Stored Procedures:

Business Logic in an N-Tiered Architecture

Preface

This little paper is derived (plagiarized) from a number of sources. In many cases I used their text verbatim; in others I rewrote if for concision, clarity, and flow. I added my own comments in places. This is not intended to be a scholarly work, and does not meet with scholastic standards for attributions. There are links in the text to some of the source documents, but these attributions do not meet with generally accepted standards for attributing text. Therefore, feel free to call this plagiarism. Since this is the case, you would do well not to cite this little paper, but instead cite the links in question.

Introduction

Why are we still using Stored Procedures in databases? It was once considered good form to embed business logic into the database, but separating the business logic from the data store is now generally considered an industry best practice (although not in every case.) Stored procedures have been on the decline for at least a decade, ever since the N-tier architecture replaced client-server. The decline has only been accelerated by the adoption of Object-Oriented (OO) languages like Java, C#, Python, etc.

If we are using an object-oriented framework with Python (for example), we have no need to deal with stored procedures. On the other hand, do we really need to forbid all stored procedures? Do we really need to boil the sea? While stored procedures are no longer a best practice, neither are they something to be avoided at all costs. Like most things in life, there are costs and benefits to be weighed, and informed engineering and business decisions made.

Issues involving Stored Procedures

1. The N-Tier Architecture separates presentation, logic, and data. Note that these are logical layers rather than physical tiers; some argue you could map the logic tier/layer to the database just as easily as you could to an application. However, this not only ignores the architectural pattern defined by the N-Tier concept, but the manner in which N-Tier works in practice. The term "tier" implies some inter-process communication is taking place, something that would not happen if the database was both the logic and data tier.

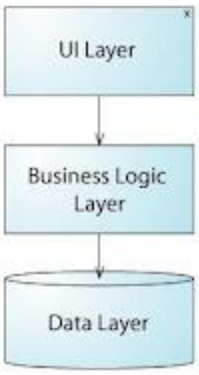

Figure 1: N-Tier Architecture

2. Stored procedures break your application into two languages and platforms according to rules that are often random, especially as the database is modified over time. Stored procedures can be (and often are) implemented as quick fixes rather than carefully thought out modifications, breaking the N-Tier architectural pattern and causing maintenance problems downstream.

