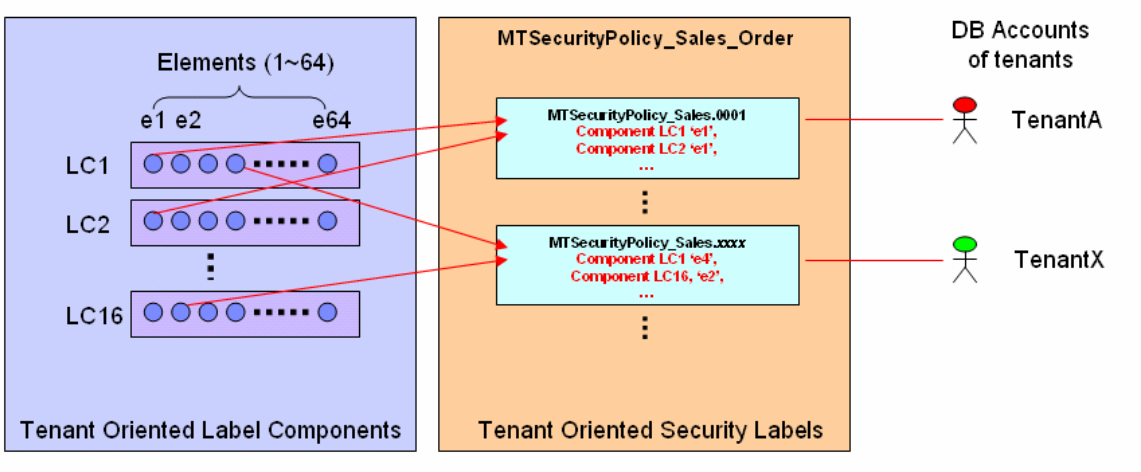

Figure 4 Tenant Isolated Security Labels with Two Elements

2. Scalability and Performance

Cost effective scalability is very important for multi-tenant applications. In an ideal situation, the maximum number of tenants supported should increase in direct proportion to the increase in resources, while still keeping the performance metrics of each tenant in a pre-defined and acceptable level.

There are two kinds of typical approaches to scaling:

- *Scale Up* (Vertical scaling) via adding more resources (CPU, memory, disks IO etc.) to the existing machines. This is an easy-to-use and manageable approach. However, it may not provide linear scalability. As you add resources there is some overhead in resource management that limits the scalability of single systems.
- *Scale Out* (Horizontal scaling) via adding additional machines to the existing system. Compared with scale up, this approach provides a more cost-effective and smooth scalability, since it can incrementally extend the system by adding more resources to an initially low-cost hardware set. Although scale out may inevitably increase the management complexity, it can also improve the reliability and availability of the system, in some case because of redundancy.

DB2 provides many kinds of technologies to support both scale up and scale out mechanisms effectively. You may refer to the paper *High Availability and Scalability Guide for DB2* and *DB2 Integrated Cluster Environment Deployment Guide* for more details.

In this section, we focus on exploring how to leverage and design effective data partitioning mechanisms at both the Database and Table levels when implementing multi-tenant scenarios.

2.1 Database Partitioning

The database partitioning feature (DPF) of DB2 Viper extends the capability of DB2 in the parallel, multi-partition environment, improving the performance and scalability of very large databases. In the scale up scenario, we can create more than one database partition on the same physical machine to take advantage of the SMP architecture. While in the scale-out scenario, partitions can be created in multiple physical machines. Each partition has its own common memory, CPUs, disk controls and disks.

The data within a database can be distributed across one or more partitions associated with the database partition groups. A distribution key is a column (or group of columns) that is used to determine the partition in which a particular row of data is stored. The distribution key value is hashed to generate the partition map index value (Range from 0 to 4095), which maps to the database partition the record resides in.

In the multi-tenant context, there are primarily two types of database partitioning patterns: using application-based distribution keys, or using tenant-based distribution keys. These represent two principles in determining how to organize and isolate tenants' data amongst multiple partitions, which has an impact on many important aspects of the multi-tenant applications, such as performance, scalability and isolation.

The remainder of this section will introduce these two patterns and their pros and cons to identify the scenarios in which each of these approaches would be best followed.

