



Optimized Database Design for Software-as-a-Service (SaaS)

Noel Vega
Senior Database Administrator
TOMOS Software, LLC





Software-as-a-Service

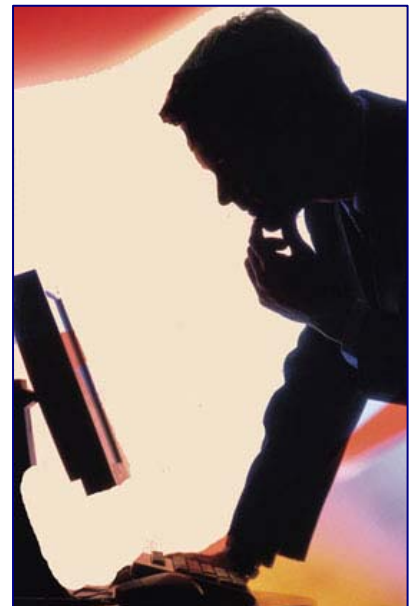
In today's economic climate, many firms large and small have difficult financial decisions to make. Balancing a budget with the need for quality software can be a tricky task for even the most seasoned administrator. With large enterprise systems, this juggling act is even trickier. In addition to the purchase price, licensing fees, hardware and other up-front costs, there are hidden costs to be incurred during the lifetime of the product. These include upgrade fees, maintenance and, in many cases, staffing of in-house resources to administer the software. It is not uncommon for system costs to exceed \$1 million (USD).

Lately, a new model has emerged to give companies a cheaper alternative to in-house solutions: Software-as-a-Service (SaaS). A client pays a per-user fee for a predefined period of time to access the solution. In exchange, the software resides in one central location, managed outside of each client. All clients access the same software, though certain features may be enabled or disabled for specific clients depending on what they choose to pay for. As the software is upgraded, all clients receive the upgrade without any physical intervention on their end. Hardware, networking, even disaster recovery are all handled without any staffing costs.

Design Considerations

Enterprise applications are designed to provide maximum uptime and superior performance. In many cases, companies hire dedicated resources for the sole purpose of resolving issues in a timely manner and to minimize system outages. In order to compete with these products, SaaS offerings must, at a minimum, be comparably resistant to outages and performance degradation; costs savings alone cannot justify a drop in productivity or the sudden loss of data.

At the core of most reliable SaaS offerings is a well-designed, secure, fault-tolerant database design. Perhaps more so than for their enterprise counterparts, SaaS databases must be able to keep problems isolated as much as possible. While it is true that a database crash could bring down an enterprise application, causing a company to lose productivity until the issue is remedied, a crash of a poorly-designed SaaS database could cause an outage for *multiple* companies, possibly causing all of them to lose faith in the product and take their business elsewhere. On the other hand, if reliability is favored at the expense of performance, companies may not feel that the application can scale to their needs and will refuse to sign up in the first place. Finally, as a SaaS database will contain data from multiple companies, it is imperative to design it in a secure, isolated fashion. Failing to ensure that no client can see another's data could spell the end for a SaaS offering.



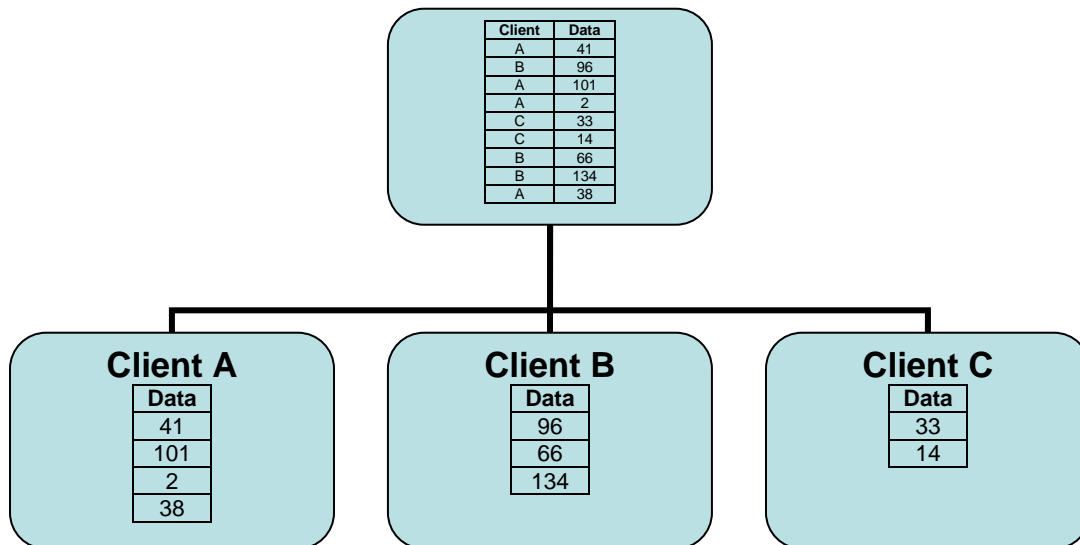


A Sample SaaS Database Design

Database crashes are a nightmare for any business. While almost all systems and environments are backed up to one degree or another, there is the potential for data loss, system outage and other problems to be dealt with. As mentioned earlier, this nightmare is worse for a SaaS product, as it could lead to outages for all clients if not properly planned for.