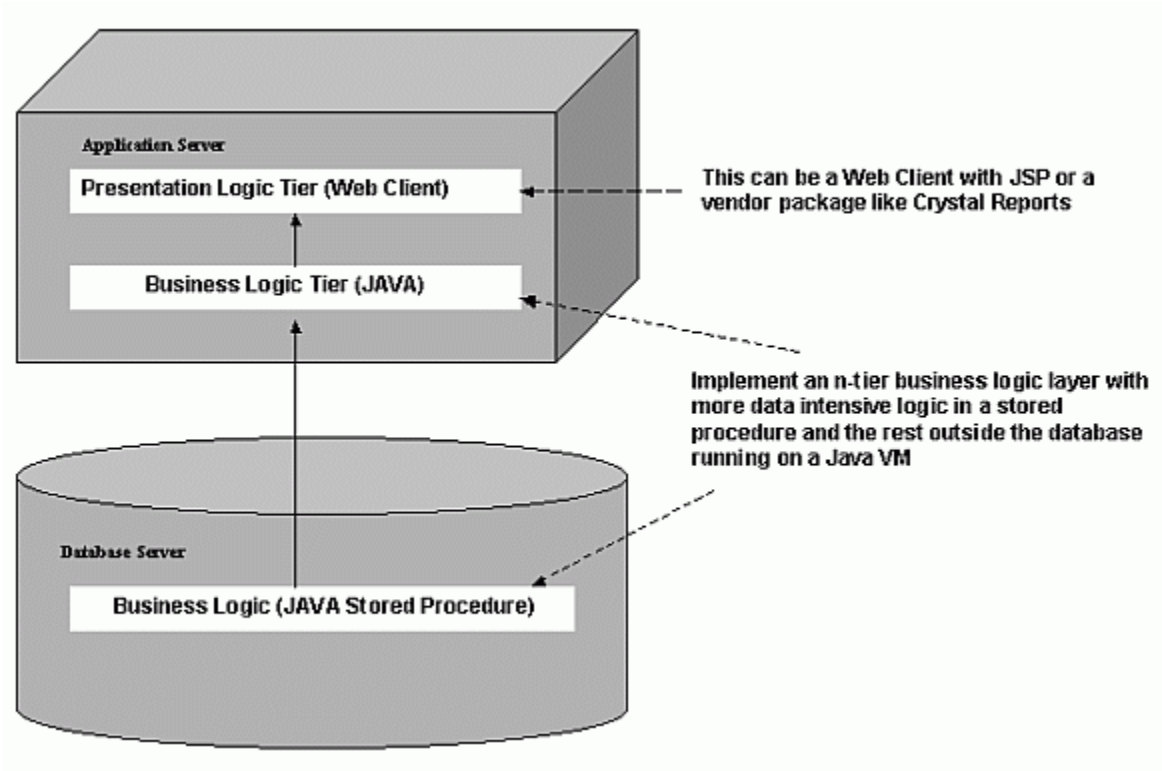

Figure 2: N-Tier with Business Logic Split Between Layers

3. In general, the proper place for business logic is in the logic tier of the application, not in the database. Putting logic in the DB is mixing up the tiers. The DB should be the data tier and not used as an all purpose computing engine.
    a. Database Administrators (DBA) will likely disagree with this, as modern Database Management Systems (DBMS) have the capability to perform much

of the business logic computations; it may seem like a waste to let that capability go unused.

b. Stored procedures and triggers that are not part of the business logic may have a place within the database. In particular, stored procedures can be used as part of the general data management process.

4. Writing your application in OO languages generally requires some sort of Object-Relational Mapping (ORM).
    a. With the logic-tier and data-tier separated, DB schema changes are isolated from ORM changes, allowing some independence between applications and database schema.
    b. The use of application and/or business logic embedded into the data tier through triggers and stored procedures makes the case for OO programming much less valid, and makes the ORM an incomplete representation of the application logic.

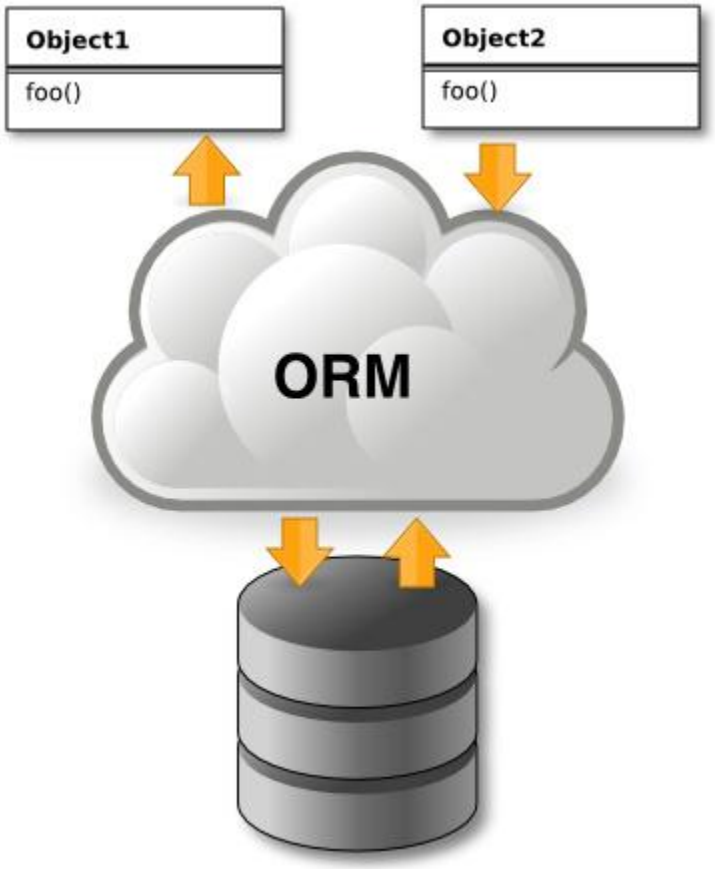

Figure 3: Object-Relational Mapping

5. You can't test stored procedures as seamlessly as the rest of the application logic code in your conventional unit test projects.

6. Stored procedures aren’t as conducive to test-first programming while writing code.
    a. Stored procedures are not necessarily conductive of test-first practices, but not everything that is computable can be test-first'ed.

    b. Running any of the Assert methods on the stored procedure at least requires a DB connection, which by definition makes it more of an integration test than a unit test.