2.1.1 Application-based Distribution Key

This pattern is a traditional and typical database partitioning approach. It chooses one or more good distribution key candidates according to the application-specific domain knowledge. In this case, one tenant's data may be stored across multiple partitions simultaneously. Figure 5 shows an example of this approach.

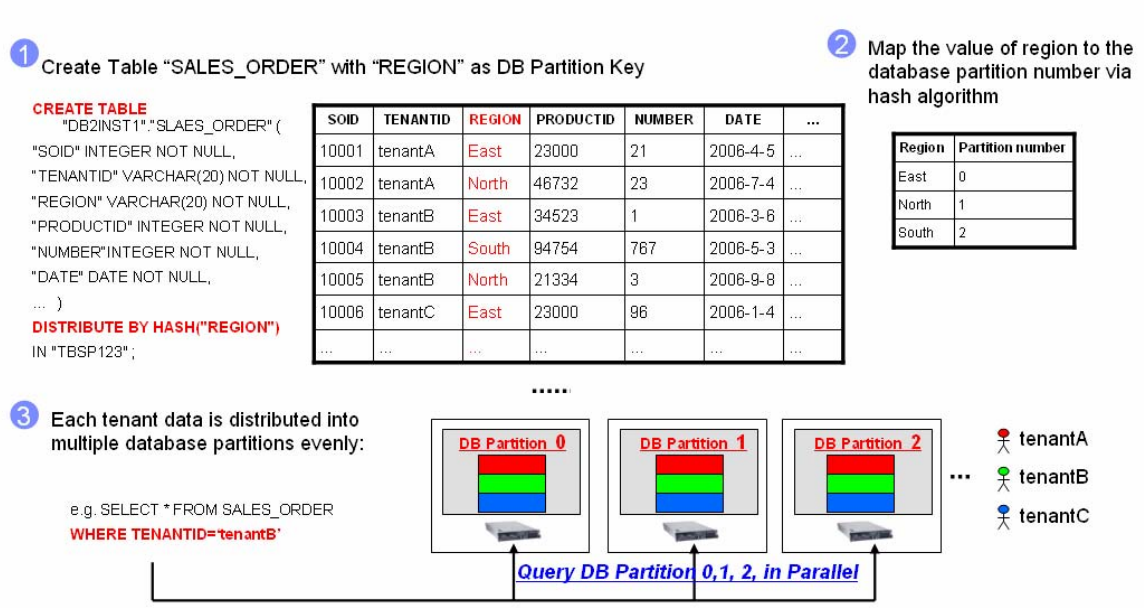

Figure 5 Sample of Application-based Database Partitioning

Generally, the distribution key is specified with the CREATE TABLE statement. In this sample, the column "REGION" of the "SALES_ORDER" table is selected as the distribution key via statement "*DISTRIBUTED BY HASH("REGION")*". By using the hash algorithm, the order records of different regions will be distributed into different partitions evenly, regardless of which tenants they belong to. For each query, the central coordinator will forward the request to multiple nodes and return the data it receives to the tenant.

The main merit of this approach is the good load balancing among multiple partitions & nodes via selecting a suitable distribution key. However, since this approach does not take the characteristics of a multi-tenant scenario into consideration, it lacks the kind of effective isolation support required amongst tenants, which has impacts in the following areas:

- ✧ Performance Isolation: If a tenant issues a large volume of high-cost (or even malicious) requests in a short period, all tenants of the shared partition group may suffer from serious performance degradation or frustration.

- ✧ Availability Isolation: Since all of the tenants' data are distributed across the whole partition group, the failure of any partition may result in service unavailability to all tenants.

- ✧ Management Isolation: It's difficult to migrate, backup and restore data in a tenant-isolated way using this partitioning model.

Choosing a good distribution key is important for the even distribution of data across partitions and maximizing the use of collocated joins. Collocation between joining tables

means having the matching rows of the joining tables on the same database partition. This avoids the database manager having to ship rows between partitions.

2.1.2 Tenant-based Distribution Key

This approach stores each tenant's data in a single partition. It can simply use the column "*TenantId*" as the distribution key. However, to improve flexibility, we create a new column "*DBPKey*" in the table as the distribution key and associate a tenantid with the DBPKey. Using this approach, we needn't bind the partition of one tenant with its fixed tenant id, and can map one tenant to any specific partition freely by assigning or changing the "*DBPKey*" value directly.