One design which addresses this issue involves assigning each client their own database, rather than hosting all data together in one. Each database would contain a copy of all tables necessary to store client-specific data. One central database would contain tables for handling any application-specific tasks such as user authentication, logging and more. If the central database were to crash, the application would be rendered unavailable – but no client data would be at risk. Similarly, if a client database were to crash, the outage would be limited to that client only; no other client would be prevented from accessing the system, nor would any data loss affect any other client.

The design also has the advantage of improving performance for clients across the board. By giving each client their own database, table sizes are necessarily reduced, leading to improved lookup times. For instance, assume a given table contains 10,000,000 rows for 1000 clients (an average of 10000 rows per client) under a schema where all client data is maintained in a common database. By splitting the schema up into one database per client, each client now has a copy of that table with only their data in it, meaning that the single table with ten million rows is now 1000 different tables with only 10000 rows in each on average.



There are, however, pitfalls to avoid when implementing this design. For instance, code referencing this data must be absolutely sure to alias all table references. Not doing so could return the correct table from the wrong client, causing that client's private data to be displayed to an unauthorized user. This can be avoided by assigning each user to a specific client and maintaining a specific client database name in the central database for each client, such that a specific username maps to one and only one client database name. This database name can then be stored by calling code and used to reference the proper client tables or stored routines.



A second pitfall relates to schema maintenance. In a traditional model (where all data is in a common database), updating a specific table's schema involves the execution of a single set of statements. Under the proposed SaaS design, however, the same set of statements must be executed against all client databases. Failing to do so could cause functionality to work for some clients but not for all. A well-designed tool can manage this for you; for instance, you can design a PHP page interfacing with a MySQL SaaS database to look up all client databases and execute an update statement against each. The same page could be used to run OPTIMIZE TABLE statements for a specific table across all schemas. Care must be taken to restrict access to such a page, of course.

Backup Procedures

As with all good applications, proper backup procedures and a solid disaster recovery plan should be in place to ensure a prompt resumption of service in the event of an outage. For SaaS companies, this involves a minimum of 2 Web servers and 2 database servers. Optimally 4 different machines would be used for this; however, if only 2 machines are used for this purpose they should serve the same purpose for both the Web server and the database server on that box. This means that both the Web server and the database server on one machine should be Production servers, while those on the other machine should be backup servers.

Minimally, database schema changes should be pushed to both a Production machine and at least one backup machine during code deployment cycles. Each database – Production and all backups – should schedule nightly flat file backups to allow any of the servers to revert to an older state in the event of a corruption or crash. After the nightly backups are done, the Production backup should then overwrite all backup server database instances to ensure that the version of each database as of the most recent end of day is available on all machines.

If networking and hardware costs are not an issue, a replication or clustering setup is the ideal database setup for maximum availability and minimized loss of data. Through a replication setup, any statement issued to a master database instance that modifies either the schema or data in that database instance is also sent and executed against all slave database instances connected to the master for replication. In a cluster setup, multiple machines serve as a single logical database instance and share the duties of storing data. If correctly designed, data is redundantly stored and updated in multiple nodes simultaneously such that the failure of one node does not result in an outage of the database.



About The Author:

Noel Vega is the Senior Database Administrator for TOMOS Software, LLC and was instrumental in the design and creation of the database schema for TOMOS. He has over 4 years of experience in software quality assurance as both a functional and performance test engineer and has successfully implemented automated and manual software testing projects at many Fortune 500 firms.

He holds a BS in Computer Science from Columbia University and is a Certified MySQL Developer, Certified MySQL DBA, Certified MySQL Cluster DBA, Zend Certified Engineer for PHP 5, and Sun Certified Java Programmer for Java 5.

About TOMOS Software, LLC:

TOMOS Software, LLC is a Delaware corporation headquartered in New York City. The TOMOS product is *the* light-weight, SaaS solution for Application Lifecycle Quality Management, created by a team of senior management and technology experts specializing in software application lifecycle management.

For more information on TOMOS and TOMOS Software, LLC, visit www.reachsimplicity.com or e-mail us at info@reachsimplicity.com.