7. Stored procedures aren't as easy to debug as application code in your IDE.
    a. Debugging shouldn't be an issue since stored procedures should contain nothing more than easy-to-verify SQL statements and cursors.
    b. Debugging should take place by first testing and debugging the SQL statements in code, and then moved into stored procedures.

8. It is difficult to maintain versioning control & source control of Stored Procedures vs. normal code.
    a. A stored procedure is a text file that you upload in your database engine (which takes it, compiles it and installs it for execution.) Normal code is part of the application itself.
    b. Conceivably, the stored procedure source code could be saved in, say, CVS, ClearCase or whichever Software Configuration Management (SCM) program that was in use. (How often is this done, human nature and time constraints being what they are? And even if this is done, it is a matter of policy rather than a systemic constraint.)

9. While stored procedures give you code reuse and encapsulation, this comes at the cost of other agile design goals, such as maintainability. Moreover, this is not the only way to achieve code reuse and encapsulation.
    a. Over time, stored procedures and triggers can proliferate in the database to the point where making a change to the database has unanticipated consequences.
    b. Over time, you'll likely end up with multiple, incompatible SPs that do essentially the same job, with no benefit.

10. Implementing stored procedures in the data-tier limits portability and ties you to a particular DB (vendor lock-in.)

11. While some say stored procedures protect you from SQL injection attacks, it is actually the parameterized querying that protects you --- which you can easily do in plain text SQL querying. If your stored proc is using any type of dynamic SQL along with a string parameter, you've opened yourself up to a SQL injection attack.

12. The hardest thing to scale out is the database. If you have your business logic in the database, should you ever need the DB to scale it will be more difficult and expensive. If the business logic is in the logic-tier of the application, adding another Windows/Linux server is cheaper and easier then adding a DB server and a paying for another DB license. It is easier to scale the app layer than the database.

How not to use Stored Procedures

1. Don't put complex logic beyond data gathering (and perhaps some transformations) in them. It is ok to put some data massaging logic in them, or to aggregate the result of multiple queries with them. But that's about it. Anything beyond that would qualify as business logic which should reside somewhere else.
2. Don't use them as your sole mechanism of defense against SQL injection. You leave them there in case something bad makes it to them, but there should be a slew of defensive logic in front of them - client-side validation/scrubbing, server-side

validation/scrubbing, possibly transformation into types that make sense in your domain model, and finally getting passed to parametrized statements (which could be parametrized SQL statements or parametrized stored procedures.)
3. Don't make databases the only place containing your stored procedures. Your stored procedures should be treated just as you treat your C# or Java source code. That is, source control the textual definition of your stored procedures.
4. Stored procedures should not contain low-level decision logic, and they should never simply encapsulate an otherwise simple query. There are few benefits and many drawbacks.

How & Where to use Stored Procedures

1. One good reason to use a stored procedure is when you must make a complex, multi-staged query that pulls from multiple collated sources.
2. Your application requires data that needs to be transposed or aggregated from multiple queries or views. You can offload that from the application into the db. Here you have to do a performance analysis since a) database engines are more efficient that app servers in doing these things, but b) app servers are (sometimes) easier to scale horizontally.
3. Use stored procedures for fine grain access control. You do not want someone running Cartesian joins in your DB, and you cannot just forbid people from executing arbitrary SQL statements like that either. A typical solution is to allow arbitrary SQL statements in development and UAT environments, while forbidding them in system test and production environments. Any statement that must make it to system test or production goes into a stored procedure, code-reviewed by both developers and DBAs.
4. Any valid need to run a SQL statement not in a stored procedure goes through a different username/account and connection pool (with the usage highly monitored and discouraged.)
5. In systems like Oracle, you can get access to LDAP, or create symlinks to external databases (say calling a stored procedure on a business partner's DB via VPN.) This is an easy route to spaghetti code. (This is true for all programming paradigms, and sometimes you have specific business/environment requirements for which this is the only solution.) Stored procedures help encapsulate that nastiness in one place alone, close to the data and without having to traverse to the app server.
6. Whether you run this on the DB as a stored procedure or on your app server depends on the trade-off analysis that you, as an engineer, have to make. Both options have to be analyzed and justified with some type of analysis. Saying putting business logic in the logic-tier is a best practice does not mean that all stored procedures in the data-tier is a bad practice; saying otherwise is an engineering cop-out.
7. In situations where you simply cannot scale up your application server (i.e. no budget for new hardware or cloud instances) but with plenty of capacity on the DB back-end (this is more typical that many people care to admit), it pays to move business logic to stored procedures. This can lead to anemic domain models, but sometimes you can make a business case for bad engineering decisions.