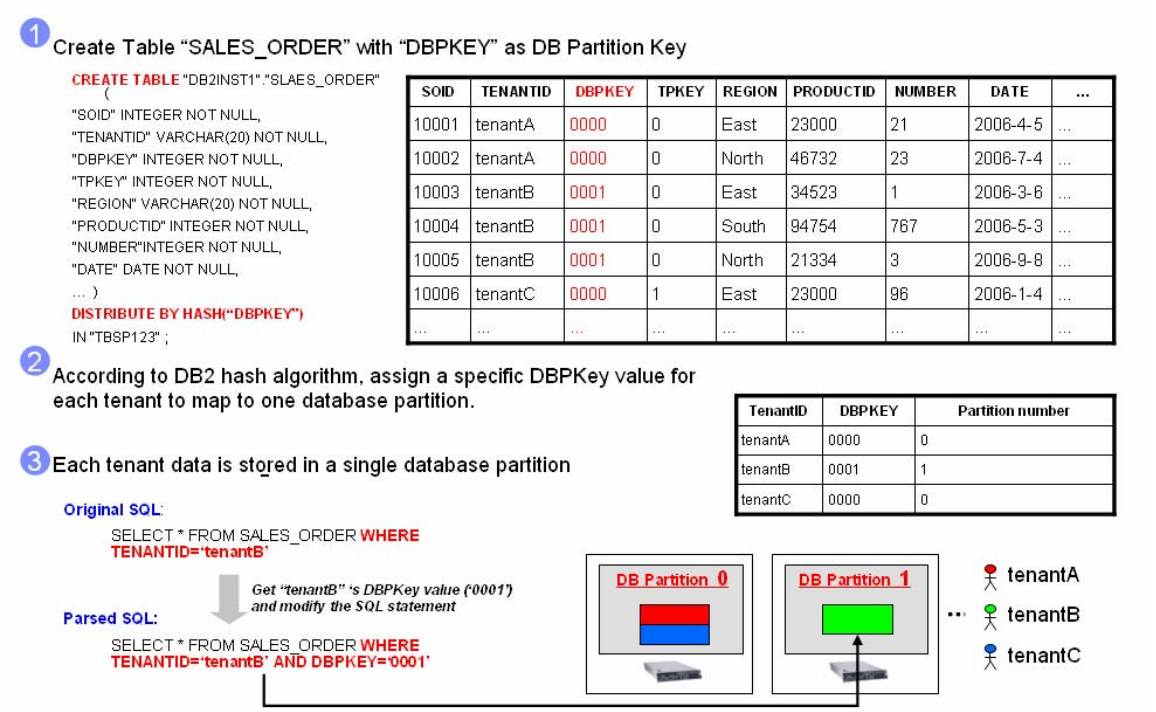

Figure 6 Sample of Application-based Database Partitioning

As illustrated in Figure 6, when a new tenant is on-boarding, the SaaS application operator will allocate a fixed *TenantId* and a *DBPKey* value mapping to a certain database partition. This metadata is stored in a repository to maintain the mapping relationship between tenants and partitions. When a data access request is received, we will first retrieve the tenant's *DBPKey* value from the metadata repository, and attach the sub clause "*DBPKEY=xyz*" to the original SQL statement. Then the database manager can automatically route the request to the corresponding partition the tenant resides.

As a tenant-aware (but application independent) partitioning approach, it provides better isolation support among tenants, since all data and operations of one tenant are contained in a single partition. The failure of a certain partition will not impact the service

availability to tenants in other partitions. In the management aspect, it's easier to implement functions such as tenant data migration, backup and restore etc., since no cross-partition operations are needed. This distribution key selection approach also eliminates the risks associated with cross partition joins.

Furthermore, in order to do better load balancing among partitions and optimize the overall database performance, we may design algorithms or policies to effectively migrate/distribute tenants among partitions via monitoring the load of tenants and partitions.

To sum up, using application-based distribution keys is more suitable for large enterprises that have relatively large volumes of data and heavy loads, since it can provide better parallel performance via effective load balance mechanisms across multiple partitions and machines. While for SMB tenants whose data size and transaction loads can be satisfied by a single database partition, the tenant-based distribution key is a better choice due to its more powerful isolation capability.

