Ruminations on Multi-Tenant Databases

Dean Jacobs, Stefan Aulbach

Technische Universität München
Institut für Informatik - Lehrstuhl III (I3)
Boltzmannstr. 3
D-85748 Garching bei München
{dean.jacobs, stefan.aulbach}@in.tum.de

Abstract: This is a position paper on multi-tenant databases. As motivation, it first describes the emerging marketplace of hosted enterprise services and the importance of using multi-tenancy to handle high traffic volumes at low cost. It then outlines the main requirements on multi-tenant databases: scale up by consolidating multiple tenants onto the same server and scale out by providing an administrative framework that manages a farm of such servers. Finally it describes three approaches to implementing multi-tenant databases and compares them based on some simple experiments. The main conclusion is that existing database vendors need to enhance their products to better support multi-tenancy.

1 Hosted Services and Multi-Tenancy

In the hosted service model [GM02a, GM02b, Wa03], a service provider develops an application and operates the system that hosts it. Customers access the application over the Internet using industry-standard web browsers or Web Services clients. As the Internet has matured, hosted services have appeared for an increasingly wide variety of enterprise applications, including ones that manage sales, marketing, support, human resources, planning, manufacturing, inventory, financials, purchasing, and compliance [Th06]. While hosted services are attractive to all segments of the market, they are particularly appealing to small- to medium-sized businesses, which often lack the resources to maintain a complex data center. Hosted services are exploiting such "greenfield" opportunities to expand the overall size of the market.

In comparison to traditional on-premises solutions, hosted services can reduce the total cost of ownership of an application by aggregating customers together and leveraging economy of scale. This principle applies to both capital expenditures, e.g., for hardware and software, and operational expenditures, e.g., for bandwidth and personnel. Because a hosted service is focused on one application, the infrastructure and the procedures for managing it can be highly optimized: well-known examples here include the Google File System [GGL03] and Hotmail [Sm05]. Such optimizations are essential to support large numbers of small- to medium-sized businesses, which would otherwise be prohibitively expensive.

Multi-tenancy is an optimization for hosted services in which multiple customers are consolidated onto the same operational system, a technique pioneered by *salesforce.com* [Sa06, Co06]. Multi-tenancy allows pooling of resources, which improves utilization by eliminating the need to provision each customer for their maximum load. Multi-tenancy can also improve management efficiencies by providing a uniform framework for administering the system. A multi-tenant system should support both scale up and scale out [De99]: scale up by consolidating multiple customers onto the same server and scale out by having the administrative framework span a farm of such servers. Scale out is required because it is not cost effective to scale up a single server indefinitely.

The administrative framework of a multi-tenant system should support the ability to migrate a customer from one server to another within the farm. For example, a customer might start out on a server that manages trial accounts, be moved to a server that manages small production accounts, and grow until it is moved to a dedicated server that manages only one account. Migration should also be possible between farms, for example, to allow customers to be moved from one data center to another. The administrative framework should also support rolling upgrade, where the servers in a farm are upgraded to a new version of the application one at a time. Since it is difficult to generate realistic Internet-scale workloads for testing, rolling upgrades are needed to gain experience with new versions before they are put widely into production.

2 Requirements on Multi-Tenant Databases

Multi-tenancy can be applied at the database layer of a hosted service, where it can be very effective due to the high cost of provisioning and operating databases. This section argues that, in addition to managing customers' private data, a multi-tenant database should manage customer metadata and shared public data.

The administrative framework of a multi-tenant database should maintain metadata about customers, such as their contact information, their location in the farm, and the features they are allowed to access. Certain administrative operations will need to access this metadata alongside of customer data. For example, a service provider might want to scan the metadata to find all customers in a given region and then determine which of those customers have more than a certain amount of data. To support such operations, the administrative framework should offer a unified query language that obviates the need for a general-purpose programming language with embedded queries. In addition to being easier to use, this approach makes it simpler to execute administrative operations in bulk on individual databases in the farm.

The target application for a hosted service will generally have a base schema that specifies all of its standard data. A multi-tenant database should maintain an instance of this base schema for each customer. The unified query language should ensure that DDL statements for modifying the base schema and DML statements for transforming existing data within it are applied to all customers in the farm, within the context of a rolling upgrade. The ability to perform such operations in bulk on the individual databases is essential to minimize downtime during an upgrade. Many business applications, such as CRM or ERP, allow customers to extend the base schema, e.g., by adding new columns to existing tables and adding new tables. The administrative framework should maintain the specification of each customer's schema extensions as part of their metadata.

A service provider may want to maintain public information, such as area code data, census data, streaming stock prices, or industry best practices, that is accessible to all customers. Such shared data will generally be read-only, although there are cases where space-saving technologies like copy-on-write [PB03] should be used to allow customers to privately update the data. A multi-tenant database should allow customers to seamlessly access shared data alongside of their private data. This will entail either replicating the shared data on each server in the farm or providing a mechanism to access it remotely.

3 Implementing Multi-Tenant Databases

This section describes and compares three approaches to implementing multi-tenant databases: shared machine, shared process, and shared table. These approaches are increasingly better at pooling resources and executing administrative operations in bulk. However they increasingly break down the isolation between customers, weakening security and increasing contention for resources.