http://programmers.stackexchange.com/a/66084

Analysis of using Stored Procedures in the Data-Tier

Nesting

Let's say you have a stored procedure that returns a dataset you want to use; how can you use it in future stored procedures? The mechanisms on SQL Server for that are not very good. EXEC INTO...only works to one or two levels of nesting. Or you have to pre-define a work table and have it process keyed. Or you need to pre-create a temp table and have the procedure populate it. But what if two people call a temp table the same thing in two different procedures that they never planned to use at the same time? In any normal programming language, you can just return an array from a function or point to an object/global structure shared between them (except for functional languages where you'd return the data structure as opposed to just changing a global structure...)

Code Reuse

How about code reuse? If you start putting common expressions into UDFs (or even worse sub queries) you will slow the code to a halt. You can't call a stored procedure to perform a calculation for a column (unless you use a cursor, pass the column values in one by one, and then update your table/dataset somehow). So basically to get the most performance, you need to cut/paste common expressions all over the place which is a maintenance nightmare. With a programming language you can create a function to generate the common SQL and then call it from everywhere when building the SQL string. Then if you need to adjust the formula you can make the change in a single place.

Error Handling

How about error handling? SQL Server has many errors that immediately stop the stored procedure from executing and some that even force a disconnection. Since 2005 there is try/catch but there are still a number of errors that cannot be caught. Also the same thing happens with code duplication on the error handling code and you really cannot pass exceptions around as easily or bubble them up to higher levels as easily as most programming languages.

Speed

A lot of operations on datasets are not SET oriented. If you try to do row oriented stuff, either you are going to use a cursor, or you are going to use a "cursor" (when developers often query each row one by one and store the contents into @ variables just like a cursor, even though this is often slower than a FORWARD_ONLY cursor). Some operations in SQL Server run for an extended period of time, but when rewritten in Perl will finish in minutes. When a scripting language that is 20-80x slower than C smokes SQL in performance, you definitely have no business writing row oriented operations in SQL.

Programs with a lot of client SQL can become slow because of poor thinking about transaction boundaries. One of the reasons stored procedures appear to be fast is that stored procedures force very, very careful design of transactions. ORM's don't magically force careful transaction design. Transaction design still has to be done just as carefully as it was when writing stored procedures.

Common Language Runtime (CLR)

SQL Server does have CLR integration and a lot of these issues go away if you use CLR stored procedures. But many DBAs don't know how to write .NET programs, or they turn off

the CLR due to security concerns and stick with Transact SQL. Also even with the CLRs you still have issues of sharing data between multiple procedures in an efficient way.

T-SQL, CLR Integration, & Managed Code

Transact-SQL is specifically designed for direct data access and manipulation in the database. While Transact-SQL excels at data access and management, it does not have programming constructs that make data manipulation and computation easy. For example, Transact-SQL does not support arrays, collections, for-each loops, bit shifting, or classes. While some of these constructs can be simulated in Transact-SQL, managed code has integrated support for these constructs. Depending on the scenario, these features can provide a compelling reason to implement certain database functionality in managed code.

Managed code is better suited than Transact-SQL for calculations and complicated execution logic, and features extensive support for many complex tasks, including string handling and regular expressions. With the functionality found in the .NET Framework Library, you have access to thousands of pre-built classes and routines. These can be easily accessed from any stored procedure, trigger or user defined function. The Base Class Library (BCL) includes classes that provide functionality for string manipulation, advanced math operations, file access, cryptography, and more.

> Note: While many of these classes are available for use from within CLR code in SQL Server, those that are not appropriate for server-side use (for example, windowing classes), are not available.

One of the benefits of managed code is type safety, or the assurance that code accesses types only in well-defined, permissible ways. Before managed code is executed, the CLR verifies that the code is safe. For example, the code is checked to ensure that no memory is read that has not previously been written. The CLR can also help ensure that code does not manipulate unmanaged memory.

http://msdn.microsoft.com/en-us/library/ms254498(v=vs.80).aspx

Transact-SQL facilities

Take a look at the Transact-SQL facilities. String Manipulation? The Java String Class/Tokenizer/Scanner/Regex classes are far superior. The same goes for Hash Tables/Linked Lists/etc. The Java Collection and the .NET framework are way more evolved languages than Transact SQL; in fact, T-SQL is not properly a programming language, as it is only useful in a single problem domain.