2.2 Table Partitioning

Table partitioning provides a way of creating a table where ranges of the data in the table are stored separately. Its merits can be shown through both performance optimization and management flexibility aspects. On one hand, it can potentially boost query performance through data partition elimination. On the other hand, it also makes managing table data much easier by providing partition roll-in and roll-out capability by using attach and detach clauses on the 'ALTER TABLE' statement. As we will explain in the following section, these two features are valuable in multi-tenant scenarios.

Determining the range selection criteria is one of the essential elements to table partitioning. Range selection defines the table partitioning key column, how many partitions you will have in the table, the volume of data that will be rolled-out, and limits to the data that the table will hold. Similar to database partitioning, we can add a "*TPKey*" column in the shared table as the table partitioning key. Each tenant will be assigned a specific *TPKey* value to identify its corresponding table partition. In practice, some application-based columns can be combined with the *TPKey* to build a more intricate table partitioning range.

For example, as illustrated in Figure 7, the *TPKey* and *REGION* columns of the "*SALES_ORDER*" table are combined together to form the union key. For the requests of one tenant, we may retrieve its corresponding *TPKey* value from the metadata repository, and add it to the original SQL statement for table partition elimination. The partition elimination is the ability of the optimizer to determine that certain ranges do not need to be accessed at all for a query. This greatly improves the performance of the multi-tenant applications, since all requests from one tenant will only be redirected to its dedicated table partitions. For example, a query request of tenant A will only operate over table partitions PAT_0E, PAT_0N and PAT_0S. Furthermore, by isolating different tenants'

data within dedicated table partitions, we can easily (un)load and migrate a tenant's data from the shared table using Viper roll-in and roll-out features. These features allow data partitions to be easily added or removed from the table without having to take the database offline.

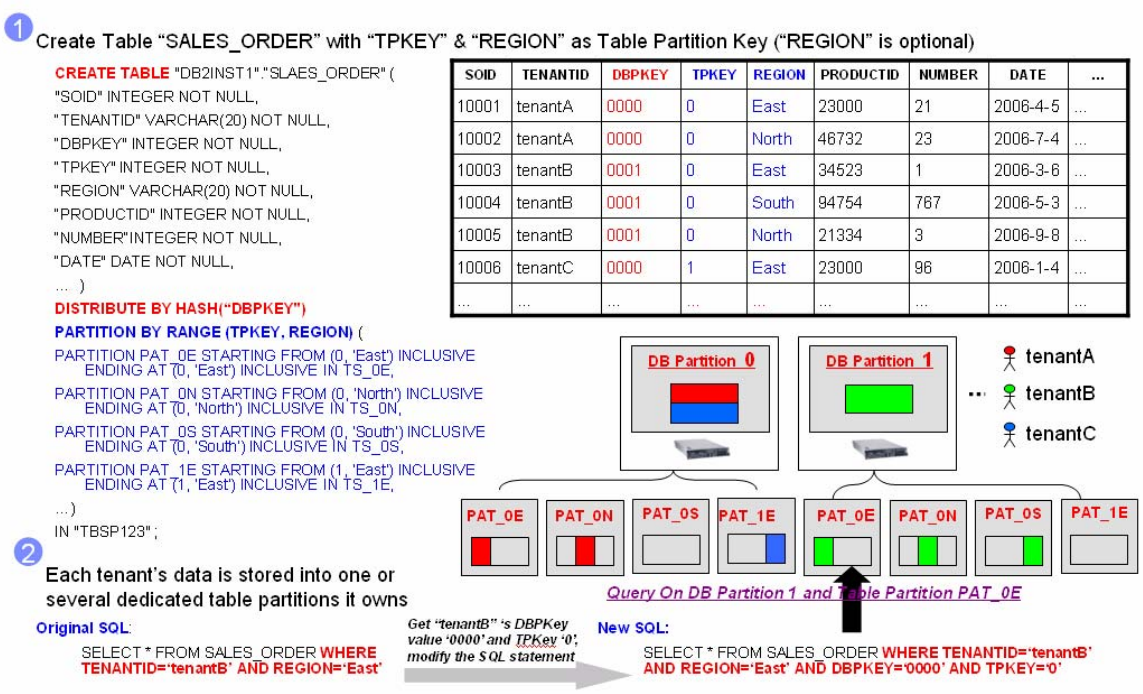

Figure 7 Sample of Tenant-based Table Partitioning

3. Maintenance

Per-tenant data backup and restore is a very important feature that should be provided in a multi-tenant scenario. However, in the shared schema/tables environment, it's difficult since all of the tenants' data are stored together. In this section, we propose a practical approach to realizing this feature based on the database and table partitioning technologies introduced before.