The discussion presents the results of some preliminary experiments on the memory and disk usage of five databases: PostgreSQL [Po06], MaxDB [Ma06], and three commercial databases cleverly code-named Commercial1, Commercial2, and Commercial3. Unless otherwise indicated, no specialized tuning was performed on these databases. The experiments used a simplified version of the CRM schema offered by *salesforce.com* containing eleven tables, each with about twelve fields.

The experiments were focused on the shared process approach: they created different numbers of schema instances, each with eleven empty tables, and loaded them into memory in the same database process. The experiments are preliminary in that the tables were not populated and no queries were executed against them. The results are shown in Table 1. In all cases, the storage requirements were close to being linear in the number of schema instances, hence only the end points at 1 and 10,000 instances are presented. PostgreSQL and Commercial1 produced various errors that prevented the experiments from completing and the final numbers are extrapolated. These errors were generally fixed limits on internal structures, such as the number of objects in a table space. All numbers are in megabytes.

	Memory 1 instance	Memory 10,000 instances	Disk 1 instance	Disk 10,000 instances
PostgreSQL	55	79	4	4,488
MaxDB	80	80	3	1,168
Commercial1	171	616	200	414,210
Commercial2	74	2,061	3	693
Commercial3	273	359	1	13,630

Table 1. Storage Requirements for Schemas Instances (in megabytes)

3.1 Shared Machine

In the shared machine approach, each customer gets their own database process and multiple customers share the same machine. An example of this approach is CasJobs, which supports analysis of data in the Sloan Digital Sky Survey database [Mu05]. This approach is popular in practice because it does not require modifying the implementation of the database. In addition, it does not substantially reduce customer isolation, particularly if each process is placed in its own virtual machine.

The main limitation of this approach is that it does not pool memory. The first column of Table 1 shows that the memory requirements to handle one schema instance vary from 55 MB to 273 MB across the five databases. This result indicates that this approach cannot scale beyond tens of active customers per server. Another limitation of this approach is that each database requires its own connection pool on each application server, so sockets will not be shared among customers. To improve this situation, the implementation could be modified to share memory and sockets among co-located database processes, possibly at a level below those processes.

Because isolation between customers is strong in this approach, executing administrative operations in bulk is not feasible: each database will execute queries on its own. In addition, providing seamless access to shared data requires a mechanism to access it remotely, otherwise it will end up being replicated multiple times on the same server. Customer migration is straightforward however, in that it entails simply moving files from one server to another.

An interesting issue is brought to light by the third column of Table 1, which shows the disk requirements to handle one schema instance. Commercial1 pre-allocates 200 MB for the database, although this could be reduced by additional tuning. Under this approach, this practice could result in terabytes of wasted disk space when multiplied across thousands of small businesses.

3.2 Shared Process

In the shared process approach, each customer gets their own tables and multiple customers share the same database process. For most databases, it does not matter whether each customer gets their own schema, since schemas are implemented using a lightweight prefixing mechanism. However, it is useful if each customer gets their own physical table space so that customer migration entails simply moving files from one server to another. In addition, it allows the system administrator to balance the I/O load by distributing customers across different backing disks.

This approach is considerably better at pooling memory. The second column of Table 1 shows that the memory requirements to handle 10,000 schema instances vary from 79 MB to 2,061 MB across the five databases. MaxDB stands out in that the memory requirements were the same regardless of the number of schema instances, suggesting that the data dictionary is being paged. Further experiments are required to determine the performance implications of this technique. In any case, this result indicates that this approach should easily scale up to thousands of active customers per server, a two orders of magnitude improvement over the shared machine approach. Further improvements could be provided by modifying the implementation to keep only one copy of the schema and having each instance refer to it. This technique could also be applied to the disk, although the third and fourth columns of Table 1 indicate that it would save only gigabytes of storage.

Since there is only one database in this approach, customers can share connection pools. There is a well-known downside to connection pooling however: all connections must be associated with a fixed principal who can access everything. This means that both security and the management of resource contention have to be handled at the application layer. Thus errors in the application code could allow one customer to access another customer's tables or prevent them from getting their fair share of resources. This problem could be mitigated by allowing the principal associated with a database connection to be picked up from the application server. Support for this feature is beginning to appear in databases today.

To execute administrative operations in bulk in this approach, the metadata embodied in the schemas has to be made available as data and it must be possible to create operations that are parameterized over the domain of table names. Some support for these features is available in databases today. These capabilities are also required in the administrative framework to handle queries that go across the individual databases in the farm.

In this approach, administrative operations such as adding customers, removing customers, and extending the base schema entail executing DDL statements. This requirement can be problematic for some databases, which behave poorly if schemas are modified while the system is in operation.

3.3 Shared Table

In the shared table approach, data from many customers is stored in the same tables, as illustrated in Figure 1. A `TenantId` column is added to each table to identify the owner of each row. Every application query is expected to specify a single value for this column. To allow customers to extend the base schema, each table is given a fixed set of additional generic columns. These columns might be of type VARCHAR, as shown in Figure 1, or they might have a mix of types. The data for the *n*-th new column of a table for each customer is placed in the *n*-th generic column of the appropriate type, after performing any necessary type conversions.