Scalability

In general, the hardest thing to scale out is the database. If all your business logic is in the database, then when the database becomes too slow you are going to have serious challenges. If you have your business logic in a business layer, you can just add more caching and more business servers to boost performance. Traditionally, adding another server to install Windows/Linux and run .NET/Java is much cheaper than adding another database server and another SQL Server instance. SQL Server does have more clustering support now, so if you do have a lot of money, you can add clustering or even do some log shipping to make multiple read only copies. But overall this will cost much more than just having a write behind cache or something.

Business Logic

Surprisingly, the business logic is not always the same across the enterprise. In an ideal world you could put all the logic in stored procedures and share that logic between applications. But quite often the logic differs based on the applications and your stored procedures end up becoming overly complex monoliths that people are afraid to change, because they cannot determine the implications of changing. However, with a good object oriented language you can code a data access layer which has some standard interface/hooks that each application can override to their own needs.

Security

Stored procedures can be good for security. You can cut all the access to the underlying tables and only allow access through the stored procedures. With some modern techniques like XML you can have stored procedures that do batch updates. Then all access is controlled through the stored procedures so as long as they are secure/correct the data can have more integrity.

However, security at the DB level isn't granular enough to make context-aware decisions. Because of performance and management overhead, it's unusual to have per-user connections as well - so you still need some level of authorization in your app code. You can use role based logins, but you will need to create them for new roles, maintain which role you're running as, switch connections to do "system level" work like logging, etc. And, in the end, if your app is pwned - so is your connection to the DB, meaning you've gained nothing.

SQL Injection

The SQL injection argument doesn't really apply so much anymore since we have parameterized queries on the programming language side. Also, really even before parameterized queries, a little replace ("'", "''") worked most of the time as well (although there are still tricks to use to go past the end of the string to get what you want).

Large Datasets

On the plus side, stored procedures are more efficient for working with a big dataset and applying a multiple queries/criteria to shrink it down before returning it to the business layer. If you have to send a bunch of huge datasets to the client application and break down the data at the client it will be much more inefficient than just doing all the work at the server.

Consultants, Vendors, and Lock-in

What's best for a company is very often not best for a consulting firm or other software vendor. A smart company desires to have a permanent advantage over its competitors. By contrast a software vendor wants to be able to hawk the same solution to all the businesses in a particular industry, for the lowest cost. If they are successful in this, there will be no net competitive advantage for the client.

Applications developed by consultants come and go, but the corporate database lives forever. One of the primary things an RDBMS does is to keep junk data from entering the database. This can involve stored procedures. If the logic is good logic and highly unlikely to change from year to year, why should it not be in the database, keeping it internally consistent, irrespective of whatever application is written to use the database? Years later

someone will have a question they want to ask of the database and it will be answerable if junk has been prevented from entering the DB.

On the other hand, DB vendors want to promote lock-in, and could well encourage the use of things like triggers and stored procedures. While there may be good technical reasons for this, it could also be a case of the software vendor securing your long-term commitment to their product, and thereby enhancing their long-term bottom line.

Summary

Overall, SQL and Transact SQL are great languages for querying/updating data. But developers should avoid stored procedures when coding any type of logic, doing string manipulation, or doing file manipulation --- you'd be surprised what you can do with a command shell. From a code maintainability point of view, stored procedures are a nightmare. Also, stored procedures are a problem when it comes time to change your DB platform, creating vendor lock-in. There are circumstances where stored procedures may be appropriate---the decision requires both a technical and a business case analysis. In some cases, while the best technical solution may be to put business logic into the logic-tier, the business case might require the use of stored procedures instead. Ultimately, there are compelling arguments for avoiding using stored procedures in an n-tier architecture, where the advantages for putting business logic in the logic-tier generally outweigh the convenience of using stored procedures.

http://programmers.stackexchange.com/a/65810

See Also:

http://www.codinghorror.com/blog/2004/10/who-needs-stored-procedures-anyways.html

http://programmers.stackexchange.com/questions/65742/stored-procedures-a-bad-practice-at-one-of-worlds-largest-it-software-consulting

http://ora-00001.blogspot.com/2011/07/mythbusters-stored-procedures-edition.html

http://c2.com/cgi/wiki?StoredProceduresAreEvil