DB2 provides table space level online incremental backup and roll-forward support in a partitioned environment. You may refer to paper *Data Recovery and High Availability Guide and Reference* for details. The following are two samples:

(1) Online incremental backup of tablespace *tbsp1* in database *testdb*:

db2_all " db2 BACKUP DB testdb TABLESPACE tbsp1 ONLINE INCREMENTAL TO /home/db2inst1/BACKUPS include logs"

(2) Online roll-forward of tablespace *tbsp1* in database *testdb*:

db2_all 'db2 ROLLFORWARD DB testdb to 2007-10-31-14.21.56.245378 and stop TABLESPACE(tbsp1) online'

Our idea is to store each tenant's data from multiple shared tables in a dedicated tablespace isolated from other tenants via well designed database and table partitioning mechanisms.

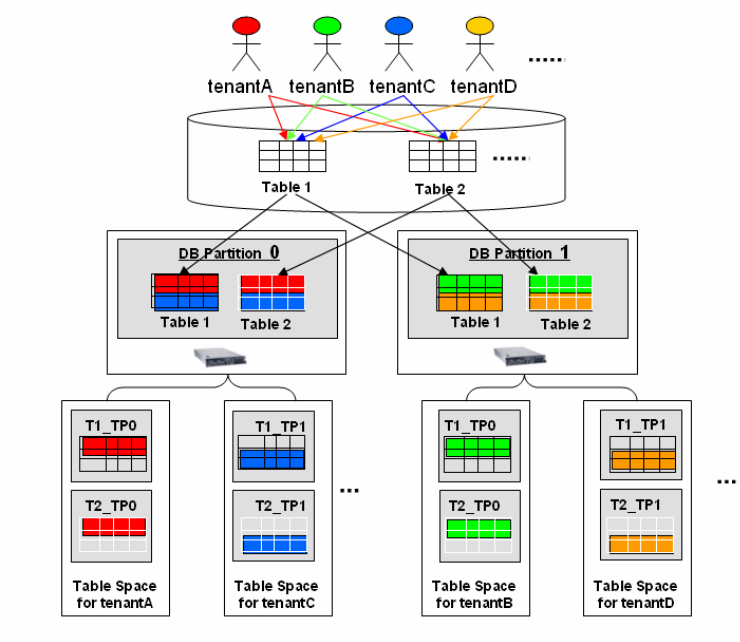

Figure 8 Sample of Tenant-based Table Space Management

As illustrated in Figure 8, by taking advantage of the tenant-based database and table partitioning approaches introduced previously, all data of one tenant will be stored in a single database partition, with a dedicated table partition for each shared table it utilizes. Within the database partition, we further allocate a dedicated tablespace for the tenant to contain all of its table partitions. This results in all data of the tenant being stored in the dedicated tablespace. In this way, we can easily backup and restore one tenant's data at the tablespace level without impacting other tenants sharing the same set of tables.

4. Extensibility

Typically, when utilizing a shared database schema in a multi-tenant environment, the task of customizing a database table to add tenant-specific data can be complex. Each shared table needs to contain a column that identifies the tenant (i.e. a 'tenant id' field), and the tenant data is accessed by filtering on that column. In terms of providing tenant-specific data elements, there are a number of potential implementation strategies that are often used, such as the following:

- User views: views created against shared database tables that only include the data for a single tenant. Access to the views is granted to the tenant of that data.

The views select the rows by tenant id, and include the specific columns in the table that the tenant is interested in. When adding new tenant-specific columns, the views of the other tenants do not need to be recreated as long as the other tenants do not want to include the new column.