TenantId	Account	Name	…	Val0	…	Val100
1041	0021	Acme		1/3/95		----
1041	0029	Ball		3/7/72		----
1053	0016	Gump		red		----
1053	0049	Wonk		blue		----

Figure 1. `Account` Table in the Shared Table Approach

This approach is clearly the best at pooling resources. Its ability to scale up is limited only by the number of rows the database can hold, which should offer several orders of magnitude improvement over the shared process approach. Administrative operations can be executed in bulk simply by executing queries that range over the `TenantId` column. And since there is only one database, customers can share connection pools.

There are however several significant problems with this approach. First, since files on the disk have intermingled data from multiple customers, migration requires executing queries against the operational system. Second, such intermingling can impact the performance of accessing a customer's data, since it may be spread out across many pages. Third, security can be pushed down into the database only if different access privileges can be assigned to different rows in the same table. Fourth, the use of generic columns is feasible only if the database has a compact representation for sparse tables. Fifth, if typing of the generic columns has been abandoned, it will be hard to use column-oriented features such as indexes and integrity constraints.

The sixth and biggest problem with this approach is that queries intended for a single customer have to contend with data from all customers, which compromises query optimization. In particular, optimization statistics aggregate across all customers and table scans go across all customers. Moreover, if one customer requires an index on a column, then all customers have to have that index. To improve this situation, the implementation could be modified to treat the `TenantId` column as if it defined separate tables. Alternatively, this problem could be handled above the database in a query preprocessor that performed its own optimizations.

4 Conclusions

This paper has argued that multi-tenant databases are essential for hosted services to manage high traffic volumes at low cost. Multi-tenant databases may also be useful behind the firewall of a single organization, e.g., to handle multiple branches that have the same schema.

This paper described and compared three approaches to implementing multi-tenant databases. The authors' opinion is that, among these, shared-process is the most promising approach. Shared-machine will continue to be used in circumstances where security concerns are paramount. Modifications to increase resource pooling could provide moderate improvements in scalability, however they may not be worth the effort of performing delicate inter-process coordination. Shared-table swings too far in the other direction, potentially compromising performance, customer migration, security, and typing. Shared-process can be made more scalable by maintaining only one copy of the base schema, and more secure by having the principal associated with a database connection be picked up from the application server. In any case, it is clear that existing database vendors need to enhance their products to better support multi-tenancy.

It is interesting to note that support for multi-tenancy generally requires blurring the line between data and metadata: shared-process offers bulk execution of administrative queries by allowing them to be parameterized over the domain of table names, while shared-table repairs query processing by having the `TenantId` column implicitly define tables. Since these features require only limited support for dynamic manipulation of schemas, it should be possible to support them without compromising query optimization.

Acknowledgements This paper benefited greatly from discussions with Alfons Kemper, Mike Carey, Jim Gray, Adam Messinger, Anno Langen, and the people who started it all, the developers at salesforce.com.

References

[GM02a] Greschler, D.; Mangan, T.: Networking lessons in delivering 'Software as a Service' - Part I. Int. J. Network Mgmt 12, 2002.

[GM02b] Greschler, D.; Mangan, T.: Networking lessons in delivering 'Software as a Service' - Part II. Int. J. Network Mgmt 12, 2002.

[Wa03] Walsh, K. R.: Analyzing the Application ASP Concept: Technologies, Economies, and Strategies. CACM Vol. 46, No. 8 August, 2003.

[Th06] THINKstrategies: http://saas-showplace.com/. 2006.

[GGL03] Ghemawat, S.; Gobioff, H.; Leung, S.: The Google File System. Proc. 19th ACM Symp. on Operating Systems Principles, New York, 2003.

[Sm05] Smoot, P.: A Conversation with Phil Smoot. ACM Queue Vol.3 Issue 10, 2005.

[Sa06] http://www.salesforce.com/, 2006.

[Co06] Salesforce.com boasts of architectural superiority. Computer Business Review Online. http://www.cbronline.com/article_news.asp?guid=A6156ED5-55F7-432B-95BC-7CF413E1BB88&z=rc_CRM, 2006.

[De99] Devlin, B.; Gray, J.; Laing, B.; Spix, G.: Scalability Terminology: Farms, Clones, Partitions, and Packs: RACS and RAPS. MSR-TR-99-85, December 1999.

[PB03] Peterson, Z. N. J.; Burns, R. C.: Ext3cow: The Design, Implementation, and Analysis of Metadata for a Time-Shifting File System. Technical Report HSSL-2003-03, Hopkins Storage Systems Lab, The Johns Hopkins University, 2003.

[Po06] http://www.postgresql.org/, 2006.

[Ma06] http://www.mysql.com/products/maxdb/, 2006

[Mu05] Mullane, W.; Li, N.; Nieto-Santisteban, M.; Szalay, A.; Thakar, A.; Gray, J.: Batch is back: CasJobs, serving multi-TB data on the Web. Proc. IEEE Int. Conf. on Web Services, 2005.