- Pre-allocated columns: Creating a fixed number of additional columns in each table with a generic definition (such as varchar (x)). The content for these columns would be enforced through application logic in the user interface, i.e., enforcing numerics, data format etc, allowing each tenant to use the columns in the manner in which they see fit. This approach has several drawbacks. Since the columns exist for every row in the table, space is being consumed for those tenants that are not utilizing the additional columns. For those tenants which require a large number of custom columns, once you've used up all of the additional columns, you need to alter the table to add new columns, potentially affecting all of the tenants.
- Extension tables: tables that use the tenant id as key, and store additional tenant-specific data elements through the use of a record identifier in conjunction with the tenant id. In this scenario, the tenant id in the original table is used to join to an extension table to pull in additional tenant specific data, data which is not contained in the base table utilized by all the tenants. This is a recommended approach over pre-allocated columns since only tenants who use the additional fields will have them allocated.

This secion focuses on a fourth implementation strategy for shared schema, namely the use of the pureXML capabilities of DB2 Viper. Viper stores XML data in a hierarchical structure that naturally reflects the structure of XML. This structure along with new indexing techniques allows DB2 to efficiently manage this data and eliminate the complex and time-consuming parsing typically required for XML. The use of this approach raises the following two questions:

1. XML storage has been around for quite a while. What benefit does storage in pureXML format have over the current storage mechanisms, i.e. CLOB format and shredding?
2. Regarding extensibility, what is the advantage of storing data in pureXML format over storage in tradition SQL format?

4.1 PureXML vs Traditional XML Storage

Traditional XML storage requires storing whole XML documents in a CLOB column, or 'shredding' the XML document into multiple relational tables. When XML is stored as a CLOB, the XML document is retrieved as a single object, with no ability to query based on the data contained within the XML document. PureXML provides query capability on all of the elements within the XML document. It also provides indexing and search capability on those elements. On the other hand, shredding XML documents into multiple tables is time consuming and difficult. The benefit of pureXML is that the data is stored in a single table column, yet you are provided the benefit of indexing and search on the individual xml elements like you are in a shredding approach. So it is the ease of storage and access that differentiates pureXML from XML shredding.

4.2 PureXML vs SQL format

Using pureXML, the XML document is stored in a single db2 column. Therefore, the ddl for the table remains constant, regardless of the format of the XML elements. In a multi-tenant shared environment, this eliminates the need for DDL changes once the table has been defined (assuming all tenant-specific data is stored in the XML column), and provides the tenant with the ability to customize their data without affecting other tenants. Of course, regardless of whether you store the data in traditional SQL columns or XML, all database changes need to be considered in conjunction with the application level changes, to ensure that the application code and display code can accommodate the database changes.

Figure 9 below illustrates how the data for two tenants in a separate schema environment would be stored in custom tables. Extending those tables with new columns could be done without affecting other tenants.

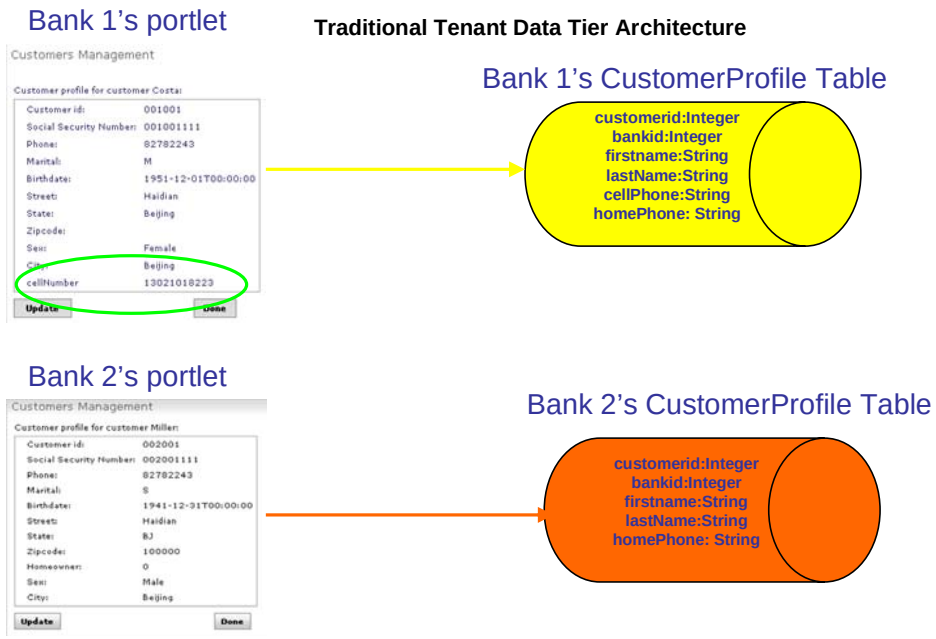

Figure 9 Data Storage in an isolated environment

Utilizing DB2 Viper, we created a single table in a shared schema environment that had two columns: tenant id and customerprofile. The customerprofile column was defined as

XML. The XSD for that column could differ by row – db2 allows for multiple xsds on the same xml column.

In Figure 10 you can see that Bank1's XML documents contain the element cell phone, while Bank2's doesn't. The schema for the table is the same, allowing both tenants' data to be stored in the same table. In this scenario the same XSD was used (see Figure 11), although it doesn't have to be.

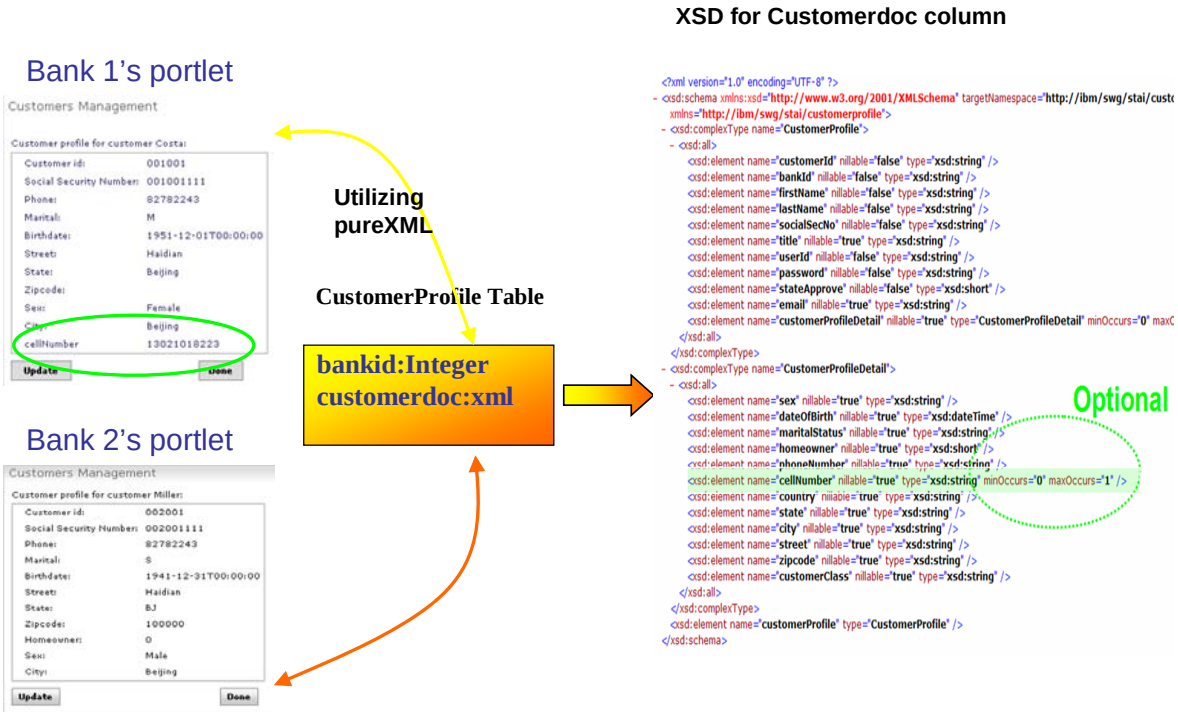

Figure 10 Utilizing pureXML data storage

```
- <xsd:schema targetNamespace="http://ibm/swg/stai/customerprofile">
  - <xsd:complexType name="CustomerProfile">
    - <xsd:all>
        <xsd:element name="customerId" nillable="false" type="xsd:string"/>
        <xsd:element name="bankId" nillable="false" type="xsd:string"/>
        <xsd:element name="firstName" nillable="false" type="xsd:string"/>
        <xsd:element name="lastName" nillable="false" type="xsd:string"/>
        <xsd:element name="socialSecNo" nillable="false" type="xsd:string"/>
        <xsd:element name="title" nillable="true" type="xsd:string"/>
        <xsd:element name="userId" nillable="false" type="xsd:string"/>
        <xsd:element name="password" nillable="false" type="xsd:string"/>
        <xsd:element name="stateApprove" nillable="false" type="xsd:short"/>
        <xsd:element name="email" nillable="true" type="xsd:string"/>
        <xsd:element name="customerProfileDetail" nillable="true" type="CustomerProfileDetail" minOccurs="0" maxOccurs=
      </xsd:all>
    </xsd:complexType>
  - <xsd:complexType name="CustomerProfileDetail">
    - <xsd:all>
        <xsd:element name="sex" nillable="true" type="xsd:string"/>
        <xsd:element name="dateOfBirth" nillable="true" type="xsd:dateTime"/>
        <xsd:element name="maritalStatus" nillable="true" type="xsd:string"/>
        <xsd:element name="homeowner" nillable="true" type="xsd:short"/>
        <xsd:element name="phoneNumber" nillable="true" type="xsd:string"/>
        <xsd:element name="cellNumber" nillable="true" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="country" nillable="true" type="xsd:string"/>
        <xsd:element name="state" nillable="true" type="xsd:string"/>
        <xsd:element name="city" nillable="true" type="xsd:string"/>
        <xsd:element name="street" nillable="true" type="xsd:string"/>
        <xsd:element name="zipcode" nillable="true" type="xsd:string"/>
        <xsd:element name="customerClass" nillable="true" type="xsd:string"/>
      </xsd:all>
    </xsd:complexType>
    <xsd:element name="customerProfile" type="CustomerProfile"/>
  </xsd:schema>
```

Figure 11 Customer profile XSD

As mentioned previously, database extensions need to be considered along with the changes required to the application and display code. In our scenario we simplified tenant data configurability through the use of an XML column to hold all tenant data, allowing for ease of data configuration. In addition we were able to simplify the application code changes required to implement the tenant specific portlets by utilizing the same XSD for the two tenants with optional XML elements. This allowed us to use the same web services for both tenants.

Conclusion

In conclusion, we've provided techniques on how to address multi-tenant technical challenges at the data tier, by utilizing the capabilities of DB2 and DB2 Viper. We recommend using Label-based access control (LBAC), to guarantee the security of your data. LBAC uses the DBMS to enforce database security, alleviating the tenant of the threat of SQL injection. In terms of scalability and performance, we focused on database and tablespace partitioning. The result of our analysis is that using application-based distribution keys is more suitable for large enterprises that have relatively large volumes of data and heavy loads, since it can provide better parallel performance via effective load balance mechanisms across multiple partitions and machines. Partitioning using a tenant-based distribution key is a better choice for smaller tenants whose data size and transaction loads can be satisfied by a single database partition. This type of partitioning provides powerful isolation capability which is required in a multi-tenant shared schema environment. In terms of maintenance, in order to provide tenant-isolated maintenance, you should store each tenant's data in its own tablespace. By using this approach you can use the DB2 tablespace backup and restore capabilities and only affect a single tenant's data at a time. And finally, regarding extensibility, by using pureXML you can isolate tenant-specific data in a single xml column in each table, providing flexibility for making changes to that data without affecting other tenants, and without requiring the use of a DBA to make DDL changes.

About the Author

Chang Jie Guo is a research staff member working on next generation service within IBM China Research Lab in Beijing, China. In rencent years, he focus on some key technologies in Software as a Service (SaaS) area including massive multi-tenancy, agile business process mangement (BPM) and Web 2.0 etc.
Contact: guocj@cn.ibm.com

Mary Taylor is an IT Specialist in the Strategic Technology Architecture and Incubation team in Software Group. She's been in IBM for over 20 years. She's currently working in the SOA area, focusing on how to compose Composite Business Services (CBS) in a Software as a Service (SaaS) environment.
Contact: marytaylor@us.ibm.com

Additional Resources:

DB2 V9 Administration Guide: Implementation

DB2 Label-Based Access Control: A Practical Guide

High Availability and Scalability Guide for DB2

DB2 Integrated Cluster Environment Deployment Guide

Data Recovery and High Availability Guide and Reference

Program with XML for DB2, Part 2: Leverage database support for XML in your application architecture